Fully Deniable Interactive Encryption

Ran Canetti* Sunoo Park† Oxana Poburinnaya‡

July 2, 2020

Abstract

Deniable encryption (Canetti *et al.*, Crypto 1996) enhances secret communication over public channels, providing the additional guarantee that the secrecy of communication is protected even if the parties are later coerced (or willingly bribed) to expose their entire internal states: plaintexts, keys and randomness. To date, constructions of deniable encryption — and more generally, interactive deniable communication — only address restricted cases where only *one* party is compromised (Sahai and Waters, STOC 2014). The main question — whether deniable communication is at all possible if *both* parties are coerced at once — has remained open.

We resolve this question in the affirmative, presenting a communication protocol that is *fully deniable* under coercion of both parties. Our scheme has three rounds, assumes subexponentially secure indistinguishability obfuscation and one-way functions, and uses a short global reference string that is generated once at system set-up and suffices for an unbounded number of encryptions and decryptions.

Of independent interest, we introduce a new notion called *off-the-record deniability*, which protects parties even when their claimed internal states are inconsistent (a case not covered by prior definitions). Our scheme satisfies both standard deniability and off-the-record deniability.

*Boston University and Tel Aviv University. Email: canetti@bu.edu
†MIT and Harvard. Email: sunoo@csail.mit.edu
‡Boston University. Email: oxanapob@bu.edu

Contents

1 Introduction

The ability to communicate secret information without having any prior shared secrets is a central pillar of modern cryptography [DH76, RSA78, GM84]. However, standard definitions and existing algorithms for secure communication only guarantee security if the parties' local randomness remains hidden. If the parties' secret keys or randomness are exposed, say as a result of coercion or bribery, secrecy is no longer guaranteed. Moreover, the transcript in common encryption and key exchange schemes often "commits" the sender to the plaintext, in that each transcript is consistent with only one plaintext and randomness.

To address this issue, Canetti, Dwork, Naor and Ostrovsky [CDNO96] introduced the notion of *deniable encryption*, which provides a mechanism for preserving the secrecy of communicated plaintexts even in the face of post-communication coercion or bribery.[1] Specifically, deniable encryption (or, more generally, deniable interactive communication) introduces additional algorithms, called *faking algorithms,* that are not present in standard secure communication definitions. The faking algorithms allow the communicating parties to present fake internal states (including keys and randomness) that make any communication transcript appear consistent with any plaintext of the parties' choice. Concretely, an adversary should not be able to tell whether the sender and the receiver gave it the true keys, randomness, and plaintext, or fake ones.

When the communicating parties have a secret key that was shared ahead of time, deniable encryption can be simple. The classic one-time-pad scheme is perfectly deniable: having sent $c = k \oplus m$, the parties can claim that they sent any plaintext m' by claiming that $k' = c \oplus m'$ is their true key. However, shared-key deniable schemes fail to address the crucial question of how to deniably agree on a shared key in the first place. Indeed, existing key exchange protocols are "committing" in a way that precludes deniability. For instance, in Diffie-Hellman key exchange, there exists only one key consistent with any given transcript, so it is impossible to equivocate a one-time pad key generated using Diffie-Hellman key exchange. Thus, the core question here is how to deniably transmit a value (or equivalently, to establish a shared key) *without any pre-shared secrets.*

This setting turns out to be much more challenging. Even the restricted case where only the *sender* is coerced (or bribed) was fully resolved only much later, assuming indistinguishability obfuscation, in the breakthrough work of Sahai and Waters [SW14].[2] The case where only the *receiver* is coerced or bribed follows from the sender-only case via a general transformation, at the cost of an additional message [CDNO96]; hence, the [SW14] scheme implies a 3-round receiver-deniable protocol. This transformation can also be extended to handle the case where the adversary may coerce either party, *but only one of the two*. Furthermore, as demonstrated by Bendlin, Nielsen, Nordholt, and Orlandi [BNNO11], *any* receiver-deniable encryption protocol must take at least three rounds of communication.

Constructing *bideniable* encryption protocols, namely encryption protocols that guarantee deniability in the unrestricted case where both the sender and the receiver can be simultaneously coerced or bribed, has remained open:

Do there exist bideniable encryption protocols, with any number of rounds?

Bideniability is a significantly stronger property than any of the restricted variants above, where the adversary

[1]While our results address both bribery and coercion, bribery might be the better setting to keep in mind. Indeed, protecting against bribery is more challenging in that the parties are *incentivized* to disclose all internal state, including all random choices.

[2]Prior to [SW14], we only had the partial solution of [CDNO96], where the adversary's distinguishing advantage decreases linearly with ciphertext size; in particular, to get indistinguishability with negligibly small advantage, one has to send superpolynomially long ciphertexts.

only learns the internal state of either the sender or the receiver.[3] Indeed, when both parties are coerced, the adversary obtains a complete transcript of an execution, including all the random choices, inputs and outputs of both parties. This means that the adversary can now fully run this execution, step by step, and compare it against the recorded communication. Even so, as long as the sender and receiver follow the protocol during the actual exchange of messages, bideniability guarantees that any (real or fake) internal state provided by the parties looks just as plausible as any other (real or fake) one.

Off-the-record deniability. When the attacker bribes or coerces *both* parties, a new concern emerges: what happens if the plaintext claimed by the sender is different from that claimed by the receiver? This could arise in various scenarios: the parties might simply not have the chance to coordinate a story in advance (e.g., if they are separated and interrogated); or the parties might be incentivized to tell different stories (e.g., to protect themselves or those close to them); or the parties might find themselves incentivized to "defect" on each other as in a prisoner's dilemma. Still, standard bideniability (as defined by [CDNO96]) provides no guarantees for these cases.

1.1 Our Contributions

Our first contribution is defining a security guarantee, called *off-the-record deniability*, that holds even in the above setting, where the coerced (or bribed) parties' responses are inconsistent with each other.[4] Off-the-record deniability achieves protection akin to an ideal, physically protected communication channel where the communication leaves no trace behind. That is, off-the-record deniability guarantees that the communication transcript does not help the attacker to determine which of the two parties is telling the truth, if any. This holds even if the parties deviate from the protocol — as long as the deviation happens after the actual protocol execution completes. In other words, off-the-record deniability guarantees protection for each party *independently of the other party's actions*. This contrasts with standard bideniability where, for the security guarantee to hold, *both parties must lie, and their claims must be consistent*.

Our second and main contribution is the first encryption protocol that is both bideniable and off-the-record deniable. We call such protocols *fully deniable.* A fully deniable protocol provides protection akin to an ideal channel, in that after message transmission, each party can claim that the message was any value whatsoever (say, from a pre-specified domain) and the attacker has no way to tell which party, if any, is telling the truth.

We stress that, prior to this work, even the existence of bideniable encryption (without the additional off-the-record property) was an open question.

Theorem 1. *Assuming subexponentially secure indistinguishability obfuscation and subexponentially secure one-way functions, there exists a* 3*-message interactive bit encryption scheme that is **fully deniable** (i.e., both **bideniable** and **off-the-record-deniable**) in the common reference string model. In addition, the receiver's deniability is* public *(i.e., the true random coins of the receiver are not required to compute fake randomness of the receiver).*

Our common reference string (CRS) consists of six obfuscated programs: three for the sender (programs $\mathsf{P1}$ and $\mathsf{P3}$ for generating the first and third messages, respectively, and the sender faking program SFake) and

[3] We note that a related but different concept, *multi-distributional bideniability*, has previously been considered in a setting where both parties are attacked [OPW11]; see Appendix A.

[4] The *off-the-record messaging protocol* [BGB04] is a messaging protocol that shares our motivation of enabling encrypted communications as close as possible to an ideal private channel, but is otherwise unrelated to our off-the-record deniability notion.

	Type of deniability	Secure against adversaries that may...
I.	Sender-deniable	coerce S but not R
II.	Receiver-deniable	coerce R but not S
III.	Sender-or-receiver deniable	coerce either S or R but not both
IV.	Bideniable	coerce both S and R, who claim the same plaintext
V.	Off-the-record deniable	coerce both S and R, who claim two different plaintexts

Table 1: *Taxonomy of deniable encryption.* In each case, the adversary sees a communication transcript between sender S and receiver R. "Coercion" means the adversary may demand from a party an explanation (internal state) consistent with a plaintext m of the adversary's choice. Note: I∧II⇔III and IV∨V⇒I∧II∧III.

three for the receiver (programs P2 for generating the second message, the decryption program Dec, and the receiver faking program RFake). The scheme instructs the parties to run the obfuscated programs on their relevant inputs and uniformly chosen random inputs.

Designing the scheme requires addressing two main classes of challenges. First, the operation of the six programs should follow a certain "internal logic" — which ends up being necessary even in an idealized model where parties only have *oracle access* to the encryption scheme programs. We make this logic explicit by constructing and proving security of a fully deniable encryption scheme in this idealized model. While this construction is not used directly in the full-fledged scheme, it highlights key design difficulties. Interestingly, while many cryptographic primitives are trivial to construct in this idealized model, deniable encryption is still highly non-trivial; indeed, *our technical overview (section 2) is fully devoted to building deniable encryption in the idealized model.*

The next challenge lies in translating this idealized protocol to one that is provably secure when the programs are (a) actual programs and (b) protected only by indistinguishability obfuscation (IO), not ideal obfuscation. Here, we use the sophisticated tools developed in [KLW15, CHJV14, BPR15, BPW16] that were developed for dealing with obfuscated programs that are designed to be repeatedly run on inputs that were generated earlier by the program itself. Our situation is however significantly more complex: We have several programs that are designed to take inputs from each other with a specific and context-dependent set of constraints. We thus develop additional tools and abstractions that allow us to deal with this more complicated setting.

We now turn to discussing our definitions and constructions in more detail.

1.2 Fully Deniable Interactive Encryption: Definition in a Nutshell

Deniable interactive encryption is equipped with algorithms to generate protocol messages, to decrypt, and to generate fake randomness. We present the definition for the three-message case, since our protocol has three messages. A scheme consists of six programs $\mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}$ as follows.

- Program P1, run by the sender, takes as input a message m and sender random string s, and outputs a first message μ_1.
- Program P2, run by the receiver, takes as input a message μ_1 and receiver random string r, and outputs second message μ_2.
- Program P3, run by the sender, takes as input s, m, μ_1, μ_2 and outputs a third message μ_3.
- Program Dec, run by the receiver, takes as input r, μ_1, μ_2, μ_3 and outputs plaintext $\tilde{m}$.

- Program SFake takes as input the public transcript of the protocol (namely messages μ_1, μ_2, μ_3), the sender randomness s, the message m, a fake message m', and potentially some additional random input ρ_S, and outputs a fake random string $s_{m'}$ that is intended to explain the transcript as an encryption of m'.
- Program RFake takes as input the public transcript, the receiver randomness r, the message m, a fake message m', and potentially some additional random input ρ_R, and outputs a fake random string $r_{m'}$ that is intended to explain the transcript as decrypting to m'.

We define *correctness* in the natural way: if the sender runs $\mathsf{P1}, \mathsf{P3}$ with plaintext m and uniformly chosen s, and the receiver runs $\mathsf{P2}, \mathsf{Dec}$ with uniformly chosen r, the receiver must decrypt $\tilde{m} = m$ except with negligible probability.

Bideniability requires that no PPT adversary can distinguish the following two distributions: (1) a protocol transcript for plaintext m', and both parties' true random coins for that transcript; and (2) a protocol transcript for plaintext m and fake random coins which make that transcript decrypt to m'. That is,

$$(\mathsf{tr}(s, r, m'), s, r) \approx_c (\mathsf{tr}(s, r, m), s_{m'}, r_{m'}) \; , \tag{1}$$

where s, r are uniformly random, $\mathsf{tr}(s, r, m)$ is the transcript from running the protocol to transmit m with random inputs s for the sender and r for the receiver, $s_{m'} = \mathsf{SFake}(s, m, m', \mathsf{tr}(s, r, m); \rho_S)$, $r_{m'} = \mathsf{RFake}(r, m, m', \mathsf{tr}(s, r, m); \rho_R)$, and $\approx_c$ denotes computational indistinguishability.

Off-the-record deniability requires that no PPT adversary can distinguish between the following three cases:

- **The sender tells the truth** and **the receiver lies**. That is, the adversary sees a transcript for plaintext m, the sender's true random coins, and fake random coins from the receiver consistent with m'.
- **The sender lies** and **the receiver tells the truth**. That is, the adversary sees a transcript for plaintext m', fake random coins from the sender consistent with m, and the receiver's true random coins.
- **Both the sender and the receiver lie**. That is, the adversary sees a transcript for plaintext m'', fake random coins from the sender consistent with m, and fake random coins from the receiver consistent with m'.

That is,

$$(\mathsf{tr}(s, r, m), s, r_{m'}) \approx_c (\mathsf{tr}(s, r, m'), s_m, r) \approx_c (\mathsf{tr}(s, r, m''), s_m, r_{m'}) \; , \tag{2}$$

where s, r, tr are defined as in (1), and $s_m, r_{m'}$ are fake coins produced by running faking algorithms on the corresponding transcript.

Observe that bideniability implies that $\mathsf{tr}(s, r, m) \approx_c \mathsf{tr}(s, r, m')$, so a bideniable scheme is also semantically secure. Similarly, off-the-record deniable schemes are also semantically secure.

Full deniability. A scheme is **fully deniable** if it is both bideniable and off-the-record deniable. Full deniability provides protection akin to an ideal secure channel in that the parties can freely claim any plaintext was sent or received, and which guarantees protection even when parties' claims do not match.

1.3 A Very Brief Overview of the Construction

Our starting point is an elegant technique from [SW14] that transforms any randomized algorithm A (with domain X and range Y) into a "deniable version" using IO. The technique creates two obfuscated programs A' and F, where: A' is the "deniable version" of A; and F is a "faking algorithm" that, for any input $(x, y) \in X \times Y$, outputs randomness ρ such that $A'(x; \rho) = y$. Using this technique, for any protocol, we can equip parties with a way to "explain" any given protocol message sent: that is, to produce fake randomness which makes that protocol message consistent with any plaintext of the parties' choice.

Based on this, a first attempt at a bideniable scheme might be to apply the [SW14] technique to an arbitrary public-key encryption scheme to create obfuscated programs for encryption, decryption, sender-fake and receiver-fake — and then use the sender-fake and receiver-fake programs to "explain" the protocol messages one by one. However, this does not yield a bideniable encryption scheme: the [SW14] technique is guaranteed to work only when applied to independent algorithm executions, but here the algorithms are run on the same keys and randomness, protocol messages are interrelated, and any convincing overall explanation must consist of a sequence of *consistent* explanations across the algorithms.[5] The problem in a nutshell is that although the [SW14] technique could create a deniable version of any single program, applying the technique separately to the key generation, encryption, and decryption programs fails to achieve deniability with respect to the programs' *joint* behavior.

More concretely, it is problematic that the adversary can manipulate any given transcript and randomness to generate certain "related" transcripts and randomness, and then try running the decryption algorithm on different combinations of them. Next, we give some intuition as to why this is a problem.

The Accumulating Attack. A fake r (i.e., receiver randomness) can be viewed as a string which "encodes" or "remembers", explicitly or implicitly, an instruction to decrypt a certain transcript to a certain fake plaintext. An adversary can run RFake iteratively on a given r (and a series of related transcripts) to successively obtain $r_1, r_2, \dots$, hoping that each new application of RFake will add a new (ith) instruction into the "memory" of r_i in addition to all the preceding instructions. Since r_i is a bounded-length string which, information-theoretically, can carry only a fixed amount of information, sooner or later, one of the instructions will be lost from the "memory" of r_{i^*} for some i^*. It can be shown that, assuming r was fake, an adversary running RFake many times can obtain some r_i which does not carry the original r's instruction, and thus decrypts the original transcript honestly. (This attack first appeared in [BNNO11] in the two-message setting and was used to demonstrate impossibility of two-message receiver-deniable encryption.)

While the above attack does not carry over to the three-message case generically, it stills remains valid for many protocols: namely, for those protocols where it is easy, given the challenge transcript tr^*, to find transcripts "related" to tr^*. Here "related" means that these transcripts can be successfully decrypted using the same true randomness r that was used to generate tr^*. (In particular, in the two-message case, it is always easy to generate related transcripts (pk, c) by setting $pk = pk^*$ and setting c to be a fresh ciphertext with respect to pk.)

Therefore our approach, based on the above ideas, involves: (1) designing a protocol that prevents the adversary from computing related transcripts that force receiver randomness to "accumulate" information as described above, and then (2) applying the [SW14] technique to the algorithms for generating each message of this protocol. For the first step, we design such a protocol in the *oracle-access model*, where everyone (parties and adversaries) has only oracle access to the programs for computing protocol messages. Then, we

[5] Indeed, if this approach worked, it would yield two-message bideniable encryption, which is impossible [BNNO11].

adapt the construction to the setting where everyone has access to program code, obfuscated under IO.

Step 1 of our plan — designing a protocol resistant to the Accumulating Attack — itself consists of two key steps, further detailed below: (1a) design a "Base Protocol" that resists only some attacks, then (1b) augment the base protocol using the ideas of a *level system* and *comparison-based decryption*, to obtain a protocol secure in the oracle-access model (the "Idealized Protocol").

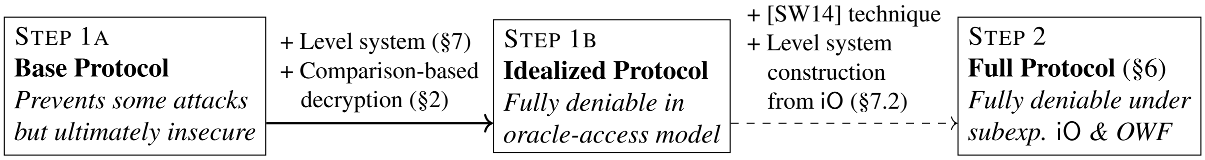

Figure 1: **The construction, step by step.** The second arrow is dashed because while *conceptually* the Idealized Protocol is a stepping-stone to the Full Protocol, *technically* the Full Protocol requires very different techniques, and must be proven from scratch rather than "building on" the Idealized Protocol.

STEP 1A: We design the Base Protocol in the oracle-access model as follows. The first message μ_1 is a PRF output for input (s, m) where s is the sender randomness s and m is the plaintext. The second message μ_2 is a PRF output for input (r, μ_1) where r is the receiver randomness r. The third message μ_3 is an encryption of (m, μ_1, μ_2). All keys for PRFs and encryption are hidden inside these programs (and not known to anyone, including parties). After exchanging μ_1, μ_2, μ_3 with the sender, the receiver runs Dec, which decrypts the ciphertext μ_3 and outputs m. In addition, the programs contain certain consistency checks: Dec returns an output only if it gets the correct r (i.e., consistent with μ_2), and $\mathsf{P3}$ only returns an output if it gets the correct s (i.e., consistent with μ_1).

The intuition for this design is as follows. The first two messages serve as "hashes" of the parties' internal state so far, and the next two programs — $\mathsf{P3}$ and Dec — produce output only if the parties "prove" to the programs (by giving randomness consistent with these "hashes") that they are continuing to execute the protocol on the same inputs used to generate these hashes. This design aims to prevent the adversary from computing related transcripts (and thus prevent the Accumulating Attack): for instance, an adversary must not be able to reuse μ_1, μ_2 from a transcript (μ_1, μ_2, μ_3) to compute a new ${\mu_3}'$ such that $(\mu_1, \mu_2, {\mu_3}')$ is also a valid transcript with respect to the same r. Section 2 gives more intuition about this.

STEP 1B: Unfortunately, the intuition from Step 1a is only partially correct: it turns out that it is still possible to generate related transcripts, although the design above indeed protects against "most" ways of generating them. Concretely, we describe a method Ω (detailed in Section 2.1) to compute a series of related transcripts differing only in the third message. Importantly, Ω is generic: it works for *any* three-message bideniable encryption scheme. Ω takes any transcript (μ_1, μ_2, μ_3) and, applied iteratively, produces a "chain" of valid transcripts $\mathsf{tr}_1 = (\mu_1, \mu_2, {\mu_3}^{(1)}), \mathsf{tr}_2 = (\mu_1, \mu_2, {\mu_3}^{(2)})$, and so on. However, the scheme from Step 1a importantly ensures that Ω is the *only* way to compute valid related transcripts: this is crucial for the security proof.

It remains to ensure that the adversary cannot learn the true plaintext from the chain of related transcripts produced using Ω (e.g., by performing the Accumulating Attack). To do this, we augment the Base Protocol with a *level system*, under which each ${\mu_3}^{(i)}$, generated using Ω, encodes a number which we call a *level*, which is set to that transcript's own *index* i.[6] Concretely, ${\mu_3}^{(i)}$ is an encryption of (m, μ_1, μ_2, i). Additionally, any fake randomness r_i — generated by running RFake on $(\mu_1, \mu_2, {\mu_3}^{(i)})$ — also encodes the level i of the

[6] Since Ω is inherently applied sequentially, the index i of each transcript produced by Ω is well defined.

transcript used to generate this r_i. The level i is encrypted, and so hidden from parties and the adversary, but the programs can decrypt and learn i using their internal keys. To complete the Idealized Protocol, we modify the decryption algorithm such that any fake r_i associated with level i may be used to decrypt transcripts with ${\mu_3}^{(j)}$ where $j > i$ ("correctness forward"), but decryption will fail (i.e., output $\perp$) if attempted with respect to r_i and ${\mu_3}^{(j)}$ where $j < i$ ("oblivious past"). We call this *comparison-based decryption behavior*.

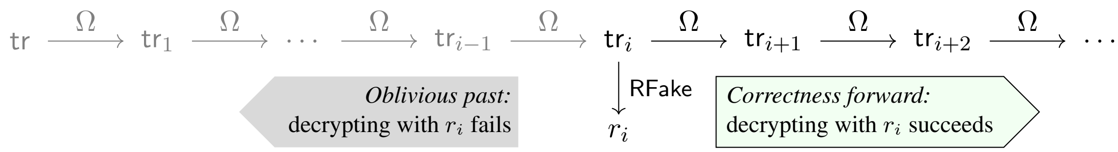

Figure 2: Comparison-based decryption behavior

The Idealized Protocol, just described, is fully deniable in the oracle-access model. In particular, it prevents the Accumulating Attack: intuitively, this is because comparison-based decryption ensures that an iteratively faked r only encodes the most recent faked plaintext, rather than accumulating a sequence of past fake plaintexts.

STEP 2: We obtain the Full Protocol by applying the [SW14] technique to the Idealized Protocol, which enables the parties to use *obfuscated programs* (not oracle access) to compute protocol messages and to generate fake randomness. Proving security of the resulting protocol based on IO presents a number of challenges. To start with, the security argument in the oracle-access model relies heavily on certain outputs of programs being hard to find provided the corresponding inputs are hard to find. To make the analogous argument with respect to IO, we need to show that such inputs *don't exist* (rather than being hard to find). Furthermore, as part of our construction, we introduce and construct a special primitive that could be called "deterministic order-revealing encryption": a variant of deterministic encryption where $\mathsf{Enc}(0)$ and $\mathsf{Enc}(1)$ must be indistinguishable, even given programs which homomorphically increment ciphertexts (producing $\mathsf{Enc}(2), \mathsf{Enc}(3)$ and so on up to some superpolynomial bound) *and* homomorphically compare them. (Intuitively, homomorphic comparison enables the comparison-based decryption behavior; see section 2). To argue security of this special deterministic encryption, we employ different primitives and techniques from the literature, including the *asymmetrically contrained encryption* from [CHJV14], and the proof techniques from [BPR15] to argue unreachability of the end of a superpolynomially-long chain.

This concludes the brief overview of our scheme. An in-depth technical overview of the scheme, including the intuition for the construction and the proof, can be found in section 2 (technical overview). Impatient readers may wish to jump ahead to the Idealized Protocol program descriptions in figures 6 and 7 or refer to the complete description of the Full Protocol in section 6.

1.4 On Verifiability of the Result

Several readers and reviewers expressed concern that it would be difficult to verify the result. While this concern is understandable given the intimidating size of the paper, we argue that the proof is less laborious than it looks and can be understood in a reasonable time by a knowledgeable reader.[7]

[7] Ironically, our efforts to make the proof as easily readable and verifiable as possible contributed to the size of this paper. For instance, for modularity we split the proof of security into 4 logical steps. Since each of the "core" changes requires some puncturing both before and after the change is made, such splitting introduced redundant hybrids which would not be necessary, had we instead written the whole proof in one shot. We also present the full code of the programs in each hybrid where the code changes happen,

1. While the total length of the proofs in this paper totals about 150 pages, we note that these proofs, like most cryptographic proofs, are very structured: they consist of a sequence of hybrid distributions with an argument of why each pair of subsequent distributions are indistinguishable. What makes this paper different is that *these distributions take a lot of space to describe*: since we frequently alter the code of the programs in the hybrid distributions, just the description of each hybrid distribution alone can easily take between 1/3 of a page (when no programs are changed) to 3 pages (when programs of both the sender and the receiver are changed).

   Yet, what matters for reading and verifying the proof is understanding the *difference* between two subsequent distributions (to verify that they are indeed indistinguishable). In this sense our proofs are no different than other cryptographic proofs: at each step we only make a simple atomic change, such as puncturing a PRF, switching the value to random, using the property of an extractor, changing the code of an obfuscated program, and so on.

2. While the number of hybrid distributions is somewhat high (approaching a hundred), the majority of the changes are very straighforward and easy to check (e.g., puncturing keys). For instance, the proof of security of the level system, despite taking almost 100 pages, almost entirely consists of applying one of only two changes at each step: either puncturing a public key of a special encryption scheme (with a reduction to iO security), or puncturing a corresponding secret key (with a reduction to a special security property of that encryption scheme).

3. Finally and perhaps most importantly, the proofs are fairly modular (the proof of security of deniable encryption is split into 4 logical steps, and the proof of security of a level scheme is splitted into 3). Even within each of these steps, there is a high-level strategy behind a low-level puncturing and "iO gymnastics" (the strategy for the encryption scheme itself is described at the end of the technical overview section, and the strategy for the level system is described in the level system section).

Thus, we encourage a curious reader to take a look at the proofs, and we are happy to explain the result or answer any questions by mail or in person.

1.5 Variants of Deniable Encryption and Other Related Concepts

Next, we overview some variants of deniable encryption, deniable communication, and surrounding concepts. While they are not directly relevant to this work, clarifying these concepts may prevent confusion.

- **Post-execution vs. adaptive coercion.** This paper considers coercion that happens after protocol execution. A broader definition, *adaptive* coercion, would capture coercion at some (arbitrary) point during the protocol execution (with uncoerced parties possibly unaware of the coercion).
- **Private vs. public deniability.** The deniability of the sender (or receiver, or both) is *public* [SW14] if the corresponding faking algorithm does not require the true randomness or the true plaintext as input. Our scheme has public receiver deniability (our RFake has syntax $\mathsf{RFake}(m', \mathsf{tr}; \rho_R)$). This means that anyone, not just the receiver, can produce fake random coins for the receiver. Note that any publicly deniable faking algorithm must be randomized: otherwise, the adversary could easily check if a claimed r is fake by comparing it to $\mathsf{RFake}(m', \mathsf{tr})$.

even if the change would be very small if described incrementally (such as replacing a full key with a punctured key): we believe this is beneficial for readability, given the number of hybrid distributions.

- **"Coordinated" schemes.** One can also consider "coordinated" schemes [OPW11] where a single faking algorithm takes as input the true coins of *both the sender and the receiver* at the same time. Such schemes require coordination between the sender and the receiver in order to compute fake randomness. Our scheme does not require coordination, but we note that prior to this work, even coordinated fully bideniable schemes were not known.

Deniable encryption is related to a number of other cryptographic concepts:

- **Incoercible key exchange** is equivalent to deniable encryption. The former can be used to establish deniable one-time-pad keys for encryption. The latter enables a sender to pick a random key and transmit it deniably to a receiver.

- **Flexible deniability.** [CDNO96] also proposed a weaker deniability notion, variously called *flexible deniability*, *multi-distributional deniability* ([OPW11, BNNO11, Dac12, AFL16, CIO16]), or *dual-scheme deniability* ([GKW17]). In a nutshell, this notion considers a setting where the coercer does not know which scheme is actually in use, and the coerced party has the freedom to "lie" in an undetectable way regarding the scheme that was actually used. (Equivalently, this notion assumes that the coercer does not expect to see some of the randomness used by the coerced party.) We note that none of the schemes in [OPW11, BNNO11, Dac12, AFL16, CIO16] are deniable in a setting where the coercer knows the scheme used in full and expects to see all the random coins of the coerced party. Appendix A provides detailed discussion of this notion and its limitations.

- **Non-committing (adaptively secure) encryption (NCE, [CFGN96])** is weaker than deniable encryption, and designed for a different purpose. NCE requires that a simulator can generate dummy ciphertexts that can later be opened to any plaintext. The differences with deniable encryption are twofold. First, deniable encryption enables faking of real ciphertexts (that carry plaintexts), while NCE ciphertexts can *either* be faked (if simulated) *or* carry a plaintext (if real). Thus, in NCE, parties cannot fake; only the simulator can. Secondly, fake opening on behalf of all parties in NCE is done by the same entity, the simulator, while in deniable encryption the sender and the receiver fake independently of each other.

  Bideniable encryption is strictly stronger than NCE: any bideniable encryption is also an NCE [CDNO96], but two-message NCE (e.g., [CDMW09]) is provably not bideniable, due to the three-message lower bound of [BNNO11].

- **Deniable authentication.** Deniable encryption is incomparable to deniable authentication. Deniable authentication allows the receiver of a message to authenticate the message's origin and contents, while preventing the receiver from convincing a third party who did not directly witness the communication that the message came from the sender (see, e.g., [DKSW09]). In contrast, in deniable encryption, the third party (adversary) may directly witness the communicated ciphertext and learn whether the parties have communicated with each other. The goal of deniable encryption is not to hide *whether* a party participated in a communication, but rather to preserve secrecy of the communication *contents* — even when parties are coerced (separately or jointly) to reveal their internal secrets.

1.6 Prior Work on Deniable Encryption

The definition of deniable encryption was introduced in 1996 by [CDNO96]. However, the techniques of that time fell short of achieving deniability: in fact, [CDNO96] presented a construction where the distinguishing advantage between real and fake opening was inversely proportional to the length of the ciphertext, thus requiring superpolynomially long ciphertexts in order to achieve cryptographic deniability. It was not until 2014 that Sahai and Waters presented the first (and, to date, the only) construction of sender-deniable encryption [SW14]. Their construction is based on indistinguishability obfuscation.

The [SW14] scheme can be transformed into a three-message *receiver*-deniable protocol using a generic transformation from sender- to receiver-deniable encryption (due to [CDNO96]) at the cost of one additional round, as follows: the *receiver* first deniably sends a random bit b to the *sender* deniably using the sender-deniable protocol, then the sender sends $b \oplus m$ to the receiver in the final round. Furthermore, if the sender sends $b \oplus m$ using the sender-deniable protocol rather than in the clear, the resulting scheme will be *sender-or-receiver*-deniable: that is, deniable against adversaries that coerce either one but not both of the parties. This final step incurs no additional rounds if (as in [SW14]) the message needs not be decided until the last round of the sender-deniable protocol. However, all these constructions rely heavily on the fact one of the parties' internal states remains hidden, and therefore fail to achieve bideniability.

Several prior works have focused on proving lower bounds for deniable encryption. [CDNO96] showed that a certain class of schemes cannot achieve better distinguishing advantage than inverse polynomial. [Dac12] extended this result to a broader class of constructions, showing that the same holds for *any* black-box construction of sender-deniable encryption from simulatable encryption. [Nie02] showed that any non-committing encryption, including bideniable encryption, can only reuse its public key an *a priori* bounded number of times; and therefore deniable communication must be interactive, even if two messages. Using different techniques, [BNNO11] showed that two-message receiver-deniable schemes, and hence also bideniable schemes, do not exist.

1.7 Organization of the Paper

The rest of the paper is organized as follows. Section 2 gives an informal yet almost complete description of the scheme, and outlines the main proof steps. Section 3 **formally defines bideniable and off-the-record deniable encryption**. Section 4 details the Idealized Protocol and security proof in the oracle-access model. Section 5 covers preliminaries: iO, puncturable PRFs, and other cryptographic primitives necessary for our construction.

Section 7 formally defines, constructs, and proves security of the **level system** which is an essential building block of our deniable encryption scheme. Section 6 gives a **complete description of our deniable encryption scheme** and states 4 main lemmas from which security of the scheme follows. Finally, Sections 8 and 9 give the **full proofs** of bideniability and off-the-record deniability of our Full Protocol.

2 Towards the Scheme: Technical Overview

This section provides an informal yet almost complete overview of our construction. The primary purpose of this section is to guide the reader through the process of designing the scheme, outlining concrete attacks and corresponding protection mechanisms. This should be helpful for readers who want to gain some intuition about the scheme and its security, but are not willing to read the whole 250-page full version [**?**], and for readers seeking to design a scheme from weaker assumptions (several issues described in this overview inhere in any 3-round deniable encryption, and could arise in schemes with more rounds too).

In this overview we describe the scheme in the oracle-access model. That is, we assume that all parties and the adversary have oracle access to programs $\mathsf{P1}, \mathsf{P2}, \mathsf{P3}$ (which generate the three messages of the protocol), decryption program Dec, and faking programs SFake, RFake.

We build our scheme in two main steps. As a first attempt, we try to avoid the known attacks on the 2-message case by considering a 3-message scheme. Next, we discuss some attacks and augment our scheme with levels and comparison-based decryption behavior, which yields our final scheme.

2.1 A First Attempt

Recall the [SW14] technique, mentioned above, that transforms any algorithm into a deniable version using indistinguishability obfuscation. Given this, a natural attempt to build deniable encryption is to take any (2-message) public-key encryption scheme and use the [SW14] technique to make each of its algorithms Gen, Enc, and Dec deniable. Concretely, the [SW14] technique takes any randomized algorithm A (with domain X and range Y) and outputs two obfuscated programs A' and F, where: A' is the "deniable version" of A; and F is a "faking algorithm" that, for any input $(x, y) \in X \times Y$, outputs randomness ρ such that $A'(x;\rho) = y$. Using this technique, we can take any protocol and equip parties with a way to "explain" any given protocol message they send: that is, to produce fake randomness which makes that protocol message consistent with any plaintext of the parties' choice.

This approach would allow, for example, the receiver to create a fake sk' decrypting a given ciphertext c to any plaintext of its choice. This sk' would even be indistinguishable from the real sk, to an adversary that only sees the purported secret key. But the problem is that the adversary sees other related information: e.g., it has the public key, so can run the encryption algorithm and generate outputs related to sk. The [SW14] technique does not work when applied to multiple programs with interrelated outputs: such as Gen, Enc and Dec.

Let us now outline the result of [BNNO11]: impossibility of bideniable encryption in 2 messages. This will give us insight on how to construct a 3-message bideniable encryption scheme while "avoiding" the impossibility. Also, the [BNNO11] result yields a concrete attack on the first-attempt scheme outlined above.

Impossibility of the 2-message case ([BNNO11]). [BNNO11] shows that even receiver-deniable (as opposed to bideniable) schemes are impossible with two messages. Their result is unconditional. Their proof shows that any 2-message receiver-deniable encryption scheme, even for a single-bit plaintext, can be used to deniably send any polynomial number of plaintexts, simply by reusing the first message (pk) and sending multiple second messages $c_1, \ldots, c_N$ (where N is arbitrarily, but polynomially, large); then they show that all these ciphertexts can be faked *simultaneously* using a *single* fake decryption key. This implies a method for compressing an arbitrary string beyond what is information-theoretically possible, as follows. To compress a string $b_1, \ldots, b_N$ from N bits (where N is larger than $|\mathsf{sk}|$) to $|\mathsf{sk}|$ bits: (1) prepare N encryptions of 0 under a single pk (call them $c_1, \ldots, c_N$);[8] (2) compute $\mathsf{sk}^{(1)} \leftarrow \mathsf{RFake}(\mathsf{sk}, c_1, b_1)$, $\mathsf{sk}^{(2)} \leftarrow \mathsf{RFake}(\mathsf{sk}^{(1)}, c_2, b_2), \ldots,$ $\mathsf{sk}^{(N)} \leftarrow \mathsf{RFake}(\mathsf{sk}^{(N-1)}, c_N, b_N)$. The final string $\mathsf{sk}^{(N)}$ is a compressed description of $b_1, \ldots, b_N$, since it is shorter than N and since the original string can be recovered by decrypting each b_i as $\mathsf{Dec}(sk^{(N)}, c_i)$. Since most strings cannot be compressed, we have a contradiction.

Stated differently, this impossibility says that a secret key which was faked multiple times to lie about different ciphertexts has to "remember" or store information about each lie; but information-theoretically, it cannot

[8]These ciphertexts do not depend on the string to be compressed and thus can be thought of as public parameters of the compression protocol.

remember more information than its length allows. Thus, at some point, such a secret key has to "forget" previous lies, and then it can be used to decrypt the original ciphertext to its real plaintext. That is, there is always an attack on any 2-message scheme, which roughly goes as follows. Assume the adversary sees a ciphertext c, claimed to encrypt plaintext m', together with a fake sk' that decrypts c to m'; but in reality, c encrypts m. The adversary can generate $N > |\mathsf{sk}|$ ciphertexts $c_1, \ldots, c_N$ as above, and run RFake iteratively to compute $\mathsf{sk}^{(N)}$ as above, and then compute $\mathsf{Dec}(\mathsf{sk}^{(N)}; c) = m$ to learn the true plaintext.

In summary, the core issue with the 2-message schemes is that for a single receiver message (i.e., pk) it is possible to efficiently generate many different sender messages (i.e., ciphertexts), such that all these ciphertexts are valid ciphertexts with respect to the same receiver key; this, in turn, means we must be able to use a single secret key to fake all the ciphertexts at once, which is information-theoretically impossible.

What would be the analogous argument in the 3-message case? Consider a 3-message scheme with messages (μ_1, μ_2, μ_3). If the scheme has the property that, given a receiver message μ_2, one can efficiently generate many different sender messages ${\mu_1}^{(i)}, {\mu_3}^{(i)}$ yielding valid transcripts $({\mu_1}^{(i)}, \mu_2, {\mu_3}^{(i)})$, then the scheme is subject to the [BNNO11] impossibility. For example, consider a 3-message scheme where the third message is a fresh encryption under freshly sampled random coins: this enables generating many third messages ${\mu_3}^{(i)}$ for any given μ_1, μ_2, and applying the [BNNO11] argument shows that any fake receiver randomness must remember a lie for each ${\mu_3}^{(i)}$, so this scheme is susceptible to the same attack as two-message schemes.

Base Protocol. Now we present our Base Protocol, which is insecure but will be augmented later to achieve a secure version. The scheme has parties first exchange two PRF values, then has the sender encrypt its plaintext m into a ciphertext μ_3 using program $\mathsf{P3}$, which the receiver can decrypt using program Dec. Before presenting the scheme formally, we give motivation for the design.

With the [BNNO11] impossibility in mind, a natural approach to building a 3-message scheme is to ensure that for any given first two messages μ_1, μ_2, only one consistent third message μ_3 can be efficiently computed. The Base Protocol achieves this using the following ideas.

1. The first message μ_1 "commits" to the sender's coins s and message m.
2. The third message μ_3 is a deterministic, symmetric-key encryption of m under a key K that is hardwired in programs $\mathsf{P3}$ and Dec and is unknown to parties.
3. $\mathsf{P3}(s, m, \mu_1, \mu_2)$ does a validity check before its output: if μ_1 is indeed a "commitment" to s and m, $\mathsf{P3}$ outputs μ_3; otherwise, it outputs $\perp$.

In other words, the only way for the sender to generate a valid μ_3 is to "prove" to $\mathsf{P3}$ that it is running $\mathsf{P3}$ on the same s, m used to compute μ_1. Thus, as long as K remains secret and the ciphertexts are sufficiently sparse, for any μ_1, μ_2, there is only one (efficiently computable) consistent μ_3.

So far, since μ_3 is computed under the same key K in each execution and it is not randomized, all executions with the same m yield the same μ_3, which is clearly insecure. To fix this, we let μ_3 encrypt not only m, but the first two messages μ_1, μ_2 as well, forcing different executions to have different third messages.

We have not yet discussed how the second message μ_2 should be computed, which actually depends on an extension of the attack based on [BNNO11], described above. Recall that we wanted it to be hard to compute multiple transcripts with the same μ_2: say, $({\mu_1}^{(i)}, \mu_2, {\mu_3}^{(i)})$. In fact, we also want it to be hard to convert a transcript (μ_1, μ_2, μ_3) with receiver randomness r into a different transcript $({\mu_1}', {\mu_2}', {\mu_3}')$ consistent with the same r, since it is possible to extend the attack to this case as well. With this in mind, we design the protocol as follows.

1. The second message μ_2 is a pseudorandom function output $\mathsf{PRF}(r, \mu_1)$, for a PRF key that is hardwired into P2 and Dec and not known to the parties.[9] The PRF inputs are the receiver randomness r and the first message μ_1.
2. $\mathsf{Dec}(r, \mu_1, \mu_2, \mu_3)$ does a validity check before decryption: if μ_2 is the correct PRF output for input (r, μ_1), Dec outputs m; otherwise, it outputs $\perp$.

Thus, the only way for the receiver to decrypt is to "prove" to Dec that it is running Dec on a valid r (consistent with μ_2). This ensures that it is hard to transform a transcript (μ_1, μ_2, μ_3) into a different (μ_1', μ_2', μ_3') consistent with the same receiver randomness r, since that would require finding μ_1', μ_2' such that $\mu_2' = \mathsf{PRF}(r, \mu_1')$, for an unknown r and an unknown PRF key.

We conclude this protocol design with a couple of final notes. First, we instantiate our "commitment" using a PRF as well, with its key hardwired into programs $\mathsf{P1}, \mathsf{P3}$ and not known to parties (thus, both μ_1 and μ_2 are PRF outputs). Secondly, we augment each program P1, P2, P3, Dec with a "trapdoor step" which makes each of these programs separately deniable, in the spirit of the [SW14] technique. Finally, we make the validity check inside Dec accept if $\mathsf{P2}(r, \mu_1) = \mu_2$, rather than if $\mathsf{PRF}(r, \mu_1) = \mu_2$; the difference is that P2 also accepts "fake" values which are not real preimages of the PRF. We make a similar modification to P3: its validity check verifies that $\mathsf{P1}(s, m) = \mu_1$ and therefore would also accept fake s which is not a real opening of the "commitment". These changes are necessary because without them, an adversary could use the validity check to test whether a given s is a real (PRF) preimage of μ_1 or a fake one.

We present the programs $\mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}$ as described so far, in fig. 3. For readability, the program includes comments to explain what the code is doing. Despite the somewhat dense code, the programs are very structured, and in a nutshell they behave as follows.

- Each program has a main step which is triggered when the program is run on uniformly random s or r (which is the case during an honest execution);
- Programs $\mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}$ each have a trapdoor step which is triggered when the programs are given fake randomness (which has a special format recognizable to the programs). The set of fake randomness is sufficiently sparse that the trapdoor steps are almost never triggered on uniform s or r. Fake randomness contains an "instruction" of how the program should behave.
- Programs P3 and Dec run validity checks, as described and motivated above.
- Programs SFake and RFake generate fake randomness which can be recognized by other programs.

In particular, during an honest execution with uniformly random s and r and plaintext m, the parties exchange messages μ_1, μ_2, μ_3 (computed by programs $\mathsf{P1}, \mathsf{P2}, \mathsf{P3}$, respectively), as follows: $\mu_1 = \mathsf{PRF}(s, m)$, $\mu_2 = \mathsf{PRF}(r, \mu_1)$, $\mu_3 = \mathsf{Enc}_K(m, \mu_1, \mu_2)$.[10] The receiver decrypts (μ_1, μ_2, μ_3) by running $\mathsf{Dec}(r, \mu_1, \mu_2, \mu_3)$, which verifies that $\mathsf{PRF}(r, \mu_1) = \mu_2$, then decrypts μ_3 and outputs m.

If the parties want to claim to a coercing adversary that they transmitted a different plaintext $\hat{m}$, they can use $\mathsf{SFake}, \mathsf{RFake}$ to compute fake s' and r', which are random-looking strings with $\hat{m}$, μ_1, μ_2, and μ_3 encrypted inside. If the adversary decrypts the transcript (μ_1, μ_2, μ_3) with fake $r' = \mathsf{Enc}_{K_R}(\hat{m}, \mu_1, \mu_2, \mu_3, \rho_R)$, it will get $\hat{m}$ as a result (via the trapdoor step of the decryption program). Similarly, the other programs, when given

[9] In this high-level description we omit PRF keys to simplify notation.

[10] Note that s, m (and r, μ_1) are both *inputs* to the PRF, not keys; we omit PRF keys for simplicity of notation.

[12] We treat s, r as non-random inputs, even though they are supposed to be uniformly chosen, since they are reused across different programs.

Base Protocol programs: first attempt at deniable encryption.

Program $\mathsf{P1}(s, m)$

1. **Trapdoor step:** if $\mathsf{Dec}_{K_S}(s) = (m', {\mu_1}', {\mu_2}', {\mu_3}')$ and $m' = m$, then return ${\mu_1}'$ //if s is fake and encodes m, output encoded ${\mu_1}'$
2. **Main step:** Return $\mu_1 \leftarrow \mathsf{PRF}(s, m)$. //otherwise output $\mathsf{PRF}(s, m)$

Program $\mathsf{P2}(r, \mu_1)$

1. **Trapdoor step:** if $\mathsf{Dec}_{K_R}(r) = (m', {\mu_1}', {\mu_2}', {\mu_3}')$ and ${\mu_1}' = \mu_1$, then return ${\mu_2}'$. //if r is fake and encodes μ_1, output encoded ${\mu_2}'$
2. **Normal step:** else return $\mathsf{PRF}(r, \mu_1)$. //otherwise output $\mathsf{PRF}(r, \mu_1)$

Program $\mathsf{P3}(s, m, \mu_1, \mu_2)$

1. **Validity check:** if $\mathsf{P1}(s, m) \neq \mu_1$, then abort;
2. **Trapdoor step:** if $\mathsf{Dec}_{K_S}(s) = (m', {\mu_1}', {\mu_2}', {\mu_3}')$ and $(m', {\mu_1}', {\mu_2}') = (m, \mu_1, \mu_2)$, then return ${\mu_3}'$. //if s is fake and encodes correct (m, μ_1, μ_2), output encoded ${\mu_3}'$
3. **Normal step:** else return $\mathsf{Enc}_K(m, \mu_1, \mu_2)$.//otherwise encrypt m

Program $\mathsf{Dec}(r, \mu_1, \mu_2, \mu_3)$

1. **Validity check:** if $\mathsf{P2}(r, \mu_1) \neq \mu_2$, then abort;
2. **Trapdoor step:** if $\mathsf{Dec}_{K_R}(r) = (m', {\mu_1}', {\mu_2}', {\mu_3}')$ and $({\mu_1}', {\mu_2}', {\mu_3}') = (\mu_1, \mu_2, \mu_3)$, then return m'. //if r is fake and encodes correct (μ_1, μ_2, μ_3), output encoded m'
3. **Normal step:** else decrypt $(m'', {\mu_1}'', {\mu_2}'') \leftarrow \mathsf{Dec}_K(\mu_3)$. If $({\mu_1}'', {\mu_2}'' = \mu_1, \mu_2)$ then output m'', else abort. //otherwise decrypt honestly

Program $\mathsf{SFake}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3; \rho_S)$

1. **Validity check:** if $\mathsf{P1}(s, m) \neq \mu_1$, then abort;
2. **Normal step:** else return $\mathsf{Enc}_{K_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \rho_S)$ // output fake s with fake plaintext and the transcript inside.

Program $\mathsf{RFake}(\hat{m}, \mu_1, \mu_2, \mu_3; \rho_R)$

1. **Normal step:** return $\mathsf{Enc}_{K_R}(\hat{m}, \mu_1, \mu_2, \mu_3, \rho_R)$ // output fake r with fake plaintext and the transcript inside

Figure 3: Base Protocol programs: first attempt at deniable encryption. $\mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}$ are deterministic;[12] SFake, RFake are randomized.

fake s' or r' as input, employ their trapdoor steps as well, making each protocol message appear consistent with $\hat{m}$.

The problem with the Base Protocol. We designed our scheme above with specific attacks in mind, but is it secure against all attacks? The answer is "almost": it is relatively easy to show security of the scheme in an idealized model where parties (and the adversary) have only oracle access to the programs, *but only as long as the adversary cannot query the* SFake *oracle.* Concretely, the adversary can use SFake to mount a certain attack Ω on the scheme, but this turns out to be the *only* possible type of attack. In section 2.2, we describe a special protection mechanism — comparison-based decryption behavior — which, when added to the protocol, prevents this type of attack and yields a scheme that is fully deniable even if the adversary has an access to *all* oracles including SFake. (And in the full version [**?**], we prove this result even when the adversary can see *the code* of all programs, obfuscated under IO).

Let's unpack why the protocol described so far is insecure. Recall that we wanted μ_1 to serve as a "commitment", and we wanted $\mathsf{P3}$ to output μ_3 only if the sender uses the same s and m in the commitment and as input to P3. This was important to make sure that for any μ_1, μ_2, at most one consistent μ_3 is efficiently computable. Then, however, we said that P3 should perform its validity check with respect to the whole program P1 and not just the commitment; in particular, the validity check in P3 accepts not only the true opening of the commitment, but also fake s. The problem is that P1, due to its trapdoor step, is not binding: given any ${\mu_1}^* = \mathsf{PRF}(s^*, m_0)$ and $m_1 \neq m_0$, it is easy to generate a different s_1 that passes the verification check. In fact, SFake does exactly that: given $(s^*, m_0, m_1, {\mu_1}^*, \mu_2, \mu_3)$ for some μ_2, μ_3, it outputs s_1 such that $\mathsf{P1}(s_1, m_1) = {\mu_1}^*$.

While this is not yet a concrete attack, it exposes a problem with our initial hope of a committing first message: sender deniability guarantees the first message is easily invertible, potentially with respect to inconsistent plaintexts m, so μ_1 cannot be committing. Thus, it is easy to create many fake s_i consistent with μ_1, and therefore many third messages ${\mu_3}^{(1)}, {\mu_3}^{(2)}, \ldots$, all consistent with a given $({\mu_1}^*, {\mu_2}^*)$. A procedure Ω that does this is detailed in fig. 4. For our purposes, the key features of the attack Ω are as follows.

- To generate such a ${\mu_3}^{(i)}$, encrypting some m_1 for a given $({\mu_1}^*, {\mu_2}^*)$, one has to run P3 on certain fake sender randomness s_i.
- P3 can recognize when it is being used to generate such a ${\mu_3}^{(i)}$. (This is because P3 will be run on a "mixed input": that is, P3 should be run on $s, m, {\mu_1}^*, {\mu_2}^*$, and a fake s_i that encodes the same ${\mu_1}^*$ but different $\tilde{\mu_2} \neq {\mu_2}^*$.)
- The only way to generate such fake s_i efficiently is to run SFake (on a transcript *different* from the one being attacked: specifically, with a different second message).

A procedure Ω to generate a new third message encrypting m_1
and consistent with given first and second messages μ_1, μ_2.

Inputs to $\Omega({\mu_1}^*, {\mu_2}^*, {\mu_3}^*, s^*, m^*, m_1)$ are: transcript $({\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, sender randomness s^* (which could be real or fake), plaintext m^*, and new desired plaintext m_1:

1. Compute an auxiliary transcript $\widetilde{\mathsf{tr}} = ({\mu_1}^*, \tilde{\mu_2}, \tilde{\mu_3})$ with the same first message ${\mu_1}^*$, but different second message $\tilde{\mu_2}$, by choosing fresh receiver randomness $\tilde{r}$ and setting $\widetilde{\mathsf{tr}} \leftarrow \mathsf{tr}(s^*, \tilde{r}, m^*)$. Note that the first message of this transcript is $\mathsf{P1}(s^*, m^*) = {\mu_1}^*$.
2. Compute $s_1 \leftarrow \mathsf{SFake}(s^*, m^*, m_1, {\mu_1}^*, \tilde{\mu_2}, \tilde{\mu_3})$. Note that s_1 is fake randomness which remembers $m_1, {\mu_1}^*$ and a new $\tilde{\mu_2} \neq {\mu_2}^*$.
3. Compute ${\mu_3}^{(1)} \leftarrow \mathsf{P3}(s_1, m_1, {\mu_1}^*, {\mu_2}^*)$.

Ω can now be repeated on input ${\mu_1}^*, {\mu_2}^*, {\mu_3}^{(1)}, s_1, m_1, m_2$ to generate ${\mu_3}^{(2)}$, and so on.

Figure 4: Procedure Ω to compute many third messages consistent with given μ_1, μ_2.

Since it is easy to generate many third messages, our scheme is subject to the same attack as all 2-message schemes: namely, the adversary can generate many ciphertexts ${\mu_3}^{(i)}$, fake each of them to compute an N-times-faked $r^{(N)}$, and then use it to correctly decrypt the original ${\mu_3}^*$. While this attack inheres in all 2-message schemes [BNNO11], in the 3-message case we can fix it. We do so by introducing *levels* and *comparison-based decryption behavior*, which specify how the decryption program should behave when the adversary tries to use such $r^{(N)}$ to decrypt a transcript $({\mu_1}^*, {\mu_2}^*, {\mu_3}^{(i)})$ or a challenge transcript $({\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$.

2.2 Levels, Comparison-Based Decryption, and the Final Scheme

Comparison-based decryption behavior. Let $(\mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$ be a challenge transcript. For any superpolynomial T and $j \in \{0, \ldots, T\}$, let r_j be the output of RFake on transcript $(\mu_1{}^*, \mu_2{}^*, \mu_3{}^{(j)})$, where $\mu_3{}^{(j)}$ is computed by j iterations of Ω. Let $\mu_3{}^{(0)}$ denote the challenge $\mu_3{}^*$. When Dec is run on r_j and $\mu_3{}^{(i)}$, for $i, j \in \{0, \ldots, T\}$, *comparison-based decryption behavior* requires the following.

1. *Oblivious past:* When $j > i$, Dec outputs $\perp$.
2. *Correctness forward:* When $j < i$, Dec decrypts $\mu_3{}^{(i)}$ correctly (as long as consistency checks pass).
3. When $j = i$, Dec should decrypt $\mu_3{}^{(i)}$ according to the instruction in fake r_j.

That is, if an adversary creates fake r_j using $\mu_3{}^{(j)}$, the jth in the sequence of ciphertexts, this r_j can be used to honestly decrypt ciphertexts "after" $\mu_3{}^{(j)}$, but cannot be used to decrypt ciphertexts "before" $\mu_3{}^{(j)}$; and naturally, r_j decrypts $\mu_3{}^{(j)}$ itself according to the instruction inside fake r_j.

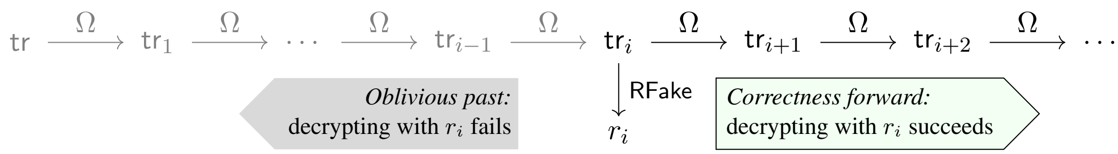

Figure 5: Comparison-based decryption behavior

Comparison-based decryption behavior prevents the attack described above, despite the fact that Ω enables the adversary to generate many third messages. Next, we give some intuition as to why. Recall that the attack had the adversary generate a fake r_j (by faking a ciphertext sequence $\mu_3{}^{(1)}, \mu_3{}^{(2)}, \ldots$) and then return to the challenge $\mu_3{}^*$ and decrypt it. Thus, a natural idea to mitigate this attack is to make Dec output $\perp$ whenever fake r_j is used to try to decrypt the initial $\mu_3{}^* = \mu_3{}^{(0)}$.[13] This simple modification indeed stops the attack, but introducing it alone would break security. To maintain security, we need to make sure that Dec on inputs r_j, $\mu_3{}^{(i)}$ should output $\perp$ for *all* $j > i$, and not just $j > i = 0$.[14] In other words, the "oblivious past" rule is the "minimum" modification which prevents fake r_j from decrypting $\mu_3{}^* = \mu_3{}^{(0)}$ *and* maintains security of the scheme.

Finally, the "correctness forward" rule *must* hold as well, since it is implied by sender-deniability. As a result, the behavior of the decryption program depends on the comparison of "indices" of the transcript and the receiver randomness; therefore, we call this comparison-based decryption behavior.

Implementing comparison-based decryption behavior: levels.

Next, we consider how to construct our programs such that comparison-based decryption behavior holds. When we run Dec on some μ_3 and some r, how does it know whether μ_3 is "forward" of r in the chain (meaning Dec should decrypt honestly), or "in the past" with respect to r (meaning Dec should output $\perp$)?

[13] Such a restriction is not possible in the 2-message case, in contrast to the 3-message case. This relates to the fact that our procedure Ω which generates $\mu_3{}^{(i)}$ is "one way", i.e., it is easy to generate $\mu_3{}^{(i+1)}$ from $\mu_3{}^{(i)}$, but it could be hard — and *is* hard, in our scheme — to generate $\mu_3{}^{(i)}$ from $\mu_3{}^{(i+1)}$. In contrast, in any 2-message scheme, there is no order on the ciphertexts; they are always easy to generate.

[14] To see this, suppose Dec outputs $\perp$ whenever $r_j, j > 0$ is used to decrypt $\mu_3{}^* = \mu_3{}^{(0)}$. Now consider trying to decrypt some $\mu_3{}^{(i)}$ with, say, r_{i+3}. r_3 does not decrypt $\mu_3{}^{(0)}$, and the difference between $(\mu_3{}^{(0)}, r_3)$ and $(\mu_3{}^{(i)}, r_{i+3})$ is that $\mu_3{}^{(0)}$ was generated with truly random s whereas $\mu_3{}^{(i)}$ used s_i which was faked i times. Sender deniability requires these two cases be indistinguishable, so $\mu_3{}^{(i)}$ must not be decrypted by r_{i+3}.

To this end, we introduce *levels*. That is, we have all fake sender randomness, all fake receiver randomness, and all third message ${\mu_3}^{(i)}$ also encrypt a number ℓ between 0 and some superpolynomial T, as follows.

- Fake sender randomness encrypts, among other things, a level ℓ which is *how many times this randomness was faked*. (E.g., to compute fake randomness, the sender would normally run SFake once, so the level ℓ of the resulting fake randomness is 1. If it runs SFake on the resulting randomness again, its level ℓ will be 2, and so on).
- Each potential third message ${\mu_3}^{(i)}$ also encrypts, in addition to m and μ_1, μ_2, its level, which is *its index i in the chain*. Note that the algorithm Ω which computes ${\mu_3}^{(i)}$ outputs ${\mu_3}^{(1)}$, ${\mu_3}^{(2)}, \ldots$ sequentially, and therefore their index i is well defined. In an honest execution, the level of μ_3 is always set to 0.
- Fake receiver randomness encrypts, among other things, a level ℓ which is *the level of its "parent" transcript* (i.e., the transcript which was used as input to RFake). (E.g., to compute fake randomness, the receiver would normally run RFake on the honest transcript, which has level 0, so the resulting fake randomness would have level 0 too).

We claim that storing this "level" information in fake randomness and third messages is enough for the scheme to maintain the level information accurately and follow comparison-based decryption behavior. For instance, Dec can decide its output behavior by comparing the levels inside r and μ_3. RFake can record the correct level of r by copying the level of its parent ciphertext. SFake can maintain the correct number of times something was faked, by reading the level in its input s and incrementing it. P3, as discussed above, can detect when it is being run within Ω, and it can put inside its output third message the level it copied from input s; since generating each new μ_3 requires once-more fake s, the level in s — i.e., the number of times it was faked — corresponds to the index of μ_3 in the chain.

Our final protocol in the oracle-access model. We present our final protocol (albeit still in the oracle model) in figs. 6–7. This scheme is a provably secure deniable encryption scheme in the oracle access model, as we show in section 4.

The structure of the final protocol programs is summarized below.

- Each program has a main step which is triggered when the program is run on uniformly random s or r, which is the case during an honest execution.
- Programs P1, P2, P3, and Dec also have a trapdoor step which is triggered when the programs receive fake randomness (which has a special format recognizable to the programs). The set of fake randomness is sufficiently sparse that the trapdoor step is almost never triggered on uniformly chosen s or r. Fake randomness contains an "instruction" of how the program should behave on some particular input.
- Programs P3 and Dec also have a "mixed input" step which serves to prevent attacks using Ω to generate many third messages μ_3. P3's mixed input step copies the level from its input s into the third message μ_3, ensuring μ_3 encrypts its own index in the sequence. Dec's mixed input step implements comparison-based decryption behavior by comparing the levels of μ_3 and r.

  The mixed input steps are triggered when the programs receive fake s (or r) as input, but the program's other inputs do not match the inputs in the instruction inside s (or r). P3 enters its mixed input step when its input and fake s contain the same μ_1 but different second messages, and Dec enters its mixed input step when its input and fake r contain the same μ_1, μ_2 but different third messages.
- Programs P3 and Dec's output behavior depends on validity checks, as in the Base Protocol (and for

the same reasons as in the Base Protocol).

- Programs SFake and RFake generate fake randomness that is recognizable to the other programs, and maintain accurate level information inside the fake randomness as follows: SFake increments the level of sender randomness with respect to its input sender randomness (unless the latter is honest, in which case SFake sets the level to 0); and RFake copies the level from the parent transcript into fake randomness.

Programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$.

Program $\mathsf{P1}(s, m)$

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{Dec}_{K_S}(s)$; if $\mathsf{out} = \text{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m = m'$ then return ${\mu_1}'$; //if s is fake and encodes m, output encoded ${\mu_1}'$
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{PRF}_{k_S}(s, m)$. //otherwise output $\mathsf{PRF}(s, m)$

Program $\mathsf{P3}(s, m, \mu_1, \mu_2)$

1. **Validity check:** if $\mathsf{P1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{Dec}_{K_S}(s)$; if $\mathsf{out} = \text{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', {\mu_1}', {\mu_2}'$ then return ${\mu_3}'$; //if s is fake and encodes correct (m, μ_1, μ_2), output encoded ${\mu_3}'$
3. **Mixed input step:** If $m, \mu_1 = m', {\mu_1}'$ but $\mu_2 \neq {\mu_2}'$ then return $\mu_3 \leftarrow \mathsf{Enc}_K(m, \mu_1, \mu_2, \ell')$; //if s is fake and encodes correct (m, μ_1) but incorrect ${\mu_2}'$, encrypt m with level copied from s
4. **Main step:**
   (a) Return $\mu_3 \leftarrow \mathsf{Enc}_K(m, \mu_1, \mu_2, 0)$. //otherwise encrypt m with level 0

Program $\mathsf{SFake}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

1. **Validity check:** if $\mathsf{P1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{Dec}_{K_S}(s)$; if $\mathsf{out} = \text{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1 = m', {\mu_1}'$ then
      i. If $\ell \geq T$ then abort;
      ii. Return $\mathsf{Enc}_{K_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell + 1)$. //if input s is already fake then output new fake s with fake plaintext, the transcript, and incremented level
3. **Main step:**
   (a) Return $\mathsf{Enc}_{K_S}(\hat{m}, \mu_1, \mu_2, \mu_3, 1)$. //otherwise output fake s with fake plaintext, the transcript, and level 1

Figure 6: Programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$.

The interesting cases of protocol execution are summarized next.

- **Normal protocol execution.** Executing the programs on randomly chosen s^*, r^* and plaintext m_0^* triggers the main step, yielding outputs ${\mu_1}^* = \mathsf{PRF}(s^*, m_0^*)$, ${\mu_2}^* = \mathsf{PRF}(r^*, {\mu_1}^*)$, and ${\mu_3}^* = \mathsf{Enc}_K(m_0^*, {\mu_1}^*, {\mu_2}^*, 0)$, where the last 0 is the level. Dec, given the resulting transcript as input, outputs m_0^* via its main step.

- **Fake randomness of parties.** A sender wishing to claim that it sent plaintext $m_1^* \neq m_0^*$ can run SFake to obtain fake s' encoding $(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, 1)$, where the last 1 is the level. A receiver wishing

Programs $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$**.**

Program $\mathsf{P2}(r, \mu_1)$

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{Dec}_{K_R}(r)$; if $\mathsf{out} = '\mathsf{fail}'$ then goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', L', \hat{\rho})$;
   (b) If $\mu_1 = {\mu_1}'$ then return ${\mu_2}'$; //if r is fake and encodes μ_1, output encoded ${\mu_2}'$
2. **Main step:**
   (a) Return $\mu_2 \leftarrow \mathsf{PRF}_{k_R}(r, \mu_1)$. //otherwise output $\mathsf{PRF}(r, \mu_1)$

Program $\mathsf{Dec}(r, \mu_1, \mu_2, \mu_3)$

1. **Validity check:** if $\mathsf{P2}(r, \mu_1) \neq \mu_2$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{Dec}_{K_R}(r)$; if $\mathsf{out}' = '\mathsf{fail}'$ then goto main step; else parse out' as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell', \hat{\rho})$;
   (b) if $\mu_1, \mu_2, \mu_3 = {\mu_1}', {\mu_2}', {\mu_3}'$ then return m'; //if r is fake and encodes correct (μ_1, μ_2, μ_3), output encoded m'
   (c) $\mathsf{out} \leftarrow \mathsf{Dec}_K(\mu_3)$; if $\mathsf{out}'' = '\mathsf{fail}'$ then abort, else parse out'' as $(m'', {\mu_1}'', {\mu_2}'', \ell'')$;
3. **Mixed input step:** If $\mu_1, \mu_2 = {\mu_1}', {\mu_2}'$ but $\mu_3 \neq {\mu_3}'$ then
   (a) If $({\mu_1}', {\mu_2}') = ({\mu_1}'', {\mu_2}'')$ and $\ell' < \ell''$ then return m''; //if r is fake and encodes correct (μ_1, μ_2) but incorrect ${\mu_3}'$, decrypt honestly or abort, depending on whether the level in r is smaller than in μ_3 or not
   (b) Else abort.
4. **Main step:**
   (a) $\mathsf{out} \leftarrow \mathsf{Dec}_K(\mu_3)$; if $\mathsf{out} = '\mathsf{fail}'$ then abort, else parse out as $(m'', {\mu_1}'', {\mu_2}'', \ell'')$;
   (b) If $(\mu_1, \mu_2) = ({\mu_1}'', {\mu_2}'')$ then return m''; //otherwise decrypt honestly
   (c) Else abort.

Program $\mathsf{RFake}(\hat{m}, \mu_1, \mu_2, \mu_3; \rho)$

1. $\mathsf{out} \leftarrow \mathsf{Dec}_K(\mu_3)$; if $\mathsf{out} = '\mathsf{fail}'$ then abort, else parse out as $(m'', {\mu_1}'', {\mu_2}'', \ell'')$;
2. Return $r' \leftarrow \mathsf{Enc}_{K_R}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell'', \mathsf{prg}(\rho))$. // output fake r with fake plaintext, the transcript, and the level copied from μ_3

Figure 7: Programs $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$.

to claim that it received $m_1^* \neq m_0^*$ can run RFake to obtain fake r' encoding $(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, 0)$, where the last 0 is the level. Executing programs on fake s' or fake r' and m_1^* triggers the trapdoor step, so programs will output the values hardwired into the fake s' or r'. Thus, $\mathsf{P1}$ will output ${\mu_1}^*$, $\mathsf{P2}$ will output ${\mu_2}^*$, $\mathsf{P3}$ will output ${\mu_3}^*$, and Dec will output m_1^* via their trapdoor steps, making the transcript, originally for plaintext m_0^*, appear consistent with m_1^*.

- **Efficiently computable related transcripts.** It is only possible to compute related transcripts of the form $({\mu_1}^*, {\mu_2}^*, \mu_3)$, where $\mu_3 = \mathsf{Enc}_K(m, {\mu_1}^*, {\mu_2}^*, \ell)$, $\ell \geq 1$; moreover, the only way of doing so is to use the procedure Ω described above (which invokes SFake). Trying to compute μ_3 a for such transcript will cause program $\mathsf{P3}$ to execute its "mixed input step", ensuring that such μ_3 receives level $\ell \geq 1$; for this, it is important that SFake increments the level inside s. Trying to decrypt such a related transcript $({\mu_1}^*, {\mu_2}^*, \mu_3)$ will cause program Dec to execute its "mixed input step", ensuring that the requisite decryption behavior is observed (that fake r decrypts correctly transcripts with larger level, but fails to decrypt transcripts with smaller level); for this, it is important that RFake copies the level from the transcript to r.

Outline of security proof in oracle-access model. Since the proof even in this simpler (oracle-access) model is somewhat lengthy, we only outline the main steps here, with intuition for each. The proof proceeds in four main hybrid steps. We start with a real execution corresponding to plaintext m_0^*, where the adversary receives real randomness s^*, r^*.

- **Step I: indistinguishability of sender explanations.** Instead of giving the adversary real s^*, we give it $s' = \mathsf{Enc}_{K_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell = 0)$ (note that this s' contains level 0, unlike fake randomness produced by SFake which contains level at least 1).

  Intuitively, the reason why we can switch from s^* to s' indistinguishably is because all programs treat s^* and s' indistinguishably. That is:

  – either the programs output the same value, possibly via different branches of execution (e.g., $\mathsf{P1}$ on input (s^*, m_0^*) outputs $\mu_1{}^*$ via its main step and on input (s', m_0^*) outputs $\mu_1{}^*$ via its trapdoor step);

  – or the programs execute the same code, possibly outputting different results (e.g., $\mathsf{P1}$, on input (s^*, m_1^*) or (s', m_1^*), evaluates a PRF on its input and outputs the result).

  The above, and the fact that s' is pseudorandom, allow us to change s^* to s' (similarly to the [SW14] proof for sender-deniable encryption).

- **Step II: indistinguishability of receiver explanations.** Instead of giving the adversary real r^*, we give it fake r', i.e., $r' = \mathsf{Enc}_{K_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell = 0, \rho_R)$. Unlike in Step I, here there is a transcript with respect to which the decryption program treats r^* and r' distinguishably.

  Recall that r^* honestly decrypts *all* related transcripts, while r' only honestly decrypts "forward", i.e., for related transcripts with level $\ell \geq 1$. Thus, the two programs may treat level-0 transcripts differently. Consider a transcript $(\mu_1{}^*, \mu_2{}^*, \overline{\mu_3{}^*})$, where $\overline{\mu_3{}^*} = \mathsf{Enc}_K(m_1^*, \mu_1{}^*, \mu_2{}^*, \ell = 0)$ is like $\mu_3{}^*$ except that it encrypts the wrong plaintext m_1^*. This transcript decrypts correctly to m_1^* with r^*, but decrypting it with r' returns $\perp$ due to the level comparison logic.

  This single transcript makes r^* and r' distinguishable. As a result, the proof of Step I does not work here. Therefore, we first move to a hybrid where this "differing" transcript doesn't exist, as follows. First, since s^* (the preimage of PRF output $\mu_1{}^*$) is not part of the distribution anymore, we can move $\mu_1{}^*$ outside the PRF image. Then we argue that $\mathsf{P3}$ never outputs $\overline{\mu_3{}^*}$:

  – *The main step cannot output* $\overline{\mu_3{}^*}$, since it is executed only if the validity check passes *via a correct PRF preimage*, which now does not exist.

  – *The mixed step cannot output* $\overline{\mu_3{}^*}$, since $\mathsf{P3}$ can only output a ciphertext with level 0 (like $\overline{\mu_3{}^*}$) via the mixed step if its input randomness has level 0, and such input randomness is hard to find since SFake never outputs randomness with level 0.

  – *The trapdoor step cannot output* $\overline{\mu_3{}^*}$, since $\mathsf{P3}$ can only output $\overline{\mu_3{}^*}$ via the trapdoor step if it receives as input fake randomness that has $\overline{\mu_3{}^*}$ inside to begin with. Since there are no other means of computing $\overline{\mu_3{}^*}$, such randomness is also hard to find.

  Once the differing transcript $(\mu_1{}^*, \mu_2{}^*, \overline{\mu_3{}^*})$ is eliminated, we can switch r^* to r' similarly to Step I.

- **Step III: indistinguishability of plaintexts.** The next step is to switch $\mu_3{}^*$ from encrypting m_0^* to encrypting m_1^*. This is done by "detaching" $\mu_3{}^*$ from its key K in programs $\mathsf{P3}$ and Dec. Concretely:

- P3 can only output $\mu_3{}^*$ via the trapdoor thread (which does not use the key K). The reason is very similar to the case-by-case analysis of P3 in Step II: the main step requires a PRF preimage, which does not exist, and the mixed step requires level-0 sender randomness, which is hard to find.
- Dec can only "decrypt" $\mu_3{}^*$ via the trapdoor thread (which, again, does not use K). To guarantee this, we first move $\mu_2{}^*$ outside of the PRF image (this is possible since r^* is no longer part of the distribution). Then $\mu_3{}^*$ is never decrypted via the main step because the preimage for $\mu_2{}^*$ does not exist. Further, $\mu_3{}^*$ cannot be decrypted via the mixed step either, because the "correctness forward" decryption rule outputs $\perp$ unless the input receiver randomness has level smaller than the level in $\mu_3{}^*$, and this is not possible since $\mu_3{}^*$ has the smallest possible level, 0.

In other words, neither P3 nor Dec need to use K to encrypt or decrypt $\mu_3{}^*$. Therefore we can "detach" K and $\mu_3{}^*$ and change the plaintext to m_1^*.

Note that the transcript now contains m_1^*, and both sender and receiver randomness s', r' are consistent with m_0^*. However, the proof is not finished yet, since parties cannot produce such s' themselves (since s' has level 0).

- **Step IV: indistinguishability of levels.** The last step is to change the level inside s' from 0 to 1, i.e., let $s' = \mathsf{Enc}_{K_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell = 1)$. To understand the challenge of this step, it is instructive to take a "level-centric" perspective: let's put aside that the scheme is about transmitting plaintexts, and instead think of fake s as an encryption of level (0 or 1), think of $\mu_3{}^*$ as an encryption of level 0, and think of the programs of deniable encryption as implementing homomorphic operations on encrypted levels. For example, program SFake outputs fake randomness which is an encryption of incremented level, and thus implements a homomorphic Increment operation on levels. Program Dec compares levels inside μ_3 and r and, based on that, decrypts or outputs $\perp$, and thus it implements a homomorphic isLess function on levels, which reveals (in the clear) if one level is smaller than the other.

  From this perspective, step IV essentially requires switching s' from an encryption of 0 to an encryption of 1, while the adversary has access to homomorphic functions Increment and isLess.[15] In the oracle-access model, it can be easily shown that polynomially bounded adversaries cannot distinguish between $\mathsf{Enc}(0)$ and $\mathsf{Enc}(1)$, even given oracle access to isLess and Increment, as long as the largest allowed level T is superpolynomial: this is because the adversary can only generate polynomial-length sequences of encryptions — $\mathsf{Enc}(1), \mathsf{Enc}(2), \ldots$ or $\mathsf{Enc}(2), \mathsf{Enc}(3), \ldots$ (depending on whether the challenge ciphertext was $\mathsf{Enc}(0)$ or $\mathsf{Enc}(1)$) — and the oracles' behavior will be identical on both sequences.

This concludes the proof outline in the oracle-access model. We underline that in the actual construction we need special types of PRFs, encryption schemes, and a special *level system* primitive in order to prove security with iO. The proofs of steps I-III in the final construction roughly follow the same outline (sometimes with several hybrids per each logical step), but the proof of the step IV (indistinguishability of levels) requires substantial additional work when the adversary possesses the code of the programs. We outline the intuition and the main steps of the proof for this step in section 7.3.

[15] Recall that the adversary also has $\mu_3{}^*$ which is an encryption of level 0. For simplicity, we ignore this fact in this high-level overview.

3 Defining Bideniable and Off-the-Record Deniable Encryption

We present the definition of interactive deniable encryption, or, more formally, interactive deniable message transmission. In Section 3.1 we present the definition in the CRS model; this definition corresponds to our main construction. In Section 3.2 we present the definition for the idealized, oracle access model.

3.1 Deniability in the CRS Model

Syntax. An interactive deniable encryption scheme π consists of seven algorithms $\pi = (\mathsf{Setup}, \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake})$, where Setup is used to generate the public programs (i.e., the CRS), programs $\mathsf{P1}$, $\mathsf{P3}$ and SFake are used by the sender, and programs $\mathsf{P2}$, Dec and RFake are used by the receiver. Let $\mathsf{tr} = \pi(s, r, m)$ denote the transcript of a protocol execution on input plaintext m, sender randomness s, and receiver randomness r, i.e., the sequence of three messages sent in the protocol execution. That is, $\pi(s, r, m) = \mathsf{tr} = (\mu_1, \mu_2, \mu_3)$, where $\mu_1 = \mathsf{P1}(s, m)$, $\mu_2 = \mathsf{P2}(r, \mu_1)$, and $\mu_3 = \mathsf{P3}(s, m, \mu_1, \mu_2)$.

The faking algorithms have the following syntax: $\mathsf{SFake}(s, m, m', \mathsf{tr}; \rho)$ expects to take as input a transcript tr along with the true random coins s and true plaintext m which were used to compute tr, and a desired fake plaintext m'. SFake is randomized and ρ denotes its randomness. RFake has the same syntax except that it expects receiver randomness r instead of sender randomness s.

Bideniable and off-the-record-deniable encryption in the CRS model. Next, we define standard and off-the-record deniability for interactive deniable encryption in the CRS model. For simplicity, we focus on bit encryption. The definitions are naturally extensible to multi-bit plaintexts.

Formally, the deniable encryption algorithms should take the CRS as input. We omit this for notational simplicity as it is unnecessary in our construction (where the CRS contains the programs, and the programs do not take the CRS as input).

Definition 1. Bideniable bit encryption in the CRS model. $\pi = (\mathsf{Setup}, \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake})$ *is a 3-message bideniable interactive encryption scheme for message space* $\mathcal{M} = \{0, 1\}$, *if it satisfies the following correctness and bideniability properties.*

- **Correctness:** *There exists a negligible function* $\nu(\lambda)$ *such that for at least a* $(1 - \nu)$ *fraction of randomness* $r_{\mathsf{Setup}} \in \{0,1\}^{|r_{\mathsf{Setup}}|}$, *for any* $m \in \mathcal{M}$,

$$\Pr\left[\; m' \neq m \quad : \quad \begin{array}{l} \mathsf{CRS} \leftarrow \mathsf{Setup}(r_{\mathsf{Setup}}) \\ s \leftarrow \{0,1\}^{|s|} \\ r \leftarrow \{0,1\}^{|r|} \\ \mathsf{tr} \leftarrow \pi(s, r, m) \\ m' \leftarrow \mathsf{Dec}(r, \mathsf{tr}) \end{array} \;\right] \leq \nu(\lambda)\ .$$

- **Bideniability:** *No PPT adversary* Adv *has more than negligible advantage in the following game, for any* $m_0, m_1 \in \mathcal{M}$*:*

  1. *The challenger chooses random* r_{Setup} *and generates* $\mathsf{CRS} \leftarrow \mathsf{Setup}(r_{\mathsf{Setup}})$. *It also chooses a bit* b *at random.*

  2. *If* $b = 0$, *then the challenger behaves as follows:*

(a) It chooses random s^*, r^* *and computes* $\mathsf{tr}^* = \pi(s^*, r^*, m_0)$.

(b) It gives the adversary $(\mathsf{CRS}, m_0, m_1, s^*, r^*, \mathsf{tr}^*)$.

3. If $b = 1$*, then the challenger behaves as follows:*

(a) It chooses random s^*, r^* *and computes* $\mathsf{tr}^* \leftarrow \pi(s^*, r^*, m_1)$.

(b) It sets $s' \leftarrow \mathsf{SFake}(s^*, m_1, m_0, \mathsf{tr}^*; \rho_S)$ *for random* ρ_S.

(c) It sets $r' \leftarrow \mathsf{RFake}(r^*, m_1, m_0, \mathsf{tr}^*; \rho_R)$ *for random* ρ_R.

(d) It gives the adversary $(\mathsf{CRS}, m_0, m_1, s', r', \mathsf{tr}^*)$.

4. Adv *outputs* b' *and wins if* $b = b'$.

Next, we define off-the-record deniability. We define it for an arbitrary message space instead of bit encryption, since having $|\mathcal{M}| > 2$ allows for an extra case when plaintexts claimed by the sender, by the receiver, and the real plaintext are three different strings (case $b = 2$ in the definition below).

Definition 2. Off-the-record deniable encryption in the CRS model. *We say that a scheme is off-the-record deniable, if it satisfies correctness as above and also has the following property.*

Off-the-record deniability: No PPT adversary Adv wins with more than negligible advantage in the following game, for any $m_0, m_1, m_2 \in \mathcal{M}$:

1. The challenger chooses random r_{Setup} and generates $\mathsf{CRS} \leftarrow \mathsf{Setup}(r_{\mathsf{Setup}})$. It also chooses random $b \in \{0, 1, 2\}$.
2. If $b = 0$, then the challenger generates the following variables:
   (a) The challenger chooses random s^*, r^* and computes $\mathsf{tr}^* \leftarrow \pi(s^*, r^*, m_0)$;
   (b) It sets $r' \leftarrow \mathsf{RFake}(r^*, m_0, m_1, \mathsf{tr}^*; \rho_R)$ for randomly chosen ρ_R.
   (c) It gives the adversary $(\mathsf{CRS}, m_0, m_1, m_2, s^*, r', \mathsf{tr}^*)$.
3. If $b = 1$, then the challenger generates the following variables:
   (a) The challenger chooses random s^*, r^* and computes $\mathsf{tr}^* \leftarrow \pi(s^*, r^*, m_1)$;
   (b) It sets $s' \leftarrow \mathsf{SFake}(s^*, m_1, m_0, \mathsf{tr}^*; \rho_S)$ for randomly chosen ρ_S.
   (c) It gives the adversary $(\mathsf{CRS}, m_0, m_1, m_2, s', r^*, \mathsf{tr}^*)$.
4. If $b = 2$, then the challenger generates the following variables:
   (a) The challenger chooses random s^*, r^* and computes $\mathsf{tr}^* \leftarrow \pi(s^*, r^*, m_2)$;
   (b) It sets $s' \leftarrow \mathsf{SFake}(s^*, m_2, m_0, \mathsf{tr}^*; \rho_S)$ for randomly chosen ρ_S.
   (c) It sets $r' \leftarrow \mathsf{RFake}(r^*, m_2, m_1, \mathsf{tr}^*; \rho_R)$ for randomly chosen ρ_R.
   (d) It gives the adversary $(\mathsf{CRS}, m_0, m_1, m_2, s', r', \mathsf{tr}^*)$.
5. Adv outputs b' and wins if $b = b'$.

We say that an encryption scheme is bideniable (resp., off-the-record deniable) with (t, ε)-security, if the distinguishing advantage of any any size-t adversary in the bideniability (resp., off-the-record deniability) game is at most ε.

Single-execution security implies multi-execution security. In definitions 4 and 2, the CRS is global (i.e., non-programmable). These definitions do not involve simulation and the same set of programs is used throughout. Furthermore, even though definitions 4 and 2 consider a single protocol execution, a simple hybrid argument shows that security of a single execution implies security of arbitrarily polynomially many executions with the same set of programs.[16]

Definition 3. Public receiver deniability. *A deniable scheme has* public receiver-deniability *if the receiver faking algorithm* RFake *takes as input only the transcript* tr *and fake plaintext* m' *(not true random coins of the receiver* r^* *and true plaintext* m*).*

3.2 Deniability in The Oracle Access Model

In the oracle access model the algorithms $\mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}$ are replaced by oracles. That is, an interactive deniable encryption scheme π in the oracle access model consists of six oracles $\pi = (\mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake})$. As before, oracles $\mathsf{P1}$, $\mathsf{P3}$ and SFake are used by the sender, and oracles $\mathsf{P2}$, Dec and RFake are used by the receiver. As begfore, we let the transcript $\mathsf{tr} = \pi(s, r, m)$ of an execution of the scheme on inputs m and random input s of the sender, and random input r of the receiver denote the sequence of three messages sent in this execution. That is, $\pi(s, r, m) = \mathsf{tr} = (\mu_1, \mu_2, \mu_3)$, where $\mu_1 = \mathsf{P1}(s, m)$, $\mu_2 = \mathsf{P2}(r, \mu_1)$, and $\mu_3 = \mathsf{P3}(s, m, \mu_1, \mu_2)$.

The faking oracles have the following syntax: $\mathsf{SFake}(s, m, m', \mathsf{tr}; \rho)$ expects to take a transcript tr along with the true random coins s and true plaintext m, which were used to compute tr. It also needs the desired fake plaintext m', and its own randomness ρ. RFake follows the same syntax except that it expects the receiver randomness r instead of sender randomness s.

Deniable encryption in the oracle access model. For the oracle access model, we concentrate on plain bideniability. As before, the definitions can be naturally extended to multi-bit plaintexts.

Definition 4. Bideniable bit encryption in the Oracle Access Model. $\pi = (\mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake})$ *is a 3-message bideniable interactive encryption scheme for message space* $\mathcal{M} = \{0, 1\}$*, if it satisfies the following correctness and bideniability properties:*

- **Correctness:** *For any* $m \in \mathcal{M}$ $\Pr[m' \neq m : s \leftarrow \{0,1\}^{|s|}, r \leftarrow \{0,1\}^{|r|}, \mathsf{tr} \leftarrow \pi(s, r, m), m' \leftarrow \mathsf{Dec}(r, \mathsf{tr})] = 0$*. (Here the probability is taken over the initial random choices of the oracles and rhw choices of* s *and* r*.*
- **Bideniability:** *No PPT adversary* Adv *wins with more than negligible advantage in the following game, for any* $m_0, m_1 \in \mathcal{M}$*:*
  1. *The challenger samples the six oracles and gives* Adv *access to them. It also chooses a bit* b *at random.*
  2. *If* $b = 0$*, then the challenger behaves as follows:*
     (a) *It chooses random* s^*, r^* *and computes* $\mathsf{tr}^* = \pi(s^*, r^*, m_0)$*.*
     (b) *It gives the adversary* $(m_0, m_1, s^*, r^*, \mathsf{tr}^*)$*.*
  3. *If* $b = 1$*, then the challenger behaves as follows:*

[16] We can change all executions from real to fake one by one, where the reduction from a single-execution security will generate other executions on its own, since knowing the CRS (but not its generation randomness) suffices to run all programs.

(a) *It chooses random* s^*, r^* *and computes* $\mathsf{tr}^* \leftarrow \pi(s^*, r^*, m_1)$*;*

(b) $s' \leftarrow \mathsf{SFake}(s^*, m_1, m_0, \mathsf{tr}^*; \rho_S)$ *and* $r' \leftarrow \mathsf{RFake}(r^*, m_1, m_0, \mathsf{tr}^*; \rho_R)$*, for randomly chosen* ρ_S, ρ_R*.*

(c) *It gives the adversary* $(m_0, m_1, s', r', \mathsf{tr}^*)$.

4. Adv *outputs* b' *and wins if* $b = b'$.

4 Deniable Encryption in Oracle-Access model

In this section we construct and prove security of our deniable encryption scheme assuming that parties and adversaries only have oracle access to the programs of deniable encryption.

We stress that our main result — deniable encryption in the CRS model described in section 6 — does not use any results from this section and can be read independently. The goal of this section is to describe a simplified construction with a relatively short proof of security, to help the reader verify the result.

Our scheme is described on fig. 8, and it assumes that all parties — senders, receivers, and adversaries — have access to oracles described in fig. 9, fig. 10. These oracles compute messages of deniable encryption, as well as compute fake random coins for parties.

Notation and primitives.

Let s and r denote thte randomness of the sender and the receiver, respectively, and let μ_1, μ_2, μ_3 denote the three messages of the protocol. $\mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}$ are the oracles computing the corresponding messages of deniable encryption, performing decryption, and faking coins for the sender and the receiver, respectively. For instance, to compute the first message, the sender should query the oracle $\mathsf{P1}$ on input (s^*, m) for uniformly chosen s^*.

We now specify the syntax. $\mathsf{P1}(s, m)$ takes as input sender randomness s and plaintext m and outputs the first message μ_1. $\mathsf{P2}(r, \mu_1)$ takes as input receiver randomness r and first message μ_1 and outputs the second message μ_2. $\mathsf{P3}(s, m, \mu_1, \mu_2)$ takes as input sender randomness s, plaintext m, and protocol messages μ_1, μ_2 and outputs the last message μ_3. $\mathsf{Dec}(r, \mu_1, \mu_2, \mu_3)$ takes as input receiver randomness r and protocol messages μ_1, μ_2, μ_3 and outputs the plaintext m. $\mathsf{SFake}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$ takes as input sender randomness s, true plaintext m, new (fake) plaintext $\hat{m}$, and protocol messages μ_1, μ_2, μ_3 and outputs fake randomness s' which makes μ_1, μ_2, μ_3 look consistent with $\hat{m}$. $\mathsf{RFake}(\hat{m}, \mu_1, \mu_2, \mu_3)$ takes as input new (fake) plaintext $\hat{m}$ and protocol messages μ_1, μ_2, μ_3 and outputs fake randomness r' which makes μ_1, μ_2, μ_3 look consistent with $\hat{m}$.

Our oracles use hashes H_1, H_2, H_3 and encryption schemes with keys K_S, K_R, K. We underline that these primitives are "ideal": that is, the description of each hash H_1, H_2, H_3 is a table $\{(x_i, y_i)\}$ specifying the output y_i for each input x_i. The description of each key K_S, K_R, K is a table specifying the ciphertext c_i for each input x_i; all three encryption schemes are deterministic (that is, they only take the plaintext as input, and do not sample any additional random coins.). All these primitives are ideal in the sense that all images of all encryption schemes and hashes are chosen uniformly at random.

For convenience, we denote by $\mathcal{S}, \mathcal{R}, \mathcal{M}, \mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3$ the image of sender-fake encryption scheme (with key K_S), receiver-fake encryption scheme (with key K_R), the main encryption scheme (with key K), and hashes

H_1, H_2, H_3, respectively.

In our construction fake randomness is itself an encryption of several variables - e.g. fake sender randomness s encrypts $m, \mu_1, \mu_2, \mu_3, \ell$. Because of this, it is convenient to refer to a particular "field" of a decrypted value, which we will denote, following programming languages notation, by $\mathsf{Dec}_{K_S}(s).m$, $\mathsf{Dec}_{K_S}(s).\mu_1$, $\mathsf{Dec}_{K_S}(s).\mu_2$, $\mathsf{Dec}_{K_S}(s).\mu_3$, $\mathsf{Dec}_{K_S}(s).\ell$; similarly, we will be referring to different fields of μ_3 by using $\mathsf{Dec}_K(\mu_3).m$, $\mathsf{Dec}_K(\mu_3).\mu_1$, $\mathsf{Dec}_K(\mu_3).\mu_2$, $\mathsf{Dec}_K(\mu_3).\ell$.

The choice of parameters. We set T to be superpolynomial in the security parameter (e.g. $T = \lambda^{\log \lambda}$). We set the size of each ciphertext and hash image to be large enough so that the image of each encryption or hash is sparse. In particular, let us set $|\mu_1| = |\mu_2| = 2\lambda$, $|\mu_3| = 7\lambda$, $|s| = |r| = 16\lambda$, $|\ell| = \lambda$. Further, we make the images size of H_1 and H_2 to be 2^λ each. This means that H_1 is a function from $|s| + 1$ bits to 2λ bits (with image size 2^λ), H_2 is a function from $|\mu_1| + |r|$ bits to 2λ bits (with image size 2^λ), H_3 is a function from λ bits to 2λ bits. This choice of parameters ensures that each encryption and hash have sufficiently sparse images and therefore the probability of randomly chosen string to be in their image is negligible in λ[17]. Finally, note that the set $\mathcal{S}$ should be sparse enough so that $T * |\mathcal{S}|/2^{|s|}$ remains negligible (indeed, this is true for our choice of parameters: the size of $\mathcal{S}$ is $2^{|m|+|\mu_1|+|\mu_2|+|\mu_3|+|\ell|} = 2^{1+2\lambda+2\lambda+7\lambda+\lambda} < 2^{13\lambda}$, $T < 2^\lambda$, and $2^{|s|} = 2^{16\lambda}$).

Finally, we note that this choice of parameters also ensures correctness of encryption schemes with keys K, K_S, K_R: namely, it ensures that for any fixed $\mu_3 \in \mathcal{M}$, the probability over the choice of K that it has more than one preimage is negligible (indeed, this probability is bounded by $|\mathcal{M}|2^{-|\mu_3|} < 2^{-\lambda}$). The same holds for any fixed s over the choice of key K_S, and any fixed r over the choice of key K_R.

4.1 Construction

The protocol is described in fig. 8. It simply instructs parties to run the programs $\mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}$ to encrypt and decrypt, and SFake, RFake to fake (described in fig. 9 and 10). Note that deniability of the receiver is *public*, since the knowledge of randomness of the receiver is not required in order to run RFake.

We assume that a program outputs $\perp$ if any of its underlying primitives outputs $\perp$, except where it is explicitly written otherwise. For instance, if a program tries to decrypt a ciphertext which is not in the image of the corresponding encryption scheme, this program outputs $\perp$.

4.2 Proof of correctness and security.

In short, correctness of the scheme follows from correctness of underlying ideal encryption and the fact that the sets $\mathcal{S}$ and $\mathcal{R}$ of fake randomness are sparse (the latter is important because oracles do not perform correct encryption / decryption operations when run on randomness from $\mathcal{S}, \mathcal{R}$, which parties may accidentally pick as their random coins).

[17]The exact choice of parameters comes from the following: the purpose of setting $|\mu_1| = |\mu_2| = 2\lambda$ is to make sure that the images of hashes H_1, H_2 are sparse enough (each hash H_1, H_2 has 2^λ different images). By setting $|\mu_3| = 7\lambda$, we make sure the set of valid ciphertexts μ_3 under key K is also sparse (indeed, note that the size of the plaintext which is encrypted in μ_3 is $|m| + |\mu_1| + |\mu_2| + |\ell| = 1 + 2\lambda + 2\lambda + \lambda < 6\lambda$). Finally, by setting $|s| = |r| = 16\lambda$ we make sure that the set of valid ciphertexts under keys K_S, K_R is sparse as well: indeed, note that the size of plaintexts encrypted inside fake s, r is at most $|m| + |\mu_1| + |\mu_2| + |\mu_3| + |\ell| + |H_3(\rho)| < 1 + 2\lambda + 2\lambda + 7\lambda + \lambda + 2\lambda < 15\lambda$.

Programs: $\mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}$, described in fig. 9, fig. 10. These programs are only accessible via oracle access.

Our interactive deniable encryption:

Inputs: plaintext $m \in \{0,1\}$ of the sender.

1. **Message 1:** The sender chooses random s^*, computes ${\mu_1}^* \leftarrow \mathsf{P1}(s^*, m)$ and sends ${\mu_1}^*$ to the receiver.
2. **Message 2:** The receiver chooses random r^*, computes ${\mu_2}^* \leftarrow \mathsf{P2}(r^*, {\mu_1}^*)$ and sends ${\mu_2}^*$ to the sender.
3. **Message 3:** The sender computes ${\mu_3}^* \leftarrow \mathsf{P3}(s^*, m, {\mu_1}^*, {\mu_2}^*)$ and sends ${\mu_3}^*$ to the receiver.
4. The receiver runs $m' \leftarrow \mathsf{Dec}(r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$.

Sender Coercion:

Inputs: real plaintext $m \in \{0,1\}$, fake plaintext $\hat{m} \in \{0,1\}$, real random coins s^* of the sender, and the protocol transcript ${\mu_1}^*, {\mu_2}^*, {\mu_3}^*$.

1. Upon coercion, the sender computes fake randomness $s' \leftarrow \mathsf{SFake}(s^*, m, \hat{m}, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$.

Receiver Coercion:

Inputs: fake plaintext $\hat{m} \in \{0,1\}$ and the protocol transcript ${\mu_1}^*, {\mu_2}^*, {\mu_3}^*$.

1. Upon coercion, the receiver chooses random ρ^* and computes fake randomness $r' \leftarrow \mathsf{RFake}(\hat{m}, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*; \rho^*)$.

Figure 8: Our interactive deniable encryption scheme.

More concretely, recall that s^*, chosen uniformly at random, belongs to set $\mathcal{S}$ only with negligible probability; the same holds for r^* and $\mathcal{R}$. This means that, in the protocol execution for plaintext m and uniformly chosen s^*, r^*, except with negligible probability, the transcript $({\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$ will be generated as follows:

- ${\mu_1}^* = H_1(s^*, m)$;
- ${\mu_2}^* = H_2(r^*, {\mu_1}^*)$;
- ${\mu_3}^* = \mathsf{Enc}_K(m, {\mu_1}^*, {\mu_2}^*, 0)$,

and therefore $\mathsf{Dec}(r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$ will return the correct plaintext m via the main step.

To prove security of the scheme, we show that for any (potentially unbounded) adversary $\mathcal{A}$ which makes only polynomial number of queries to the oracle, the distributions H_0 and $H_{11,7}$ are statistically indistinguishable, where H_0 corresponds to the output of the adversary which sees the real execution of the protocol for plaintext m_0, together with real randomness s^*, r^*, and $H_{11,7}$ corresponds to the output of the adversary which sees the execution of the protocol for plaintext m_1, together with fake randomness s', r' which makes it look consistent with m_0. To prove this, we consider intermediate hybrid distributions $\{H_i\}$ and show that for each i the distributions H_i and H_{i-1} are statistically indistinguishable. For this proof in the oracle-access model, we will consider the case $m_0 \neq m_1 \in \{0,1\}$.

Below we describe each hybrid experiment. For convenience, we mark the changes from the previous experiment in red.

By writing $\mathcal{A}^O(x)$ we mean the output of adversary $\mathcal{A}$ on input x, where the adversary has oracle access to algorithm O.

Oracles $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$**.**

Oracle $\mathsf{P1}(s, m)$

Inputs: sender randomness s, plaintext m.

Hardwired values: key K_S of sender-fake encryption scheme, hash H_1 with sparse image.

1. **Trapdoor step:**
   (a) If $s \in \mathcal{S}$ and $\mathsf{Dec}_{K_S}(s).m = m$, then return $\mathsf{Dec}_{K_S}(s).\mu_1$;
2. **Main step:**
   (a) Else return $H_1(s, m)$.

Oracle $\mathsf{P3}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, plaintext m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: key K_S of sender-fake encryption scheme, key K of main encryption scheme.

1. **Validity check:**
   (a) If $\mathsf{P1}(s, m) \neq \mu_1$ then $\perp$;
2. **Trapdoor step:**
   (a) If $s \in \mathcal{S}$ and $(\mathsf{Dec}_{K_S}(s).m, \mathsf{Dec}_{K_S}(s).\mu_1, \mathsf{Dec}_{K_S}(s).\mu_2) = (m, \mu_1, \mu_2)$ then return $\mathsf{Dec}_{K_S}(s).\mu_3$;
3. **Mixed input step:**
   (a) Else if $s \in \mathcal{S}$ and $(\mathsf{Dec}_{K_S}(s).m, \mathsf{Dec}_{K_S}(s).\mu_1) = (m, \mu_1)$ then return $\mathsf{Enc}_K(m, \mu_1, \mu_2, \mathsf{Dec}_{K_S}(s).\ell)$;
4. **Main step:**
   (a) Else return $\mathsf{Enc}_K(m, \mu_1, \mu_2, 0)$.

Oracle $\mathsf{SFake}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real plaintext m, fake plaintext $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: key K_S of sender-fake encryption scheme, upper bound T.

1. **Validity check:**
   (a) If $\mathsf{P1}(s, m) \neq \mu_1$ then $\perp$;
2. **Trapdoor step:**
   (a) If $s \in \mathcal{S}$ and $(\mathsf{Dec}_{K_S}(s).m, \mathsf{Dec}_{K_S}(s).\mu_1) = (m, \mu_1)$ then
      i. If $\mathsf{Dec}_{K_S}(s).\ell = T$ then $\perp$;
      ii. Else return $\mathsf{Enc}_{K_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \mathsf{Dec}_{K_S}(s).\ell + 1)$.
3. **Main step:**
   (a) Else return $\mathsf{Enc}_{K_S}(\hat{m}, \mu_1, \mu_2, \mu_3, 1)$.

Figure 9: Programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$. $\mathcal{S}, \mathcal{R}, \mathcal{M}, \mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3$ denote the image of sender-fake encryption scheme (with key K_S), receiver-fake encryption scheme (with key K_R), the main encryption scheme (with key K), and hashes H_1, H_2, H_3, respectively.

Oracles $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$**.**

Oracle $\mathsf{P2}(r, \mu_1)$

Inputs: receiver randomness r, the first message μ_1 in the protocol.

Hardwired values: key K_R of receiver-fake encryption scheme, hash H_2 with sparse image.

1. **Trapdoor step:**
   (a) If $r \in \mathcal{R}$ and $\mathsf{Dec}_{K_R}(r).\mu_1 = \mu_1$, then return $\mathsf{Dec}_{K_R}(r).\mu_2$;
2. **Main step:**
   (a) Return $H_2(r, \mu_1)$.

Oracle $\mathsf{Dec}(r, \mu_1, \mu_2, \mu_3)$

Inputs: receiver randomness r, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: key K_R of receiver-fake encryption scheme, key K of the main encryption scheme, upper bound T.

1. **Validity check:**
   (a) If $\mathsf{P2}(r, \mu_1) \neq \mu_2$ then $\perp$;
2. **Trapdoor step:**
   (a) If $r \in \mathcal{R}$ and $(\mathsf{Dec}_{K_R}(r).\mu_1, \mathsf{Dec}_{K_R}(r).\mu_2, \mathsf{Dec}_{K_R}(r).\mu_3) = (\mu_1, \mu_2, \mu_3)$ then return $\mathsf{Dec}_{K_R}(r).m$;
3. **Mixed input step:**
   (a) If $r \in \mathcal{R}$ and $(\mathsf{Dec}_{K_R}(r).\mu_1, \mathsf{Dec}_{K_R}(r).\mu_2) = (\mu_1, \mu_2)$ then
      i. If $\mu_3 \in \mathcal{M}$ and $(\mu_1, \mu_2) = (\mathsf{Dec}_K(\mu_3).\mu_1, \mathsf{Dec}_K(\mu_3).\mu_2)$ and $\mathsf{Dec}_{K_R}(r).\ell < \mathsf{Dec}_K(\mu_3).\ell$ then return $\mathsf{Dec}_K(\mu_3).m$;
      ii. Else $\perp$.
4. **Main step:**
   (a) If $\mu_3 \in \mathcal{M}$ and $(\mathsf{Dec}_K(\mu_3).\mu_1, \mathsf{Dec}_K(\mu_3).\mu_2) = (\mu_1, \mu_2)$ then return $\mathsf{Dec}_K(\mu_3).m$;
   (b) Else $\perp$.

Oracle $\mathsf{RFake}(\hat{m}, \mu_1, \mu_2, \mu_3; \rho)$

Inputs: fake plaintext $\hat{m}$, protocol transcript μ_1, μ_2, μ_3, random coins ρ.

Hardwired values: key K_R of receiver-fake encryption scheme, key K of the main encryption scheme, function H_3 with a sparse image.

1. If $\mu_3 \in \mathcal{M}$ and $\mathsf{Dec}_K(\mu_3).\mu_1 = \mu_1$ and $\mathsf{Dec}_K(\mu_3).\mu_2 = \mu_2$ then return $\mathsf{Enc}_{K_R}(\hat{m}, \mu_1, \mu_2, \mu_3, \mathsf{Dec}_K(\mu_3).\ell, H_3(\rho))$;
2. Else $\perp$.

Figure 10: Oracles $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$. $\mathcal{S}, \mathcal{R}, \mathcal{M}, \mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3$ denote the image of sender-fake encryption scheme (with key K_S), receiver-fake encryption scheme (with key K_R), the main encryption scheme (with key K), and hashes H_1, H_2, H_3, respectively.

- $H_0 : \mathcal{A}^{\mathsf{P1},\mathsf{P2},\mathsf{P3},\mathsf{Dec},\mathsf{SFake},\mathsf{RFake}}(m_0, m_1, s^*, r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where s^*, r^* are chosen uniformly at random, $\mu_1{}^* = \mathsf{P1}(s^*, m_0)$, $\mu_2{}^* = \mathsf{P2}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{P3}(s^*, m_0, \mu_1{}^*, \mu_2{}^*)$.

  This experiment corresponds to the adversary observing the execution of the protocol with plaintext m_0, who is given true randomness s^*, r^*.

- $H_1 : \mathcal{A}^{\mathsf{P1},\mathsf{P2},\mathsf{P3},\mathsf{Dec},\mathsf{SFake},\mathsf{RFake}}(m_0, m_1, s^*, r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where s^*, r^* are chosen uniformly at random, $\mu_1{}^* = H_1(s^*, m_0)$, $\mu_2{}^* = H_2(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{Enc}_K(m_0, \mu_1{}^*, \mu_2{}^*, 0)$. If $s^* \in \mathcal{S}$ or $r^* \in \mathcal{R}$, the experiment aborts.

  This experiment is similar to the previous one, except that it aborts if $s^* \in \mathcal{S}$ or $r^* \in \mathcal{R}$, which happens with negligible probability. Thus, this experiment is statistically close to the previous one.

  Since $s^* \notin \mathcal{S}$, we explicitly write $\mu_1{}^* = H_1(s^*, m_0)$, instead of $\mu_1{}^* = \mathsf{P1}(s^*, m_0)$; similar with $\mu_2{}^*, \mu_3{}^*$.

- $H_2 : \mathcal{A}^{\mathsf{P1},\mathsf{P2},\mathsf{P3},\mathsf{Dec},\mathsf{SFake},\mathsf{RFake}}(m_0, m_1, s', r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where s^*, r^* are chosen uniformly at random, $\mu_1{}^* = H_1(s^*, m_0)$, $\mu_2{}^* = H_2(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{Enc}_K(m_0, \mu_1{}^*, \mu_2{}^*, 0)$, and $s' = \mathsf{Enc}_{K_S}(m_0, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, 0)$. If $s^* \in \mathcal{S}$ or $r^* \in \mathcal{R}$, the experiment aborts.

  This experiment is similar to the previous one, except that the adversary receives sender randomness s' which comes from a fake set $\mathcal{S}$, instead of true s^*. Note that $s' = \mathsf{Enc}_{K_S}(m_0, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, 0)$, i.e. s' so far has level 0, and contains the fake plaintext m_0 which is the same as the real plaintext.

  We argue that this experiment is identical to the previous one. Roughly, this is because all oracles, given s^* or s' as input, output either the same values or identically distributed ones. Indeed, lets analyze how s^* and s' are used within the oracles:

  1. Oracle P1 contains the following entries which include s^* or s':
     (a) Entries $s^*, m_0 \to \mu_1{}^*$ (in the main step) and $s', m_0 \to \mu_1{}^*$ (in the trapdoor step),
     (b) Entries $s^*, m_1 \to H_1(s^*, m_1)$ and $s', m_1 \to H_1(s', m_1)$ (both in the main step).
  2. Oracle P3 contains the following entries which include s^* or s':
     (a) Entries $s^*, m_0, \mu_1{}^*, \mu_2{}^* \to \mu_3{}^*$ (in the main step) and $s', m_0, \mu_1{}^*, \mu_2{}^* \to \mu_3{}^*$ (in the trapdoor step),
     (b) For every string $\mu_2 \neq \mu_2{}^*$ of the correct length, there are entries $s^*, m_0, \mu_1{}^*, \mu_2 \to \mathsf{Enc}_K(m, \mu_1{}^*, \mu_2, 0)$ (in the main step) and $s', m_0, \mu_1{}^*, \mu_2 \to \mathsf{Enc}_K(m, \mu_1{}^*, \mu_2, \mathsf{Dec}_{K_S}(s').\ell) = \mathsf{Enc}_K(m, \mu_1{}^*, \mu_2, 0)$ (in the mixed input step)[18],
     (c) For every string (m, μ_1, μ_2) of the correct length, such that $\mu_1 = H_1(s^*, m)$, there is an entry $s^*, m, \mu_1, \mu_2 \to \mathsf{Enc}_K(m, \mu_1, \mu_2, 0)$ (in the main step). Since these entries for the case $(m, \mu_1) = (m_0, \mu_1{}^*)$ were already accounted for in steps 1 and 2, here we consider the case $(m, \mu_1) \neq (m_0, \mu_1{}^*)$. For all remaining strings (m, μ_1, μ_2) there is an entry $s^*, m, \mu_1, \mu_2 \to \perp$ (in the validity check).

        In addition, for every string (m, μ_1, μ_2) of the correct length, such that $\mu_1 = H_1(s', m)$ and $(m, \mu_1) \neq (m_0, \mu_1{}^*)$[19], there is an entry $s', m, \mu_1, \mu_2 \to \mathsf{Enc}_K(m, \mu_1, \mu_2, 0)$ (in the

[18]Indeed, note that $\mathsf{Dec}_{K_S}(s').\ell = 0$.

[19]Indeed, if $(m, \mu_1) = (m_0, \mu_1{}^*)$, then P3 on input s' uses either trapdoor step or mixed input step, but never the main step.

main step). For all remaining strings (m, μ_1, μ_2) there is an entry $s^*, m, \mu_1, \mu_2 \to \perp$ (in the validity check).

3. Oracle SFake contains the following entries which include s^* or s':

   (a) For every string $(\hat{m}, \mu_2, \mu_3)$ of the correct length, there is an entry $s^*, m_0, \hat{m}, {\mu_1}^*, \mu_2, \mu_3 \to \mathsf{Enc}_{K_S}(\hat{m}, {\mu_1}^*, \mu_2, \mu_3, 1)$ (in the main step), and an entry $s', m_0, \hat{m}, {\mu_1}^*, \mu_2, \mu_3 \to \mathsf{Enc}_{K_S}(\hat{m}, {\mu_1}^*, \mu_2, \mu_3, 1)$ (in the trapdoor step)[20].

   (b) For every string $(m, \hat{m}, \mu_1, \mu_2, \mu_3)$ of the correct length, such that $\mu_1 = H_1(s^*, m)$, there is an entry $s^*, m, \hat{m}, \mu_1, \mu_2, \mu_3 \to \mathsf{Enc}_{K_S}(\hat{m}, \mu_1, \mu_2, \mu_3, 1)$ (in the main step). Since these entries for the case $(m, \mu_1) = (m_0, {\mu_1}^*)$ were already accounted for in step 1, here we consider the case $(m, \mu_1) \neq (m_0, {\mu_1}^*)$. For all remaining strings $(m, \hat{m}, \mu_1, \mu_2, \mu_3)$ there is an entry $s^*, m, \hat{m}, \mu_1, \mu_2, \mu_3 \to \perp$ (in the validity check).

   In addition, for every string $(m, \hat{m}, \mu_1, \mu_2, \mu_3)$ of the correct length, such that $\mu_1 = H_1(s', m)$ and $(m, \mu_1) \neq (m_0, {\mu_1}^*)$[21], there is an entry $s', m, \hat{m}, \mu_1, \mu_2, \mu_3 \to \mathsf{Enc}_{K_S}(\hat{m}, \mu_1, \mu_2, \mu_3, 1)$ (in the main step). For all remaining strings $(m, \hat{m}, \mu_1, \mu_2, \mu_3)$ there is an entry $s', m, \hat{m}, \mu_1, \mu_2, \mu_3 \to \perp$ (in the validity check).

Note that in all cases s^* and s' participate either in identical entries (such as cases 1(a), 2(a), 2(b), 3(a)) or in entries which have the same distribution (cases 1(b), 2(c), 3(b)), and recall that s^* and s' are themselves uniformly chosen strings. Therefore this experiment is identical to the previous one.

- H_3 : $\mathcal{A}^{\mathsf{P1,P2,P3,Dec,SFake,RFake}}(m_0, m_1, s', r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where s^*, r^* are chosen uniformly at random, ${\mu_1}^* = H_1(s^*, m_0)$, ${\mu_2}^* = H_2(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{Enc}_K(m_0, {\mu_1}^*, {\mu_2}^*, 0)$, and $s' = \mathsf{Enc}_{K_S}(m_0, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, 0)$. If $s^* \in \mathcal{S}$ or $r^* \in \mathcal{R}$, the experiment aborts. If the adversary queries any oracle on any input containing s^*, the experiment aborts.

  This experiment is similar to the previous one, except that it aborts if the adversary ever issues a query containing s^*. Note that s^* is a uniformly random variable which is independent of the oracles' output; thus the adversary could query s^* only by guessing it, which happens with negligible probability. Therefore, this experiment is statistically close to the previous one.

- H_4 : $\mathcal{A}^{\mathsf{P1,P2,P3,Dec,SFake,RFake}}(m_0, m_1, s', r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where r^* is chosen uniformly at random, ${\mu_1}^*$ is chosen uniformly at random independently of H_1, ${\mu_2}^* = H_2(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{Enc}_K(m_0, {\mu_1}^*, {\mu_2}^*, 0)$, and $s' = \mathsf{Enc}_{K_S}(m_0, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, 0)$. If $r^* \in \mathcal{R}$ or ${\mu_1}^* \in \mathcal{H}_1$, the experiment aborts.

  This experiment is similar to the previous one, except that ${\mu_1}^*$ which is given to the adversary is chosen uniformly at random, instead of being set to its proper value $H_1(s^*, m_0)$ (in particular, ${\mu_1}^*$ is different from the value $H_1(s^*, m_0)$ which is stored by the oracles). Further, we also change the experiment to abort if uniformly random ${\mu_1}^*$ is in the image $\mathcal{H}_1$ of H_1, which happens with negligible probability. Finally, note that s^* is not part of the experiment anymore and there is no need to generate it.

  Note that the only way for the adversary to check if the oracle stores ${\mu_1}^*$ or $H_1(s^*, m_0)$ is to query it on some preimage (s, m) of $H_1(s^*, m_0)$, which can only happen with negligible probability. Therefore this experiment is statistically close to the previous one.

[20] Indeed, note that $\mathsf{Dec}_{K_S}(s').\ell + 1 = 1$

[21] Indeed, if $(m, \mu_1) = (m_0, {\mu_1}^*)$, then SFake on input s' uses trapdoor step, but never the main step.

- H_5 : $\mathcal{A}^{\mathsf{P1,P2,P3,Dec,SFake,RFake}}(m_0, m_1, s', r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where r^* is chosen uniformly at random, ${\mu_1}^*$ is chosen uniformly at random independently of H_1, ${\mu_2}^* = H_2(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{Enc}_K(m_0, {\mu_1}^*, {\mu_2}^*, 0)$, and $s' = \mathsf{Enc}_{K_S}(m_0, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, 0)$. If $r^* \in \mathcal{R}$ or ${\mu_1}^* \in \mathcal{H}_1$, the experiment aborts. If the adversary queries any oracle on any input containing $s \in \mathcal{S}'_0$, where $\mathcal{S}'_0 = \left\{\mathsf{Enc}_{K_s}(\hat{m}, \mu_1, \mu_2, \mu_3, 0) : (\hat{m}, \mu_1, \mu_2, \mu_3) \in \{0,1\}^{1+|\mu_1|+|\mu_2|+|\mu_3|}\right\} \setminus s'$, the experiment aborts.

  This experiment is similar to the previous one except that it aborts if the adversary ever makes a query containing sender randomness of a fake format with level 0 (except s', which is given to the adversary).

  Note that the oracles' outputs are independent of $\mathcal{S}'_0$ (in particular, note that neither oracle outputs fake sender randomness with level 0: indeed, in the output of SFake levels start with 1), therefore the adversary cannot find such $s \in \mathcal{S}'_0$ except for guessing it, which happens with negligible probability. Therefore this experiment is statistically close to the previous one.

- H_6 : $\mathcal{A}^{\mathsf{P1,P2,P3,Dec,SFake,RFake}}(m_0, m_1, s', r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where r^* is chosen uniformly at random, ${\mu_1}^*$ is chosen uniformly at random independently of H_1, ${\mu_2}^* = H_2(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{Enc}_K(m_0, {\mu_1}^*, {\mu_2}^*, 0)$, and $s' = \mathsf{Enc}_{K_S}(m_0, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, 0)$. If $r^* \in \mathcal{R}$ or ${\mu_1}^* \in \mathcal{H}_1$, the experiment aborts. If the adversary queries any oracle on any input containing $s \in \mathcal{S}'_0$, where $\mathcal{S}'_0 = \left\{\mathsf{Enc}_{K_s}(\hat{m}, \mu_1, \mu_2, \mu_3, 0) : (\hat{m}, \mu_1, \mu_2, \mu_3) \in \{0,1\}^{1+|\mu_1|+|\mu_2|+|\mu_3|}\right\} \setminus s'$, the experiment aborts. If the adversary queries any oracle on any input containing $\overline{{\mu_3}^*} = \mathsf{Enc}_K(m_1, {\mu_1}^*, {\mu_2}^*, 0)$ the experiment aborts.

  This experiment is similar to the previous one except that it aborts if the adversary ever queries any oracle on $\overline{{\mu_3}^*} = \mathsf{Enc}_K(m_1, \mu_1{}^*, \mu_2{}^*, 0)$ (this ciphertext can be thought of as "complement" of the challenge ciphertext ${\mu_3}^* = \mathsf{Enc}_K(m_0, {\mu_1}^*, {\mu_2}^*, 0)$ since it encrypts the same ${\mu_1}^*, {\mu_2}^*, 0$, but the opposite bit m_1).

  We argue that $\overline{{\mu_3}^*}$ can only be found by the adversary by guessing certain variables, with negligible chance of success. First, we will give some intuition: we claim that the adversary can find $\overline{{\mu_3}^*}$ only by doing one of the following:

  1. Guessing $\overline{{\mu_3}^*}$;
  2. Forcing P3 to output $\overline{{\mu_3}^*}$ via trapdoor step, by running P3 on some fake s which encrypts $\overline{{\mu_3}^*}$;
  3. Forcing P3 to output $\overline{{\mu_3}^*}$ via mixed input step, by running P3 on a certain fake $s \neq s'$ with level 0.

  Intuitively, the adversary's chance of succeeding in case one is negligible due to sparseness of the encryption scheme; in the second case, to generate such an s, the adversary would have to know $\overline{{\mu_3}^*}$ to begin with; and in the third case the adversary would have to find fake $s \neq s'$ with level 0, which is not an output of any oracle and therefore it can only be guessed by the adversary with negligible probability.

  Now we give a formal argument. We claim that the ciphertext $\overline{{\mu_3}^*} = \mathsf{Enc}_K(m_1, {\mu_1}^*, {\mu_2}^*, 0)$ can be removed from the description of oracle P3, without changing the experiment. First, recall that ${\mu_1}^* \notin \mathcal{H}_1$ (otherwise the experiment aborts). This means that the only way to satisfy the validity check in P3 with ${\mu_1}^*$ is to provide P3 with an input $(s, m, {\mu_1}^*, \mu_2)$ such that $\mathsf{Dec}_{K_S}(s).m = m$, $\mathsf{Dec}_{K_S}(s).\mu_1 = {\mu_1}^*$. However, in this case oracle P3 never executes the main step (either trapdoor step or mixed input step will be executed). Therefore we can remove the description of $\overline{\mu_3{}^*}$ from K in the main step.

Second, we claim that we can remove the description of $\overline{\mu_3{}^*}$ from K in the mixed input step as well. Indeed, note that $\overline{\mu_3{}^*}$ is an encryption of level 0 (together with other values). Note that the mixed input step copies the level $\mathsf{Dec}_{K_S}(s).\ell$ into the ciphertext $\mathsf{Enc}_K(m, \mu_1, \mu_2, \mathsf{Dec}_{K_S}(s).\ell)$; this means that the only way to force the mixed input step to encrypt level 0 is to query P3 on some $s \in \mathcal{S}$ such that $\mathsf{Dec}_{K_S}(s).\ell = 0$. However, in our experiment the adversary never queries $s \in \mathcal{S}'_0$ (otherwise the experiment aborts), therefore the only level-0 s which can be queried is $s' = \mathsf{Enc}_{K_S}(m_0, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, 0)$. Finally, for P3 to output $\overline{\mu_3{}^*} = \mathsf{Enc}_K(m_1, \mu_1{}^*, \mu_2{}^*, 0)$ on inputs s', m, μ_1, μ_2 via mixed input step, its inputs (m, μ_1, μ_2) should be set to $(m_1, \mu_1{}^*, \mu_2{}^*)$. However, inputs $(s', m_1, \mu_1{}^*, \mu_2{}^*)$ to oracle P3 will not pass the validity check, since $\mathsf{P1}(s', m_1) \neq \mu_1{}^*$ (indeed, $\mu_1{}^* \notin \mathcal{H}_1$, and $\mathsf{Dec}_{K_S}(s).m = m_0 \neq m_1$, thus neither trapdoor step nor main step of P1 results in $\mu_1{}^*$).

Third, we note that, formally speaking, the string $\overline{\mu_3{}^*}$ is present in the trapdoor step of P3, since this step outputs $\mathsf{Dec}_{K_S}(s).\mu_3$, which could happen to be $\overline{\mu_3{}^*}$. However, this step contains the description of all binary strings of length $|\mu_3|$, since any such string could be equal to $\mathsf{Dec}_{K_S}(s).\mu_3$ for some s. In other words, the description of trapdoor step is independent of $\overline{\mu_3{}^*}$.

Therefore we can remove the description of $\overline{\mu_3{}^*}$ from P3 without changing the experiment. Finally, we note that all other oracles' outputs are independent of $\overline{\mu_3{}^*}$. Therefore the probability that the adversary queries $\overline{\mu_3{}^*}$ is at most the probability of guessing it, which is negligible.

Thus, this experiment is statistically close to the previous one.

- H_7 : $\mathcal{A}^{\mathsf{P1,P2,P3,Dec,SFake,RFake}}(m_0, m_1, s', r', \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where r^* is chosen uniformly at random, $\mu_1{}^*$ is chosen uniformly at random independently of H_1, $\mu_2{}^* = H_2(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{Enc}_K(m_0, \mu_1{}^*, \mu_2{}^*, 0)$, $s' = \mathsf{Enc}_{K_S}(m_0, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, 0)$, $r' = \mathsf{Enc}_{K_R}(m_0, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, 0, \hat{\rho}^*)$ for uniformly chosen $\hat{\rho}^*$. If $r^* \in \mathcal{R}$ or $\mu_1{}^* \in \mathcal{H}_1$ or $\hat{\rho}^* \in \mathcal{H}_3$, the experiment aborts. If the adversary queries any oracle on any input containing $s \in \mathcal{S}'_0$, where $\mathcal{S}'_0 = \left\{\mathsf{Enc}_{K_s}(\hat{m}, \mu_1, \mu_2, \mu_3, 0) : (\hat{m}, \mu_1, \mu_2, \mu_3) \in \{0,1\}^{1+|\mu_1|+|\mu_2|+|\mu_3|}\right\} \setminus s'$, the experiment aborts. If the adversary queries any oracle on any input containing $\overline{\mu_3{}^*} = \mathsf{Enc}_K(m_1, \mu_1{}^*, \mu_2{}^*, 0)$ the experiment aborts.

  This experiment is similar to the previous one, except that the adversary receives fake r' and not the real r^* as the randomness of the receiver.

  We argue that this experiment is identical to the previous one. Roughly, this is because all oracles, given r^* or r' as input, output either the same values or identically distributed ones; while this is not true for some bad inputs, our experiment aborts if the adversary ever queries such an input. Indeed, lets analyze how r^* and r' are used within the oracles:

  1. Oracle P2 contains the following entries which include r^* or r':
     (a) Entries $r^*, \mu_1{}^* \to \mu_2{}^*$ (in the main step) and $r', \mu_1{}^* \to \mu_2{}^*$ (in the trapdoor step),
     (b) For all $\mu_1 \neq \mu_1{}^*$, entries $r^*, \mu_1 \to H_2(r^*, \mu_1)$ and $r', \mu_1 \to H_2(r', \mu_1)$ (both in the main step).
  2. Oracle Dec contains the following entries which include r^* or r':

(a) For every string $\mu_3 = \mathsf{Enc}_K(m, {\mu_1}^*, {\mu_2}^*, \ell)$ such that $m \in \{0,1\}$, $\ell \in [0, \ldots, T]$, there is an entry $r^*, {\mu_1}^*, {\mu_2}^*, \mu_3 \to m$ (in the main step).

For every string $\mu_3 = \mathsf{Enc}_K(m, {\mu_1}^*, {\mu_2}^*, \ell)$ such that $m \in \{0,1\}$, $\ell \in [1, \ldots, T]$, there is an entry $r', {\mu_1}^*, {\mu_2}^*, \mu_3 \to m$ (in the mixed input step).

Note that entries for r' *do not* contain entries for μ_3 with level $\ell = 0$. In particular, so far we listed two entries for r^* which r' doesn't have:

i. $r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^* \to m_0$, where ${\mu_3}^* = \mathsf{Enc}_K(m_0, {\mu_1}^*, {\mu_2}^*, 0)$

ii. $r^*, {\mu_1}^*, {\mu_2}^*, \overline{\mu_3}^* \to m_1$, where $\overline{\mu_3}^* = \mathsf{Enc}_K(m_1, {\mu_1}^*, {\mu_2}^*, 0)$.

However, one of these entries for r' ($r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^* \to m_0$) appears in the trapdoor step. The other entry however is different from r^*-entry: indeed, while r^*-entry says $r^*, {\mu_1}^*, {\mu_2}^*, \overline{\mu_3}^* \to m_1$, r'-entry says $r', {\mu_1}^*, {\mu_2}^*, \overline{\mu_3}^* \to \bot$ (in the mixed input step), since the condition $\mathsf{Dec}_{K_R}(r).\ell < \mathsf{Dec}_K(\mu_3).\ell$ is violated due to both levels being 0. However, our experiment aborts if the adversary ever queries any oracle on input $\overline{\mu_3}^*$, and therefore the fact that Dec outputs different output on input r' or r^* doesn't change the distribution of the experiment, since such "differing input" is not queried by the adversary.

(b) For every string (μ_1, μ_2, μ_3) of the correct length, such that $\mu_2 = H_2(r^*, \mu_1)$ and $\mu_3 = \mathsf{Enc}_K(m, \mu_1, \mu_2, \ell)$ for $m \in \{0,1\}$, $\ell \in [0, \ldots, T]$, there is an entry $r^*, \mu_1, \mu_2, \mu_3 \to m$ (in the main step). Since these entries for the case $(\mu_1, \mu_2) = ({\mu_1}^*, {\mu_2}^*)$ were already accounted for in steps 1 and 2, here we consider the case $(\mu_1, \mu_2) \neq ({\mu_1}^*, {\mu_2}^*)$. For all strings (μ_1, μ_2, μ_3) which are not already considered, there is an entry $r^*, \mu_1, \mu_2, \mu_3 \to \bot$.

For every string (μ_1, μ_2, μ_3) of the correct length, such that $\mu_2 = H_2(r', \mu_1)$ and $\mu_3 = \mathsf{Enc}_K(m, \mu_1, \mu_2, \ell)$ for $m \in \{0,1\}$, $\ell \in [0, \ldots, T]$, there is an entry $r', \mu_1, \mu_2, \mu_3 \to m$ (in the main step). (Note that in this case $(\mu_1, \mu_2) \neq ({\mu_1}^*, {\mu_2}^*)$, since $\mu_2 \neq H_2(r', \mu_1)$) For all strings (μ_1, μ_2, μ_3) which are not already considered, there is an entry $r', \mu_1, \mu_2, \mu_3 \to \bot$.

3. Oracle RFake doesn't r as input.

Note that in all cases r^* and r' participate either in identical entries (such as case 1(a)) or in identically distributed ones (cases 2(a), 2(b)), and recall that r^* and r' are themselves uniformly chosen strings. Therefore this experiment is identical to the previous one.

- $H_8 : \mathcal{A}^{\mathsf{P1},\mathsf{P2},\mathsf{P3},\mathsf{Dec},\mathsf{SFake},\mathsf{RFake}}(m_0, m_1, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where ${\mu_1}^*$ is chosen uniformly at random independently of H_1, ${\mu_2}^*$ is chosen uniformly at random independently of H_2, ${\mu_3}^* = \mathsf{Enc}_K(m_0, {\mu_1}^*, {\mu_2}^*, 0)$, $s' = \mathsf{Enc}_{K_S}(m_0, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, 0)$, $r' = \mathsf{Enc}_{K_R}(m_0, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, 0, \hat{\rho}^*)$ for uniformly chosen $\hat{\rho}^*$. If ${\mu_1}^* \in \mathcal{H}_1$ or ${\mu_2}^* \in \mathcal{H}_2$ or $\hat{\rho}^* \in \mathcal{H}_3$, the experiment aborts. If the adversary queries any oracle on any input containing $s \in \mathcal{S}'_0$, where $\mathcal{S}'_0 = \left\{\mathsf{Enc}_{K_s}(\hat{m}, \mu_1, \mu_2, \mu_3, 0) : (\hat{m}, \mu_1, \mu_2, \mu_3) \in \{0,1\}^{1+|\mu_1|+|\mu_2|+|\mu_3|}\right\} \setminus s'$, the experiment aborts. If the adversary queries any oracle on any input containing $\overline{{\mu_3}^*} = \mathsf{Enc}_K(m_1, {\mu_1}^*, {\mu_2}^*, 0)$ the experiment aborts.

This experiment is similar to the previous one except that ${\mu_2}^*$ is chosen uniformly at random, independently of the value $H_2(r^*, {\mu_1}^*)$. In addition, r^* is now not part of the experiment and doesn't have to be generated. Further, we also make the experiment abort if uniformly random ${\mu_2}^*$ is in the image $\mathcal{H}_2$ of H_2, which happens with negligible probability.

Note that the only way for the adversary to check if the oracle stores ${\mu_2}^*$ or $H_2(r^*, {\mu_1}^*)$ is to query it on some preimage (r, μ_1) of $H_2(r^*, {\mu_1}^*)$, which can only happen with negligible probability. Therefore this experiment is statistically close to the previous one.

- H_9 : $\mathcal{A}^{\mathsf{P1,P2,P3,Dec,SFake,RFake}}(m_0, m_1, s', r', {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*})$, where ${\mu_1}^*$ is chosen uniformly at random independently of H_1, ${\mu_2}^*$ is chosen uniformly at random independently of H_2, $\overline{{\mu_3}^*} = \mathsf{Enc}_K(m_1, {\mu_1}^*, {\mu_2}^*, 0)$, $s' = \mathsf{Enc}_{K_S}(m_0, {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*}, 0)$, $r' = \mathsf{Enc}_{K_S}(m_0, {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*}, 0, \hat{\rho}^*)$ for uniformly chosen $\hat{\rho}^*$. If ${\mu_1}^* \in \mathcal{H}_1$ or ${\mu_2}^* \in \mathcal{H}_2$ or $\hat{\rho}^* \in \mathcal{H}_3$, the experiment aborts. If the adversary queries any oracle on any input containing $s \in \mathcal{S}'_0$, where $\mathcal{S}'_0 = \left\{\mathsf{Enc}_{K_s}(\hat{m}, \mu_1, \mu_2, \mu_3, 0) : (\hat{m}, \mu_1, \mu_2, \mu_3) \in \{0,1\}^{1+|\mu_1|+|\mu_2|+|\mu_3|}\right\} \setminus s'$, the experiment aborts. If the adversary queries any oracle on any input containing ${\mu_3}^* = \mathsf{Enc}_K(m_0, {\mu_1}^*, {\mu_2}^*, 0)$ the experiment aborts.

  In this experiment we switch the roles of ${\mu_3}^*$ and $\overline{{\mu_3}^*}$: that is, we give the adversary $\overline{{\mu_3}^*} = \mathsf{Enc}_K(m_1, {\mu_1}^*, {\mu_2}^*, 0)$ encrypting m_1, instead of ${\mu_3}^* = \mathsf{Enc}_K(m_0, {\mu_1}^*, {\mu_2}^*, 0)$ encrypting m_0. Next, we use $\overline{{\mu_3}^*}$ instead of ${\mu_3}^*$ to generate fake s', r'. Next, we make the experiment abort if the adversary queries any input containing ${\mu_3}^* = \mathsf{Enc}_K(m_0, {\mu_1}^*, {\mu_2}^*, 0)$, instead of $\overline{{\mu_3}^*} = \mathsf{Enc}_K(m_1, {\mu_1}^*, {\mu_2}^*, 0)$ as before.

  We claim that this experiment is identical to the previous one. Let us analyze how ${\mu_3}^*$ and $\overline{{\mu_3}^*}$ are used in the programs:

  1. Program P3:
     (a) In the trapdoor step, for every entry of the form $(s, m, \mu_1, \mu_2) \to {\mu_3}^*$, there is an entry $(\overline{s}, m, \mu_1, \mu_2) \to \overline{{\mu_3}^*}$ (and vice versa), where s and $\overline{s}$ are such that $\mathsf{Dec}_{K_S}(s).\mu_3 = {\mu_3}^*$, $\mathsf{Dec}_{K_S}(\overline{s}).\mu_3 = \overline{{\mu_3}^*}$, and all other fields of $\overline{s}$ and s are the same.

     (b) In the mixed input step, we can remove all entries containing ${\mu_3}^*$, $\overline{{\mu_3}^*}$, without changing the experiment. This is because of the following: in order for P3 to output ${\mu_3}^*$ or $\overline{{\mu_3}^*}$ via mixed input step, it should be run on inputs $(s, m, {\mu_1}^*, {\mu_2}^*)$ for some s, m such that $s \in \mathcal{S}$ and s has level 0. Recall that our experiment aborts if the adversary queries any level-0 s except for s'. Finally, in order for $\mathsf{P3}(s', m, {\mu_1}^*, {\mu_2}^*)$ to output non-$\perp$ on input s', m should be equal to m_0 to pass the validity check, in which case P3 uses the trapdoor step (and outputs $\overline{{\mu_3}^*}$); in particular, doesn't use the mixed input step.

     (c) Finally, in the main step we can also remove all entries containing ${\mu_3}^*$, $\overline{{\mu_3}^*}$ without changing the experiment. Indeed, since ${\mu_1}^* \notin \mathcal{H}_1$, they only way to pass the the validity check in P3 with ${\mu_1}^*$ is to give it some $s \in \mathcal{S}$, which forces P3 to execute either trapdoor step or mixed input step.

  2. Program SFake has the same set of entries for all possible strings μ_3 of proper length;

  3. Program Dec:
     (a) In the trapdoor step, for every entry of the form $(r, \mu_1, \mu_2, {\mu_3}^*) \to m$, there is an entry $(\overline{r}, \mu_1, \mu_2, \overline{{\mu_3}^*}) \to m$ (and vice versa), where r and $\overline{r}$ are such that $\mathsf{Dec}_{K_R}(r).\mu_3 = {\mu_3}^*$, $\mathsf{Dec}_{K_R}(\overline{r}).\mu_3 = \overline{{\mu_3}^*}$, and all other fields of $\overline{r}$ and r are the same.

     (b) In the mixed input step, we can remove all entries containing ${\mu_3}^*$, $\overline{{\mu_3}^*}$, without changing the experiment. This is because of the following: in order for Dec to output non-$\perp$ via mixed input step, the condition $\mathsf{Dec}_{K_R}(r).\ell < \mathsf{Dec}_K(\mu_3).\ell$ should hold. However, both ${\mu_3}^*$ and $\overline{{\mu_3}^*}$ have level 0, therefore there doesn't exist r which satisfies this consition.

(c) Finally, in the main step we can remove all entries containing ${\mu_3}^*$, $\overline{{\mu_3}^*}$, without changing the experiment, since ${\mu_2}^* \notin \mathcal{H}_2$, and therefore, if the input passes the the validity check with ${\mu_2}^*$, it must be that $r \in \mathcal{R}$, which forces Dec to execute either trapdoor step or mixed input step.

4. Program RFake has the same set of entries for strings ${\mu_3}^*$ and $\overline{{\mu_3}^*}$, since the only information from μ_3 used by RFake is its level, which is the same (0) in ${\mu_3}^*$ and $\overline{{\mu_3}^*}$.

To conclude the argument, it remains to note that other programs do not use ${\mu_3}^*$ nor $\overline{{\mu_3}^*}$, and that in both experiments 8 and 9 fake randomness of the sender and the receiver corresponds to the claimed third message: that is, in experiment 8 s' and r' are both generated using ${\mu_3}^*$, and in experiment 9 s' and r' are both generated using $\overline{{\mu_3}^*}$. Thus, this experiment is identical to the previous one.

- $H_{11,1}$: $\mathcal{A}^{\mathsf{P1_1,P2,P3_1,Dec,SFake_1,RFake}}(m_0, m_1, s', r', {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*})$, where ${\mu_1}^*$ is chosen uniformly at random independently of H_1, ${\mu_2}^*$ is chosen uniformly at random independently of H_2, $\overline{{\mu_3}^*} = \mathsf{Enc}_K(m_1, {\mu_1}^*, {\mu_2}^*, 0)$, $s' = \mathsf{Enc}_{K_S}(m_0, {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*}, 1)$, $r' = \mathsf{Enc}_{K_R}(m_0, {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*}, 0, \hat{\rho}^*)$ for uniformly chosen $\hat{\rho}^*$. If ${\mu_1}^* \in \mathcal{H}_1$ or ${\mu_2}^* \in \mathcal{H}_2$ or $\hat{\rho}^* \in \mathcal{H}_3$, the experiment aborts. If the adversary queries any oracle on any input containing $s \in \mathcal{S}_0 \cup \mathcal{S}'_1$, where $\mathcal{S}_0 = \left\{\mathsf{Enc}_{K_s}(\hat{m}, \mu_1, \mu_2, \mu_3, 0) : (m, \mu_1, \mu_2, \mu_3) \in \{0,1\}^{1+|\mu_1|+|\mu_2|+|\mu_3|}\right\}$, and $\mathcal{S}'_1 = \left\{\mathsf{Enc}_{K_s}(\hat{m}, {\mu_1}^*, \mu_2, \mu_3, 1) : (m, \mu_2, \mu_3) \in \{0,1\}^{1+|\mu_2|+|\mu_3|}\right\} \setminus s'$, the experiment aborts. If the adversary queries any oracle on any input containing ${\mu_3}^* = \mathsf{Enc}_K(m_0, {\mu_1}^*, {\mu_2}^*, 0)$ the experiment aborts.

In this experiment we change the encryption table of the key K_S and adjust the code of the programs of the sender, as shown on fig. 13, to preserve the distribution of the experiment. (For convenience of verification, we also rewrote the code of the original programs of the sender but made the bound on ℓ explicit on fig. 11). In addition, we change s' from $s' = \mathsf{Enc}_{K_S}(m_0, {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*}, 0)$ to $s' = \mathsf{Enc}_{K_S}(m_0, {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*}, 1)$, and we change the set (which aborts the experiment when being queried by the adversary) from $\mathcal{S}'_0$ to $\mathcal{S}_0 \cup \mathcal{S}'_1$.

We now describe the changes in detail. First, we change the key K_S of a sender-fake encryption scheme as follows. Recall that key K_S is a table of all plaintext-ciphertext pairs, i.e. a table containing entries of the form $s \leftrightarrow (m, \mu_1, \mu_2, \mu_3, \ell)$ for all strings $s, m, \mu_1, \mu_2, \mu_3$ of the proper length and $\ell \in [0, T]$. In this experiment we replace each entry where $\mu_1 = {\mu_1}^*$ with another entry: that is, for each entry of the form $s \leftrightarrow (m, {\mu_1}^*, \mu_2, \mu_3, \ell)$ we replace it with another entry $s \leftrightarrow (m, {\mu_1}^*, \mu_2, \mu_3, \ell + 1)$, thus incrementing by 1 the value of level in some ciphertexts. In particular, the set of levels for which encryptions exist changes to $[1, T+1]$ from $[0, T]$ for $\mu_1 = {\mu_1}^*$. Note that this change affects the challenge s', which is switched from $\mathsf{Enc}_{K_S}(m_0, {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*}, 0)$ to $\mathsf{Enc}_{K_S}(m_0, {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*}, 1)$.

Second, we change the code of the programs so that they subtract 1 from the level of affected ciphertexts s, before using it, thus nullifying the change from the above and preserving the distribution. The resulting code is presented on fig. 13, and the changes are highlighted in red. Below we list the changes:

- In the mixed input step of program P3, we consider the cases $\mu_1 = {\mu_1}^*$ and $\mu_1 \neq {\mu_1}^*$ separately. For the case $\mu_1 \neq {\mu_1}^*$ the code remains unchanged. For the case $\mu_1 = {\mu_1}^*$, the program checks that the level in s is within $[1, T+1]$ (instead of $[0, T]$), and the program outputs an encryption of $\mathsf{Dec}_{K_S}(s).\ell - 1$ instead of $\mathsf{Dec}_{K_S}(s).\ell$.

- In the trapdoor step of program SFake, we consider the cases $\mu_1 = {\mu_1}^*$ and $\mu_1 \neq {\mu_1}^*$ separately. For the case $\mu_1 \neq {\mu_1}^*$ the code remains unchanged. For the case $\mu_1 = {\mu_1}^*$, the program checks that the level in s is within $[1, T]$ (instead of $[0, T-1]$).
- In the main step of program SFake, we do not need to make any changes to the program. Recall that the experiment aborts if ${\mu_1}^* \in \mathcal{H}_1$, and therefore the main step of SFake cannot be triggered on input ${\mu_1}^*$. As a result, the main step of SFake never needs to encrypt ${\mu_1}^*$ and is therefore not affected by our change of shifting the levels by 1.

Due to adjusted code, this experiment is identical to the previous one.

- $H_{11,2}$: $\mathcal{A}^{\mathsf{P1}_2,\mathsf{P2}_2,\mathsf{P3}_2,\mathsf{Dec}_2,\mathsf{SFake}_2,\mathsf{RFake}_2}(m_0, m_1, s', r', {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*})$, where ${\mu_1}^*$ is chosen uniformly at random independently of H_1, ${\mu_2}^*$ is chosen uniformly at random independently of H_2, $\overline{{\mu_3}^*} = \mathsf{Enc}_K(m_1, {\mu_1}^*, {\mu_2}^*, 0)$, $s' = \mathsf{Enc}_{K_S}(m_0, {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*}, 1)$, $r' = \mathsf{Enc}_{K_R}(m_0, {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*}, 0, \hat{\rho}^*)$ for uniformly chosen $\hat{\rho}^*$. If ${\mu_1}^* \in \mathcal{H}_1$ or ${\mu_2}^* \in \mathcal{H}_2$ or $\hat{\rho}^* \in \mathcal{H}_3$, the experiment aborts. If the adversary queries any oracle on any input containing $s \in \mathcal{S}_0 \cup \mathcal{S}_1'$, where $\mathcal{S}_0 = \left\{\mathsf{Enc}_{K_s}(\hat{m}, \mu_1, \mu_2, \mu_3, 0) : (m, \mu_1, \mu_2, \mu_3) \in \{0,1\}^{1+|\mu_1|+|\mu_2|+|\mu_3|}\right\}$, and $\mathcal{S}_1' = \left\{\mathsf{Enc}_{K_s}(\hat{m}, {\mu_1}^*, \mu_2, \mu_3, 1) : (m, \mu_2, \mu_3) \in \{0,1\}^{1+|\mu_2|+|\mu_3|}\right\} \setminus s'$, the experiment aborts. If the adversary queries any oracle on any input containing ${\mu_3}^* = \mathsf{Enc}_K(m_0, {\mu_1}^*, {\mu_2}^*, 0)$ the experiment aborts.

In this experiment we change the encryption table of the key K and adjust the code of the programs of the sender and the receiver, as shown on fig. 14, fig. 15. (For convenience of verification, we also rewrote the code of the original programs of the receiver but made the bound on ℓ explicit on fig. 12).

We now describe the changes in detail. First, we change the key K of the main encryption scheme as follows. Recall that key K is a table of all plaintext-ciphertext pairs, i.e. a table containing entries of the form $\mu_3 \leftrightarrow (m, \mu_1, \mu_2, \ell)$ for all strings m, μ_1, μ_2 of the proper length and $\ell \in [0, T]$. In this experiment we replace each entry where $\mu_1 = {\mu_1}^*$, except for $\overline{{\mu_3}^*}$ and ${\mu_3}^*$, with another entry: that is, for each entry of the form $\mu_3 \leftrightarrow (m, {\mu_1}^*, \mu_2, \ell)$ we replace it with another entry $\mu_3 \leftrightarrow (m, {\mu_1}^*, \mu_2, \ell+1)$ (as long as $\mu_3 \neq {\mu_3}^*, \overline{{\mu_3}^*}$), thus incrementing by 1 the value of level in some ciphertexts. In particular, the set of levels for which encryptions exist changes to $[1, T+1]$ from $[0, T]$ for $\mu_1 = {\mu_1}^*$. Note that this change does not affect the challenge ${\mu_3}^*$.

Second, we change the code of the programs so that they add 1 to the level before encrypting with key K, and subtract 1 from the level of decrypted μ_3 before using it, thus nullifying the change from the above and preserving the distribution. The resulting code is presented on fig. 14, fig. 15, and the changes are highlighted in red. Below we list the changes:

- In the mixed input step of program P3, we consider the cases $\mu_1 = {\mu_1}^*$ and $\mu_1 \neq {\mu_1}^*$ separately. For the case $\mu_1 \neq {\mu_1}^*$ the code remains unchanged. For the case $\mu_1 = {\mu_1}^*$, the program outputs an encryption of $\mathsf{Dec}_{K_S}(s).\ell$ instead of $\mathsf{Dec}_{K_S}(s).\ell - 1$.

  Recall that the levels of the two ciphertexts - ${\mu_3}^*$ and $\overline{{\mu_3}^*}$ encrypting $(1, {\mu_1}^*, {\mu_2}^*, 0)$ and $(0, {\mu_1}^*, {\mu_2}^*, 0)$ - were not incremented in a table of K. However, the mixed input step never needs to encrypt $(0, {\mu_1}^*, {\mu_2}^*, 0)$ or $(1, {\mu_1}^*, {\mu_2}^*, 0)$. Indeed, to force P3 to output an encryption of, say, $(0, {\mu_1}^*, {\mu_2}^*, 0)$ via mixed input step, one has to run it on fake s with level 0(i.e. $S \in \mathcal{S}_0$), but the experiment aborts in this case.

- In the mixed input step of program Dec, we consider the cases $\mu_1 = {\mu_1}^*$ and $\mu_1 \neq {\mu_1}^*$ separately. For the case $\mu_1 \neq {\mu_1}^*$ the code remains unchanged. For the case $\mu_1 = {\mu_1}^*$, the program checks that the level in μ_3 is within $[1, T+1]$ (instead of $[0, T]$), and compares $\mathsf{Dec}_{K_R}(r).\ell$ against $\mathsf{Dec}_K(\mu_3).\ell - 1$ instead of $\mathsf{Dec}_K(\mu_3).\ell$, to account for an incremented levels of some ciphertexts.

  Recall that the levels of the two ciphertexts - ${\mu_3}^*$ and $\overline{{\mu_3}^*}$ - were not incremented in a table of K. However, the mixed input step never outputs non-$\perp$ on input ${\mu_3}^*$ or $\overline{{\mu_3}^*}$. Indeed, this is because the condition $\mathsf{Dec}_{K_R}(r).\ell < \mathsf{Dec}_K(\mu_3).\ell$ can never be satisfied due to level of ${\mu_3}^*$, $\overline{{\mu_3}^*}$ being 0.

- In program RFake, we consider three cases: $\mu_3 = {\mu_3}^*, \overline{{\mu_3}^*}$, and else $\mu_1 = {\mu_1}^*$ and $\mu_1 \neq {\mu_1}^*$. For the case $\mu_1 \neq {\mu_1}^*$ and $\mu_3 = {\mu_3}^*, \overline{{\mu_3}^*}$ the code remains unchanged. For the case $\mu_1 = {\mu_1}^*$, the program checks that the level in μ_3 is within $[1, T+1]$ (instead of $[0, T]$), and decrements $\mathsf{Dec}_K(\mu_3).\ell$ by one before using it to compute the output.

Due to adjusted code, this experiment is identical to the previous one.

- $H_{11,3} : \mathcal{A}^{\mathsf{P1}_2, \mathsf{P2}_3, \mathsf{P3}_2, \mathsf{Dec}_3, \mathsf{SFake}_2, \mathsf{RFake}_3}(m_0, m_1, s', r', {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*})$, where ${\mu_1}^*$ is chosen uniformly at random independently of H_1, ${\mu_2}^*$ is chosen uniformly at random independently of H_2, $\overline{{\mu_3}^*} = \mathsf{Enc}_K(m_1, {\mu_1}^*, {\mu_2}^*, 0)$, $s' = \mathsf{Enc}_{K_S}(m_0, {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*}, 1)$, $r' = \mathsf{Enc}_{K_R}(m_0, {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*}, 0, \hat{\rho}^*)$ for uniformly chosen $\hat{\rho}^*$. If ${\mu_1}^* \in \mathcal{H}_1$ or ${\mu_2}^* \in \mathcal{H}_2$ or $\hat{\rho}^* \in \mathcal{H}_3$, the experiment aborts. If the adversary queries any oracle on any input containing $s \in \mathcal{S}_0 \cup \mathcal{S}'_1$, where $\mathcal{S}_0 = \left\{\mathsf{Enc}_{K_s}(\hat{m}, \mu_1, \mu_2, \mu_3, 0) : (m, \mu_1, \mu_2, \mu_3) \in \{0,1\}^{1+|\mu_1|+|\mu_2|+|\mu_3|}\right\}$, and $\mathcal{S}'_1 = \left\{\mathsf{Enc}_{K_s}(\hat{m}, {\mu_1}^*, \mu_2, \mu_3, 1) : (m, \mu_2, \mu_3) \in \{0,1\}^{1+|\mu_2|+|\mu_3|}\right\} \setminus s'$, the experiment aborts. If the adversary queries any oracle on any input containing ${\mu_3}^* = \mathsf{Enc}_K(m_0, {\mu_1}^*, {\mu_2}^*, 0)$ the experiment aborts.

In this experiment we change the encryption table of the receiver-faking key K_R and adjust the code of the programs of the receiver, as shown on fig. 16.

We now describe the changes in detail. First, we change the key K_R of the receiver faking scheme as follows. Recall that key K_R is a table of all plaintext-ciphertext pairs, i.e. a table containing entries of the form $r \leftrightarrow (\hat{m}, \mu_1, \mu_2, \mu_3, \ell, \hat{\rho})$ for all strings $m, \mu_1, \mu_2, \mu_3, \hat{\rho}$ of the proper length and $\ell \in [0, T]$. In this experiment we replace each entry where $\mu_1 = {\mu_1}^*$, except for $(\mu_1, \mu_2, \mu_3, \ell) = ({\mu_1}^*, {\mu_2}^*, {\mu_3}^*, 0)$ and $(\mu_1, \mu_2, \mu_3, \ell) = ({\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*}, 0)$, with another entry: that is, for each entry of the form $r \leftrightarrow (m, {\mu_1}^*, \mu_2, \mu_3, \ell, \hat{\rho})$ we replace it with another entry $r \leftrightarrow (m, {\mu_1}^*, \mu_2, \mu_3, \ell + 1, \hat{\rho})$ (as long as $(\mu_2, \mu_3, \ell) \neq ({\mu_2}^*, {\mu_3}^*, 0)$ and $(\mu_2, \mu_3, \ell) \neq ({\mu_2}^*, \overline{{\mu_3}^*}, 0)$), thus incrementing by 1 the value of level in some ciphertexts. Note that this change does not affect the challenge r'.

Second, we change the code of the programs so that they add 1 to the level before encrypting with key K_R, and subtract 1 from the level of decrypted r before using it, thus nullifying the change from the above and preserving the distribution. The resulting code is presented on fig. 16, and the changes are highlighted in red. Below we list the changes:

- In the mixed input step of program Dec, we consider the cases $\mu_1 = {\mu_1}^*$ and $\mu_1 \neq {\mu_1}^*$ separately. For the case $\mu_1 \neq {\mu_1}^*$ the code remains unchanged. For the case $\mu_1 = {\mu_1}^*$, the program uses $\mathsf{Dec}_{K_R}(r).\ell - 1$ instead of $\mathsf{Dec}_{K_R}(r).\ell$ when comparing to $\mathsf{Dec}_K(\mu_3).\ell - 1$.

Recall that the levels of the ciphertexts of the form $(m, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, 0, \hat{\rho})$ were not incremented in a table of K_R. However, we claim that, even though we have adjusted the code without adjusting the table of K_R for these cases, the distribution of the experiment doesn't change. Indeed, to force Dec to output non-$\perp$ via mixed input step, given some r which is an encryption of $(m, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, 0, \hat{\rho})$, one has to run it on inputs $(r, \mu_1{}^*, \mu_2{}^*, \mu_3)$ where μ_3 is itself an encryption of $\mu_1{}^*, \mu_2{}^*$. Consider the following three cases of μ_3 which encrypts $\mu_1{}^*, \mu_2{}^*$:

  * $\mu_3 = \mu_3{}^*$. In this case the trapdoor step would have been executed, and the program never reaches the mixed input step.
  * $\mu_3 = \overline{\mu_3{}^*}$. In this case the experiment aborts.
  * $\mu_3 \neq \mu_3{}^*$ and $\mu_3 \neq \overline{\mu_3{}^*}$. In this case the level of μ_3 is at least 2 (indeed, $\mu_3{}^*$ and $\overline{\mu_3{}^*}$ are the only two possible ciphetexts with level 0, and all other levels were incremented by 1, thus there are no level-1 ciphertexts). But if $\mathsf{Dec}_K(\mu_3).\ell \geq 2$, then the conditions $\mathsf{Dec}_{K_R}(r).\ell - 1 < \mathsf{Dec}_K(\mu_3).\ell - 1$ and $\mathsf{Dec}_{K_R}(r).\ell < \mathsf{Dec}_K(\mu_3).\ell - 1$ are equivalent, since $\mathsf{Dec}_{K_R}(r).\ell = 0$.

- In program RFake, we consider three cases: $\mu_3 = \mu_3{}^*, \overline{\mu_3{}^*}$, and else $\mu_1 = \mu_1{}^*$ and $\mu_1 \neq \mu_1{}^*$. For the case $\mu_1 \neq \mu_1{}^*$ and $\mu_3 = \mu_3{}^*, \overline{\mu_3{}^*}$ the code remains unchanged. For the case $\mu_1 = \mu_1{}^*$, the program encrypts the value $\mathsf{Dec}_K(\mu_3).\ell$ instead of $\mathsf{Dec}_K(\mu_3).\ell - 1$.

Due to adjusted code, this experiment is identical to the previous one.

- $H_{11,4}$: $\mathcal{A}^{\mathsf{P1}_2, \mathsf{P2}_3, \mathsf{P3}_2, \mathsf{Dec}_3, \mathsf{SFake}_2, \mathsf{RFake}_3}(m_0, m_1, s', r', \mu_1{}^*, \mu_2{}^*, \overline{\mu_3{}^*})$, where $\mu_1{}^*$ is chosen uniformly at random independently of H_1, $\mu_2{}^*$ is chosen uniformly at random independently of H_2, $\overline{\mu_3{}^*} = \mathsf{Enc}_K(m_1, \mu_1{}^*, \mu_2{}^*, 0)$, $s' = \mathsf{Enc}_{K_S}(m_0, \mu_1{}^*, \mu_2{}^*, \overline{\mu_3{}^*}, 1)$, $r' = \mathsf{Enc}_{K_R}(m_0, \mu_1{}^*, \mu_2{}^*, \overline{\mu_3{}^*}, 0, \hat{\rho}^*)$ for uniformly chosen $\hat{\rho}^*$. If $\mu_1{}^* \in \mathcal{H}_1$ or $\mu_2{}^* \in \mathcal{H}_2$ or $\hat{\rho}^* \in \mathcal{H}_3$, the experiment aborts. If the adversary queries any oracle on an input $s \in \mathcal{S}$ such that $\mathsf{Dec}_{K_S}(s).\mu_1 = \mu_1{}^*$ and $(\mathsf{Dec}_{K_S}(s).\ell = 0$ or $\mathsf{Dec}_{K_S}(s).\ell = T + 1$ or $\mathsf{Dec}_{K_S}(s).\ell = T)$, the experiment aborts. If the adversary queries any oracle on an input $r \in \mathcal{R}$ such that $\mathsf{Dec}_{K_R}(r).\mu_1 = \mu_1{}^*$ and $(\mathsf{Dec}_{K_R}(r).\ell = 0$ or $\mathsf{Dec}_{K_R}(R).\ell = T + 1)$, the experiment aborts. If the adversary queries any oracle on an input $\mu_3 \in \mathcal{M}$ such that $\mathsf{Dec}_K(\mu_3).\mu_1 = \mu_1{}^*$ and $(\mathsf{Dec}_K(\mu_3).\ell = 0$ or $\mathsf{Dec}_K(\mu_3).\ell = T + 1)$, the experiment aborts. If the adversary queries any oracle on any input containing $s \in \mathcal{S}_0 \cup \mathcal{S}'_1$, where $\mathcal{S}_0 = \left\{\mathsf{Enc}_{K_s}(\hat{m}, \mu_1, \mu_2, \mu_3, 0) : (m, \mu_1, \mu_2, \mu_3) \in \{0,1\}^{1+|\mu_1|+|\mu_2|+|\mu_3|}\right\}$, and $\mathcal{S}'_1 = \left\{\mathsf{Enc}_{K_s}(\hat{m}, \mu_1{}^*, \mu_2, \mu_3, 1) : (m, \mu_2, \mu_3) \in \{0,1\}^{1+|\mu_2|+|\mu_3|}\right\} \setminus s'$, the experiment aborts. If the adversary queries any oracle on any input containing $\mu_3{}^* = \mathsf{Enc}_K(m_0, \mu_1{}^*, \mu_2{}^*, 0)$ the experiment aborts.

This experiment is similar to the previous one, except that it aborts if the adversary queries any variable encrypting $\mu_1{}^*$ with level 0 or $T+1$. Informally, both happen with negligible probability because none of the programs ever output a value with $\mu_1{}^*$ and level 0, and because finding any value with level T requires the adversary to query SFake at least $T - 1$ times, which is infeasible for a polynomial-time adversary since T is superpolynomial.

More formally, note that the following is required for the adversary to find level-0 variables encrypting $\mu_1{}^*$: to output r with level 0 , the adversary needs to run RFake on μ_3 with level 0; to find μ_3 with level 0, it needs to run P3 on s with level 0 or $s \notin \mathcal{S}$. Since such μ_3 should have $\mu_1{}^*$ encrypted, and $\mu_1{}^* \notin \mathcal{H}_1$, there doesn't exist $s \notin \mathcal{S}$ which outputs such μ_3. Further, $s \in \mathcal{S}$ with level 0 is never an output of any program, and therefore the adversary can only guess any of these values, which happens with negligible probability.

Now we show that the adversary queries any variable with level T with at most negligible probability. Concretely, let ε be the sparseness of the sender-fake encryption scheme, i.e. $\varepsilon = \frac{|\mathcal{S}|}{2^{|s|}}$. We claim that the probability that any polynomial-time adversary queries the programs on $s \in \mathcal{S}$ with level $T+1$ or T is at most $T\varepsilon$, which we are going to show by proving that if the adversary makes n queries, the probability of asking a query containing $s \in \mathcal{S}$ with $\ell \in [n+1, T+1]$ is at most εn. In turn, $\varepsilon n < \varepsilon T$ since the number of queries is polynomial. We prove this statement by induction:

For the base case, note that for $n = 1$, the probability that the adversary queries $s \in \mathcal{S}$ with level $\ell \in [2, \ldots, T+1]$ is bounded by the probability of guessing $s \in \mathcal{S}$, which is equal to ε.

Assume the hypothesis holds for n. Assume the adversary makes $n+1$ queries and happens to ask a query containing $s \in \mathcal{S}$ with $\ell \in [n+1, T+1]$. Let's split this probability by considering the case when it did or did not query $s \in \mathcal{S}$ with level $\ell - 1$ within first n queries. The first event happens with probability at most εn by induction hypothesis, and the second happens with probability at most ε since the adversary can only guess such $s \in \mathcal{S}$. Thus the probability of the adversary succeeding with $n+1$ queries is at most $\varepsilon(n+1)$, thus proving induction hypothesis for $n+1$.

Finally, note that in order to query any μ_3 or r with level T or $T+1$, the adversary needs to either guess it or query its "parent" variable with level T or $T+1$ first (that is, s in case of μ_3, and μ_3 in case of r), and therefore the probability ad the adversary querying any variable with level T or $T+1$ is negligible.

- $H_{11,5}$: $\mathcal{A}^{\mathsf{P1,P2,P3,Dec,SFake,RFake}}(m_0, m_1, s', r', {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*})$, where ${\mu_1}^*$ is chosen uniformly at random independently of H_1, ${\mu_2}^*$ is chosen uniformly at random independently of H_2, $\overline{{\mu_3}^*} = \mathsf{Enc}_K(m_1, {\mu_1}^*, {\mu_2}^*, 0)$, $s' = \mathsf{Enc}_{K_S}(m_0, {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*}, 1)$, $r' = \mathsf{Enc}_{K_R}(m_0, {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*}, 0, \hat{\rho}^*)$ for uniformly chosen $\hat{\rho}^*$.If ${\mu_1}^* \in \mathcal{H}_1$ or ${\mu_2}^* \in \mathcal{H}_2$ or $\hat{\rho}^* \in \mathcal{H}_3$, the experiment aborts. If the adversary queries any oracle on an input $s \in \mathcal{S}$ such that $\mathsf{Dec}_{K_S}(s).\mu_1 = {\mu_1}^*$ and ($\mathsf{Dec}_{K_S}(s).\ell = 0$ or $\mathsf{Dec}_{K_S}(s).\ell = T+1$ or $\mathsf{Dec}_{K_S}(s).\ell = T$), the experiment aborts. If the adversary queries any oracle on an input $r \in \mathcal{R}$ such that $\mathsf{Dec}_{K_R}(r).\mu_1 = {\mu_1}^*$ and ($\mathsf{Dec}_{K_R}(r).\ell = 0$ or $\mathsf{Dec}_{K_R}(R).\ell = T+1$), the experiment aborts. If the adversary queries any oracle on an input $\mu_3 \in \mathcal{M}$ such that $\mathsf{Dec}_K(\mu_3).\mu_1 = {\mu_1}^*$ and ($\mathsf{Dec}_K(\mu_3).\ell = 0$ or $\mathsf{Dec}_K(\mu_3).\ell = T+1$), the experiment aborts. If the adversary queries any oracle on any input containing $s \in \mathcal{S}_0 \cup \mathcal{S}_1'$, where $\mathcal{S}_0 = \left\{\mathsf{Enc}_{K_s}(\hat{m}, \mu_1, \mu_2, \mu_3, 0) : (m, \mu_1, \mu_2, \mu_3) \in \{0,1\}^{1+|\mu_1|+|\mu_2|+|\mu_3|}\right\}$, and $\mathcal{S}_1' = \left\{\mathsf{Enc}_{K_s}(\hat{m}, {\mu_1}^*, \mu_2, \mu_3, 1) : (m, \mu_2, \mu_3) \in \{0,1\}^{1+|\mu_2|+|\mu_3|}\right\} \setminus s'$, the experiment aborts. If the adversary queries any oracle on any input containing ${\mu_3}^* = \mathsf{Enc}_K(m_0, {\mu_1}^*, {\mu_2}^*, 0)$ the experiment aborts.

  In this experiment we revert to using the original programs of the deniable encryption scheme, without any modifications.

  We argue that this experiment is identical to the previous one. Indeed, note that there are the following differences between the programs in this experiment and the previous experiment:

  - Some programs check that the level is in $[1, \ldots, T+1]$ instead of $[0, \ldots, T]$, for some variables encrypting ${\mu_1}^*$. Also, program SFake_2 checks that the level is in $[1, \ldots, T]$ instead of $[0, \ldots, T-1]$. This change in the programs doesn't change the distribution of the experiment, since the experiment aborts when the adversary queries variables with levels $0, T, T+1$ encrypting ${\mu_1}^*$.

  – Program Dec_3 checks the levels in the mixed input step by checking that $\mathsf{Dec}_{K_R}(r).\ell - 1 < \mathsf{Dec}_K(\mu_3).\ell - 1$, instead of $\mathsf{Dec}_{K_R}(r).\ell < \mathsf{Dec}_K(\mu_3).\ell$. The two conditions are equivalent and therefore this change doesn't affect the functionality of the program.

- $H_{11,6}$: $\mathcal{A}^{\mathsf{P1,P2,P3,Dec,SFake,RFake}}(m_0, m_1, s', r', {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*})$, where ${\mu_1}^*$ is chosen uniformly at random independently of H_1, ${\mu_2}^*$ is chosen uniformly at random independently of H_2, $\overline{{\mu_3}^*} = \mathsf{Enc}_K(m_1, {\mu_1}^*, {\mu_2}^*, 0)$, $s' = \mathsf{Enc}_{K_S}(m_0, {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*}, 1)$, $r' = \mathsf{Enc}_{K_R}(m_0, {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*}, 0, \hat{\rho}^*)$ for uniformly chosen $\hat{\rho}^*$.

  That is, we remove the condition that we abort when the adversary queries certain values. This is statistically close to the previous experiment, since the adversary can only query these values with negligible probability. The argument is similar to the argument made in previous hybrid distributions where these abort conditions were introduced.

- $H_{11,7}$: $\mathcal{A}^{\mathsf{P1,P2,P3,Dec,SFake,RFake}}(m_0, m_1, s', r', {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*})$, where s^*, r^* are chosen uniformly at random, ${\mu_1}^* = \mathsf{P1}(s^*, m_1)$, ${\mu_2}^* = \mathsf{P2}(r^*, {\mu_1}^*)$, $\overline{{\mu_3}^*} = \mathsf{Enc}_K(m_1, {\mu_1}^*, {\mu_2}^*, 0)$, $s' = \mathsf{Enc}_{K_S}(m_0, {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*}, 1)$, $r' = \mathsf{Enc}_{K_R}(m_0, {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*}, 0, H_3(\rho^*))$ for uniformly chosen ρ^*.

  That is, we set ${\mu_1}^*$ and ${\mu_2}^*$ to be their values in the protocol execution with m_1, and we set r' to use $H_3(\rho^*)$ instead of an independent random value as randomness.

  The reasoning is similar to the reasoning used in experiments H_1 to H_5 and H_8.

  Note that this experiment corresponds to an execution on input m_1, where s', r' are both fake and consistent with m_0.

5 Preliminaries: IO, DIO, and ACE

5.1 Indistinguishability Obfuscation for Circuits

Definition 5 (Indistinguishability Obfuscation (iO))**.** *A uniform PPT machine* iO *is called an* indistinguishability obfuscator *if the following conditions are satisfied:*

- *For all security parameters $\lambda \in \mathbb{N}$, for all $C \in \mathcal{C}_\lambda$, for all inputs x, we have that*

$$\Pr[C'(x) = C(x) : C' \leftarrow \mathsf{iO}(1^\lambda, C)] = 1$$

- *There is a polynomial p such that for every circuit $C \in \mathcal{C}_\lambda$, it holds that $|\mathsf{iO}(c)| \le p(|C|)$.*

- *For any (not necessarily uniform) PPT distinguisher D, there exists a negligible function α such that the following holds: For all security parameters $\lambda \in \mathbb{N}$, for all circuit families $C_0 = \{C_\lambda^0\}_{\lambda \in \mathbb{N}}, C_1 = \{C_\lambda^1\}_{\lambda \in \mathbb{N}}$ of size $|C_\lambda^0| = |C_\lambda^1|$, we have that if $C_\lambda^0(x) = C_\lambda^1(x)$ for all inputs x, then*

$$\Big|\Pr\big[D(\mathsf{iO}(1^\lambda, C_\lambda^0)) = 1\big] - \Pr\big[D(\mathsf{iO}(1^\lambda, C_\lambda^1)) = 1\big]\Big| \le \mathsf{negl}(\lambda).$$

We say that indistinguishability obfuscation is $(t(\lambda), \varepsilon(\lambda))$-secure if the distinguishing advantage of all distinguishers of size $t(\lambda)$ is at most $\varepsilon(\lambda)$.

Oracles $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$**.**

Oracle $\mathsf{P1}(s, m)$
Inputs: sender randomness s, plaintext m.
Hardwired values: key K_S of sender-fake encryption scheme, hash H_1 with sparse image.

1. **Trapdoor step:**
   (a) If $s \in \mathcal{S}$ and $\mathsf{Dec}_{K_S}(s).m = m$, then return $\mathsf{Dec}_{K_S}(s).\mu_1$;
2. **Main step:**
   (a) Else return $H_1(s, m)$.

Oracle $\mathsf{P3}(s, m, \mu_1, \mu_2)$
Inputs: sender randomness s, plaintext m, the first and the second messages μ_1, μ_2 in the protocol.
Hardwired values: key K_S of sender-fake encryption scheme, key K of main encryption scheme.

1. **Validity check:**
   (a) If $\mathsf{P1}(s, m) \neq \mu_1$ then $\perp$;
2. **Trapdoor step:**
   (a) If $s \in \mathcal{S}$ and $(\mathsf{Dec}_{K_S}(s).m, \mathsf{Dec}_{K_S}(s).\mu_1, \mathsf{Dec}_{K_S}(s).\mu_2) = (m, \mu_1, \mu_2)$ then return $\mathsf{Dec}_{K_S}(s).\mu_3$;
3. **Mixed input step:**
   (a) Else if $s \in \mathcal{S}$ and $(\mathsf{Dec}_{K_S}(s).m, \mathsf{Dec}_{K_S}(s).\mu_1) = (m, \mu_1)$ then
      i. If $\mathsf{Dec}_{K_S}(s).\ell \notin [0, \ldots, T]$ then $\perp$
      ii. Else return $\mathsf{Enc}_K(m, \mu_1, \mu_2, \mathsf{Dec}_{K_S}(s).\ell)$
4. **Main step:**
   (a) Else return $\mathsf{Enc}_K(m, \mu_1, \mu_2, 0)$.

Oracle $\mathsf{SFake}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$
Inputs: sender randomness s, real plaintext m, fake plaintext $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.
Hardwired values: key K_S of sender-fake encryption scheme, upper bound T.

1. **Validity check:**
   (a) If $\mathsf{P1}(s, m) \neq \mu_1$ then $\perp$;
2. **Trapdoor step:**
   (a) If $s \in \mathcal{S}$ and $(\mathsf{Dec}_{K_S}(s).m, \mathsf{Dec}_{K_S}(s).\mu_1) = (m, \mu_1)$ then
      i. If $\mathsf{Dec}_{K_S}(s).\ell \notin [0, \ldots, T-1]$ then $\perp$
      ii. Else return $\mathsf{Enc}_{K_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \mathsf{Dec}_{K_S}(s).\ell + 1)$.
3. **Main step:**
   (a) Else return $\mathsf{Enc}_{K_S}(\hat{m}, \mu_1, \mu_2, \mu_3, 1)$.

Figure 11: Oracles $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$. The code of the programs is unchanged, but we make the bounds on ℓ explicit in relevant places; we highlight them in red for convenience.

Oracles $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$**.**

Oracle $\mathsf{P2}(r, \mu_1)$

Inputs: receiver randomness r, the first message μ_1 in the protocol.

Hardwired values: key K_R of receiver-fake encryption scheme, hash H_2 with sparse image.

1. **Trapdoor step:**
   (a) If $r \in \mathcal{R}$ and $\mathsf{Dec}_{K_R}(r).\mu_1 = \mu_1$, then return $\mathsf{Dec}_{K_R}(r).\mu_2$;
2. **Main step:**
   (a) Return $H_2(r, \mu_1)$.

Oracle $\mathsf{Dec}(r, \mu_1, \mu_2, \mu_3)$

Inputs: receiver randomness r, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: key K_R of receiver-fake encryption scheme, key K of the main encryption scheme, upper bound T.

1. **Validity check:**
   (a) If $\mathsf{P2}(r, \mu_1) \neq \mu_2$ then $\perp$;
2. **Trapdoor step:**
   (a) If $r \in \mathcal{R}$ and $(\mathsf{Dec}_{K_R}(r).\mu_1, \mathsf{Dec}_{K_R}(r).\mu_2, \mathsf{Dec}_{K_R}(r).\mu_3) = (\mu_1, \mu_2, \mu_3)$ then return $\mathsf{Dec}_{K_R}(r).m$;
3. **Mixed input step:**
   (a) If $r \in \mathcal{R}$ and $(\mathsf{Dec}_{K_R}(r).\mu_1, \mathsf{Dec}_{K_R}(r).\mu_2) = (\mu_1, \mu_2)$ then
      i. If $\mu_3 \in \mathcal{M}$ and $(\mu_1, \mu_2) = (\mathsf{Dec}_K(\mu_3).\mu_1, \mathsf{Dec}_K(\mu_3).\mu_2)$ and $\mathsf{Dec}_{K_R}(r).\ell < \mathsf{Dec}_K(\mu_3).\ell$ and $\mathsf{Dec}_{K_R}(r).\ell \in [0, \ldots, T]$ and $\mathsf{Dec}_K(\mu_3).\ell \in [0, \ldots, T]$ return $\mathsf{Dec}_K(\mu_3).m$;
      ii. Else $\perp$.
4. **Main step:**
   (a) If $\mu_3 \in \mathcal{M}$ and $(\mathsf{Dec}_K(\mu_3).\mu_1, \mathsf{Dec}_K(\mu_3).\mu_2) = (\mu_1, \mu_2)$ then return $\mathsf{Dec}_K(\mu_3).m$;
   (b) Else $\perp$.

Oracle $\mathsf{RFake}(\hat{m}, \mu_1, \mu_2, \mu_3; \rho)$

Inputs: fake plaintext $\hat{m}$, protocol transcript μ_1, μ_2, μ_3, random coins ρ.

Hardwired values: key K_R of receiver-fake encryption scheme, key K of the main encryption scheme, function H_3 with a sparse image.

1. If $\mu_3 \in \mathcal{M}$ and $\mathsf{Dec}_K(\mu_3).\ell \in [0, \ldots, T]$ and $\mathsf{Dec}_K(\mu_3).\mu_1 = \mu_1$ and $\mathsf{Dec}_K(\mu_3).\mu_2 = \mu_2$ then return $\mathsf{Enc}_{K_R}(\hat{m}, \mu_1, \mu_2, \mu_3, \mathsf{Dec}_K(\mu_3).\ell, H_3(\rho))$;
2. Else $\perp$.

Figure 12: Oracles $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$. The code of the programs is unchanged, but we make the bounds on ℓ explicit in relevant places; we highlight them in red for convenience.

Oracles $\mathsf{P1}_1, \mathsf{P3}_1, \mathsf{SFake}_1$.

Oracle $\mathsf{P1}_1(s, m)$

Inputs: sender randomness s, plaintext m.

Hardwired values: key K_S of sender-fake encryption scheme, hash H_1 with sparse image.

1. **Trapdoor step:**
   (a) If $s \in \mathcal{S}$ and $\mathsf{Dec}_{K_S}(s).m = m$, then return $\mathsf{Dec}_{K_S}(s).\mu_1$;
2. **Main step:**
   (a) Else return $H_1(s, m)$.

Oracle $\mathsf{P3}_1(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, plaintext m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: key K_S of sender-fake encryption scheme, key K of main encryption scheme.

1. **Validity check:**
   (a) If $\mathsf{P1}_1(s, m) \neq \mu_1$ then $\perp$;
2. **Trapdoor step:**
   (a) If $s \in \mathcal{S}$ and $(\mathsf{Dec}_{K_S}(s).m, \mathsf{Dec}_{K_S}(s).\mu_1, \mathsf{Dec}_{K_S}(s).\mu_2) = (m, \mu_1, \mu_2)$ then return $\mathsf{Dec}_{K_S}(s).\mu_3$;
3. **Mixed input step:**
   (a) Else if $s \in \mathcal{S}$ and $(\mathsf{Dec}_{K_S}(s).m, \mathsf{Dec}_{K_S}(s).\mu_1) = (m, \mu_1)$ and $\mu_1 = {\mu_1}^*$ then
      i. If $\mathsf{Dec}_{K_S}(s).\ell \notin [1, \ldots, T+1]$ then $\perp$
      ii. Else return $\mathsf{Enc}_K(m, \mu_1, \mu_2, \mathsf{Dec}_{K_S}(s).\ell - 1)$;
   (b) Else if $s \in \mathcal{S}$ and $(\mathsf{Dec}_{K_S}(s).m, \mathsf{Dec}_{K_S}(s).\mu_1) = (m, \mu_1)$ and then
      i. If $\mathsf{Dec}_{K_S}(s).\ell \notin [0, \ldots, T]$ then $\perp$
      ii. Else return $\mathsf{Enc}_K(m, \mu_1, \mu_2, \mathsf{Dec}_{K_S}(s).\ell)$;
4. **Main step:**
   (a) Else return $\mathsf{Enc}_K(m, \mu_1, \mu_2, 0)$.

Oracle $\mathsf{SFake}_1(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real plaintext m, fake plaintext $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: key K_S of sender-fake encryption scheme, upper bound T.

1. **Validity check:**
   (a) If $\mathsf{P1}_1(s, m) \neq \mu_1$ then $\perp$;
2. **Trapdoor step:**
   (a) If $s \in \mathcal{S}$ and $(\mathsf{Dec}_{K_S}(s).m, \mathsf{Dec}_{K_S}(s).\mu_1) = (m, \mu_1)$ and $\mu_1 = {\mu_1}^*$ then
      i. If $\mathsf{Dec}_{K_S}(s).\ell \notin [1, \ldots, T]$ then $\perp$
      ii. Else return $\mathsf{Enc}_{K_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \mathsf{Dec}_{K_S}(s).\ell + 1)$.
   (b) Else if $s \in \mathcal{S}$ and $(\mathsf{Dec}_{K_S}(s).m, \mathsf{Dec}_{K_S}(s).\mu_1) = (m, \mu_1)$ then
      i. If $\mathsf{Dec}_{K_S}(s).\ell \notin [0, \ldots, T-1]$ then $\perp$
      ii. Else return $\mathsf{Enc}_{K_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \mathsf{Dec}_{K_S}(s).\ell + 1)$.
3. **Main step:**
   (a) Else return $\mathsf{Enc}_{K_S}(\hat{m}, \mu_1, \mu_2, \mu_3, 1)$.

Figure 13: Oracles $\mathsf{P1}_1, \mathsf{P3}_1, \mathsf{SFake}_1$.

Oracles $\mathsf{P1}_2, \mathsf{P3}_2, \mathsf{SFake}_2$**.**

Oracle $\mathsf{P1}_2(s, m)$
Inputs: sender randomness s, plaintext m.
Hardwired values: key K_S of sender-fake encryption scheme, hash H_1 with sparse image.

1. **Trapdoor step:**
   (a) If $s \in \mathcal{S}$ and $\mathsf{Dec}_{K_S}(s).m = m$, then return $\mathsf{Dec}_{K_S}(s).\mu_1$;
2. **Main step:**
   (a) Else return $H_1(s, m)$.

Oracle $\mathsf{P3}_2(s, m, \mu_1, \mu_2)$
Inputs: sender randomness s, plaintext m, the first and the second messages μ_1, μ_2 in the protocol.
Hardwired values: key K_S of sender-fake encryption scheme, key K of main encryption scheme.

1. **Validity check:**
   (a) If $\mathsf{P1}_2(s, m) \neq \mu_1$ then $\perp$;
2. **Trapdoor step:**
   (a) If $s \in \mathcal{S}$ and $(\mathsf{Dec}_{K_S}(s).m, \mathsf{Dec}_{K_S}(s).\mu_1, \mathsf{Dec}_{K_S}(s).\mu_2) = (m, \mu_1, \mu_2)$ then return $\mathsf{Dec}_{K_S}(s).\mu_3$;
3. **Mixed input step:**
   (a) Else if $s \in \mathcal{S}$ and $(\mathsf{Dec}_{K_S}(s).m, \mathsf{Dec}_{K_S}(s).\mu_1) = (m, \mu_1)$ and $\mu_1 = \mu_1{}^*$ then
      i. If $\mathsf{Dec}_{K_S}(s).\ell \notin [1, \ldots, T+1]$ then $\perp$
      ii. Else return $\mathsf{Enc}_K(m, \mu_1, \mu_2, \mathsf{Dec}_{K_S}(s).\ell)$;
   (b) Else if $s \in \mathcal{S}$ and $(\mathsf{Dec}_{K_S}(s).m, \mathsf{Dec}_{K_S}(s).\mu_1) = (m, \mu_1)$ then
      i. If $\mathsf{Dec}_{K_S}(s).\ell \notin [0, \ldots, T]$ then $\perp$
      ii. Else return $\mathsf{Enc}_K(m, \mu_1, \mu_2, \mathsf{Dec}_{K_S}(s).\ell)$;
4. **Main step:**
   (a) Else return $\mathsf{Enc}_K(m, \mu_1, \mu_2, 0)$.

Oracle $\mathsf{SFake}_2(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$
Inputs: sender randomness s, real plaintext m, fake plaintext $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.
Hardwired values: key K_S of sender-fake encryption scheme, upper bound T.

1. **Validity check:**
   (a) If $\mathsf{P1}_2(s, m) \neq \mu_1$ then $\perp$;
2. **Trapdoor step:**
   (a) If $s \in \mathcal{S}$ and $(\mathsf{Dec}_{K_S}(s).m, \mathsf{Dec}_{K_S}(s).\mu_1) = (m, \mu_1)$ and $\mu_1 = \mu_1{}^*$ then
      i. If $\mathsf{Dec}_{K_S}(s).\ell = 0$ or $\mathsf{Dec}_{K_S}(s).\ell = T+1$ then $\perp$;
      ii. Else return $\mathsf{Enc}_{K_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \mathsf{Dec}_{K_S}(s).\ell + 1)$.
   (b) Else if $s \in \mathcal{S}$ and $(\mathsf{Dec}_{K_S}(s).m, \mathsf{Dec}_{K_S}(s).\mu_1) = (m, \mu_1)$ then
      i. If $\mathsf{Dec}_{K_S}(s).\ell = T$ then $\perp$;
      ii. Else return $\mathsf{Enc}_{K_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \mathsf{Dec}_{K_S}(s).\ell + 1)$.
3. **Main step:**
   (a) Else return $\mathsf{Enc}_{K_S}(\hat{m}, \mu_1, \mu_2, \mu_3, 1)$.

Figure 14: Oracles $\mathsf{P1}_2, \mathsf{P3}_2, \mathsf{SFake}_2$.

Oracles $\mathsf{P2}_2, \mathsf{Dec}_2, \mathsf{RFake}_2$**.**

Oracle $\mathsf{P2}_2(r, \mu_1)$

Inputs: receiver randomness r, the first message μ_1 in the protocol.

Hardwired values: key K_R of receiver-fake encryption scheme, sparse hash H_2.

1. **Trapdoor step:**
   (a) If $r \in \mathcal{R}$ and $\mathsf{Dec}_{K_R}(r).\mu_1 = \mu_1$, then return $\mathsf{Dec}_{K_R}(r).\mu_2$;
2. **Main step:**
   (a) Return $H_2(r, \mu_1)$.

Oracle $\mathsf{Dec}_2(r, \mu_1, \mu_2, \mu_3)$

Inputs: receiver randomness r, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: key K_R of receiver-fake encryption scheme, key K of the main encryption scheme, upper bound T.

1. **Validity check:**
   (a) If $\mathsf{P2}_2(r, \mu_1) \neq \mu_2$ then $\perp$;
2. **Trapdoor step:**
   (a) If $r \in \mathcal{R}$ and $(\mathsf{Dec}_{K_R}(r).\mu_1, \mathsf{Dec}_{K_R}(r).\mu_2, \mathsf{Dec}_{K_R}(r).\mu_3) = (\mu_1, \mu_2, \mu_3)$ then return $\mathsf{Dec}_{K_R}(r).m$;
3. **Mixed input step:**
   (a) Else if $r \in \mathcal{R}$ and $(\mathsf{Dec}_{K_R}(r).\mu_1, \mathsf{Dec}_{K_R}(r).\mu_2) = (\mu_1, \mu_2)$ then
      i. If $\mu_3 \in \mathcal{M}$ and $(\mu_1, \mu_2) = (\mathsf{Dec}_K(\mu_3).\mu_1, \mathsf{Dec}_K(\mu_3).\mu_2)$and $\mathsf{Dec}_K(\mu_3).\mu_1 = {\mu_1}^*$ and $\mathsf{Dec}_{K_R}(r).\ell < \mathsf{Dec}_K(\mu_3).\ell - 1$ and $\mathsf{Dec}_{K_R}(r).\ell \in [0, \ldots, T]$ and $\mathsf{Dec}_K(\mu_3).\ell \in [1, \ldots, T+1]$ then return $\mathsf{Dec}_K(\mu_3).m$;
      ii. Else if $\mu_3 \in \mathcal{M}$ and $(\mu_1, \mu_2) = (\mathsf{Dec}_K(\mu_3).\mu_1, \mathsf{Dec}_K(\mu_3).\mu_2)$ and $\mathsf{Dec}_{K_R}(r).\ell < \mathsf{Dec}_K(\mu_3).\ell$ and $\mathsf{Dec}_{K_R}(r).\ell \in [0, \ldots, T]$ and $\mathsf{Dec}_K(\mu_3).\ell \in [0, \ldots, T]$ then return $\mathsf{Dec}_K(\mu_3).m$;
      iii. Else $\perp$.
4. **Main step:**
   (a) Else if $\mu_3 \in \mathcal{M}$ and $(\mathsf{Dec}_K(\mu_3).\mu_1, \mathsf{Dec}_K(\mu_3).\mu_2) = (\mu_1, \mu_2)$ then
      i. Else return $\mathsf{Dec}_K(\mu_3).m$;
   (b) Else $\perp$.

Oracle $\mathsf{RFake}_2(\hat{m}, \mu_1, \mu_2, \mu_3; \rho)$

Inputs: fake plaintext $\hat{m}$, protocol transcript μ_1, μ_2, μ_3, random coins ρ.

Hardwired values: key K_R of receiver-fake encryption scheme, key K of the main encryption scheme, function H_3 with a sparse image.

1. If $\mu_3 = {\mu_3}^*$ or $\mu_3 = \overline{{\mu_3}^*}$ then
   (a) $(\mu_1, \mu_2) = ({\mu_1}^*, {\mu_2}^*)$ then return $\mathsf{Enc}_{K_R}(\hat{m}, \mu_1, \mu_2, \mu_3, 0, H_3(\rho))$;
   (b) Else $\perp$;
2. Else if $\mu_3 \in \mathcal{M}$ and $\mathsf{Dec}_K(\mu_3).\mu_1 = {\mu_1}^*$ and $\mathsf{Dec}_K(\mu_3).\ell \in [1, \ldots, T+1]$ and $\mathsf{Dec}_K(\mu_3).\mu_1 = \mu_1$ and $\mathsf{Dec}_K(\mu_3).\mu_2 = \mu_2$ then return $\mathsf{Enc}_{K_R}(\hat{m}, \mu_1, \mu_2, \mu_3, \mathsf{Dec}_K(\mu_3).\ell - 1, H_3(\rho))$;
3. Else if $\mu_3 \in \mathcal{M}$ and $\mathsf{Dec}_K(\mu_3).\ell \in [0, \ldots, T]$ and $\mathsf{Dec}_K(\mu_3).\mu_1 = \mu_1$ and $\mathsf{Dec}_K(\mu_3).\mu_2 = \mu_2$ then return $\mathsf{Enc}_{K_R}(\hat{m}, \mu_1, \mu_2, \mu_3, \mathsf{Dec}_K(\mu_3).\ell, H_3(\rho))$;
4. Else $\perp$.

Figure 15: Oracles $\mathsf{P2}_2, \mathsf{Dec}_2, \mathsf{RFake}_2$.

Oracles $\mathsf{P2}_3, \mathsf{Dec}_3, \mathsf{RFake}_3$**.**

Oracle $\mathsf{P2}_3(r, \mu_1)$

Inputs: receiver randomness r, the first message μ_1 in the protocol.

Hardwired values: key K_R of receiver-fake encryption scheme, sparse hash H_2.

1. **Trapdoor step:**
   (a) If $r \in \mathcal{R}$ and $\mathsf{Dec}_{K_R}(r).\mu_1 = \mu_1$, then return $\mathsf{Dec}_{K_R}(r).\mu_2$;
2. **Main step:**
   (a) Return $H_2(r, \mu_1)$.

Oracle $\mathsf{Dec}_3(r, \mu_1, \mu_2, \mu_3)$

Inputs: receiver randomness r, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: key K_R of receiver-fake encryption scheme, key K of the main encryption scheme, upper bound T.

1. **Validity check:**
   (a) If $\mathsf{P2}_3(r, \mu_1) \neq \mu_2$ then $\perp$;
2. **Trapdoor step:**
   (a) If $r \in \mathcal{R}$ and $(\mathsf{Dec}_{K_R}(r).\mu_1, \mathsf{Dec}_{K_R}(r).\mu_2, \mathsf{Dec}_{K_R}(r).\mu_3) = (\mu_1, \mu_2, \mu_3)$ then return $\mathsf{Dec}_{K_R}(r).m$;
3. **Mixed input step:**
   (a) Else if $r \in \mathcal{R}$ and $(\mathsf{Dec}_{K_R}(r).\mu_1, \mathsf{Dec}_{K_R}(r).\mu_2) = (\mu_1, \mu_2)$ then
      i. If $\mu_3 \in \mathcal{M}$ and $(\mu_1, \mu_2) = (\mathsf{Dec}_K(\mu_3).\mu_1, \mathsf{Dec}_K(\mu_3).\mu_2)$and $\mathsf{Dec}_K(\mu_3).\mu_1 = {\mu_1}^*$ and $\mathsf{Dec}_{K_R}(r).\ell - 1 < \mathsf{Dec}_K(\mu_3).\ell - 1$ and $\mathsf{Dec}_{K_R}(r).\ell \in [1, \ldots, T+1]$ and $\mathsf{Dec}_K(\mu_3).\ell \in [1, \ldots, T+1]$ then return $\mathsf{Dec}_K(\mu_3).m$;
      ii. Else if $\mu_3 \in \mathcal{M}$ and $(\mu_1, \mu_2) = (\mathsf{Dec}_K(\mu_3).\mu_1, \mathsf{Dec}_K(\mu_3).\mu_2)$ and $\mathsf{Dec}_{K_R}(r).\ell < \mathsf{Dec}_K(\mu_3).\ell$ and $\mathsf{Dec}_{K_R}(r).\ell \in [0, \ldots, T]$ and $\mathsf{Dec}_K(\mu_3).\ell \in [0, \ldots, T]$ then return $\mathsf{Dec}_K(\mu_3).m$;
      iii. Else $\perp$.
4. **Main step:**
   (a) Else if $\mu_3 \in \mathcal{M}$ and $(\mathsf{Dec}_K(\mu_3).\mu_1, \mathsf{Dec}_K(\mu_3).\mu_2) = (\mu_1, \mu_2)$ then
      i. Else return $\mathsf{Dec}_K(\mu_3).m$;
   (b) Else $\perp$.

Oracle $\mathsf{RFake}_3(\hat{m}, \mu_1, \mu_2, \mu_3; \rho)$

Inputs: fake plaintext $\hat{m}$, protocol transcript μ_1, μ_2, μ_3, random coins ρ.

Hardwired values: key K_R of receiver-fake encryption scheme, key K of the main encryption scheme, function H_3 with a sparse image.

1. If $\mu_3 = {\mu_3}^*$ or $\mu_3 = \overline{{\mu_3}^*}$ then
   (a) $(\mu_1, \mu_2) = ({\mu_1}^*, {\mu_2}^*)$ then return $\mathsf{Enc}_{K_R}(\hat{m}, \mu_1, \mu_2, \mu_3, 0, H_3(\rho))$;
   (b) Else $\perp$;
2. If $\mu_3 \in \mathcal{M}$ and $\mathsf{Dec}_K(\mu_3).\mu_1 = {\mu_1}^*$ and $\mathsf{Dec}_K(\mu_3).\ell \in [1, \ldots, T+1]$ and $\mathsf{Dec}_K(\mu_3).\mu_1 = \mu_1$ and $\mathsf{Dec}_K(\mu_3).\mu_2 = \mu_2$ then return $\mathsf{Enc}_{K_R}(\hat{m}, \mu_1, \mu_2, \mu_3, \mathsf{Dec}_K(\mu_3).\ell, H_3(\rho))$;
3. Else if $\mu_3 \in \mathcal{M}$ and $\mathsf{Dec}_K(\mu_3).\ell \in [0, \ldots, T]$ and $\mathsf{Dec}_K(\mu_3).\mu_1 = \mu_1$ and $\mathsf{Dec}_K(\mu_3).\mu_2 = \mu_2$ then return $\mathsf{Enc}_{K_R}(\hat{m}, \mu_1, \mu_2, \mu_3, \mathsf{Dec}_K(\mu_3).\ell, H_3(\rho))$;
4. Else $\perp$.

Figure 16: Oracles $\mathsf{P2}_3, \mathsf{Dec}_3, \mathsf{RFake}_3$.

5.2 Equivalence of iO and diO for programs differing on one point

In the proof of security of the level system we use the following lemma from [BPR15] (which is a special case of theorem 6.2 from [BCP14], with exact parameters):

Lemma 1. *([BPR15, BCP14]) Let* iO *be a* (t, δ)*-secure indistinguishability obfuscator for P/poly. There exists a PPT oracle-aided extractor* E*, such that for any* $t^{O(1)}$*-size distinguisher* D*, and two equal-size circuits* C_0, C_1 *differing on exactly one input* x^**, the following holds. Let* C'_0, C'_1 *be padded versions of* C_0, C_1 *of size* $s \geq 3 \cdot |C_0|$*. If* $|\mathsf{Pr}[D(\mathsf{iO}(C'_0)) = 1] - \mathsf{Pr}[D(\mathsf{iO}(C'_1)) = 1]| = \eta \geq \delta(s)^{o(1)}$*, then* $\mathsf{Pr}[x^* \leftarrow E^{D(\cdot)}(1^{1/\eta}, C_0, C_1)] \geq 1 - 2^{-\Omega(s)}$.

5.3 Puncturable Pseudorandom Functions and their variants

Puncturable PRFs. In puncrurable PRFs it is possible to create a key that is punctured at a set S of polynomial size. A key k punctured at S (denoted $k\{S\}$) allows evaluating the PRF at all points not in S. Furthermore, the function values at points in S remain pseudorandom even given $k\{S\}$.

Definition 6. *A puncturable pseudorandom function family for input size* $n(\lambda)$ *and output size* $m(\lambda)$ *is a tuple of algorithms* $\{\mathsf{Sample}, \mathsf{Puncture}, \mathsf{Eval}\}$ *such that the following properties hold:*

- **Functionality preserved under puncturing:** *For any PPT adversary A which outputs a set* $S \subset \{0,1\}^n$*, for any* $x \notin S$*,*

$$\mathsf{Pr}[F_k(x) = F_{k\{S\}}(x) : k \leftarrow \mathsf{Sample}(1^\lambda), k\{S\} \leftarrow \mathsf{Puncture}(k, S)] = 1.$$

- **Pseudorandomness at punctured points:** *For any PPT adversaries* A_1, A_2*, define a set* S *and state* state *as* $(S, \mathsf{state}) \leftarrow A_1(1^\lambda)$*. Then*

$$\mathsf{Pr}[A_2(\mathsf{state}, S, k\{S\}, F_k(S))] - \mathsf{Pr}[A_2(\mathsf{state}, S, k\{S\}, U_{|S|\cdot m(\lambda)})] < \mathsf{negl}(\lambda),$$

  where $F_k(S)$ *denotes concatenated PRF values on inputs from S, i.e.* $F_k(S) = \{F_k(x_i) : x_i \in S\}$*.*

The GGM PRF [GGM84] satisfies this definition.

Statistically injective puncturable PRFs. Such PRFs are injective with overwhelming probability over the choice of a key. Sahai and Waters [SW14] show that if F is a puncturable PRF with arbitrary input length n and output length $m \geq 2n + \lambda$, and h is 2-universal hash function, then $\mathsf{F}'_{k,h} = \mathsf{F}_k(x) \oplus h(x)$ is a statistically injective puncturable PRF with probability $1 - 2^{-\lambda}$ over the choice of a key.

Extracting puncturable PRFs. Such PRFs have a property of a strong extractor: even when a full key is known, the output of the PRF is statistically close to uniform, as long as there is enough min-entropy in the input. Sahai and Waters [SW14] show that if the input has min-entropy at least $m + 2\lambda + 2$ (where m is the output size), then such PRF can be constructed from any puncturable PRF F as $\mathsf{F}'_{k,h} = h(\mathsf{F}_k(x))$, where h is 2-universal hash function; it can be shown that the output of this PRF together with the key is $2^{-\lambda}$-close to the uniform distribution.

Sparse computationally extracting puncturable PRFs. We need a slightly modified version of extracting PRFs: we relax the extracting requirement from statistical to computational, but require our PRF to have a sparse image. Such a PRF can be built from computationally extracting PRF by applying a PRF on top of it [CPR17].

Definition 7. *A PRF family with a key k mapping $\{0,1\}^{n(\lambda)}$ to $\{0,1\}^{l(\lambda)}$ is a sparse computationally extracting family for min-entropy $t(\lambda)$, if, in addition to the standard definition of a puncturable PRF, the following two conditions hold:*

- **Sparseness:** $\mathsf{Pr}[r \in Im(\mathsf{F}_k) : k \leftarrow \mathsf{Sample}(1^\lambda), r \leftarrow U_l] < \nu(\lambda)$ *for some negligible function ν;*
- **Computational extractor:** *If distribution X has min-entropy at least $t(\lambda)$, then with overwhelming probability over the choice of key k for any PPT adversary $\mathcal{A}$*

$$|\ \mathsf{Pr}\left[\ \mathcal{A}(k, \mathsf{F}_k(x)) = 1 \mid x \leftarrow X\ \right] - \mathsf{Pr}\left[\ \mathcal{A}(k, r) = 1 \mid r \leftarrow U_I\ \right]\ | < \mathsf{negl}(\lambda).$$

We say that such a PRF is $(t(\lambda), \varepsilon(\lambda))$-secure, if for any t-sized distinguishers the distinguishing advantage in the puncturable PRF game and in the computational extractor game is at most ε, and sparseness $\nu(\lambda) < \varepsilon(\lambda)$.

[CPR17] show that, assuming one-way functions, such PRFs exist if $t(\lambda)$, the entropy of the input, is at least $m/2 + 2\lambda + 2$, and m is superlogarithmic. Their construction uses a PRF with security parameter λ and a PRG with security parameter $m/2$ and therefore the construction can be made exponentially secure, by requiring (possibly stronger) subexponential security of the underlying PRF and PRG.

5.4 Asymmetrically constrained encryption (ACE) and its relaxed variant

ACE at a high level. Asymmetrically constrained encryption ([CHJV14], see also the journal version [BCG+18]), or ACE for short, is a public-key, *deterministic* encryption scheme with special security properties. Intuitively, it allows to puncture both the public key and the secret key, at possibly different sets, such that $\mathsf{EK}\{m\}$ doesn't allow to compute the encryption of m, and $\mathsf{DK}\{m\}$ doesn't allow to decrypt the encryption of m. The scheme has to satisfy the following security properties, which we only roughly outline in this paragraph (see the formal definition below for precise correctness and security requirements):

- **Indistinguishability of ciphertexts:** $\mathsf{Enc}_{\mathsf{EK}}(m_0)$ and $\mathsf{Enc}_{\mathsf{EK}}(m_1)$ are indistinguishable even given punctured $\mathsf{EK}\{m_0, m_1\}$, $\mathsf{DK}\{m_0, m_1\}$ (or given EK, DK punctured at bigger sets including m_0, m_1). Intuitively, the adversary can neither encrypt m_0, m_1 nor decrypt $\mathsf{Enc}_{\mathsf{EK}}(m_0)$ and $\mathsf{Enc}_{\mathsf{EK}}(m_1)$, and thus cannot distinguish between encryptions of m_0, m_1.
- **Security of constrained decryption:** Given $\mathsf{EK}\{U\}$, it is hard to distinguish between $\mathsf{DK}\{S_0\}$ and $\mathsf{DK}\{S_1\}$, where $S_0 \subseteq S_1 \subseteq U$. Intuitively, the adversary cannot distinguish between these two cases since it is hard to find a "differing ciphertext" $\mathsf{Enc}_{\mathsf{EK}}(m)$, $m \in S_1 \setminus S_0$, which $\mathsf{DK}\{S_0\}$ and $\mathsf{DK}\{S_1\}$ decrypt differently (to m and $\perp$). Such ciphertexts are hard to find since such $m \in U$, and EK is punctured at U.

Relaxed ACE at a high level. In addition to ACE, we require a slightly different version of it, which we call a *relaxed ACE*. Relaxed ACE does not require indistinguishability of ciphertexts, but instead requires a different property called *symmetry*. We show how to modify the construction of [CHJV14] to build relaxed ACE with small security loss in constrained decryption game for certain sets. More concretely, we have the following differences:

- In [CHJV14], security of constrained decryption allows for security loss proportional to the size of $S_1 \setminus S_0$, since they change $\mathsf{DK}\{S_0\}$ to $\mathsf{DK}\{S_1\}$, one point at a time. This is too much in our case, since our sets have size $2^{O(\lambda)}$. However, our sets have nice structure (e.g. all strings ending with the same suffix, or all such strings except one), and we can slightly modify the construction such that security loss is only polynomial on such sets. Essentially, our ciphertexts, instead of having a single signature of a plaintext like in [CHJV14], have signatures of each prefix of the plaintext, which allows to puncture DK at a lot of points at once (this technique is similar to [GPS16]).

- We require additional property which we call *symmetry*. To define it, we first need a syntactically different way of puncturing the decryption key. In [CHJV14] puncturing is plaintext-based (i.e. the punctured key $\mathsf{DK}\{m\}$ has the description of the plaintext but not the ciphertext). We need, in addition to that, a ciphertext-based way to puncture (we denote it as $\mathsf{DK}\{c\}$). Symmetry then says that distributions $(c^*, c', \mathsf{EK}\{m\}, \mathsf{DK}\{c^*, c'\})$ and $(c', c^*, \mathsf{EK}\{m\}, \mathsf{DK}\{c^*, c'\})$ are indistinguishable, where m is an arbitrary plaintext, c' is its ciphertext, and c^* is randomly chosen. We note that for ciphertext-based punctured key symmetry is the only required security property, although we still require all applicable correctness properties.

Definition of ACE. Now we present a formal definition:

Definition 8. *[CHJV14], [BCG$^+$18] An asymmetrically constrained encryption (ACE) scheme is a 5-tuple of PPT algorithms* $(\mathsf{Setup}, \mathsf{GenEK}, \mathsf{GenDK}, \mathsf{Enc}, \mathsf{Dec})$ *satisfying syntax, correctness, security of constrained decryption, and selective indistinguishability of ciphertexts as described below.*

Syntax. The algorithms $(\mathsf{Setup}, \mathsf{GenEK}, \mathsf{GenDK}, \mathsf{Enc}, \mathsf{Dec})$ have the following syntax.

- **Setup:** $\mathsf{Setup}(1^\lambda, 1^n, 1^s)$ is a randomized algorithm that takes as input the security parameter λ, the message length n, and a "circuit succinctness" parameter s, all in unary. Setup then outputs a secret key SK. We think of secret keys as consisting of two parts: an encryption key EK and a decryption key DK.

  Let $\mathcal{M} = \{0,1\}^n$ denote the message space.

- **(Constrained) Key Generation:** Let $S \subset \mathcal{M}$ be any set whose membership is decidable by a circuit C_S. We say that S is *admissible* if $|C_S| \leq s$. Intuitively, the set size parameter s denotes the upper bound on the size of circuit description of sets to which encryption and decryption keys can be constrained.

  - $\mathsf{GenEK}(SK, C_S)$ takes as input the secret key SK of the scheme and the description of circuit C_S for an admissible set S. It outputs an encryption key $EK\{S\}$. We write EK to denote $EK\{\emptyset\}$.
  - $\mathsf{GenDK}(SK, C_S)$ also takes as input the secret key SK of the scheme and the description of circuit C_S for an admissible set S. It outputs a decryption key $DK\{S\}$. We write DK to denote $DK\{\emptyset\}$.

  Unless mentioned otherwise, we will only consider admissible sets $S \subset \mathcal{M}$.

- **Encryption:** $\mathsf{Enc}(EK', m)$ is a deterministic algorithm that takes as input an encryption key EK' (that may be constrained) and a message $m \in \mathcal{M}$ and outputs a ciphertext c or the reject symbol $\perp$.

- **Decryption:** $\mathsf{Dec}(DK', c)$ is a deterministic algorithm that takes as input a decryption key DK' (that may be constrained) and a ciphertext c and outputs a message $m \in \mathcal{M}$ or the reject symbol $\perp$.

Correctness. An ACE scheme is correct if the following properties hold:

1. *Correctness of Decryption*: For all n, all $m \in \mathcal{M}$, all sets $S, S' \subset \mathcal{M}$ s.t. $m \notin S \cup S'$,

$$\Pr\left[\mathsf{Dec}(DK, \mathsf{Enc}(EK, m)) = m \,\middle|\, \begin{array}{l} SK \leftarrow \mathsf{Setup}(1^\lambda), \\ EK \leftarrow \mathsf{GenEK}(SK, C_{S'}), \\ DK \leftarrow \mathsf{GenDK}(SK, C_S) \end{array} \right] = 1.$$

   Informally, this says that $\mathsf{Dec} \circ \mathsf{Enc}$ is the identity on messages which are in neither of the punctured sets.

2. *Equivalence of Constrained Encryption*: Let $SK \leftarrow \mathsf{Setup}(1^\lambda)$. For any message $m \in \mathcal{M}$ and any sets $S, S' \subset \mathcal{M}$ with m not in the symmetric difference $S \Delta S'$ (i.e., we are requiring that m is in both S and S' or m is in neither S nor S').

$$\Pr\left[\mathsf{Enc}(EK, m) = \mathsf{Enc}(EK', m) \,\middle|\, \begin{array}{l} SK \leftarrow \mathsf{Setup}(1^\lambda), \\ EK \leftarrow \mathsf{GenEK}(SK, C_S), \\ EK' \leftarrow \mathsf{GenEK}(SK, C_{S'}) \end{array} \right] = 1.$$

3. *Unique Ciphertexts*: With high probability over $SK \leftarrow \mathsf{Setup}(1^\lambda)$, it holds for any c and c' that if $\mathsf{Dec}(DK, c) = \mathsf{Dec}(DK, c') \neq \perp$, then $c = c'$.

4. *Safety of Constrained Decryption*: For all strings c, all $S \subset \mathcal{M}$,

$$\Pr\left[\mathsf{Dec}(DK, c) \in S \,\middle|\, SK \leftarrow \mathsf{Setup}(1^\lambda), DK \leftarrow \mathsf{GenDK}(SK, C_S) \right] = 0$$

   This says that a punctured key $DK\{S\}$ will never decrypt a string c to a message in S.

5. *Equivalence of Constrained Decryption*: If $\mathsf{Dec}(DK\{S\}, c) = m \neq \perp$ and $m \notin S'$, then $\mathsf{Dec}(DK\{S'\}, c) = m$.

Security of Constrained Decryption. Intuitively, this property says that for any two sets S_0, S_1, no adversary can distinguish between the constrained key $DK\{S_0\}$ and $DK\{S_1\}$, even given additional auxiliary information in the form of a constrained encryption key EK' and ciphertexts $c_1, \ldots, c_t$. To rule out trivial attacks, EK' is constrained at least on $S_0 \Delta S_1$. Similarly, each c_i is an encryption of a message $m \notin S_0 \Delta S_1$.

Formally, we describe security of constrained decryption as a multi-stage game between an adversary adv and a challenger.

- *Setup:* $\mathcal{A}$ chooses sets S_0, S_1, U s.t. $S_0 \Delta S_1 \subseteq U \subseteq \mathcal{M}$ and sends their circuit descriptions (C_{S_0}, C_{S_1}, C_U) to the challenger. adv also sends arbitrary polynomially many messages $m_1, \ldots, m_t$ such that $m_i \notin S_0 \Delta S_1$.

  The challenger chooses a bit $b \in \{0, 1\}$ and computes the following:

  1. $SK \leftarrow \mathsf{Setup}(1^\lambda)$,
  2. $DK\{S_b\} \leftarrow \mathsf{GenDK}(SK, C_{S_b})$,
  3. $EK \leftarrow \mathsf{GenEK}(SK, \emptyset)$,
  4. $c_i \leftarrow \mathsf{Enc}(EK, m_i)$ for every $i \in [t]$, and
  5. $EK\{U\} \leftarrow \mathsf{GenEK}(SK, C_U)$.

Finally, it sends the tuple $(EK\{U\}, DK\{S_b\}, \{c_i\})$ to adv.

- *Guess:* $\mathcal{A}$ outputs a bit $b' \in \{0, 1\}$.

The advantage of $\mathcal{A}$ in this game (on security parameter λ) is defined as $\mathsf{adv}_{\mathcal{A}} = \left|\Pr[b' = b] - \frac{1}{2}\right|$. We require that for all PPT $\mathcal{A}$, $\mathsf{adv}_{\mathcal{A}}(\lambda)$ is $\mathsf{negl}(\lambda)|S_1 \setminus S_0|$.

Selective Indistinguishability of Ciphertexts. Intuitively, this property says that no adversary can distinguish encryptions of m_0 from encryptions of m_1, even given certain auxiliary information. The auxiliary information corresponds to constrained encryption and decryption keys EK', DK', as well as some ciphertexts $c_1, \dots, c_t$. In order to rule out trivial attacks, EK' and DK' should both be punctured on at least $\{m_0, m_1\}$, and none of $c_1, \dots, c_t$ should be an encryption of m_0 or m_1. Let both $\mathcal{F}_1$ and $\mathcal{F}_2$ be sub-exponentially secure.

Formally, we require that for all sets $S, U \subset \mathcal{M}$, for all $m_0^*, m_1^* \in S \cap U$, and all $m_1, \dots, m_t \in \mathcal{M} \setminus \{m_0^*, m_1^*\}$, the distribution

$$EK\{S\}, DK\{U\}, c_0^*, c_1^*, c_1, \dots, c_t$$

is computationally indistinguishable from

$$EK\{S\}, DK\{U\}, c_1^*, c_0^*, c_1, \dots, c_t$$

in the probability space defined by sampling $SK \leftarrow \mathsf{Setup}(1^\lambda)$, $EK \leftarrow \mathsf{GenEK}(SK, \emptyset)$, $EK\{S\} \leftarrow \mathsf{GenEK}(SK, C_S)$, $DK\{U\} \leftarrow \mathsf{GenDK}(SK, C_U)$, $c_b^* \leftarrow \mathsf{Enc}(EK, m_b^*)$, and $c_i \leftarrow \mathsf{Enc}(EK, m_i)$.

As shown in [CHJV14], there exists subexponentially secure ACE assuming subexponentially secure injective PRGs and iO. We note that their construction and the proof can be based on injective OWFs instead of injective PRGs, similar to the proof of our relaxed ACE (section C).

Definition of relaxed ACE. As noted earlier, we also consider a relaxed ACE where indistinguishability of ciphertexts doesn't necessarily hold. Instead, we require a different property called *symmetry*, and we show how to modify the construction of [CHJV14] to build relaxed ACE with small security loss in the constrained decryption game for certain sets.

Definition 9. *A relaxed asymmetrically constrained encryption (relaxed ACE) scheme for message space* $\{0, 1\}^n$ *and suffix parameter t is a 6-tuple of PPT algorithms* $(\mathsf{Setup}, \mathsf{GenEK}, \mathsf{GenDK}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Puncture})$ *satisfying the the following:*

1. **Syntax:** $\mathsf{Setup}, \mathsf{GenEK}, \mathsf{GenDK}, \mathsf{Enc}, \mathsf{Dec})$ have syntax as in the definition of ACE. Ciphertext-based puncturing algorithm $\mathsf{Puncture}(SK, c_1, c_2)$ is an algorithm which takes as input the secret key SK, a ciphertext c_2 and a random string c_1 of the same length and outputs a ciphertext-based punctured key $DK\{c_1, c_2\}$. (We use this notation to distinguish ciphertext-based puncturing $DK\{c_1, c_2\}$ from plaintext-based puncturing $DK\{S\}$, where S is a set of *plaintexts*).

2. **Correctness:** We require all correctness properties as in the ACE definition. In addition, we require **correctness of decryption** and **equivalence of constrained decryption** to hold even for ciphertext-based punctured decryption keys. Namely, if $DK\{c_1, c_2\} = \mathsf{Puncture}(SK, c_1, c_2))$ where c_1 is random and c_2 is $\mathsf{Enc}(EK, m)$, then we require that the mentioned properties hold for the constrained set $S = \{m\}$.

3. **Security:** We require **security of constrained decryption** (from the definition of ACE) to hold for the case when *there are no plaintext queries*, and only for the case when $S_1 \setminus S_0$ is either of the form

S_{suf} (that is, a set of all strings ending with arbitrary, but the same for all strings, suffix suf of length t), or of the form $S_{\mathsf{suf}} \setminus \{m^*\}$ (where again suf has the size t, and m^* also ends with suf). Further, we require that distinguishing advantage depends on $|S_1 \setminus S_0|$ at most logarithmically; in particular, it should be negligible even when $|S_1 \setminus S_0| = O(2^\lambda)$ (alternatively, we can require that the advantage is smaller than a concrete negligible function).

In addition, for ciphertext-based punctured key we require a property called **symmetry**, which is defined with respect to the following game.

1. $\mathcal{A}$ chooses plaintext m and sends it to the challenger. Let $U = S_{\mathsf{suffix}_t(m)}$ be the set of all strings ending with the same t bits as m. the challenger computes the following:
2. $SK \leftarrow \mathsf{Setup}(1^\lambda)$,
3. c_1 is chosen at random from $\{0,1\}^{|c|}$;
4. $EK \leftarrow \mathsf{GenEK}(SK, \emptyset)$,
5. $EK\{U\} \leftarrow \mathsf{GenEK}(SK, U)$,
6. $c_2 \leftarrow \mathsf{Enc}(EK, m)$
7. $DK\{c_1, c_2\} \leftarrow \mathsf{Puncture}(SK, c_1, c_2)$,
8. Finally the challenger chooses random b and gives the adversary $(c_1, c_2, EK\{U\}, DK\{c_1, c_2\})$ if $b = 0$ and $(c_2, c_1, EK\{U\}, DK\{c_1, c_2\})$ if $b = 1$;
9. $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$.

The advantage of $\mathcal{A}$ in this game (on security parameter λ) is defined as $\mathsf{adv}_{\mathcal{A}} = \left|\Pr[b' = b] - \frac{1}{2}\right|$. We require that for all PPT $\mathcal{A}$, $\mathsf{adv}_{\mathcal{A}}(\lambda)$ is negligible in λ (alternatively, we can require that it is smaller than a concrete negligible function).

In the appendix (section C) we show that there exists subexponentially secure relaxed ACE assuming subexponentially secure OWFs and iO.

Sparse relaxed ACE. We remark that our relaxed ACE from appendix C has sparse image, that is, the probability that a randomly chosen string of a proper length is a valid ACE ciphertext is at most $2^{-\lambda}$.

6 Construction of interactive deniable encryption

In this section we describe a construction of interactive deniable encryption for a single-bit message space.

Notation.

We denote by s and r the variables corresponding to randomness of the sender and the receiver, respectively, and let μ_1, μ_2, μ_3 denote the three messages of the protocol. $\mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}$ are programs of the deniable encryption.

$\mathsf{P1}(s, m)$ takes as input sender randomness s and plaintext m and outputs the first message μ_1. $\mathsf{P2}(r, \mu_1)$ takes as input receiver randomness r and first message μ_1 and outputs the second message μ_2. $\mathsf{P3}(s, m, \mu_1, \mu_2)$ takes as input sender randomness s, plaintext m, and protocol messages μ_1, μ_2 and outputs the last message μ_3. $\mathsf{Dec}(r, \mu_1, \mu_2, \mu_3)$ takes as input receiver randomness r and protocol messages μ_1, μ_2, μ_3 and outputs

The CRS: Programs $\mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}$ (fig. 18, fig. 19)), obfuscated under iO.

Our Interactive deniable encryption:

Inputs: plaintext $m \in \{0, 1\}$ of the sender.

1. **Message 1:** The sender chooses random s^*, computes $\mu_1{}^* \leftarrow \mathsf{P1}(s^*, m)$ and sends $\mu_1{}^*$.
2. **Message 2:** The receiver chooses random r^*, computes $\mu_2{}^* \leftarrow \mathsf{P2}(r^*, \mu_1{}^*)$ and sends $\mu_2{}^*$.
3. **Message 3:** The sender computes $\mu_3{}^* \leftarrow \mathsf{P3}(s^*, m, \mu_1{}^*, \mu_2{}^*)$ and sends $\mu_3{}^*$.
4. The receiver runs $m' \leftarrow \mathsf{Dec}(r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$.

Sender Coercion:

Inputs: real plaintext $m \in \{0, 1\}$, fake plaintext $\hat{m} \in \{0, 1\}$, real random coins s^* of the sender, and the protocol transcript $\mu_1{}^*, \mu_2{}^*, \mu_3{}^*$.

1. Upon coercion, the sender computes fake randomness $s' \leftarrow \mathsf{SFake}(s^*, m, \hat{m}, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$.

Receiver Coercion:

Inputs: fake plaintext $\hat{m} \in \{0, 1\}$ and the protocol transcript $\mu_1{}^*, \mu_2{}^*, \mu_3{}^*$.

1. Upon coercion, the receiver chooses random ρ^* and computes fake randomness $r' \leftarrow \mathsf{RFake}(\hat{m}, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*; \rho^*)$.

Figure 17: Our interactive deniable encryption scheme.

the plaintext m. $\mathsf{SFake}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$ takes as input sender randomness s, true plaintext m, new (fake) plaintext $\hat{m}$, and protocol messages μ_1, μ_2, μ_3 and outputs fake randomness s' which makes μ_1, μ_2, μ_3 look consistent with $\hat{m}$. $\mathsf{RFake}(\hat{m}, \mu_1, \mu_2, \mu_3)$ takes as input new (fake) plaintext $\hat{m}$ and protocol messages μ_1, μ_2, μ_3 and outputs fake randomness r' which makes μ_1, μ_2, μ_3 look consistent with $\hat{m}$.

To avoid cumbersome notation, we use the same name for both unobfuscated and obfuscated programs. In particular, the parties and the adversary only see obfuscated programs and never the actual code of the programs. For example, on fig. 17 the instruction to the sender to run $\mathsf{P1}$ means taking the obfuscation of the program $\mathsf{P1}$ from the CRS and running it.

Everywhere throughout the paper we will be assuming that any program outputs $\perp$, if any of its underlying primitives outputs $\perp$, except for the cases where it is explicitly written otherwise.

6.1 Construction

The protocol is described in fig. 17. It simply instructs parties to run the programs from the CRS, which consists of 6 obfuscated programs $\mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}$ (described in fig. 18, fig. 19). Note that deniability of the receiver is *public*, since the knowledge of randomness of the receiver is not required in order to run RFake.

In the introduction we described the reasons behind the logic of the programs we are using. Here we give an overview of the overall structure of protocol messages and fake randomness. For simplicity, for this discussion we will use integer levels to count how many times s was faked (in the full construction, the programs of deniable encryption make use of a "level system" primitive instead of integers in the clear; level systems are defined formally in Section 7).

The structure of protocol messages. The first two messages in the protocol are simply "hashes" (implemented as a PRF) of internal state of parties so far: that is, μ_1 is $\mathsf{PRF}(s, m)$ and μ_2 is $\mathsf{PRF}(r, \mu_1)$. The third message μ_3 is an encryption of m, μ_1, μ_2, and level 0. After running the protocol, the receiver can run Dec which decrypts μ_3 and outputs m.

The structure of fake randomness. Fake randomness s' of the sender is an encryption (under a special sender-fake key which is known to programs but not known to parties) of $m', \mu_1', \mu_2', \mu_3'$, and level 1. This encryption has pseudorandom ciphertexts, and for an external observer s' looks like a truly random value. Programs can decrypt s' using hardwired key and interpret $(m', \mu_1', \mu_2', \mu_3', \ell')$ as an instruction to output μ_1' on input m' (for program P1) and an instruction to output μ_3' on input m', μ_1', μ_2' (for program P3). Thus, such s' makes the transcript look consistent with m', regardless of the actual plaintext which was used to generate the transcript.

Similarly, fake randomness r' of the receiver is an encryption (under a special receiver-fake key which is known to programs but not known to parties) of $m', \mu_1', \mu_2', \mu_3'$, and level 0 (together with $\mathsf{prg}(\rho)$ which is for randomizing this ciphertext). This encryption has pseudorandom ciphertexts, and for an external observer r' looks like a truly random value. Programs can decrypt r' using hardwired key and interpret $(m', \mu_1', \mu_2', \mu_3', L')$ as an instruction to output μ_2' on input μ_1' (for program P2) and an instruction to output m' on input μ_1', μ_2', μ_3' (for program Dec). Thus, such r' makes the transcript looks consistent with m' (and in particular decrypts it to m'), regardless of the actual plaintext which was used to generate the transcript.

Both programs P3, Dec also have special instructions for the "mixed input" case, i.e., for the case when P3 gets as input fake s' encrypting $(m', \mu_1', \mu_2', \mu_3', \ell')$, but input μ_2 of the program P3 is different from μ_2' in s' (in case of Dec, when μ_3' in fake r' is different from input μ_3 to Dec). The correct treatment of the mixed case is crucial for security of the scheme. See the explanation in the introduction for the logic of the programs on mixed inputs.

6.2 Building blocks and main theorem stating security

6.2.1 Level system

The *level system*, mentioned in earlier sections of this paper, is a primitive introduced in this work that is a critical building block of our deniable encryption protocol. This subsection provides detailed intuition about the level system primitive followed by a formal definition (the latter being a prerequisite to formally stating the security guarantees of our main construction). This subsection's scope is purely definitional; see Section 7 for a construction and security proof.

Motivation and overview. The idea of a level system is to have an encryption scheme which allows to increment ciphertexts and compare them homomorphically. However, in order for this encryption to be useful in our construction of deniable protocol, we require the following properties of this "encryption scheme":[22]

- There should be two types of ciphertexts, which we call *single-tag levels* and *double-tag levels*;
- A single-tag level is an encryption of number i between 0 and upper bound T, together with some string $m_1 \in M_1$, which we call *a tag*. (In our construction of deniable encryption, we use the first

[22] Note that even though we call it encryption, we don't require this primitive to have decryption.

message of the deniable protocol as a tag. This is done to "tie" the level to the instance of the protocol).

- A double-tag level is an encryption of number i between 0 and upper bound T, together with two tags $m_1 \in M_1, m_2 \in M_2$. (In our construction of deniable encryption, we use the first and the second messages of the deniable protocol as tags. This, again, is done to "tie" the level to the instance of the protocol).
- It should be possible to perform the following operations:
  1. Sample a single-tag level 0 for any tag m_1;
  2. Homomorphically increment the value inside any single-tag level (keeping its tag the same);
  3. Transform any single-tag level into a double-tag level, for any second tag m_2 (the value and the first tag remain the same);
  4. Compare two double-tag levels, as long as their both tags are the same;
  5. Given any level, retrieve its tag(s).

Notation. We use notation $[i, m_1]$ to denote a single-tag level with value i and tag m_1. We also use ℓ_i to denote a single-tag level with value i, when the tag is clear from the context.

We use notation $[i, m_1, m_2]$ to denote a double-tag level with value i and tags m_1, m_2. We also use L_i to denote a double-tag level with value i, when its tags are clear from the context.

Security property. The security requirement of a level system is that it should be hard to distinguish between $\ell_0^* = [0, m_1^*], L_0^* = [0, m_1^*, m_2^*]$ and $\ell_1^* = [1, m_1^*], L_0^* = [0, m_1^*, m_2^*]$, even given (limited) ability to perform homomorphic operations described above.

This will be used in the proof of security of deniable encryption scheme as follows. Recall that in that proof we need to start with the real transcript and real randomness s, r (having levels L_0^*, ℓ_0^*, L_0^*, respectively) and eventually switch to the (same) real transcript but fake randomness s', r' (with levels L_0^*, ℓ_1^*, L_0^*). We can use security of the level system in the proof of deniable encryption as follows: given challenge ℓ_b^*, L_0^* (where $\ell_b^* = [b, m_1^*]$, $b \in \{0, 1\}$, $L_0^* = [0, m_1^*, m_2^*]$), we use ℓ_b^* inside fake s and we use L_0^* inside the transcript and fake r. Since security of levels only holds when programs are punctured, in the proof of deniable encryption we first move to a hybrid with only punctured level programs, and then invoke security of the level system.

Definition We start with describing the syntax of a level system for tag space M and upper bound T:

- $\mathsf{Setup}(1^\lambda; T; \mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}; r_{\mathsf{Setup}}) \rightarrow \mathsf{PP} = (\mathsf{P}_{\mathsf{GenZero}}, \mathsf{P}_{\mathsf{Increment}}, \mathsf{P}_{\mathsf{Transform}}, \mathsf{P}_{\mathsf{isLess}}, \mathsf{P}_{\mathsf{RetrieveTag}}, \mathsf{P}_{\mathsf{RetrieveTags}})$ is a randomized algorithm which takes as input security parameter, the largest allowed level T, description of programs, and randomness. It uses random coins to sample all necessary keys for each program[23], and outputs those programs obfuscated under iO.
- $\mathsf{GenZero}(m_1) \rightarrow \ell$ is a deterministic algorithm which takes message $m_1 \in M$ as input and outputs a string $\ell = [0, m_1]$, which is a single-tag level with tag m_1 and value 0. We also require that there exists a punctured version of this algorithm denoted $\mathsf{GenZero}[m_1^*](m_1)$ which outputs $'\mathsf{fail}'$ on input m_1^*.
- $\mathsf{Increment}(\ell) \rightarrow \ell'$ is a deterministic algorithm which takes a single-tag level $\ell = [i, m_1]$ for some $0 \leq i \leq T - 1, m_1 \in M$, and outputs a single-tag level with the same tag and incremented value, i.e.

[23]We assume that Setup is implicitly given generation algorithms for all underlying primitives of the programs.

$\ell' = [i+1, m_1]$. If $i \geq T$, it instead outputs $'\mathsf{fail}'$.

- $\mathsf{Transform}(\ell, m_2) \to \ell$ is a deterministic algorithm which takes a single-tag level $\ell = [i, m_1]$ for some $0 \leq i \leq T, m_1 \in M$, and some message $m_2 \in M$, and outputs $L = [i, m_1, m_2]$, which is a double-tag level with tags m_1, m_2, and value i. We also require that there exists a punctured version of this algorithm denoted $\mathsf{Transform}[(\ell^*, m_2^*)](\ell, m_2)$ which outputs $'\mathsf{fail}'$ on input (ℓ^*, m_2^*).
- $\mathsf{isLess}(L', L'') \to \mathsf{out} \in \{\mathsf{true}, \mathsf{false}\}$ is a deterministic algorithm which takes as input two double-tag levels $L' = [i', m_1', m_2']$ and $L'' = [i'', m_1'', m_2'']$. If $(m_1', m_2') \neq (m_1'', m_2'')$, then it outputs $'\mathsf{fail}'$. Otherwise it outputs true if $i' < i''$ and false if $i' \geq i''$.
- $\mathsf{RetrieveTag}(\ell) \to m_1$ is a deterministic algorithm which takes a single-tag level ℓ and outputs its tag.
- $\mathsf{RetrieveTags}(L) \to (m_1, m_2)$ is a deterministic algorithm which takes a double-tag level L and outputs both tags.

We emphasize that all programs except Setup are deterministic.

Definition 10. *A tuple of parametrized, deterministic*[24] *algorithms*

$$(\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}, \mathsf{GenZero}[m_1^*], \mathsf{Transform}[l^*, m_2^*])$$

is a level system *for tag space M, if algorithms have syntax described above, and the correctness and security properties described below hold.*

Notation: *Let T be superpolynomial in λ, and* $\mathsf{PP} = (\mathsf{P}_{\mathsf{GenZero}}, \mathsf{P}_{\mathsf{Increment}}, \mathsf{P}_{\mathsf{Transform}}, \mathsf{P}_{\mathsf{isLess}}, \mathsf{P}_{\mathsf{RetrieveTag}}, \mathsf{P}_{\mathsf{RetrieveTags}}) \leftarrow \mathsf{Setup}(1^\lambda; T; \mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ *for randomly chosen r_{Setup}.*

Next, let $m_1^* \in M$, $m_2^* \in M$, and let ℓ^* be an arbitrary string (not necessarily a level). Let $\mathsf{PP}' = (\mathsf{P}'_{\mathsf{GenZero}}, \mathsf{P}'_{\mathsf{Increment}}, \mathsf{P}'_{\mathsf{Transform}}, \mathsf{P}'_{\mathsf{isLess}}, \mathsf{P}'_{\mathsf{RetrieveTag}}, \mathsf{P}'_{\mathsf{RetrieveTags}}) \leftarrow \mathsf{Setup}(1^\lambda, T, \mathsf{GenZero}[m_1^*], \mathsf{Increment}, \mathsf{Transform}[(\ell^*, m_2^*)], \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ *with the same randomness r_{Setup} as above.*

For any fixed r_{Setup} consider the following notation:

- *For every $m_1 \in M$ denote $[0, m_1] = \mathsf{P}_{\mathsf{GenZero}}(m_1)$;*
- *For every $m_1 \in M$, $1 \leq i \leq T$ denote $[i, m] = \mathsf{P}_{\mathsf{Increment}}([i-1, m])$;*
- *For every $m_2 \in M$ and every $[i, m_1]$, where $0 \leq i \leq T, m_1 \in M$, denote $[i, m_1, m_2] = \mathsf{P}_{\mathsf{Transform}}([i, m_1], m_2)$.*

Correctness: *The following properties should hold, except with negligible probability over the choice of r_{Setup}:*

- **Uniqueness of levels:**
  - *For all $\ell \notin \{[i, m_1] : 0 \leq i \leq T, m_1 \in M\}$:*

[24] We prefer to use the notion of parametrized, deterministic algorithms to keep the definition simple. To formally define this notion, consider a randomized Turing machine with the restriction that the number of random bits written on its random tape is fixed and independent of the input (only dependent on security parameter λ). Such a Turing machine can first use these random coins to generate all necessary parameters (e.g., keys) and then run the actual code of the algorithm using generated parameters. In particular, we assume that this TM has the code of all necessary generation algorithms.

- * $\mathsf{P}_{\mathsf{Increment}}(\ell) = '\mathsf{fail}'$;
  * $\mathsf{P}_{\mathsf{Transform}}(\ell, m_2) = '\mathsf{fail}'$ *for any* $m_2 \in M$;
  * $\mathsf{P}_{\mathsf{RetrieveTag}}(\ell) = '\mathsf{fail}'$.
- *For all* $L \notin \{[i, m_1, m_2] : 0 \le i \le T, m_1 \in M, m_2 \in M\}$:
  * $\mathsf{P}_{\mathsf{isLess}}(L, L') = '\mathsf{fail}'$, $\mathsf{P}_{\mathsf{isLess}}(L', L) = '\mathsf{fail}'$, *for any string* L';
  * $\mathsf{P}_{\mathsf{RetrieveTags}}(L) = '\mathsf{fail}'$.

- **Upper bound is respected:** *For every* $m_1 \in M$ $\mathsf{P}_{\mathsf{Increment}}([T, m_1]) = '\mathsf{fail}'$.
- **Correctness of comparison:** *For every* $m_1, m_2 \in M$ *and for every* $0 \le i, j \le T$:
  - $\mathsf{P}_{\mathsf{isLess}}([i, m_1, m_2], [j, m_1, m_2]) = \mathsf{true}$ *for* $i < j$,
  - $\mathsf{P}_{\mathsf{isLess}}([i, m_1, m_2], [j, m_1, m_2]) = \mathsf{false}$ *for* $i \ge j$.
- **Comparison is possible only on matching levels:** *If* $(m_1', m_2') \neq (m_1'', m_2'')$, *then* $\mathsf{P}_{\mathsf{isLess}}([i, m_1', m_2'], [j, m_1'', m_2'']) = '\mathsf{fail}'$ *for all* i, j.
- **Correctness of tags retrieval:** *For every* $m_1, m_2 \in M$ *and for every* $0 \le i \le T$:
  - $\mathsf{P}_{\mathsf{RetrieveTag}}([i, m_1]) = m_1$,
  - $\mathsf{P}_{\mathsf{RetrieveTags}}([i, m_1, m_2]) = (m_1, m_2)$.
- **Functionality is preserved under puncturing:**
  - $\mathsf{P}_{\mathsf{GenZero}}(m) = \mathsf{P}'_{\mathsf{GenZero}}(m)$ *for all* $m \in M$, $m \neq m_1^*$;
  - $\mathsf{P}_{\mathsf{Increment}}(\ell) = \mathsf{P}'_{\mathsf{Increment}}(\ell)$ *for all strings* ℓ;
  - $\mathsf{P}_{\mathsf{Transform}}(\ell, m_2) = \mathsf{P}'_{\mathsf{Transform}}(\ell, m_2)$ *for all strings* l *and for all* $m_2 \in M$, *except* (ℓ^*, m_2^*);
  - $\mathsf{P}_{\mathsf{isLess}}(L', L'') = \mathsf{P}'_{\mathsf{isLess}}(L'', L'')$ *for all strings* L', L'';
  - $\mathsf{P}_{\mathsf{RetrieveTag}}(\ell) = \mathsf{P}'_{\mathsf{RetrieveTag}}(\ell)$ *for all strings* ℓ;
  - $\mathsf{P}_{\mathsf{RetrieveTags}}(L) = \mathsf{P}'_{\mathsf{RetrieveTags}}(L)$ *for all strings* L.

Note that it follows from the correctness properties that $[i, m_1] = [i', m_1']$ *if and only* $(i, m_1) = (i', m_1')$, *and* $[i, m_1, m_2] = [i', m_1', m_2']$ *if and only* $(i, m_1, m_2) = (i', m_1', m_2')$.

Security: *For any* $m_1^* \in M, m_2^* \in M$, *the following distributions are computationally indistinguishable:*

$$(\ell_0^*, L_0^*, \mathsf{PP}_0) \approx (\ell_1^*, L_0^*, \mathsf{PP}_1),$$

where r_{Setup} *is randomly chosen,* $\mathsf{PP} = (\mathsf{P}_{\mathsf{GenZero}}, \mathsf{P}_{\mathsf{Increment}}, \mathsf{P}_{\mathsf{Transform}}, \mathsf{P}_{\mathsf{isLess}}, \mathsf{P}_{\mathsf{RetrieveTag}}, \mathsf{P}_{\mathsf{RetrieveTags}}) \leftarrow \mathsf{Setup}(\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}; r_{\mathsf{Setup}})$,

$\ell_0^* \leftarrow \mathsf{P}_{\mathsf{GenZero}}(m_1^*)$, $\ell_1^* \leftarrow \mathsf{P}_{\mathsf{Increment}}(\ell_0^*)$, $L_0^* \leftarrow \mathsf{P}_{\mathsf{Transform}}(\ell_0^*, m_2^*)$,

$\mathsf{PP}_b \leftarrow \mathsf{Setup}(\mathsf{GenZero}[m_1^*], \mathsf{Increment}, \mathsf{Transform}[(\ell_b^*, m_2^*)], \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}; r_{\mathsf{Setup}})$.

6.2.2 Primitives required for the main construction, and their parameters

We require the primitives listed below. Note that these primitives can be constructed from iO, injective PRFs (which in turn can be constructed from standard OWFs, [SW14]) and injective OWFs (which in turn can be constructed from iO and standard OWFs, [BPW16]); thus it is enough to require iO and OWFs. By starting with subexponentially-secure iO and OWFs, we can get subexponential security of these primitives.

Definitions can be found in section 5.

Notation. We denote security parameter by λ. We parametrize sizes in our construction by $\tau(\lambda)$, which is the length of the first message in the protocol (also equal to the size of a tag for the level system, since we use μ_1, μ_2 as tags), and $T(\lambda)$, which is an upper bound of the level system.

Injective PRFs with sparse image. As shown in [SW14], for any length l there exists a family of PRFs $\{F_k\}_\lambda$ mapping l-sized inputs to $2l + \lambda$-sized outputs, such that with probability at least $1 - 2^{-\lambda}$ (over the choice of the key), the PRF is injective. Note that PRF with these parameters has exponentially sparse image, i.e. a randomly chosen element is in its image with probability $2^{-l-\lambda}$.

These PRFs are used in the construction of ACE and relaxed ACE.

Sparse extracting PRF. As shown in [SW14], for any length l, as long as the input has entropy at least $l \geq \tau/2 + 2\lambda + 2$, there exists a family of extracting PRFs $\{F_k\}_\lambda$ mapping at least l-sized inputs to $\tau/2$-sized outputs, which are strong extractors with statistical distance at most $2^{-\lambda}$. It can be shown in a simple reduction that applying a length-doubling prg to the output of such a PRF results in a (computationally) extracting PRF, such that a random string is in its image with probability $2^{\tau/2}$.

These PRFs are used to compute the first two messages in the protocol.

ACE. As shown in [CHJV14], for any plaintext length l, there exists an ACE with ciphertexts of size $3l + \lambda$ (as long as injective PRFs used are from l bits to $2l + \lambda$ bits).

ACE is used as the main encryption scheme (used to compute the third message of the protocol).

Relaxed ACE. As we show in the appendix C by modifying the construction of [CHJV14], for any plaintext length l and suffix parameter t, there exists a relaxed ACE with ciphertexts of size $(l - t + 1)(2l - t + \lambda) + \lambda$ (as long as each injective PRF F_i, $i = t, \ldots, l$, is from i bits to $2i + \lambda$ bits). . Further, ciphertexts of this ACE are sparse, with ratio of ciphertexts at most $2^{-\lambda}$. Relaxed ACE is used as an encryption scheme to generate fake sender and receiver randomness.

Length-doubling PRG. We use a prg from λ to 2λ bits. It is used in program RFake to randomize fake randomness of the receiver. (In addition, as part of the construction of a sparse extracting PRF, we also use a prg from $\tau(\lambda)/2$ to $\tau(\lambda)$ bits).

Level system. We require a level system for any superpolynomial upper bound T and any sublinear tag size.

Length of variables as a function of the first message size τ and level upper bound T. Below we express sizes in our construction (which in turn specify parameters of all primitives) as a function of the first message size $\tau(\lambda)$ and the upper bound of the level system $T(\lambda)$. We require that both $\tau(\lambda)$and $\log T(\lambda)$ are sublinear in λ. We assume that the plaintext of the deniable encryption scheme is one bit long. Somewhat abusing notation, in this discussion we will be denoting the size of the ACE ciphertext of l-size input as $\mathsf{ACE}(l)$; size of levels as $|\ell|, |L|$; size of the output of a prg as $|\mathsf{prg}|$.

- $|\mu_1| = \tau$;
- $|\mu_2| = \tau$;
- $|\ell| = |\mathsf{ACE}(|\mu_1| + \log T)| = 3(\tau + \log T) + \lambda = O(\lambda)$;
- $|L| = |\mathsf{ACE}(|\mu_1| + |\mu_2| + \log T)| = 3(2\tau + \log T) + \lambda = O(\lambda)$;
- $|\mu_3| = |\mathsf{ACE}(1+|\mu_1|+|\mu_2|+|L|)| = 3(1+2\tau+3(2\tau+\log T)+\lambda)+\lambda = 3+24\tau+9\log T+4\lambda = O(\lambda)$;
- $|s| = \mathsf{relaxedACE}(1 + |\mu_1| + |\mu_2| + |\mu_3| + |\ell|)$ (for suffix parameter $t = |\ell|$), thus the size is equal to $(1+2\tau+(3+15\tau+9\log T+4\lambda)+1)(2(1+2\tau+(3+15\tau+9\log T+4\lambda)+3(\tau+\log T)+\lambda) - (3(\tau+\log T)+\lambda)+\lambda)+\lambda = (5+17\tau+9\log T+4\lambda)(8+37\tau+21\log T+20\lambda)+\lambda = O(\lambda^2)$;
- $|r| = \mathsf{relaxedACE}(1+|\mu_1|+|\mu_2|+|\mu_3|+|L|+|\mathsf{prg}|)$ (for suffix parameter $t = |\mathsf{prg}|$), thus the size is equal to $((1+2\tau+3+24\tau+9\log T+4\lambda+3(2\tau+\log T)+\lambda+2\lambda)-2\lambda+1)(2(1+2\tau+3+24\tau+9\log T+4\lambda+3(2\tau+\log T)+\lambda+2\lambda)-2\lambda+\lambda)+\lambda = (5+32\tau+12\log T+5\lambda)(8+64\tau+24\log T+13\lambda)+\lambda = O(\lambda^2)$.

Further, since in our construction of deniable encryption we use the first message μ_1 as a tag for the level system, we need a level system for upper bound T and tag size τ.

The size of the programs, and removing layers of iO. Note that the source code on fig. 18, fig. 19 includes the description of *obfuscated* programs of the level system. In turn, the source code of programs of the level system contains ACE keys which are again obfuscations of some other programs. Thus, the CRS contains programs which have 3 layers of obfuscation.

However, this layering is only for convenience: it enables proving the security of component primitives (e.g., ACE and the level system) separately and then combine them into a bigger proof (e.g., of deniable encryption or the level system). It is possible to prove security of our deniable encryption where programs of deniable encryption are obfuscated *only once*. That is, programs of deniable encryption can use *unobfuscated* code of the programs of the level system and ACE. However, to show security in this case, one would have to "unroll" all proofs, i.e., substitute the proof of, say, ACE instead of each reduction to security of ACE in the main proof. Needless to say, writing, and more importantly, *verifying* such a proof would be very onerous (certainly from the perspective of the authors, who think of themselves as polynomially-bounded Turing machines).

Nevertheless, in appendix B we briefly explain why such a proof *could* be written. Intuitively, this holds because of the following: let's say in the proof of ACE we punctured the PRF and reduced it to security of the obfuscation (of ACE source code). Then we can do the same reduction in the "unrolled" proof, since that punctured PRF key, which is now a part of a source code of deniable encryption program, is still protected by obfuscation on top of that program.

We state our theorem with a parameter σ representing the size of the source code of the programs of the deniable encryption scheme. As long as our construction uses only one layer of iO, $\sigma = O(\lambda^3)$ (λ^3 comes from the fact that all programs of deniable encryption use keys of a relaxed ACE, which have size $O(\lambda^3)$ due to the fact each key consists of $O(\lambda)$ PRF keys, these keys are punctured in the security proof, and each punctured PRF key has size $O(\lambda^2)$).

6.2.3 Main theorem

Theorem 2. *Assume the existence of the following primitives with parameters spicified above:*

- SG, RG *are extracting puncturable PRFs with sparse image. Further, these PRFs should have a property that, given a punctured key, we can further puncture them at one more point;*
- prg *is a pseudorandom generator with a sparse image;*
- *Programs* $(\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags})$ *are the programs of a level system;*
- *sender-fake ACE (with keys* $\mathsf{EK}_S, \mathsf{DK}_S$*) is a relaxed ACE with suffix parameter equal to the size of a single-tag level of the level system; in addition, its ciphertexts should be sparse.*
- *receiver-fake ACE (with keys* $\mathsf{EK}_R, \mathsf{DK}_R$*) is a relaxed ACE with suffix parameter equal to the image length of a* prg*; in addition, its ciphertexts should be sparse.*
- *main ACE (with keys* EK, DK*);*
- iO *is a secure indistinguishability obfuscation for circuits of size* $\sigma = c \cdot \lambda^3$ *for some constant c;*

Then the protocol of fig. 17 instantiated with the programs in fig. 18 and fig. 19 is a bideniable and off-the-record deniable interactive encryption in the CRS model for 1*-bit plaintexts. More specifically, assuming that each primitive except the level system is* $(t(\lambda), \varepsilon(\lambda))$*-secure, and assuming the level system for an upper bound* T *and tag size* τ *is* $O(t(\lambda), \varepsilon_1(\lambda, T, \tau))$*-secure, the resulting deniable encryption is* $(t(\lambda), O(\varepsilon(\lambda)) + O(2^{-\tau}) + \varepsilon_1(\lambda, T, \tau))$*-secure.*

Corollary 1. *Let* $T = 2^{\lambda^{\varepsilon/2}}$*,* $\tau = \lambda^{\varepsilon/2}$*, and assume that all primitives in theorem 2 are* $(\mathsf{poly}(\lambda), 2^{-\Omega(\lambda^{\varepsilon^2/2})})$*-secure. Then the resulting deniable encryption is* $(\mathsf{poly}(\lambda), 2^{-\Omega(\lambda^{\varepsilon^2/2})})$*-secure.*

Encrypting longer plaintexts. Note that the syntax of the scheme allows to encrypt longer plaintexts. However, for simplicity we define and prove deniability and off-the-record-deniability for 1-bit plaintexts only. In appendix D we list the changes required to adapt the proof to support longer plaintexts. However, this incurs additional security loss proportional to the $|\mathcal{M}|^3$, the *cube* of the size of the plaintext space.

Programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$**.**

Program $\mathsf{P1}(s, m)$

Inputs: sender randomness s, plaintext m.

Hardwired values: decryption key DK_S of sender-fake ACE, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = '\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m = m'$ then return ${\mu_1}'$;
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, plaintext m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; decryption key DK_S of sender-fake ACE, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = '\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', {\mu_1}', {\mu_2}'$ then return ${\mu_3}'$;
   (c) If $m, \mu_1 = m', {\mu_1}'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake plaintext $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; encryption and decryption keys $\mathsf{EK}_S, \mathsf{DK}_S$ of sender-fake ACE.

1. **Validity check:** if $\mathsf{P1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = '\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1 = m', {\mu_1}'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = '\mathsf{fail}'$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 18: Programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$.

Programs $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$.

Program $\mathsf{P2}(r, \mu_1)$

Inputs: receiver randomness r, the first message μ_1 in the protocol.

Hardwired values: decryption key DK_R of receiver-fake ACE, key k_R of an extracting PRF RG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_R}(r)$; if $\mathsf{out} = {'}\mathsf{fail}{'}$ then goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', L', \hat{\rho})$;
   (b) If $\mu_1 = {\mu_1}'$ then return ${\mu_2}'$;
2. **Main step:**
   (a) Return $\mu_2 \leftarrow \mathsf{RG}_{k_R}(r, \mu_1)$.

Program $\mathsf{Dec}(r, \mu_1, \mu_2, \mu_3)$

Inputs: receiver randomness r, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P2}$, isLess, $\mathsf{RetrieveTags}$; decryption key DK_R of receiver-fake ACE, decryption key DK of the main ACE.

1. **Validity check:** if $\mathsf{P2}(r, \mu_1) \neq \mu_2$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_R}(r)$; if $\mathsf{out}' = {'}\mathsf{fail}{'}$ then goto main step; else parse out' as $(m', {\mu_1}', {\mu_2}', {\mu_3}', L', \hat{\rho})$;
   (b) if $\mu_1, \mu_2, \mu_3 = {\mu_1}', {\mu_2}', {\mu_3}'$ then return m';
   (c) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}}(\mu_3)$; if $\mathsf{out}'' = {'}\mathsf{fail}{'}$ then abort, else parse out'' as $(m'', {\mu_1}'', {\mu_2}'', L'')$;
   (d) If $\mu_1, \mu_2 = {\mu_1}', {\mu_2}'$ then
      i. If $({\mu_1}', {\mu_2}') = ({\mu_1}'', {\mu_2}'') = \mathsf{RetrieveTags}(L'')$ and $\mathsf{isLess}(L', L'') = \mathsf{true}$ then return m'';
      ii. Else abort.
3. **Main step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}}(\mu_3)$; if $\mathsf{out} = {'}\mathsf{fail}{'}$ then abort, else parse out as $(m'', {\mu_1}'', {\mu_2}'', L'')$;
   (b) If $(\mu_1, \mu_2) = ({\mu_1}'', {\mu_2}'') = \mathsf{RetrieveTags}(L'')$ then return m'';
   (c) Else abort.

Program $\mathsf{RFake}(\hat{m}, \mu_1, \mu_2, \mu_3; \rho)$

Inputs: fake plaintext $\hat{m}$, protocol transcript μ_1, μ_2, μ_3, random coins ρ.

Hardwired values: encryption key EK_R of receiver-fake ACE, decryption key DK of the main ACE.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}}(\mu_3)$; if $\mathsf{out} = {'}\mathsf{fail}{'}$ then abort, else parse out as $(m'', {\mu_1}'', {\mu_2}'', L'')$;
2. Return $r' \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_R}(\hat{m}, \mu_1, \mu_2, \mu_3, L'', \mathsf{prg}(\rho))$.

Figure 19: Programs $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$.

6.3 Proof overview

Correctness. Correctness follows from correctness of all underlying primitives and from the fact that sender-fake and receiver-fake ACE are both sparse. More concretely, assume s^* and r^* are randomly chosen coins of the sender and the receiver. Due to sparseness of ACE, s^* (resp, r^*) is outside of the image of sender-fake (resp., receiver-fake) ACE. Therefore program P1 on input s^*, m executes the main step and outputs ${\mu_1}^* = \mathsf{SG}_{k_S}(s^*, m)$, program P2 on input $r^*, {\mu_1}^*$ executes the main step and outputs ${\mu_2}^* = \mathsf{RG}_{k_R}(r^*, {\mu_1}^*)$, and program P3 on input $s^*, m, {\mu_1}^*, {\mu_2}^*$ executes the main step and outputs ${\mu_3}^* = \mathsf{Enc}_K(m, {\mu_1}^*, {\mu_2}^*, \mathsf{Transform}(\mathsf{GenZero}({\mu_1}^*), {\mu_2}^*))$. In particular, the validity check passes since indeed $\mathsf{P1}(s^*, m) = {\mu_1}^*$.

Next, program Dec on input $r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*$ executes the main step by decrypting ${\mu_3}^*$ and returning its plaintext m. In particular, validity check passes, since $\mathsf{P2}(r^*, {\mu_1}^*) = {\mu_2}^*$. Further, note that μ_1, μ_2 which are the input to Dec, ${\mu_1}'', {\mu_2}''$ which are decrypted from ${\mu_3}^*$, and the output of $\mathsf{RetrieveTags}(L'')$ are all equal to ${\mu_1}^*, {\mu_2}^*$ (recall that $L'' = \mathsf{Transform}(\mathsf{GenZero}({\mu_1}^*), {\mu_2}^*)$). Thus all checks in the main step of Dec pass and the program outputs m.

Notation. m_0^*, m_1^* denote messages chosen by the adversary. s^*, r^* denote true (chosen at random) random coins of the sender and receiver, respectively. ${\mu_1}^*, {\mu_2}^*, {\mu_3}^*$ denote the challenge transcript of the protocol, which is either $\mathsf{tr}(s^*, r^*, m_0^*)$ or $\mathsf{tr}(s^*, r^*, m_1^*)$ depending on the hybrid. s', r' denote fake random coins of the sender and receiver, respectively. We write $\mathsf{tr}(s, r, m)$ to denote the communication in the protocol with input m and randomness s and r.

By ℓ_0^* we denote a single-tag level 0 with tag ${\mu_1}^*$. By ℓ_1^* we denote a single-tag level 1 with tag ${\mu_1}^*$. By L_0^* we denote double-tag level 0 with tags ${\mu_1}^*, {\mu_2}^*$.

In addition, we will be using notation $[\mathsf{val}, \mu_1]$ and $[\mathsf{val}, \mu_1, \mu_2]$ to denote single-tag and double-tag levels with value val and tag μ_1 (or, tags μ_1, μ_2).

Main steps. We start with a distribution corresponding to transmitted plaintext $m_0^* \in \{0,1\}$ and real randomness s^* and r^* presented to the adversary. More formally, we consider the following distribution:

$\mathsf{Hyb}_A = (\mathsf{PP}, m_0^*, m_1^*, s^*, r^*, \mathsf{tr}(s^*, r^*, m_0^*))$, where s^*, r^* are randomly chosen, and $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}.

To prove security of our deniable encryption scheme, we proceed in the following steps:

1. **Indistinguishability of explanations of the sender**: we switch real (randomly chosen) s^* to fake s', which encodes plaintext m_0^*, transcript ${\mu_1}^*, {\mu_2}^*, {\mu_3}^*$, and level $\ell^* = [0, {\mu_1}^*]$, moving to the following distribution:

   $\mathsf{Hyb}_B = (\mathsf{PP}, m_0^*, m_1^*, s', r^*, \mathsf{tr}(s^*, r^*, m_0^*))$, where s^*, r^* are randomly chosen, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, and $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}.

   The proof of this step is similar in spirit to the proof of a sender-deniable encryption of Sahai and Waters [SW14], and relies on the fact that all relevant programs, given s^* or s' as input, behave in the same way for any choice of remaining inputs.

2. **Indistinguishability of explanations of the receiver**: we switch real (randomly chosen) r^* to fake r', which encodes plaintext m_0^*, transcript ${\mu_1}^*, {\mu_2}^*, {\mu_3}^*$, and level $L^* = [0, {\mu_1}^*, {\mu_2}^*]$, moving to the

following distribution:

$\mathsf{Hyb}_C = (\mathsf{PP}, m_0^*, m_1^*, s', r', \mathsf{tr}(s^*, r^*, m_0^*))$, where s^*, r^* are randomly chosen, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*, and $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}.

Unlike the previous step, here there exist inputs such that program Dec, when run on these inputs and r^* or r', produces different outputs. However, such inputs are hard to find. Thus, in security proof of this step we first use properties of ACE to "eliminate" bad inputs (i.e. to make the programs reject them), then run Sahai-Waters-like proof similar to the previous step, and finally use ACE to bring bad inputs back and restore the programs.

3. **Semantic security**: we switch the transcript from encrypting m_0^* to encrypting m_1^*, moving to the following distribution:

   $\mathsf{Hyb}_D = (\mathsf{PP}, m_0^*, m_1^*, s', r', \mathsf{tr}(s^*, r^*, m_1^*))$, where s^*, r^* are randomly chosen, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*, and $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}.

   Proving security of this step involves the following. First, similar to the previous step, we "eliminate" a ciphertext $\overline{{\mu_3}^*} = \mathsf{ACE.Enc}_{\mathsf{EK}}(1 \oplus m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, making all programs reject it (note that this ciphertext is "complementary" to the challenge ciphertext ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, meaning it encrypts the opposite bit). This allows us to modify program Dec such that decryption key DK is not used to decrypt ${\mu_3}^*, \overline{{\mu_3}^*}$. Then we use security of ACE to switch ${\mu_3}^*$ from encrypting m_0^* to m_1^*, and then revert all previous changes.

4. **Indistinguishability of levels**: we switch the level encoded in s' from $\ell_0^* = [0, {\mu_1}^*]$ to $\ell_1^* = [1, {\mu_1}^*]$ (while keeping $L_0^* = [0, {\mu_1}^*, {\mu_2}^*]$ the same), moving to the following distribution:

   $\mathsf{Hyb}_E = (\mathsf{PP}, m_0^*, m_1^*, s', r', \mathsf{tr}(s^*, r^*, m_1^*))$, where s^*, r^* are randomly chosen, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*, and $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}.

   To prove security of this step, we first use security of ACE to eliminate some bad inputs. After this, we can modify programs of deniable encryption scheme in such a way that they only use *punctured* version of the programs of the level system. Then we invoke security of the level system and finally revert previous changes.

Finally, we argue that, except with negligible probability, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$ is the same as $s' = \mathsf{SFake}(s^*, m_1^*, m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$ (indeed, this is what SFake outputs except for a negligibly small fraction of inputs). In addition, since $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*)) = \mathsf{RFake}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*; \rho^*)$, we thus obtain the following distribution:

$\mathsf{Hyb}_F = (\mathsf{PP}, m_0^*, m_1^*, s', r', \mathsf{tr}(s^*, r^*, m_1^*))$, where s^*, r^* are randomly chosen, $s' = \mathsf{SFake}(s^*, m_1^*, m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, $r' = \mathsf{RFake}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*; \rho^*)$ for randomly chosen ρ^*, and $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}.

Note that this distribution corresponds to the execution of the protocol with plaintext m_1^* and fake randomness s', r' which makes this transcript look consistent with plaintext m_0^*, and thus we proved security of our deniable encryption.

In section 8.1 for each one of the four steps we present a list of hybrids with a brief explanation of why indistinguishability between each hybrid holds. Formal security reductions can be found in section 8.2.

Off-the-record deniability. Proof of off-the-record deniability of our scheme follows the same major four steps, but in a different order and with slightly different distributions. In section 9 we explain how to modify the proof of deniability from section 8 to turn it into a proof of off-the-record deniability.

7 Level System

This section presents our level system construction and security proof. Level systems were already defined in section 6.2.1, as a building block for our deniable encryption protocol. We repeat both the intuitive motivation and formal definition here, in order to keep this section self-contained for any readers who may read it separately. (We believe the level system may be a primitive of independent interest.) Readers wishing to skip the definitional material and go straight to the construction should skip to section 7.2.

Motivation and overview. The idea of a level system is to have an encryption scheme which allows to increment ciphertexts and compare them homomorphically. However, in order for this encryption to be useful in our construction of deniable protocol, we require the following properties of this "encryption scheme":[25]

- There should be two types of ciphertexts, which we call *single-tag levels* and *double-tag levels*;
- A single-tag level is an encryption of number i between 0 and upper bound T, together with some string $m_1 \in M_1$, which we call *a tag*. (In our construction of deniable encryption, we use the first message of the deniable protocol as a tag. This is done to "tie" the level to the instance of the protocol).
- A double-tag level is an encryption of number i between 0 and upper bound T, together with two tags $m_1 \in M_1, m_2 \in M_2$. (In our construction of deniable encryption, we use the first and the second messages of the deniable protocol as tags. This, again, is done to "tie" the level to the instance of the protocol).
- It should be possible to perform the following operations:
  1. Sample a single-tag level 0 for any tag m_1;
  2. Homomorphically increment the value inside any single-tag level (keeping its tag the same);
  3. Transform any single-tag level into a double-tag level, for any second tag m_2 (the value and the first tag remain the same);
  4. Compare two double-tag levels, as long as their both tags are the same;
  5. Given any level, retrieve its tag(s).

Notation. We use notation $[i, m_1]$ to denote a single-tag level with value i and tag m_1. We also use ℓ_i to denote a single-tag level with value i, when the tag is clear from the context.

We use notation $[i, m_1, m_2]$ to denote a double-tag level with value i and tags m_1, m_2. We also use L_i to denote a double-tag level with value i, when its tags are clear from the context.

Security property. The security requirement of a level system is that it should be hard to distinguish between $\ell_0^* = [0, m_1^*], L_0^* = [0, m_1^*, m_2^*]$ and $\ell_1^* = [1, m_1^*], L_0^* = [0, m_1^*, m_2^*]$, even given (limited) ability to perform

[25] Note that even though we call it encryption, we don't require this primitive to have decryption.

homomorphic operations described above.

This will be used in the proof of security of deniable encryption scheme as follows. Recall that in that proof we need to start with the real transcript and real randomness s, r (having levels L_0^*, ℓ_0^*, L_0^*, respectively) and eventually switch to the (same) real transcript but fake randomness s', r' (with levels L_0^*, ℓ_1^*, L_0^*). We can use security of the level system in the proof of deniable encryption as follows: given challenge ℓ_b^*, L_0^* (where $\ell_b^* = [b, m_1^*]$, $b \in \{0,1\}$, $L_0^* = [0, m_1^*, m_2^*]$), we use ℓ_b^* inside fake s and we use L_0^* inside the transcript and fake r. Since security of levels only holds when programs are punctured, in the proof of deniable encryption we first move to a hybrid with only punctured level programs, and then invoke security of the level system.

7.1 Definition

We start with describing the syntax of a level system for tag space M and upper bound T:

- $\mathsf{Setup}(1^\lambda; T; \mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}; r_{\mathsf{Setup}}) \rightarrow \mathsf{PP} = (\mathsf{P}_{\mathsf{GenZero}}, \mathsf{P}_{\mathsf{Increment}}, \mathsf{P}_{\mathsf{Transform}}, \mathsf{P}_{\mathsf{isLess}}, \mathsf{P}_{\mathsf{RetrieveTag}}, \mathsf{P}_{\mathsf{RetrieveTags}})$ is a randomized algorithm which takes as input security parameter, the largest allowed level T, description of programs, and randomness. It uses random coins to sample all necessary keys for each program[26], and outputs those programs obfuscated under iO.
- $\mathsf{GenZero}(m_1) \rightarrow \ell$ is a deterministic algorithm which takes message $m_1 \in M$ as input and outputs a string $\ell = [0, m_1]$, which is a single-tag level with tag m_1 and value 0. We also require that there exists a punctured version of this algorithm denoted $\mathsf{GenZero}[m_1^*](m_1)$ which outputs $'\mathsf{fail}'$ on input m_1^*.
- $\mathsf{Increment}(\ell) \rightarrow \ell'$ is a deterministic algorithm which takes a single-tag level $\ell = [i, m_1]$ for some $0 \leq i \leq T-1, m_1 \in M$, and outputs a single-tag level with the same tag and incremented value, i.e. $\ell' = [i+1, m_1]$. If $i \geq T$, it instead outputs $'\mathsf{fail}'$.
- $\mathsf{Transform}(\ell, m_2) \rightarrow \ell$ is a deterministic algorithm which takes a single-tag level $\ell = [i, m_1]$ for some $0 \leq i \leq T, m_1 \in M$, and some message $m_2 \in M$, and outputs $L = [i, m_1, m_2]$, which is a double-tag level with tags m_1, m_2, and value i. We also require that there exists a punctured version of this algorithm denoted $\mathsf{Transform}[(\ell^*, m_2^*)](\ell, m_2)$ which outputs $'\mathsf{fail}'$ on input (ℓ^*, m_2^*).
- $\mathsf{isLess}(L', L'') \rightarrow \mathsf{out} \in \{\mathsf{true}, \mathsf{false}\}$ is a deterministic algorithm which takes as input two double-tag levels $L' = [i', m_1', m_2']$ and $L'' = [i'', m_1'', m_2'']$. If $(m_1', m_2') \neq (m_1'', m_2'')$, then it outputs $'\mathsf{fail}'$. Otherwise it outputs true if $i' < i''$ and false if $i' \geq i''$.
- $\mathsf{RetrieveTag}(\ell) \rightarrow m_1$ is a deterministic algorithm which takes a single-tag level ℓ and outputs its tag.
- $\mathsf{RetrieveTags}(L) \rightarrow (m_1, m_2)$ is a deterministic algorithm which takes a double-tag level L and outputs both tags.

We emphasize that all programs except Setup are deterministic.

[26]We assume that Setup is implicitly given generation algorithms for all underlying primitives of the programs.

Definition 11. *A tuple of parametrized, deterministic*[27] *algorithms*

$$(\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}, \mathsf{GenZero}[m_1^*], \mathsf{Transform}[l^*, m_2^*])$$

is a level system *for tag space M, if algorithms have syntax described above, and the correctness and security properties described below hold.*

Notation: *Let T be superpolynomial in λ, and* $\mathsf{PP} = (\mathsf{P}_{\mathsf{GenZero}}, \mathsf{P}_{\mathsf{Increment}}, \mathsf{P}_{\mathsf{Transform}}, \mathsf{P}_{\mathsf{isLess}}, \mathsf{P}_{\mathsf{RetrieveTag}}, \mathsf{P}_{\mathsf{RetrieveTags}}) \leftarrow \mathsf{Setup}(1^\lambda; T; \mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ *for randomly chosen r_{Setup}.*

Next, let $m_1^* \in M$, $m_2^* \in M$, and let ℓ^* be an arbitrary string (not necessarily a level). Let $\mathsf{PP}' = (\mathsf{P}'_{\mathsf{GenZero}}, \mathsf{P}'_{\mathsf{Increment}}, \mathsf{P}'_{\mathsf{Transform}}, \mathsf{P}'_{\mathsf{isLess}}, \mathsf{P}'_{\mathsf{RetrieveTag}}, \mathsf{P}'_{\mathsf{RetrieveTags}}) \leftarrow \mathsf{Setup}(1^\lambda, T, \mathsf{GenZero}[m_1^*], \mathsf{Increment}, \mathsf{Transform}[(\ell^*, m_2^*)], \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ *with the same randomness r_{Setup} as above.*

For any fixed r_{Setup} consider the following notation:

- *For every $m_1 \in M$ denote $[0, m_1] = \mathsf{P}_{\mathsf{GenZero}}(m_1)$;*
- *For every $m_1 \in M$, $1 \le i \le T$ denote $[i, m] = \mathsf{P}_{\mathsf{Increment}}([i-1, m])$;*
- *For every $m_2 \in M$ and every $[i, m_1]$, where $0 \le i \le T, m_1 \in M$, denote $[i, m_1, m_2] = \mathsf{P}_{\mathsf{Transform}}([i, m_1], m_2)$.*

Correctness: *The following properties should hold, except with negligible probability over the choice of r_{Setup}:*

- **Uniqueness of levels:**
  - *For all $\ell \notin \{[i, m_1] : 0 \le i \le T, m_1 \in M\}$:*
    - $\mathsf{P}_{\mathsf{Increment}}(\ell) = \mathsf{'fail'}$;
    - $\mathsf{P}_{\mathsf{Transform}}(\ell, m_2) = \mathsf{'fail'}$ *for any $m_2 \in M$;*
    - $\mathsf{P}_{\mathsf{RetrieveTag}}(\ell) = \mathsf{'fail'}$.
  - *For all $L \notin \{[i, m_1, m_2] : 0 \le i \le T, m_1 \in M, m_2 \in M\}$:*
    - $\mathsf{P}_{\mathsf{isLess}}(L, L') = \mathsf{'fail'}$, $\mathsf{P}_{\mathsf{isLess}}(L', L) = \mathsf{'fail'}$, *for any string L';*
    - $\mathsf{P}_{\mathsf{RetrieveTags}}(L) = \mathsf{'fail'}$.
- **Upper bound is respected:** *For every $m_1 \in M$* $\mathsf{P}_{\mathsf{Increment}}([T, m_1]) = \mathsf{'fail'}$.
- **Correctness of comparison:** *For every $m_1, m_2 \in M$ and for every $0 \le i, j \le T$:*
  - $\mathsf{P}_{\mathsf{isLess}}([i, m_1, m_2], [j, m_1, m_2]) = \mathsf{true}$ *for $i < j$,*
  - $\mathsf{P}_{\mathsf{isLess}}([i, m_1, m_2], [j, m_1, m_2]) = \mathsf{false}$ *for $i \ge j$.*

[27] We prefer to use the notion of parametrized, deterministic algorithms to keep the definition simple. To formally define this notion, consider a randomized Turing machine with the restriction that the number of random bits written on its random tape is fixed and independent of the input (only dependent on security parameter λ). Such a Turing machine can first use these random coins to generate all necessary parameters (e.g., keys) and then run the actual code of the algorithm using generated parameters. In particular, we assume that this TM has the code of all necessary generation algorithms.

- **Comparison is possible only on matching levels:** *If* $(m_1', m_2') \neq (m_1'', m_2'')$*, then* $\mathsf{P}_{\mathsf{isLess}}([i, m_1', m_2'], [j, m_1'', m_2'']) = '\mathsf{fail}'$ *for all* i, j.
- **Correctness of tags retrieval:** *For every* $m_1, m_2 \in M$ *and for every* $0 \leq i \leq T$*:*
  - $\mathsf{P}_{\mathsf{RetrieveTag}}([i, m_1]) = m_1$,
  - $\mathsf{P}_{\mathsf{RetrieveTags}}([i, m_1, m_2]) = (m_1, m_2)$.
- **Functionality is preserved under puncturing:**
  - $\mathsf{P}_{\mathsf{GenZero}}(m) = \mathsf{P}'_{\mathsf{GenZero}}(m)$ *for all* $m \in M$, $m \neq m_1^*$*;*
  - $\mathsf{P}_{\mathsf{Increment}}(\ell) = \mathsf{P}'_{\mathsf{Increment}}(\ell)$ *for all strings* ℓ*;*
  - $\mathsf{P}_{\mathsf{Transform}}(\ell, m_2) = \mathsf{P}'_{\mathsf{Transform}}(\ell, m_2)$ *for all strings* l *and for all* $m_2 \in M$*, except* (ℓ^*, m_2^*)*;*
  - $\mathsf{P}_{\mathsf{isLess}}(L', L'') = \mathsf{P}'_{\mathsf{isLess}}(L'', L'')$ *for all strings* L', L''*;*
  - $\mathsf{P}_{\mathsf{RetrieveTag}}(\ell) = \mathsf{P}'_{\mathsf{RetrieveTag}}(\ell)$ *for all strings* ℓ*;*
  - $\mathsf{P}_{\mathsf{RetrieveTags}}(L) = \mathsf{P}'_{\mathsf{RetrieveTags}}(L)$ *for all strings* L.

Note that it follows from the correctness properties that $[i, m_1] = [i', m_1']$ *if and only* $(i, m_1) = (i', m_1')$*, and* $[i, m_1, m_2] = [i', m_1', m_2']$ *if and only* $(i, m_1, m_2) = (i', m_1', m_2')$.

Security: *For any* $m_1^* \in M, m_2^* \in M$*, the following distributions are computationally indistinguishable:*

$$(\ell_0^*, L_0^*, \mathsf{PP}_0) \approx (\ell_1^*, L_0^*, \mathsf{PP}_1),$$

where r_{Setup} *is randomly chosen,* $\mathsf{PP} = (\mathsf{P}_{\mathsf{GenZero}}, \mathsf{P}_{\mathsf{Increment}}, \mathsf{P}_{\mathsf{Transform}}, \mathsf{P}_{\mathsf{isLess}}, \mathsf{P}_{\mathsf{RetrieveTag}}, \mathsf{P}_{\mathsf{RetrieveTags}}) \leftarrow \mathsf{Setup}(\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}; r_{\mathsf{Setup}})$,

$\ell_0^* \leftarrow \mathsf{P}_{\mathsf{GenZero}}(m_1^*)$, $\ell_1^* \leftarrow \mathsf{P}_{\mathsf{Increment}}(\ell_0^*)$, $L_0^* \leftarrow \mathsf{P}_{\mathsf{Transform}}(\ell_0^*, m_2^*)$,

$\mathsf{PP}_b \leftarrow \mathsf{Setup}(\mathsf{GenZero}[m_1^*], \mathsf{Increment}, \mathsf{Transform}[(\ell_b^*, m_2^*)], \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}; r_{\mathsf{Setup}})$.

7.2 Construction

We implement a level system in a natural way: we let levels to be ciphertexts (encrypting the value and the tag in a single-tag level, and the value and both tags in a double-tag level) under special encryption scheme called asymmetric constrained encryption, or ACE (8). For single-tag and double-tag levels we use two different instances of ACE, with keys $\mathsf{EK}_1, \mathsf{DK}_1$ for single-tag levels and $\mathsf{EK}_2, \mathsf{DK}_2$ for double-tag levels. We let programs of the level system (fig. 20) perform required "homomorphic" operations in a natural way, by decrypting the ciphertext and learning its value and tag, checking validity of the operation, and then outputting the result (reencrypted, when applicable).

Theorem 3. *Let:*

- λ *be a security parameter;*
- iO *be* $(\mathsf{poly}(\lambda), 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))})$*-secure indistinguishability obfuscation;*
- ACE *be an asymmetric constrained encryption scheme with* $(\mathsf{poly}(\lambda), 2^{-\Omega(\nu_{\mathsf{ACE.Indist}}(\lambda))})$*-secure indistinguishability of ciphertexts and* $(\mathsf{poly}(\lambda), 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))})$ *security of decryption;*

- *g be a $(2^{O(\nu_{\mathsf{OWF}}(\lambda'))}, 2^{-\Omega(\nu_{\mathsf{OWF}}(\lambda'))})$-secure injective one-way function mapping $\lambda' = \log T(\lambda)$-bit inputs to $\mathsf{poly}(\lambda')$-bit outputs;*
- *$\gamma(\lambda)$ be a function satisfying the following conditions:*
  - *$\gamma(\lambda) = O(\nu_{\mathsf{iO}}(\lambda))$;*
  - *$2^{\gamma(\lambda)}\mathsf{poly}(\lambda)\log T = O(2^{\nu_{\mathsf{OWF}}(\log T)})$;*

Then the scheme described on fig. 20 is a level system for upper bound $T(\lambda)$, tags of length $\tau(\lambda)$, which is $(\mathsf{poly}(\lambda), 2^{-\nu_{\mathsf{levels}}(\lambda)})$-secure, where $2^{-\nu_{\mathsf{levels}}(\lambda)}$ is equal to the following:

$$2^{-\Omega(\gamma(\lambda))}+T^{-1}(\lambda)+T(\lambda)2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}+2^{\tau(\lambda)}(T(\lambda)\cdot 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}+T(\lambda)\cdot 2^{-\Omega(\nu_{\mathsf{ACE.Indist}}(\lambda))}+2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))})).$$

Note: Here $\gamma(\lambda)$ represents distinguishing advantage between two obfuscated programs differing on one input (which is a preimage of the OWF g). The two conditions on γ are set to satisfy the requirements of theorem 1, and say that the inverter's size is small enough, and that distinguishing advantage is big enough compared to the indistinguishability guarantee of iO.

By using subexponentially-secure primitives, we obtain the following corollary:
Corollary 2. *Let:*

- *λ be a security parameter;*
- *iO be $(\mathsf{poly}(\lambda), 2^{-\Omega(\lambda^{\varepsilon})})$-secure indistinguishability obfuscation;*
- *ACE be an asymmetric constrained encryption scheme with $(\mathsf{poly}(\lambda), 2^{-\Omega(\lambda^{\varepsilon})})$-secure indistinguishability of ciphertexts and $(\mathsf{poly}(\lambda), 2^{-\Omega(\lambda^{\varepsilon})})$ security of decryption;*
- *g be a $(2^{\Omega(\lambda'^{\varepsilon})}, 2^{-\Omega(\lambda'^{\varepsilon})})$-secure injective one-way function mapping $\lambda' = \lambda^{\varepsilon/2}$-bit inputs to $\mathsf{poly}(\lambda')$-bit outputs;*
- *$\gamma(\lambda) = \lambda^{\varepsilon^2/2}$;*

Then the scheme described on fig. 20 is a level system for upper bound $T(\lambda) = 2^{\lambda^{\varepsilon/2}}$, tags of length $\tau(\lambda) = \lambda^{\varepsilon/2}$, which is $(\mathsf{poly}(\lambda), 2^{-\Omega(\lambda^{\varepsilon^2/2})})$-secure.

7.3 Overview of the proof

Correctness. Correctness properties of our level scheme immediately follow from statistical correctness of iO and correctness and uniqueness properties of ACE.

Overview of security proof. For security, we first informally describe the structure of the proof, and then give the sequence of hybrids in section 7.4 and security reductions in section 7.5. Recall that security definition requires that $(\ell_0^*, L_0^*, \mathsf{PP}_0) \approx (\ell_1^*, L_0^*, \mathsf{PP}_1)$, where PP_b are punctured, obfuscated programs. Starting from the distribution $(\ell_0^*, L_0^*, \mathsf{PP}_0)$, our proof proceeds in 3 major steps:

1. **Switching from $\boldsymbol{\ell_0^* = [0, m_1^*]}$ to $\boldsymbol{\ell_1^* = [1, m_1^*]}$.** Programs GenZero and Increment define a chain $[0, m_1] \to [1, m_1] \to \ldots \to [T, m_1] \to \perp$ for each tag m_1. In a sequence of hybrids we switch from $[0, m_1^*]$ to $[1, m_1^*]$ by switching the whole chain from $[0, m_1^*] \to [1, m_1^*] \to \ldots \to [T, m_1^*] \to \perp$ to $[1, m_1^*] \to [2, m_1^*] \to \ldots \to [T+1, m_1^*] \to \perp$.

Program GenZero(m_1)
Inputs: tag $m_1 \in M$.
Hardwired values: encryption key EK_1 of ACE.
1. output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1)$.

Program Increment(l)
Inputs: single-tag level l
Hardwired values: encryption and decryption keys $\mathsf{EK}_1, \mathsf{DK}_1$ of ACE, upper bound T.
1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = \mathsf{'fail'}$ then output $\mathsf{'fail'}$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $\mathsf{'fail'}$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(i + 1, m_1)$.

Program Transform(l, m_2)
Inputs: single-tag level l, tag $m_2 \in M$
Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, upper bound T.
1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = \mathsf{'fail'}$ then output $\mathsf{'fail'}$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $\mathsf{'fail'}$;
3. output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program isLess(L', L'')
Inputs: double-tag levels L', L''
Hardwired values: decryption key DK_2 of ACE, upper bound T.
1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if $\mathsf{out}' = \mathsf{'fail'}$ then output $\mathsf{'fail'}$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if $\mathsf{out}'' = \mathsf{'fail'}$ then output $\mathsf{'fail'}$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or or $i' < 0$ or $i'' < 0$ $(m_1', m_2') \neq (m_1'', m_2'')$ then output $\mathsf{'fail'}$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTag(l)
Inputs: single-tag level l
Hardwired values: decryption key DK_1 of ACE, upper bound T.
1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = \mathsf{'fail'}$ then output $\mathsf{'fail'}$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $\mathsf{'fail'}$;
3. Output m_1.

Program RetrieveTags(L)
Inputs: double-tag level L
Hardwired values: decryption key DK_2 of ACE, upper bound T.
1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if $\mathsf{out} = \mathsf{'fail'}$ then output $\mathsf{'fail'}$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $\mathsf{'fail'}$;
3. Output m_1, m_2.

Figure 20: Programs in our level system

As a result of this change, ℓ_0^* is switched to ℓ_1^* as desired (and in particular, the punctured point in Transform is switched from ℓ_0^* to ℓ_1^* as well). However, this change also affects the programs in the following two ways (resulting programs are in fig. 22) :

- **Wrong upper bound:** programs Increment, Transform, and RetrieveTag now have an upper bound $T+1$ (instead of T) for the case $m_1 = m_1^*$,
- **Incorrect reencryption:** program Transform, given $[i, m_1^*]$ for $0 \leq i \leq T+1$, outputs $[i - 1, m_1^*, m_2]$ instead of $[i, m_1^*, m_2]$.

2. **Restoring correct upper bound in Increment, Transform, and RetrieveTag.** In a sequence of hybrids we change the wrong upper bound $T+1$ to the correct upper bound T in relevant programs.

   Resulting programs are in fig. 23. This part of the proof uses ideas from [BPR15] to argue that the adversary can never reach the upper bound and thus the upper bound can be decreased by 1 indistinguishably.

3. **Restoring correct reencryption in Transform.** In a sequence of hybrids we make program Transform output the correct value $[i, m_1^*, m_2]$, instead of $[i-1, m_1^*, m_2]$, for all $0 \leq i \leq T$ and for all m_2.

   The proof of this step follows a by-now-standard puncturing technique (which allows to change the ciphertext in a PRF-based encryption from one plaintext to another), except that we also have to deal with program isLess which has decryption keys inside it. Intuitively, the proof still goes through even despite those decryption keys, because isLess only reveals the result of the comparison, which is not affected by our change.

   At the end of this step, we obtain original punctured programs, thus proving security of our level system.

Security loss. Steps 1 and 2 require number of hybrids proportional to the upper bound T, and step 3 requires number of hybrids proportional to $2^{|m_2|}T$. In addition, in the proof of step 2 we also lose $1/T$, thus requiring T and $2^{|m_2|}$ to be superpolynomial.

Now we describe the proof in each step in more detail. While the reader can safely skip this part and directly go to the list of hybrids (section 7.4), we suggest that the readers familiar with iO techniques take a look at this informal presentation first, since it outlines, in a succinct way, the logic behind the somewhat lengthy sequence of hybrids.

Step 1: Switching ℓ^* from $[0, m_1^*]$ to $[1, m_1^*]$.

1. We first change the chain to $[0, m_1^*] \to [1, m_1^*] \to \ldots \to [T-1, m_1^*] \to [T+1, m_1^*] \to \perp$, creating a gap between $T-1$ and $T+1$. This is done by first hardwiring the ciphertext $l_T^* = [T, m_1^*]$ into relevant programs, then puncturing keys corresponding to both $[T, m_1^*]$ and $[T+1, m_1^*]$ (the latter can be punctured since they are never used due to upper bound T), and finally switching hardwired ciphertext to $l_{T+1}^* = [T+1, m_1^*]$ and unpuncturing keys at $[T+1, m_1^*]$[28].

   Note that the keys remain punctured at the point $[T, m_1^*]$, which essentially means that from the point of view of programs there doesn't exist a valid encryption of (T, m_1^*).

   Finally, note that switching the hardwired ciphertext from $[T, m_1^*]$ to $[T+1, m_1^*]$ changes the upper

[28]Note that it is crucial for switching the ciphertext that keys are punctured at *both* points, and only one of the two ciphertexts is present in the distribution.

bound from T to $T+1$ in programs Transform and RetrieveTag.

2. Then in a sequence of hybrids we move the gap from T down to 0 a follows. Let j-th hybrid be a hybrid where the gap is at $j+1$, i.e. Increment defines a chain $[0, m_1^*] \to [1, m_1^*] \to \ldots \to [j, m_1^*] \to [j+2, m_1^*] \to \ldots \to [T, m_1^*] \to [T+1, m_1^*]$, and keys are punctured at $[j+1, m_1^*]$, meaning that there doesn't exist a valid encryption of $(j+1, m_1^*)$. We move the gap to j by first hardwiring the ciphertext $l_j^* = [j, m_1^*]$ into relevant programs, then puncturing keys corresponding to $[j, m_1^*]$ (recall that keys are already punctured at $[j+1, m_1^*]$), and finally switching hardwired ciphertext to $l_{j+1}^* = [j+1, m_1^*]$ and unpuncturing keys at $[j+1, m_1^*]$.

   Note that the keys remain punctured at the point $[j, m_1^*]$, enabling the next step.

   In addition, note that in the first step the upper bound in Increment is switched from T to $T+1$. This is due to the fact that this step switches the hardwired ciphertext from $[T-1, m_1^*]$ to $[T, m_1^*]$, and due to the fact that there is a hardwired instruction to output $[T+1, m_1^*]$, given hardwired ciphertext as input (indeed, while in the original Increment input $[T, m_1^*]$ results in $\perp$, after the change input $[T, m_1^*]$ results in $[T+1, m_1^*]$).

   Finally, note that the last step switches challenge level $\ell_0^* = [0, m_1^*]$ to $\ell_1^* = [1, m_1^*]$.

3. As a result, we obtain Increment which defines a chain $1 \to 2 \to \ldots \to T \to T+1 \to \perp$ for the tag m_1^*, and keys are punctured at $[0, m_1^*]$. We remove the puncturing using the fact that keys for $[0, m_1^*]$ are never used, since GenZero doesn't have to work on input m_1^*.

Resulting programs are in fig. 22.

Step 2: Restoring the correct upper bound of Increment, Transform, and RetrieveTag on m_1^*. Intuitively, nobody can tell whether these programs have an upper bound T or $T+1$, since the only way to test this is to check if, starting with level $[1, m_1^*]$, Increment fails after $T-1$ or T executions, which requires superpolynomial time to compute. To turn this intuition into a formal argument, we follow the proof of [BPR15]:

1. We cut the chain $1 \to 2 \to \ldots \to T \to T+1 \to \perp$ (here we omit the tag m_1^* for simplicity and compactness) at a random point as follows. We add a check "if $\mathsf{prg}(i) = S$ then abort" to Increment, where S is randomly chosen. If the prg is expanding enough, then with overwhelming probability S is outside of the prg image, and adding this line doesn't change the functionality. However, next we change S to be $\mathsf{prg}(s)$ for some random s, which cuts the line at point s: that is, Increment now defines the chain $1 \to \ldots s \to \perp, s+1 \to \ldots \to T+1 \to \perp$.

2. In a sequence of hybrids we cut the line in all points after s, obtaining the following chain: $1 \to \ldots \to s \to \perp$, $s+1 \to \perp$, $s+2 \to \perp$, ..., $T \to \perp$, $T+1 \to \perp$. Intuitively, once Increment outputs $\perp$ given $[s, m_1^*]$, it becomes impossible for an adversary to obtain $[s+1, m_1^*]$, and therefore behavior of Increment at $[s+1, m_1^*]$ can be changed to $\perp$ as well. The process can be continued. This intuition is captured by the security of constrained decryption of ACE.

   As the result, we move to a hybrid where valid encryptions of $(s+1, m_1^*), \ldots, (T+1, m_1^*)$ do not exist.

3. Then we can move the upper bound from $T+1$ back to T for the case $m_1 = m_1^*$, since programs output $\perp$ on input $[T+1, m_1^*]$ anyway. Thus, changing $T+1$ to T doesn't affect the functionality of the programs.

4. Then we can reverse all previous steps, restore the chain and eventually get original programs with correct upper bound T (except Transform, which now has the correct upper bound T, but still has incorrect behavior on inputs of the form $([i, m_1^*], m_2)$).

Resulting programs are in fig. 23.

Step 3: Restoring the correct reencryption behaviour in Transform. Note that $\mathsf{Transform}_B$ (fig. 23) defines the set of outputs $[0, m_1, m_2], \ldots, [T, m_1, m_2]$ (corresponding to inputs $([0, m_1], m_2), \ldots, ([T, m_1], m_2))$ for the case $m_1 \neq m_1^*$, and the set of outputs $[-1, m_1^*, m_2], \ldots, [T - 1, m_1^*, m_2]$ (corresponding to inputs $([0, m_1^*], m_2), \ldots, ([T, m_1^*], m_2))$ for the case $m_1 \neq m_1^*$. We change the set of outputs from $[-1, m_1^*, m_2], \ldots, [T-1, m_1^*, m_2]$ to $[0, m_1^*, m_2], \ldots, [T, m_1^*, m_2]$ by running the following sequence of steps for each possible second tag m_2:

1. We first change the set of outputs from $[-1, m_1^*, m_2], \ldots, [T-1, m_1^*, m_2]$ to $[-1, m_1^*, m_2], \ldots, [T - 2, m_1^*, m_2], [T, m_1^*, m_2]$, creating a gap between $T-2$ and T. This is done by first hardwiring the ciphertext $L_{T-1}^* = [T-1, m_1^*, m_2]$ into relevant programs (Transform, isLess, and RetrieveTags), then puncturing keys corresponding to both $[T-1, m_1^*, m_2]$ and $[T, m_1^*, m_2]$ (the latter can be punctured since they are never used due to the upper bound T), and finally switching hardwired ciphertext to $L_T^* = [T, m_1^*, m_2]$ and unpuncturing keys at $[T, m_1^*, m_2]$[29].

   Note that the keys remain punctured at the point $[T-1, m_1^*, m_2]$, which essentially means that from the point of view of programs there doesn't exist a valid encryption of $(T-1, m_1^*, m_2)$.

2. Then in a sequence of hybrids we move the gap from $T-1$ down to -1 a follows. Let j-th hybrid be a hybrid where the gap is at $j+1$, i.e. Transform outputs $[-1, m_1^*, m_2], \ldots, [j, m_1^*, m_2], [j + 2, m_1^*, m_2], \ldots, [T, m_1^*, m_2]$, and keys are punctured at $[j+1, m_1^*, m_2]$, meaning that there doesn't exist a valid encryption of $(j+1, m_1^*, m_2)$. We move the gap to j by first hardwiring the ciphertext $L_j^* = [j, m_1^*, m_2]$ into relevant programs, then puncturing keys corresponding to $[j, m_1^*, m_2]$ (recall that keys are already punctured at $[j+1, m_1^*, m_2]$), and finally switching hardwired ciphertext to $L_{j+1}^* = [j+1, m_1^*, m_2]$ and unpuncturing keys at $[j+1, m_1^*, m_2]$.

   Note that the keys remain punctured at the point $[j, m_1^*, m_2]$, enabling the next step.

   An important property of program isLess which enables switching $[j, m_1^*, m_2]$ to $[j+1, m_1^*, m_2]$ at each step is that **isLess treats both $[j, m_1^*, m_2]$ and $[j+1, m_1^*, m_2]$ in the same way**. That is, both $[j, m_1^*, m_2]$ and $[j+1, m_1^*, m_2]$ are larger than $[0, m_1^*, m_2], \ldots, [j-1, m_1^*, m_2]$, and both are smaller than $[j+2, m_1^*, m_2], \ldots, [T, m_1^*, m_2]$. Finally, both are equal when compared to themselves. The only difference in the output could have occured on inputs $([j, m_1^*, m_2], [j+1, m_1^*, m_2])$ (resulting in isLess returning true) and $([j+1, m_1^*, m_2], [j, m_1^*, m_2])$ (resulting in isLess returning false); however, in each of the two hybrids only one of the two values "exists" and the other is punctured out, thus forcing isLess to output $\perp$ on these inputs. This allows us to "swap" $[j, m_1^*, m_2]$ and $[j+1, m_1^*, m_2]$ without changing the functionality of the programs.

   Finally, note that we don't perform two last steps, i.e. switching from 0 to 1 and from -1 to 0, for the case $m_2 = m_2^*$ (indeed, that would switch the challenge value from $L_0^* = [0, m_1^*, m_2^*]$ to $L_1^* = [1, m_1^*, m_2^*]$, but it has to remain $L_0^* = [0, m_1^*, m_2^*]$ in both experiments of the security game). In fact, we don't have to switch from 0 to 1 since Transform is punctured at $[l_1^*, m_2^*]$ and outputs $'\mathsf{fail}'$ on

[29] Note that it is crucial for switching the ciphertext that keys are punctured at *both* points, and only one of the two ciphertexts is present in the distribution.

this input anyway. Further, since $[0, m_1^*]$ is hard to obtain for the adversary, we argue that $\mathsf{Transform}$ may be indistinguishably changed from outputting $[-1, m_1^*, m_2^*]$ to $[0, m_1^*, m_2^*]$ on input $[0, m_1^*], m_2^*$ (again, this intuition is formalized using security of the constrained key of the ACE).

Programs in Hyb_A

Program GenZero$[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key EK_1 of ACE, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1)$.

Program Increment(l)

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1, \mathsf{DK}_1$ of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(i+1, m_1)$.

Program Transform$[(l_0^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_0^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program isLess(L', L'')

Inputs: double-tag levels L', L''

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTag(l)

Inputs: single-tag level l

Hardwired values: decryption key DK_1 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Program RetrieveTags(L)

Inputs: double-tag level L

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1, m_2.

Figure 21: Programs in Hyb_A. In addition, in this hybrid the adversary gets $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in Hyb_B

Program GenZero$_B[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key EK_1 of ACE, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1)$.

Program Increment$_B(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1, \mathsf{DK}_1$ of ACE, tag m_1^*, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i \geq T+1$ or $i < 0$) then output $'\mathsf{fail}'$;
3. If $m_1 \neq m_1^*$ and ($i \geq T$ or $i < 0$) then output $'\mathsf{fail}'$;
4. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(i+1, m_1)$.

Program Transform$_B[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $m_1 = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then return $'\mathsf{fail}'$;
   (b) return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
4. If $m_1 \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
   (b) return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program isLess$_B(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTag$_B(l)$

Inputs: single-tag level l

Hardwired values: decryption key DK_1 of ACE, tag m_1^*, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i > T+1$ or $i < 0$) then output $'\mathsf{fail}'$;
3. If $m_1 \neq m_1^*$ and ($i > T$ or $i < 0$) then output $'\mathsf{fail}'$;
4. Output m_1.

Program RetrieveTags$_B(L)$

Inputs: double-tag level L

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1, m_2.

Figure 22: Programs in Hyb_B. In addition, in this hybrid the adversary gets $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in Hyb_C.

Program GenZero$_C[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key EK_1 of ACE, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1)$.

Program Increment$_C(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1, \mathsf{DK}_1$ of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(i+1, m_1)$.

Program Transform$_C[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
5. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program isLess$_C(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTag$_C(l)$

Inputs: single-tag level l

Hardwired values: decryption key DK_1 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Program RetrieveTags$_C(L)$

Inputs: double-tag level L

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1, m_2.

Figure 23: Programs in Hyb_C. In addition, in this hybrid the adversary gets $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in Hyb_D

Program GenZero$[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key EK_1 of ACE, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1)$.

Program Increment(l)

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1, \mathsf{DK}_1$ of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(i+1, m_1)$.

Program Transform$[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program isLess(L', L'')

Inputs: double-tag levels L', L''

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTag(l)

Inputs: single-tag level l

Hardwired values: decryption key DK_1 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Program RetrieveTags(L)

Inputs: double-tag level L

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1, m_2.

Figure 24: Programs in Hyb_D. In addition, in this hybrid the adversary gets $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

7.4 List of hybrids

For any messages m_1^*, m_2^*, consider the following distributions for randomly chosen r_{Setup}:

- $\mathsf{Hyb}_A = (\mathsf{PP}, \ell_0^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*], \mathsf{Increment}, \mathsf{Transform}[\ell_0^*, m_2^*],$ $\mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ (fig. 21), $\ell_0^* = \mathsf{GenZero}(m_1^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, m_2^*)$.
- $\mathsf{Hyb}_B = (\mathsf{PP}, \ell_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_B[m_1^*], \mathsf{Increment}_B, \mathsf{Transform}_B[\ell_1^*, m_2^*],$ $\mathsf{isLess}_B, \mathsf{RetrieveTag}_B, \mathsf{RetrieveTags}_B; r_{\mathsf{Setup}})$ (fig. 22), $\ell_0^* = \mathsf{GenZero}(m_1^*)$, $\ell_1^* = \mathsf{Increment}(\ell_0^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, m_2^*)$.
- $\mathsf{Hyb}_C = (\mathsf{PP}, \ell_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_C[m_1^*], \mathsf{Increment}_C, \mathsf{Transform}_B[\ell_1^*, m_2^*],$ $\mathsf{isLess}_C, \mathsf{RetrieveTag}_C, \mathsf{RetrieveTags}_C; r_{\mathsf{Setup}})$ (fig. 23), $\ell_0^* = \mathsf{GenZero}(m_1^*)$, $\ell_1^* = \mathsf{Increment}(\ell_0^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, m_2^*)$.
- $\mathsf{Hyb}_D = (\mathsf{PP}, \ell_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*], \mathsf{Increment}, \mathsf{Transform}[\ell_1^*, m_2^*],$ $\mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ (fig. 24), $\ell_0^* = \mathsf{GenZero}(m_1^*)$, $\ell_1^* = \mathsf{Increment}(\ell_0^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, m_2^*)$.

Note that Hyb_A is the distribution from security game for $b = 0$ and Hyb_D is the distribution from security game for $b = 1$. To prove security of the level system, we need to show that $\mathsf{Hyb}_A \approx \mathsf{Hyb}_D$, which we do in the following lemmas:

Lemma 2. (Switching from ℓ_0^* to ℓ_1^*) *For any PPT adversary $\mathcal{A}$,*

$$\mathsf{adv}_{\mathsf{Hyb}_A,\mathsf{Hyb}_B}(\lambda) \le T \cdot 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))} + T \cdot 2^{-\Omega(\nu_{\mathsf{ACE.Indist}}(\lambda))} + 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}.$$

Lemma 3. (Changing the upper bound from $T+1$ to T) *For any PPT adversary $\mathcal{A}$,*

$$\mathsf{adv}_{\mathsf{Hyb}_B,\mathsf{Hyb}_C}(\lambda) \le 2^{-\Omega(\gamma(\lambda))} + \frac{1}{T} + T \cdot 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))} + T \cdot 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}.$$

Lemma 4. (Restoring behavior of Transform) *For any PPT adversary $\mathcal{A}$,*

$$\mathsf{adv}_{\mathsf{Hyb}_C,\mathsf{Hyb}_D}(\lambda) \le 2^{\tau(\lambda)}(T \cdot 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))} + T \cdot 2^{-\Omega(\nu_{\mathsf{ACE.Indist}}(\lambda))} + 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}).$$

7.4.1 Proof of lemma 2 (Switching from ℓ_0^* to ℓ_1^*).

As described earlier, we are going to shift levels $[i, m_1^*]$ to $[i+1, m_1^*]$ one by one, starting from $i = T$. We start from Hyb_A.

- $\mathsf{Hyb}_{A,1,1}$. We give the adversary $(\mathsf{PP}, l_0^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{A,1,1}[m_1^*],$ $\mathsf{Increment}_{A,1,1}$, $\mathsf{Transform}_{A,1,1}[(l_0^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{A,1,1}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 25.

  That is, we puncture ACE key EK_1 at point $p_{T+1} = (T+1, m_1^*)$ in programs Increment and GenZero, since these programs never run encryption on p_{T+1}. Indistinguishability holds by iO.

- $\mathsf{Hyb}_{A,1,2}$. We give the adversary $(\mathsf{PP}, l_0^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{A,1,2}[m_1^*],$ $\mathsf{Increment}_{A,1,2}$, $\mathsf{Transform}_{A,1,2}[(l_0^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{A,1,2}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 26.

That is, we puncture ACE key DK_1 at the same point $p_{T+1} = (T+1, m_1^*)$ in programs Increment, Transform, and RetrieveTag. Indistinguishability holds by security of constrained decryption of ACE, since corresponding encryption key is already punctured at p_{T+1}.

Next we consider the following sequence of hybrids for $j = T, \ldots, 1$. Programs for the case $j = T$ and $j = T-1$ are written separately in order to track how the upper bound in programs is changed from T to $T+1$.

- $\mathsf{Hyb}_{A,2,j,1}$. We give the adversary $(\mathsf{PP}, l_0^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{A,2,j,1}[m_1^*]$, $\mathsf{Increment}_{A,2,j,1}$, $\mathsf{Transform}_{A,2,j,1}[(l_0^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{A,2,j,1}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 26 (for the case $j = T$), fig. 29 (for $j = T-1$), fig. 33 (for $j = T-2, \ldots, 1$).

  That is, in this hybrid EK_1 and DK_1 are punctured at $p_{j+1} = (j+1, m_1^*)$. In addition, program Increment, given $[j, m_1^*]$, outputs $[j+2, m_1^*]$. Program Transform, given $([i, m_1^*], m_2)$ for $i > j$, outputs $[i-1, m_1^*, m_2]$.

  Note that $\mathsf{Hyb}_{A,2,j,1} = \mathsf{Hyb}_{A,1,2}$ for $j = T$.

- $\mathsf{Hyb}_{A,2,j,2}$. We give the adversary $(\mathsf{PP}, l_0^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{A,2,j,2}[m_1^*]$, $\mathsf{Increment}_{A,2,j,2}$, $\mathsf{Transform}_{A,2,j,2}[(l_0^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{A,2,j,2}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 27 (for the case $j = T$), fig. 30 (for $j = T-1$), fig. 34 (for $j = T-2, \ldots, 1$).

  That is, we additionally puncture ACE keys $\mathsf{EK}_1, \mathsf{DK}_1$ at the point $p_j = (j, m_1^*)$ and hardwire $l_j^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(j, m_1^*)$ to eliminate the need to encrypt or decrypt p_j in programs GenZero, Increment, Transform, and RetrieveTag. Indistinguishability holds by iO.

- $\mathsf{Hyb}_{A,2,j,3}$. We give the adversary $(\mathsf{PP}, l_0^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{A,2,j,3}[m_1^*]$, $\mathsf{Increment}_{A,2,j,3}$, $\mathsf{Transform}_{A,2,j,3}[(l_0^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{A,2,j,3}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 28 (for the case $j = T$), fig. 31 (for $j = T-1$), fig. 35 (for $j = T-2, \ldots, 1$).

  That is, we replace $l_j^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(j, m_1^*)$ with $l_{j+1}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(j+1, m_1^*)$ in programs Increment, Transform, and RetrieveTag. Indistinguishability holds by security of ACE for punctured points p_j, p_{j+1}.

- $\mathsf{Hyb}_{A,2,j,4}$. We give the adversary $(\mathsf{PP}, l_0^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{A,2,j,4}[m_1^*]$, $\mathsf{Increment}_{A,2,j,4}$, $\mathsf{Transform}_{A,2,j,4}[(l_0^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{A,2,j,4}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 29 (for the case $j = T$), fig. 32 (for $j = T-1$), fig. 36 (for $j = T-2, \ldots, 1$).

  That is, we unpuncture ACE keys $\mathsf{EK}_1, \mathsf{DK}_1$ at the point $p_{j+1} = (j+1, m_1^*)$ and remove hardwired $l_{j+1}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(j+1, m_1^*)$ in programs GenZero, Increment, Transform, and RetrieveTag. Indistinguishability holds by iO.

  Note that $\mathsf{Hyb}_{A,2,j,4} = \mathsf{Hyb}_{A,2,j-1,1}$ for $2 \leq j \leq T$.

Next we change l_0^* to l_1^* as follows:

- $\mathsf{Hyb}_{A,2,0,1}$. We give the adversary $(\mathsf{PP}, l_0^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{A,2,0,1}[m_1^*]$, $\mathsf{Increment}_{A,2,0,1}$, $\mathsf{Transform}_{A,2,0,1}[(l_0^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{A,2,0,1}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 37.

  That is, in this hybrid EK_1 and DK_1 are punctured at $p_1 = (1, m_1^*)$. In addition, program $\mathsf{Increment}$, given $[0, m_1^*]$, outputs $[2, m_1^*]$. Program $\mathsf{Transform}$, given $([i, m_1^*], m_2)$ for $i > 0$, outputs $[i - 1, m_1^*, m_2]$.

  Note that $\mathsf{Hyb}_{A,2,0,1} = \mathsf{Hyb}_{A,2,j,4}$ for $j = 1$.

- $\mathsf{Hyb}_{A,2,0,2}$. We give the adversary $(\mathsf{PP}, l_0^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{A,2,0,2}[m_1^*]$, $\mathsf{Increment}_{A,2,0,2}$, $\mathsf{Transform}_{A,2,0,2}[(l_0^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{A,2,0,2}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 38.

  That is, we additionally puncture ACE keys $\mathsf{EK}_1, \mathsf{DK}_1$ at the point $p_0 = (0, m_1^*)$ and hardwire $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$ to eliminate the need to encrypt or decrypt p_0 in programs $\mathsf{GenZero}$, $\mathsf{Increment}$, $\mathsf{Transform}$, and $\mathsf{RetrieveTag}$. Indistinguishability holds by iO.

- $\mathsf{Hyb}_{A,2,0,3}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{A,2,0,3}[m_1^*]$, $\mathsf{Increment}_{A,2,0,3}$, $\mathsf{Transform}_{A,2,0,3}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{A,2,0,3}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 39.

  That is, we replace $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$ with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$ in programs $\mathsf{Increment}$, $\mathsf{Transform}$, and $\mathsf{RetrieveTag}$, and give l_1^* instead of l_0^* to the adversary. Indistinguishability holds by security of ACE for punctured points p_0, p_1.

- $\mathsf{Hyb}_{A,3,1}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{A,3,1}[m_1^*]$, $\mathsf{Increment}_{A,3,1}$, $\mathsf{Transform}_{A,3,1}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{A,3,1}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 40.

  That is, we unpuncture ACE keys $\mathsf{EK}_1, \mathsf{DK}_1$ at the point $p_1 = (1, m_1^*)$ and remove hardwired $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$ in programs $\mathsf{GenZero}$, $\mathsf{Increment}$, $\mathsf{Transform}$, and $\mathsf{RetrieveTag}$. Indistinguishability holds by iO.

- $\mathsf{Hyb}_{A,3,2}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{A,3,2}[m_1^*]$, $\mathsf{Increment}_{A,3,2}$, $\mathsf{Transform}_{A,3,2}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{A,3,2}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 41.

  That is, we unpuncture ACE decryption key DK_1 at the point $p_0 = (0, m_1^*)$ in programs $\mathsf{Increment}$, $\mathsf{Transform}$, and $\mathsf{RetrieveTag}$. Indistinguishability holds by security of constrained decryption of ACE, since corresponding encryption key is punctured at p_0.

- $\mathsf{Hyb}_{A,3,3}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{A,3,3}[m_1^*]$, $\mathsf{Increment}_{A,3,3}$, $\mathsf{Transform}_{A,3,3}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{A,3,3}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 42.

That is, we unpuncture ACE encryption key EK_1 at the point $p_0 = (0, m_1^*)$ in programs $\mathsf{GenZero}$, $\mathsf{Increment}$. Indistinguishability holds by iO, since these programs never encrypt p_0.

Note that $\mathsf{Hyb}_{A,3,3}$ is the same as Hyb_B.

Thus, the the advantage of the PPT adversary in distinguishing between Hyb_A and Hyb_B is at most

$$(2T+4) \cdot 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))} + (T+1) \cdot 2^{-\Omega(\nu_{\mathsf{ACE.Indist}}(\lambda))} + 2 \cdot 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))} =$$

$$T \cdot 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))} + T \cdot 2^{-\Omega(\nu_{\mathsf{ACE.Indist}}(\lambda))} + 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}.$$

Programs in $\mathsf{Hyb}_{A,1,1}$

Program GenZero$_{A,1,1}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_{T+1}\}$ of ACE punctured at the point $p_{T+1} = (T+1, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Return $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{T+1}\}}(0, m_1)$.

Program Increment$_{A,1,1}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_{T+1}\}, \mathsf{DK}_1$ of ACE punctured at $p_{T+1} = (T+1, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Return $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{T+1}\}}(i+1, m_1)$.

Program Transform$_{A,1,1}[(l_0^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_0^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{A,1,1}(l)$

Inputs: single-tag level l

Hardwired values: decryption key DK_1 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Return m_1.

Figure 25: Programs in $\mathsf{Hyb}_{A,1,1}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and RetrieveTags, together with $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{A,1,2}$ (same as $\mathsf{Hyb}_{A,2,T,1}$)

Program GenZero$_{A,2,T,1}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_{T+1}\}$ of ACE punctured at the point $p_{T+1} = (T+1, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output 'fail';
2. Return $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{T+1}\}}(0, m_1)$.

Program Increment$_{A,2,T,1}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_{T+1}\}, \mathsf{DK}_1\{p_{T+1}\}$ of ACE punctured at $p_{T+1} = (T+1, m_1^*)$, upper bound T.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_{T+1}\}}(l)$; if out = 'fail' then return 'fail'; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then return 'fail';
3. Return $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{T+1}\}}(i+1, m_1)$.

Program Transform$_{A,2,T,1}[(l_0^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_{T+1}\}$ of ACE punctured at the point $p_{T+1} = (T+1, m_1^*)$, encryption key EK_2 of ACE, single-tag level $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_0^*, m_2^*)$ then return 'fail';
2. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_{T+1}\}}(l)$; if out = 'fail' then return 'fail'; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return 'fail';
4. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{A,2,T,1}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_{T+1}\}$ of ACE punctured at the point $p_{T+1} = (T+1, m_1^*)$, upper bound T.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_{T+1}\}}(l)$; if out = 'fail' then output 'fail'; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then return 'fail';
3. Return m_1.

Figure 26: Programs in $\mathsf{Hyb}_{A,1,2}$ (same as $\mathsf{Hyb}_{A,2,T,1}$). In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and RetrieveTags, together with $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{A,2,T,2}$

Program GenZero$_{A,2,T,2}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_T, p_{T+1}\}$ of ACE punctured at points $p_T = (T, m_1^*), p_{T+1} = (T+1, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output 'fail';
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_T,p_{T+1}\}}(0, m_1)$.

Program Increment$_{A,2,T,2}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_T, p_{T+1}\}, \mathsf{DK}_1\{p_T, p_{T+1}\}$ of ACE punctured at $p_T = (T, m_1^*), p_{T+1} = (T+1, m_1^*)$, single-tag level $l_T^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(T, m_1^*)$, upper bound T.

1. If $l = l_T^*$ then output 'fail';
2. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_T,p_{T+1}\}}(l)$; if out $=$ 'fail' then output 'fail'; else parse out as (i, m_1).
3. If $i \geq T$ or $i < 0$ then output 'fail';
4. If $i = T-1$ and $m_1 = m_1^*$ then output l_T^*;
5. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_T,p_{T+1}\}}(i+1, m_1)$.

Program Transform$_{A,2,T,2}[(l_0^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_T, p_{T+1}\}$ of ACE punctured at points $p_T = (T, m_1^*), p_{T+1} = (T+1, m_1^*)$, encryption key EK_2 of ACE, single-tag level $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, tag m_2^*, single-tag level $l_T^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(T, m_1^*)$, upper bound T.

1. If $(l, m_2) = (l_0^*, m_2^*)$ then output 'fail';
2. If $l = l_T^*$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(T, m_1^*, m_2)$;
3. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_T,p_{T+1}\}}(l)$; if out $=$ 'fail' then output 'fail'; else parse out as (i, m_1).
4. If $i > T$ or $i < 0$ then output 'fail';
5. output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{A,2,T,2}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_T, p_{T+1}\}$ of ACE punctured at points $p_T = (T, m_1^*), p_{T+1} = (T+1, m_1^*)$, single-tag level $l_T^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(T, m_1^*)$, upper bound T.

1. If $l = l_T^*$ then output m_1^*;
2. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_T,p_{T+1}\}}(l)$; if out $=$ 'fail' then output 'fail'; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then output 'fail';
4. Output m_1.

Figure 27: Programs in $\mathsf{Hyb}_{A,2,T,2}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and RetrieveTags, together with $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{A,2,T,3}$

Program GenZero$_{A,2,T,3}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_T, p_{T+1}\}$ of ACE punctured at points $p_T = (T, m_1^*), p_{T+1} = (T+1, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_T,p_{T+1}\}}(0, m_1)$.

Program Increment$_{A,2,T,3}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_T, p_{T+1}\}, \mathsf{DK}_1\{p_T, p_{T+1}\}$ of ACE punctured at $p_T = (T, m_1^*), p_{T+1} = (T+1, m_1^*)$, single-tag level $l_{T+1}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(T+1, m_1^*)$, upper bound T.

1. If $l = l_{T+1}^*$ then output $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_T,p_{T+1}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
4. If $i = T-1$ and $m_1 = m_1^*$ then output l_{T+1}^*;
5. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_T,p_{T+1}\}}(i+1, m_1)$.

Program Transform$_{A,2,T,3}[(l_0^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_T, p_{T+1}\}$ of ACE punctured at points $p_T = (T, m_1^*), p_{T+1} = (T+1, m_1^*)$, encryption key EK_2 of ACE, single-tag level $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, tag m_2^*, single-tag level $l_{T+1}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(T+1, m_1^*)$, upper bound T.

1. If $(l, m_2) = (l_0^*, m_2^*)$ then output $'\mathsf{fail}'$;
2. If $l = l_{T+1}^*$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(T, m_1^*, m_2)$;
3. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_T,p_{T+1}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
4. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
5. output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{A,2,T,3}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_T, p_{T+1}\}$ of ACE punctured at points $p_T = (T, m_1^*), p_{T+1} = (T+1, m_1^*)$, single-tag level $l_{T+1}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(T+1, m_1^*)$, upper bound T.

1. If $l = l_{T+1}^*$ then output m_1^*;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_T,p_{T+1}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
4. Output m_1.

Figure 28: Programs in $\mathsf{Hyb}_{A,2,T,3}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and RetrieveTags, together with $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{A,2,T,4}$ (same as $\mathsf{Hyb}_{A,2,T-1,1}$).

Program GenZero$_{A,2,T-1,1}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_T\}$ of ACE punctured at the point $p_T = (T, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_T\}}(0, m_1)$.

Program Increment$_{A,2,T-1,1}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_T\}, \mathsf{DK}_1\{p_T\}$ of ACE punctured at $p_T = (T, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_T\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. If $i = T-1$ and $m_1 = m_1^*$ then output $\mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_T\}}(i+2, m_1)$;
4. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_T\}}(i+1, m_1)$.

Program Transform$_{A,2,T-1,1}[(l_0^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_T\}$ of ACE punctured at the point $p_T = (T, m_1^*)$, encryption key EK_2 of ACE, single-tag level $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_0^*, m_2^*)$ then output $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_T\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $m_1 = m_1^*$ and $i = T+1$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(T, m_1^*, m_2)$;
4. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
5. output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{A,2,T-1,1}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_T\}$ of ACE punctured at the point $p_T = (T, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_T\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1^*.
3. If $m \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1.

Figure 29: Programs in $\mathsf{Hyb}_{A,2,T,4}$ (same as $\mathsf{Hyb}_{A,2,T-1,1}$). In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and $\mathsf{RetrieveTags}$, together with $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{A,2,T-1,2}$

Program GenZero$_{A,2,T-1,2}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_{T-1}, p_T\}$ of ACE punctured at points $p_{T-1} = (T-1, m_1^*), p_T = (T, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{T-1},p_T\}}(0, m_1)$.

Program Increment$_{A,2,T-1,2}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_{T-1}, p_T\}, \mathsf{DK}_1\{p_{T-1}, p_T\}$ of ACE punctured at points $p_{T-1} = (T-1, m_1^*), p_T = (T, m_1^*)$, single-tag level $l_{T-1}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(T-1, m_1^*)$, upper bound T,

1. If $l = l_{T-1}^*$ then output $\mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{T-1},p_T\}}(T+1, m_1^*)$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_{T-1},p_T\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
4. If $i = T-2$ and $m_1 = m_1^*$ then output l_{T-1}^*;
5. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{T-1},p_T\}}(i+1, m_1)$.

Program Transform$_{A,2,T-1,2}[(l_0^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_{T-1}, p_T\}$ of ACE punctured at points $p_{T-1} = (T-1, m_1^*), p_T = (T, m_1^*)$, encryption key EK_2 of ACE, single-tag level $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, tag m_2^*, single-tag level $l_{T-1}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(T-1, m_1^*)$, upper bound T.

1. If $(l, m_2) = (l_0^*, m_2^*)$ then output $'\mathsf{fail}'$;
2. If $l = l_{T-1}^*$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(T-1, m_1^*, m_2)$;
3. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_{T-1},p_T\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
4. If $m_1 = m_1^*$ and $i = T+1$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(T, m_1^*, m_2)$;
5. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
6. output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{A,2,T-1,2}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_{T-1}, p_T\}$ of ACE punctured at points $p_{T-1} = (T-1, m_1^*), p_T = (T, m_1^*)$, single-tag level $l_{T-1}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(T-1, m_1^*)$, upper bound T.

1. If $l = l_{T-1}^*$ then output m_1^*;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_{T-1},p_T\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $m = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1^*.
4. If $m \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1.

Figure 30: Programs in $\mathsf{Hyb}_{A,2,T-1,2}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and RetrieveTags, together with $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{A,2,T-1,3}$

Program GenZero$_{A,2,T-1,3}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_{T-1}, p_T\}$ of ACE punctured at points $p_{T-1} = (T-1, m_1^*), p_T = (T, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{T-1},p_T\}}(0, m_1)$.

Program Increment$_{A,2,T-1,3}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_{T-1}, p_T\}, \mathsf{DK}_1\{p_{T-1}, p_T\}$ of ACE punctured at points $p_{T-1} = (T-1, m_1^*)$, $p_T = (T, m_1^*)$, single-tag level $l_T^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(T, m_1^*)$, upper bound T.

1. If $l = l_T^*$ then output $\mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{T-1},p_T\}}(T+1, m_1^*)$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_{T-1},p_T\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
4. If $i = T-2$ and $m_1 = m_1^*$ then output l_T^*;
5. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{T-1},p_T\}}(i+1, m_1)$.

Program Transform$_{A,2,T-1,3}[(l_0^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_{T-1}, p_T\}$ of ACE punctured at points $p_{T-1} = (T-1, m_1^*), p_T = (T, m_1^*)$, encryption key EK_2 of ACE, single-tag level $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, tag m_2^*, single-tag level $l_T^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(T, m_1^*)$, upper bound T.

1. If $(l, m_2) = (l_0^*, m_2^*)$ then output $'\mathsf{fail}'$;
2. If $l = l_T^*$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(T-1, m_1^*, m_2)$;
3. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_{T-1},p_T\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
4. If $m_1 = m_1^*$ and $i = T+1$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(T, m_1^*, m_2)$;
5. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
6. output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{A,2,T-1,3}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_{T-1}, p_T\}$ of ACE punctured at points $p_{T-1} = (T-1, m_1^*), p_T = (T, m_1^*)$, single-tag level $l_T^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(T, m_1^*)$, upper bound T.

1. If $l = l_T^*$ then output m_1^*;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_{T-1},p_T\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $m = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1^*.
4. If $m \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1.

Figure 31: Programs in $\mathsf{Hyb}_{A,2,T-1,3}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and RetrieveTags, together with $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{A,2,T-1,4}$

Program GenZero$_{A,2,T-1,4}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_{T-1}\}$ of ACE punctured at the point $p_{T-1} = (T-1, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output 'fail';
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{T-1}\}}(0, m_1)$.

Program Increment$_{A,2,T-1,4}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_{T-1}\}, \mathsf{DK}_1\{p_{T-1}\}$ of ACE punctured at the point $p_{T-1} = (T-1, m_1^*)$, upper bound T.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_{T-1}\}}(l)$; if out = 'fail' then output 'fail'; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i \geq T+1$ or $i < 0$) then output 'fail';
3. If $m_1 \neq m_1^*$ and ($i \geq T$ or $i < 0$) then output 'fail';
4. If $i = T-2$ and $m_1 = m_1^*$ then output $\mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{T-1}\}}(i+2, m_1)$;
5. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{T-1}\}}(i+1, m_1)$.

Program Transform$_{A,2,T-1,4}[(l_0^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_{T-1}\}$ of ACE punctured at the point $p_{T-1} = (T-1, m_1^*)$, encryption key EK_2 of ACE, single-tag level $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_0^*, m_2^*)$ then output 'fail';
2. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_{T-1}\}}(l)$; if out = 'fail' then output 'fail'; else parse out as (i, m_1).
3. If $m_1 = m_1^*$ and $i = T+1$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(T, m_1^*, m_2)$;
4. If $m_1 = m_1^*$ and $i = T$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(T-1, m_1^*, m_2)$;
5. If $i > T$ or $i < 0$ then output 'fail';
6. output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{A,2,T-1,4}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_{T-1}\}$ of ACE punctured at the point $p_{T-1} = (T-1, m_1^*)$, upper bound T.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_{T-1}\}}(l)$; if out = 'fail' then output 'fail'; else parse out as (i, m_1).
2. If $m = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then output 'fail';
   (b) Output m_1^*.
3. If $m \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output 'fail';
   (b) Output m_1.

Figure 32: Programs in $\mathsf{Hyb}_{A,2,T-1,4}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and RetrieveTags, together with $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{A,2,j,1}$

Program GenZero$_{A,2,j,1}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_{j+1}\}$ of ACE punctured at the point $p_{j+1} = (j+1, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{j+1}\}}(0, m_1)$.

Program Increment$_{A,2,j,1}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_{j+1}\}, \mathsf{DK}_1\{p_{j+1}\}$ of ACE punctured at $p_{j+1} = (j+1, m_1^*)$, index j, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_{j+1}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i \geq T+1$ or $i < 0$) then output $'\mathsf{fail}'$;
3. If $m_1 \neq m_1^*$ and ($i \geq T$ or $i < 0$) then output $'\mathsf{fail}'$;
4. If $i = j$ and $m_1 = m_1^*$ then output $\mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{j+1}\}}(i+2, m_1^*)$;
5. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{j+1}\}}(i+1, m_1)$.

Program Transform$_{A,2,j,1}[(l_0^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_{j+1}\}$ of ACE punctured at the point $p_{j+1} = (j+1, m_1^*)$, encryption key EK_2 of ACE, single-tag level $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, tag m_2^*, index j, upper bound T.

1. If $(l, m_2) = (l_0^*, m_2^*)$ then output $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_{j+1}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $m_1 = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) If $i > j+1$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
   (c) If $i = j+1$ then output $'\mathsf{fail}'$;
   (d) If $i < j+1$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.
4. If $m_1 \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{A,2,j,1}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_{j+1}\}$ of ACE punctured at the point $p_{j+1} = (j+1, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_{j+1}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1^*.
3. If $m \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1.

Figure 33: Programs in $\mathsf{Hyb}_{A,2,j,1}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and RetrieveTags, together with $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{A,2,j,2}$

Program GenZero$_{A,2,j,2}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_j, p_{j+1}\}$ of ACE punctured at points $p_j = (j, m_1^*), p_{j+1} = (j+1, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_j,p_{j+1}\}}(0, m_1)$.

Program Increment$_{A,2,j,2}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_j, p_{j+1}\}, \mathsf{DK}_1\{p_j, p_{j+1}\}$ of ACE punctured at points $p_j = (j, m_1^*), p_{j+1} = (j+1, m_1^*)$, single-tag level $l_j^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(j, m_1^*)$, index j, upper bound T,

1. If $l = l_j^*$ then output $\mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_j,p_{j+1}\}}(j+2, m_1^*)$;
2. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_j,p_{j+1}\}}(l)$; if out $= '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $m_1 = m_1^*$ and ($i \geq T+1$ or $i < 0$) then output $'\mathsf{fail}'$;
4. If $m_1 \neq m_1^*$ and ($i \geq T$ or $i < 0$) then output $'\mathsf{fail}'$;
5. If $i = j-1$ and $m_1 = m_1^*$ then output l_j^*;
6. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_j,p_{j+1}\}}(i+1, m_1)$.

Program Transform$_{A,2,j,2}[(l_0^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_j, p_{j+1}\}$ of ACE punctured at points $p_j = (j, m_1^*), p_{j+1} = (j+1, m_1^*)$, encryption key EK_2 of ACE, single-tag level $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, tag m_2^*, single-tag level $l_j^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(j, m_1^*)$, index j, upper bound T.

1. If $(l, m_2) = (l_0^*, m_2^*)$ then output $'\mathsf{fail}'$;
2. If $l = l_j^*$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, m_2)$;
3. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_j,p_{j+1}\}}(l)$; if out $= '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
4. If $m_1 = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) If $i > j+1$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
   (c) If $i = j+1$ then output $'\mathsf{fail}'$;
   (d) If $i = j$ then output $'\mathsf{fail}'$;
   (e) If $i < j$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.
5. If $m_1 \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{A,2,j,2}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_j, p_{j+1}\}$ of ACE punctured at points $p_j = (j, m_1^*), p_{j+1} = (j+1, m_1^*)$, single-tag level $l_j^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(j, m_1^*)$, upper bound T.

1. If $l = l_j^*$ then output m_1^*;
2. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_j,p_{j+1}\}}(l)$; if out $= '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $m = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1.
4. If $m \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1.

Figure 34: Programs in $\mathsf{Hyb}_{A,2,j,2}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and RetrieveTags, together with $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{A,2,j,3}$

Program GenZero$_{A,2,j,3}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_j, p_{j+1}\}$ of ACE punctured at points $p_j = (j, m_1^*), p_{j+1} = (j+1, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output 'fail';
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_j,p_{j+1}\}}(0, m_1)$.

Program Increment$_{A,2,j,3}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_j, p_{j+1}\}, \mathsf{DK}_1\{p_j, p_{j+1}\}$ of ACE punctured at points $p_j = (j, m_1^*), p_{j+1} = (j+1, m_1^*)$, single-tag level $l_{j+1}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(j+1, m_1^*)$, index j, upper bound T.

1. If $l = l_{j+1}^*$ then output $\mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_j,p_{j+1}\}}(j+2, m_1^*)$;
2. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_j,p_{j+1}\}}(l)$; if out $=$ 'fail' then output 'fail'; else parse out as (i, m_1).
3. If $m_1 = m_1^*$ and ($i \geq T+1$ or $i < 0$) then output 'fail';
4. If $m_1 \neq m_1^*$ and ($i \geq T$ or $i < 0$) then output 'fail';
5. If $i = j-1$ and $m_1 = m_1^*$ then output l_{j+1}^*;
6. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_j,p_{j+1}\}}(i+1, m_1)$.

Program Transform$_{A,2,j,3}[(l_0^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_j, p_{j+1}\}$ of ACE punctured at points $p_j = (j, m_1^*), p_{j+1} = (j+1, m_1^*)$, encryption key EK_2 of ACE, single-tag level $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, tag m_2^*, single-tag level $l_{j+1}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(j+1, m_1^*)$, index j, upper bound T.

1. If $(l, m_2) = (l_0^*, m_2^*)$ then output 'fail';
2. If $l = l_{j+1}^*$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, m_2)$;
3. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_j,p_{j+1}\}}(l)$; if out $=$ 'fail' then output 'fail'; else parse out as (i, m_1).
4. If $m_1 = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then output 'fail';
   (b) If $i > j+1$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
   (c) If $i = j+1$ then output 'fail';
   (d) If $i = j$ then output 'fail';
   (e) If $i < j$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.
5. If $m_1 \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output 'fail';
   (b) Output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{A,2,j,3}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_j, p_{j+1}\}$ of ACE punctured at points $p_j = (j, m_1^*), p_{j+1} = (j+1, m_1^*)$, single-tag level $l_{j+1}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(j+1, m_1^*)$, upper bound T.

1. If $l = l_{j+1}^*$ then output m_1^*;
2. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_j,p_{j+1}\}}(l)$; if out $=$ 'fail' then output 'fail'; else parse out as (i, m_1).
3. If $m = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then output 'fail';
   (b) Output m_1.
4. If $m \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output 'fail';
   (b) Output m_1.

Figure 35: Programs in $\mathsf{Hyb}_{A,2,j,3}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and RetrieveTags, together with $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{A,2,j,4}$.

Program GenZero$_{A,2,j,4}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_j\}$ of ACE punctured at the point $p_j = (j, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_j\}}(0, m_1)$.

Program Increment$_{A,2,j,4}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_j\}, \mathsf{DK}_1\{p_j\}$ of ACE punctured at $p_j = (j, m_1^*)$, index j, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_j\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and $(i \geq T+1$ or $i < 0)$ then output $'\mathsf{fail}'$;
3. If $m_1 \neq m_1^*$ and $(i \geq T$ or $i < 0)$ then output $'\mathsf{fail}'$;
4. If $i = j-1$ and $m_1 = m_1^*$ then output $\mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_j\}}(i+2, m_1^*)$;
5. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_j\}}(i+1, m_1)$.

Program Transform$_{A,2,j,4}[(l_0^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_j\}$ of ACE punctured at the point $p_j = (j, m_1^*)$, encryption key EK_2 of ACE, single-tag level $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, tag m_2^*, index j, upper bound T.

1. If $(l, m_2) = (l_0^*, m_2^*)$ then output $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_j\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $m_1 = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) If $i > j$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
   (c) If $i = j$ then output $'\mathsf{fail}'$;
   (d) If $i < j$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.
4. If $m_1 \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{A,2,j,4}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_j\}$ of ACE punctured at the point $p_j = (j, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_j\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1^*.
3. If $m \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1.

Figure 36: Programs in $\mathsf{Hyb}_{A,2,j,4}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and $\mathsf{RetrieveTags}$, together with $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{A,2,0,1}$

Program GenZero$_{A,2,0,1}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_1\}$ of ACE punctured at the point $p_1 = (1, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_1\}}(0, m_1)$.

Program Increment$_{A,2,0,1}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_1\}, \mathsf{DK}_1\{p_1\}$ of ACE punctured at $p_1 = (1, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_1\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i \geq T + 1$ or $i < 0$) then output $'\mathsf{fail}'$;
3. If $m_1 \neq m_1^*$ and ($i \geq T$ or $i < 0$) then output $'\mathsf{fail}'$;
4. If $i = 0$ and $m_1 = m_1^*$ then output $\mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_1\}}(i + 2, m_1^*)$;
5. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_1\}}(i + 1, m_1)$.

Program Transform$_{A,2,0,1}[(l_0^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_1\}$ of ACE punctured at the point $p_1 = (1, m_1^*)$, encryption key EK_2 of ACE, single-tag level $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_0^*, m_2^*)$ then output $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_1\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $m_1 = m_1^*$:
   (a) If $i > T + 1$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) If $i > 1$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i - 1, m_1, m_2)$;
   (c) If $i = 1$ then output $'\mathsf{fail}'$;
   (d) If $i < 1$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.
4. If $m_1 \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{A,2,0,1}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_1\}$ of ACE punctured at the point $p_1 = (1, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_1\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m = m_1^*$:
   (a) If $i > T + 1$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1^*.
3. If $m \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1.

Figure 37: Programs in $\mathsf{Hyb}_{A,2,0,1}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and $\mathsf{RetrieveTags}$, together with $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{A,2,0,2}$

Program GenZero$_{A,2,0,2}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_0, p_1\}$ of ACE punctured at points $p_0 = (0, m_1^*), p_1 = (1, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output 'fail';
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0,p_1\}}(0, m_1)$.

Program Increment$_{A,2,0,2}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_0, p_1\}, \mathsf{DK}_1\{p_0, p_1\}$ of ACE punctured at points $p_0 = (0, m_1^*), p_1 = (1, m_1^*)$, single-tag level $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, upper bound T,

1. If $l = l_0^*$ then output $\mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0,p_1\}}(2, m_1^*)$;
2. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0,p_1\}}(l)$; if out $=$ 'fail' then output 'fail'; else parse out as (i, m_1).
3. If $m_1 = m_1^*$ and ($i \geq T+1$ or $i < 0$) then output 'fail';
4. If $m_1 \neq m_1^*$ and ($i \geq T$ or $i < 0$) then output 'fail';
5. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0,p_1\}}(i+1, m_1)$.

Program Transform$_{A,2,0,2}[(l_0^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_0, p_1\}$ of ACE punctured at points $p_0 = (0, m_1^*), p_1 = (1, m_1^*)$, encryption key EK_2 of ACE, single-tag level $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_0^*, m_2^*)$ then output 'fail';
2. If $l = l_0^*$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2)$;
3. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0,p_1\}}(l)$; if out $=$ 'fail' then output 'fail'; else parse out as (i, m_1).
4. If $m_1 = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then output 'fail';
   (b) If $i > 1$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
   (c) If $i = 1$ then output 'fail';
   (d) If $i = 0$ then output 'fail';
5. If $m_1 \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output 'fail';
   (b) Output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{A,2,0,2}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_0, p_1\}$ of ACE punctured at points $p_0 = (0, m_1^*), p_1 = (1, m_1^*)$, single-tag level $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, upper bound T.

1. If $l = l_0^*$ then output m_1^*;
2. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0,p_1\}}(l)$; if out $=$ 'fail' then output 'fail'; else parse out as (i, m_1).
3. If $m = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then output 'fail';
   (b) Output m_1.
4. If $m \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output 'fail';
   (b) Output m_1.

Figure 38: Programs in $\mathsf{Hyb}_{A,2,0,2}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and RetrieveTags, together with $l_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{A,2,0,3}$

Program GenZero$_{A,2,0,3}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_0, p_1\}$ of ACE punctured at points $p_0 = (0, m_1^*), p_1 = (1, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0,p_1\}}(0, m_1)$.

Program Increment$_{A,2,0,3}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_0, p_1\}, \mathsf{DK}_1\{p_0, p_1\}$ of ACE punctured at points $p_0 = (0, m_1^*), p_1 = (1, m_1^*)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, upper bound T,

1. If $l = l_1^*$ then output $\mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0,p_1\}}(2, m_1^*)$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0,p_1\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $m_1 = m_1^*$ and ($i \geq T + 1$ or $i < 0$) then output $'\mathsf{fail}'$;
4. If $m_1 \neq m_1^*$ and ($i \geq T$ or $i < 0$) then output $'\mathsf{fail}'$;
5. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0,p_1\}}(i + 1, m_1)$.

Program Transform$_{A,2,0,3}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_0, p_1\}$ of ACE punctured at points $p_0 = (0, m_1^*), p_1 = (1, m_1^*)$, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then output $'\mathsf{fail}'$;
2. If $l = l_1^*$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2)$;
3. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0,p_1\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
4. If $m_1 = m_1^*$:
   (a) If $i > T + 1$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) If $i > 1$ then output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i - 1, m_1, m_2)$;
   (c) If $i = 1$ then output $'\mathsf{fail}'$;
   (d) If $i = 0$ then output $'\mathsf{fail}'$;
5. If $m_1 \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{A,2,0,3}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_0, p_1\}$ of ACE punctured at points $p_0 = (0, m_1^*), p_1 = (1, m_1^*)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, upper bound T.

1. If $l = l_1^*$ then output m_1^*;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0,p_1\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $m = m_1^*$:
   (a) If $i > T + 1$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1.
4. If $m \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1.

Figure 39: Programs in $\mathsf{Hyb}_{A,2,0,3}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and RetrieveTags, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{A,3,1}$.

Program GenZero$_{A,3,1}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_0\}$ of ACE punctured at the point $p_0 = (0, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(0, m_1)$.

Program Increment$_{A,3,1}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_0\}, \mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i \geq T + 1$ or $i < 0$) then output $'\mathsf{fail}'$;
3. If $m_1 \neq m_1^*$ and ($i \geq T$ or $i < 0$) then output $'\mathsf{fail}'$;
4. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(i + 1, m_1)$.

Program Transform$_{A,3,1}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_0\}$ of ACE punctured at the point $p_0 = (0, m_1^*)$, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then output $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $m_1 = m_1^*$:
   (a) If $i > T + 1$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i - 1, m_1, m_2)$;
4. If $m_1 \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{A,3,1}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_0\}$ of ACE punctured at the point $p_0 = (0, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m = m_1^*$:
   (a) If $i > T + 1$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1^*.
3. If $m \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1.

Figure 40: Programs in $\mathsf{Hyb}_{A,3,1}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and $\mathsf{RetrieveTags}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{A,3,2}$

Program GenZero$_{A,3,2}[m_1^*](m_1)$
Inputs: tag $m_1 \in M$.
Hardwired values: encryption key $\mathsf{EK}_1\{p_0\}$ of ACE punctured at the point $p_0 = (0, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(0, m_1)$.

Program Increment$_{A,3,2}(l)$
Inputs: single-tag level l
Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_0\}, \mathsf{DK}_1$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i \geq T+1$ or $i < 0$) then output $'\mathsf{fail}'$;
3. If $m_1 \neq m_1^*$ and ($i \geq T$ or $i < 0$) then output $'\mathsf{fail}'$;
4. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(i+1, m_1)$.

Program Transform$_{A,3,2}[(l_1^*, m_2^*)](l, m_2)$
Inputs: single-tag level l, tag $m_2 \in M$
Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then output $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $m_1 = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
4. If $m_1 \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{A,3,2}(l)$
Inputs: single-tag level l
Hardwired values: decryption key DK_1 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1^*.
3. If $m \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1.

Figure 41: Programs in $\mathsf{Hyb}_{A,3,2}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and $\mathsf{RetrieveTags}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{A,3,3}$

Program GenZero$_{A,3,3}[m_1^*](m_1)$
Inputs: tag $m_1 \in M$.
Hardwired values: encryption key EK_1 of ACE, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1)$.

Program Increment$_{A,3,3}(l)$
Inputs: single-tag level l
Hardwired values: encryption and decryption keys $\mathsf{EK}_1, \mathsf{DK}_1$ of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and $(i \geq T+1$ or $i < 0)$ then output $'\mathsf{fail}'$;
3. If $m_1 \neq m_1^*$ and $(i \geq T$ or $i < 0)$ then output $'\mathsf{fail}'$;
4. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(i+1, m_1)$.

Program Transform$_{A,3,3}[(l_1^*, m_2^*)](l, m_2)$
Inputs: single-tag level l, tag $m_2 \in M$
Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then output $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $m_1 = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
4. If $m_1 \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{A,3,3}(l)$
Inputs: single-tag level l
Hardwired values: decryption key DK_1 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1^*.
3. If $m \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
   (b) Output m_1.

Figure 42: Programs in $\mathsf{Hyb}_{A,3,3}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and $\mathsf{RetrieveTags}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

7.4.2 Proof of lemma 3 (Changing the upper bound from $T + 1$ to T).

As described earlier, we will fix upper bounds in programs by cutting the sequence of encryptions $[1, m_1^*] \to \ldots \to [T+1, m_1^*]$ at a random place and then cutting the sequence in all subsequent positions, then changing the upper bound, and finally restoring the line. We cut the line at a random place in the following sequence of hybrids, starting from Hyb_B:

- $\mathsf{Hyb}_{B,1,1}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{B,1,1}[m_1^*]$, $\mathsf{Increment}_{B,1,1}$, $\mathsf{Transform}_{B,1,1}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{B,1,1}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 44.

  That is, in program Increment we add an instruction to abort if $m_1 = m_1^*$ and $g(i) = I^*$, where g is an injective OWF and I^* is a random image of g. Indistinguishability holds by security of iO and OWF: since OWF is injective, the two programs differ only at a single point; as shown in [BCP14], any adversary which can distinguish between the two programs, can be also used to find the differing point, which can be used to break one-wayness of g (see lemma 1).

- $\mathsf{Hyb}_{B,1,2}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{B,1,2}[m_1^*]$, $\mathsf{Increment}_{B,1,2}$, $\mathsf{Transform}_{B,1,2}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{B,1,2}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 45.

  That is, in programs Increment and GenZero we puncture ACE encryption key EK_1 at the point $(i^* + 1, m_1^*)$. Indistinguishability holds by iO, since Increment never needs to encrypt this point, because it aborts earlier on input $[i^*, m_1^*]$. GenZero never needs to encrypt (i^*, m_1^*) as well, since it only encrypts value 0, and $i^* = 0$ only with negligible probability.

Next we run the following sequence of hybrids for $j = i^*, \ldots, T$ in order to cut the chain at all points after i^*:

- $\mathsf{Hyb}_{B,2,j,1}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{B,2,j,1}[m_1^*]$, $\mathsf{Increment}_{B,2,j,1}$, $\mathsf{Transform}_{B,2,j,1}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{B,2,j,1}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 46.

  That is, in programs GenZero, Increment, Transform, and RetrieveTag ACE encryption key EK_1 is punctured at the set $\{(i^* + 1, m_1^*), \ldots, (j + 1, m_1^*)\}$, and its decryption key DK_1 is punctured at the set $\{(i^* + 1, m_1^*), \ldots, (j, m_1^*)\}$.

  Note that $\mathsf{Hyb}_{B,2,j,1} = \mathsf{Hyb}_{B,1,2}$ for $j = i^*$.

- $\mathsf{Hyb}_{B,2,j,2}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{B,2,j,2}[m_1^*]$, $\mathsf{Increment}_{B,2,j,2}$, $\mathsf{Transform}_{B,2,j,2}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{B,2,j,2}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 47.

  That is, in programs Increment, Transform, and RetrieveTag we additionally puncture ACE decryption key DK_1 at the point $(j + 1, m_1^*)$. Indistinguishability holds by security of constrained decryption of ACE, since EK_1 is already punctured at the set which includes $(j + 1, m_1^*)$.

- $\mathsf{Hyb}_{B,2,j,3}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{B,2,j,3}[m_1^*]$, $\mathsf{Increment}_{B,2,j,3}$, $\mathsf{Transform}_{B,2,j,3}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{B,2,j,3}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 48.

  That is, we additionally puncture ACE encryption key EK_1 at the point $(j+2, m_1^*)$ in programs GenZero and Increment. Indistinguishability holds by iO, since DK_1 is punctured at the set which includes $(j+1, m_1^*)$, and thus program Increment never tries to encrypt $(j+2, m_1^*)$, aborting earlier; GenZero never needs to encrypt $(j+2, m_1^*)$ either since $j+2 \neq 0$.

  Note that $\mathsf{Hyb}_{B,2,j,3} = \mathsf{Hyb}_{B,2,j+1,1}$ for $j = i^*, \ldots, T$.

Next we change the upper bound as follows:

- $\mathsf{Hyb}_{B,3,1}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{B,3,1}[m_1^*]$, $\mathsf{Increment}_{B,3,1}$, $\mathsf{Transform}_{B,3,1}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{B,3,1}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 49.

  That is, in programs GenZero, Increment, Transform, and RetrieveTag $\mathsf{EK}_1, \mathsf{DK}_1$ are punctured at the set $\{[i^*+1, m_1^*], \ldots, [T+1, m_1^*]\}$.

  Note that $\mathsf{Hyb}_{B,3,1} = \mathsf{Hyb}_{B,2,T,2}$.

- $\mathsf{Hyb}_{B,3,2}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{B,3,2}[m_1^*]$, $\mathsf{Increment}_{B,3,2}$, $\mathsf{Transform}_{B,3,2}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{B,3,2}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 50.

  That is, in program Increment and Transform we change the upper bound from $T+1$ to T. Indistinguishability holds by iO, since DK_1 is punctured at the set which includes $(T, m_1^*), (T+1, m_1^*)$, and thus Increment anyways outputs 'fail' on input $[T, m_1^*]$, and Transform anyway outputs 'fail' on input $[T+1, m_1^*]$.

Next we run the following sequence of hybrids for $j = T, \ldots, i^*$ in order to restore the chain:

- $\mathsf{Hyb}_{B,4,j,1}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{B,4,j,1}[m_1^*]$, $\mathsf{Increment}_{B,4,j,1}$, $\mathsf{Transform}_{B,4,j,1}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{B,4,j,1}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 51.

  That is, in programs GenZero, Increment, Transform, and RetrieveTag ACE key $\mathsf{EK}_1, \mathsf{DK}_1$ are punctured at the set $\{(i^*+1, m_1^*), \ldots, (j+1, m_1^*)\}$.

  Note that $\mathsf{Hyb}_{B,4,j,1} = \mathsf{Hyb}_{B,3,2}$ for $j = T$.

- $\mathsf{Hyb}_{B,4,j,2}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{B,4,j,2}[m_1^*]$, $\mathsf{Increment}_{B,4,j,2}$, $\mathsf{Transform}_{B,4,j,2}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{B,4,j,2}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 52.

  That is, we unpuncture DK_1 in Increment, Transform, and RetrieveTag at the point $(j+1, m_1^*)$. Indistinguishability holds by security of constrained decryption of ACE, since EK_1 is punctured at the set which includes $(j+1, m_1^*)$.

- $\mathsf{Hyb}_{B,4,j,3}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{B,4,j,3}[m_1^*],$ $\mathsf{Increment}_{B,4,j,3}$, $\mathsf{Transform}_{B,4,j,3}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{B,4,j,3}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 53.

  That is, we unpuncture EK_1 in GenZero and Increment at the point $(j+1, m_1^*)$. Indistinguishability holds by iO, since GenZero never encrypts $(j+1, m_1^*)$ where $j+1 \neq 0$, and since Increment never encrypts $(j+1, m_1^*)$, since it aborts on input $[j, m_1^*]$ due to punctured DK_1.

  Note that $\mathsf{Hyb}_{B,4,j,3} = \mathsf{Hyb}_{B,4,j-1,1}$ for $j = T, \dots, i^*+1$.

Finally we remove the last remaining cut in the chain as follows:

- $\mathsf{Hyb}_{B,5,1}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{B,5,1}[m_1^*],$ $\mathsf{Increment}_{B,5,1}$, $\mathsf{Transform}_{B,5,1}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{B,5,1}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 54.

  That is, in programs Increment and GenZero ACE encryption key EK_1 is punctured at the point (i^*+1, m_1^*).

  Note that $\mathsf{Hyb}_{B,5,1} = \mathsf{Hyb}_{B,4,j,2}$ for $j = i^*$.

- $\mathsf{Hyb}_{B,5,2}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{B,5,2}[m_1^*],$ $\mathsf{Increment}_{B,5,2}$, $\mathsf{Transform}_{B,5,2}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{B,5,2}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 55.

  That is, in program Increment we add an instruction to abort if $m_1 = m_1^*$ and $g(i) = I^*$, where $I^* = g(i^*)$ for randomly chosen i^*. In addition, we remove the puncturing from EK_1 in all programs. Indistinguishability holds by iO, since Increment outputs 'fail' on $[i^*, m_1^*]$ in both cases, and since GenZero never needs to encrypt (i^*+1, m_1^*).

- $\mathsf{Hyb}_{B,5,3}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{B,5,3}[m_1^*],$ $\mathsf{Increment}_{B,5,3}$, $\mathsf{Transform}_{B,5,4}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{B,5,3}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 56.

  That is, in program Increment we remove an instruction to abort if $m_1 = m_1^*$ and $g(i) = I^*$. Indistinguishability holds by security of iO and OWF: since OWF is injective, the two programs differ only at a single point; as shown in [BCP14], any adversary which can distinguish between the two programs, can be also used to find the differing point, which can be used to break one-wayness of g (see lemma 1).

  Note that $\mathsf{Hyb}_{B,5,3} = \mathsf{Hyb}_C$.

Note that this reduction works only as long as $i^* \neq 0$, which happens with probability $\frac{1}{T}$. Thus, the the advantage of the PPT adversary in distinguishing between Hyb_B and Hyb_C is at most

$$\frac{1}{T} + 2 \cdot 2^{-\Omega(\gamma(\lambda))} + (2(T - i^* + 1) + 3) \cdot 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))} + 2(T - i^* + 1) \cdot 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))} \leq$$

$$\frac{1}{T} + 2^{-\Omega(\gamma(\lambda))} + T \cdot 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))} + T \cdot 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}.$$

Programs in Hyb_B

Program GenZero$_B[m_1^*](m_1)$
Inputs: tag $m_1 \in M$.
Hardwired values: encryption key EK_1 of ACE, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1)$.

Program Increment$_B(l)$
Inputs: single-tag level l
Hardwired values: encryption and decryption keys $\mathsf{EK}_1, \mathsf{DK}_1$ of ACE, tag m_1^*, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i \geq T+1$ or $i < 0$) then output $'\mathsf{fail}'$;
3. If $m_1 \neq m_1^*$ and ($i \geq T$ or $i < 0$) then output $'\mathsf{fail}'$;
4. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(i+1, m_1)$.

Program Transform$_B[(l_1^*, m_2^*)](l, m_2)$
Inputs: single-tag level l, tag $m_2 \in M$
Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $m_1 = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then return $'\mathsf{fail}'$;
   (b) return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
4. If $m_1 \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
   (b) return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program isLess$_B(L', L'')$
Inputs: double-tag levels L', L''
Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTag$_B(l)$
Inputs: single-tag level l
Hardwired values: decryption key DK_1 of ACE, tag m_1^*, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i > T+1$ or $i < 0$) then output $'\mathsf{fail}'$;
3. If $m_1 \neq m_1^*$ and ($i > T$ or $i < 0$) then output $'\mathsf{fail}'$;
4. Output m_1.

Program RetrieveTags$_B(L)$
Inputs: double-tag level L
Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1, m_2.

Figure 43: Programs in Hyb_B. In addition, in this hybrid the adversary gets $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{B,1,1}$.

Program GenZero$_{B,1,1}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key EK_1 of ACE, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1)$.

Program Increment$_{B,1,1}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1, \mathsf{DK}_1$ of ACE, tag m_1^*, OWF g, $I^* = g(i^*)$ for random i^*, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i \geq T+1$ or $i < 0$) then output $'\mathsf{fail}'$;
3. If $m_1 = m_1^*$ and $g(i) = I^*$) then output $'\mathsf{fail}'$;
4. If $m_1 \neq m_1^*$ and ($i \geq T$ or $i < 0$) then output $'\mathsf{fail}'$;
5. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(i+1, m_1)$.

Program Transform$_{B,1,1}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $m_1 = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then return $'\mathsf{fail}'$;
   (b) return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
4. If $m_1 \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
   (b) return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{B,1,1}(l)$

Inputs: single-tag level l

Hardwired values: decryption key DK_1 of ACE, tag m_1^*, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i > T+1$ or $i < 0$) then output $'\mathsf{fail}'$;
3. If $m_1 \neq m_1^*$ and ($i > T$ or $i < 0$) then output $'\mathsf{fail}'$;
4. Output m_1.

Figure 44: Programs in $\mathsf{Hyb}_{B,1,1}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and $\mathsf{RetrieveTags}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{B,1,2}$.

Program GenZero$_{B,1,2}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values:punctured encryption key $\mathsf{EK}_1\{p_{i^*+1}\}$ of ACE, punctured at the point $p_{i^*+1} = (i^* + 1, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{i^*+1}\}}(0, m_1)$.

Program Increment$_{B,1,2}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_{i^*+1}\}, \mathsf{DK}_1$ of ACE, punctured at $p_{i^*+1} = (i^* + 1, m_1^*)$, tag m_1^*, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i \geq T + 1$ or $i < 0$) then output $'\mathsf{fail}'$;
3. If $m_1 \neq m_1^*$ and ($i \geq T$ or $i < 0$) then output $'\mathsf{fail}'$;
4. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{i^*+1}\}}(i + 1, m_1)$.

Program Transform$_{B,1,2}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $m_1 = m_1^*$:
   (a) If $i > T + 1$ or $i < 0$ then return $'\mathsf{fail}'$;
   (b) return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i - 1, m_1, m_2)$;
4. If $m_1 \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
   (b) return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{B,1,2}(l)$

Inputs: single-tag level l

Hardwired values: decryption key DK_1 of ACE, tag m_1^*, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i > T + 1$ or $i < 0$) then output $'\mathsf{fail}'$;
3. If $m_1 \neq m_1^*$ and ($i > T$ or $i < 0$) then output $'\mathsf{fail}'$;
4. Output m_1.

Figure 45: Programs in $\mathsf{Hyb}_{B,1,2}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and $\mathsf{RetrieveTags}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{B,2,j,1}$.

Program GenZero$_{B,2,j,1}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: punctured encryption key $\mathsf{EK}_1\{S_{i^*+1,j+1}\}$ of ACE, tag m_1^*. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \dots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{S_{i^*+1,j+1}\}}(0, m_1)$.

Program Increment$_{B,2,j,1}(l)$

Inputs: single-tag level l

Hardwired values: punctured encryption and decryption keys $\mathsf{EK}_1\{S_{i^*+1,j+1}\}$, $\mathsf{DK}_1\{S_{i^*+1,j}\}$ of ACE, tag m_1^*, set $S_{i^*,j}$, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \dots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,j}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i \geq T+1$ or $i < 0$) then output $'\mathsf{fail}'$;
3. If $m_1 \neq m_1^*$ and ($i \geq T$ or $i < 0$) then output $'\mathsf{fail}'$;
4. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{S_{i^*+1,j+1}\}}(i+1, m_1)$.

Program Transform$_{B,2,j,1}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{S_{i^*+1,j}\}$ of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \dots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,j}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $m_1 = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then return $'\mathsf{fail}'$;
   (b) return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
4. If $m_1 \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
   (b) return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{B,2,j,1}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{S_{i^*+1,j}\}$ of ACE, punctured at the set $S_{i^*+1,j}$, tag m_1^*, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \dots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,j}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i > T+1$ or $i < 0$) then output $'\mathsf{fail}'$;
3. If $m_1 \neq m_1^*$ and ($i > T$ or $i < 0$) then output $'\mathsf{fail}'$;
4. Output m_1.

Figure 46: Programs in $\mathsf{Hyb}_{B,2,j,1}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and RetrieveTags, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{B,2,j,2}$.

Program GenZero$_{B,2,j,2}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: punctured encryption key $\mathsf{EK}_1\{S_{i^*+1,j+1}\}$ of ACE, tag m_1^*. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. If $m_1 = m_1^*$ then output 'fail';
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{S_{i^*+1,j+1}\}}(0, m_1)$.

Program Increment$_{B,2,j,2}(l)$

Inputs: single-tag level l

Hardwired values: punctured encryption and decryption keys $\mathsf{EK}_1\{S_{i^*+1,j+1}\}$, $\mathsf{DK}_1\{S_{i^*+1,j+1}\}$ of ACE, tag m_1^*, set $S_{i^*,j}$, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,j+1}\}}(l)$; if out $=$ 'fail' then output 'fail'; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i \geq T+1$ or $i < 0$) then output 'fail';
3. If $m_1 \neq m_1^*$ and ($i \geq T$ or $i < 0$) then output 'fail';
4. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{S_{i^*+1,j+1}\}}(i+1, m_1)$.

Program Transform$_{B,2,j,2}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{S_{i^*+1,j+1}\}$ of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return 'fail';
2. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,j+1}\}}(l)$; if out $=$ 'fail' then return 'fail'; else parse out as (i, m_1).
3. If $m_1 = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then return 'fail';
   (b) return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
4. If $m_1 \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then return 'fail';
   (b) return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{B,2,j,2}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{S_{i^*+1,j+1}\}$ of ACE, punctured at the set $S_{i^*+1,j+1}$, tag m_1^*, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,j+1}\}}(l)$; if out $=$ 'fail' then output 'fail'; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i > T+1$ or $i < 0$) then output 'fail';
3. If $m_1 \neq m_1^*$ and ($i > T$ or $i < 0$) then output 'fail';
4. Output m_1.

Figure 47: Programs in $\mathsf{Hyb}_{B,2,j,2}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and RetrieveTags, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{B,2,j,3}$.

Program GenZero$_{B,2,j,3}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: punctured encryption key $\mathsf{EK}_1\{S_{i^*+1,j+2}\}$ of ACE, tag m_1^*. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. If $m_1 = m_1^*$ then output 'fail';
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{S_{i^*+1,j+2}\}}(0, m_1)$.

Program Increment$_{B,2,j,3}(l)$

Inputs: single-tag level l

Hardwired values: punctured encryption and decryption keys $\mathsf{EK}_1\{S_{i^*+1,j+2}\}$, $\mathsf{DK}_1\{S_{i^*+1,j+1}\}$ of ACE, tag m_1^*, set $S_{i^*,j}$, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,j+1}\}}(l)$; if out = 'fail' then output 'fail'; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i \geq T+1$ or $i < 0$) then output 'fail';
3. If $m_1 \neq m_1^*$ and ($i \geq T$ or $i < 0$) then output 'fail';
4. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{S_{i^*+1,j+2}\}}(i+1, m_1)$.

Program Transform$_{B,2,j,3}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{S_{i^*+1,j+1}\}$ of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return 'fail';
2. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,j+1}\}}(l)$; if out = 'fail' then return 'fail'; else parse out as (i, m_1).
3. If $m_1 = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then return 'fail';
   (b) return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
4. If $m_1 \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then return 'fail';
   (b) return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{B,2,j,3}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{S_{i^*+1,j+1}\}$ of ACE, punctured at the set $S_{i^*+1,j+1}$, tag m_1^*, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,j+1}\}}(l)$; if out = 'fail' then output 'fail'; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i > T+1$ or $i < 0$) then output 'fail';
3. If $m_1 \neq m_1^*$ and ($i > T$ or $i < 0$) then output 'fail';
4. Output m_1.

Figure 48: Programs in $\mathsf{Hyb}_{B,2,j,3}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and RetrieveTags, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{B,3,1}$.

Program GenZero$_{B,3,1}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: punctured encryption key $\mathsf{EK}_1\{S_{i^*+1,T+1}\}$ of ACE, tag m_1^*. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{S_{i^*+1,T+1}\}}(0, m_1)$.

Program Increment$_{B,3,1}(l)$

Inputs: single-tag level l

Hardwired values: punctured encryption and decryption keys $\mathsf{EK}_1\{S_{i^*+1,T+1}\}$, $\mathsf{DK}_1\{S_{i^*+1,T+1}\}$ of ACE, tag m_1^*, set $S_{i^*,T}$, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,T+1}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i \geq T+1$ or $i < 0$) then output $'\mathsf{fail}'$;
3. If $m_1 \neq m_1^*$ and ($i \geq T$ or $i < 0$) then output $'\mathsf{fail}'$;
4. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{S_{i^*+1,T+1}\}}(i+1, m_1)$.

Program Transform$_{B,3,1}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{S_{i^*+1,T+1}\}$ of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,T+1}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $m_1 = m_1^*$:
   (a) If $i > T+1$ or $i < 0$ then return $'\mathsf{fail}'$;
   (b) return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
4. If $m_1 \neq m_1^*$:
   (a) If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
   (b) return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{B,3,1}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{S_{i^*+1,T+1}\}$ of ACE, punctured at the set $S_{i^*+1,T+1}$, tag m_1^*, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,T+1}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $m_1 = m_1^*$ and ($i > T+1$ or $i < 0$) then output $'\mathsf{fail}'$;
3. If $m_1 \neq m_1^*$ and ($i > T$ or $i < 0$) then output $'\mathsf{fail}'$;
4. Output m_1.

Figure 49: Programs in $\mathsf{Hyb}_{B,3,1}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and $\mathsf{RetrieveTags}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{B,3,2}$.

Program GenZero$_{B,3,2}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: punctured encryption key $\mathsf{EK}_1\{S_{i^*+1,T+1}\}$ of ACE, tag m_1^*. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{S_{i^*+1,T+1}\}}(0, m_1)$.

Program Increment$_{B,3,2}(l)$

Inputs: single-tag level l

Hardwired values: punctured encryption and decryption keys $\mathsf{EK}_1\{S_{i^*+1,T+1}\}$, $\mathsf{DK}_1\{S_{i^*+1,T+1}\}$ of ACE, tag m_1^*, set $S_{i^*,T}$, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,T+1}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{S_{i^*+1,T+1}\}}(i+1, m_1)$.

Program Transform$_{B,3,2}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{S_{i^*+1,T+1}\}$ of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,T+1}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
5. If $m_1 \neq m_1^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{B,3,2}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{S_{i^*+1,T+1}\}$ of ACE, punctured at the set $S_{i^*+1,T+1}$, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,T+1}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Figure 50: Programs in $\mathsf{Hyb}_{B,3,2}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and $\mathsf{RetrieveTags}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{B,4,j,1}$.

Program GenZero$_{B,4,j,1}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: punctured encryption key $\mathsf{EK}_1\{S_{i^*+1,j+1}\}$ of ACE, tag m_1^*. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{S_{i^*+1,j+1}\}}(0, m_1)$.

Program Increment$_{B,4,j,1}(l)$

Inputs: single-tag level l

Hardwired values: punctured encryption and decryption keys $\mathsf{EK}_1\{S_{i^*+1,j+1}\}$, $\mathsf{DK}_1\{S_{i^*+1,j+1}\}$ of ACE, tag m_1^*, set $S_{i^*,j}$, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,j+1}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{S_{i^*+1,j+1}\}}(i+1, m_1)$.

Program Transform$_{B,4,j,1}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{S_{i^*+1,j+1}\}$ of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,j+1}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
5. If $m_1 \neq m_1^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{B,4,j,1}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{S_{i^*+1,j+1}\}$ of ACE, punctured at the set $S_{i^*+1,j+1}$, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,j+1}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Figure 51: Programs in $\mathsf{Hyb}_{B,4,j,1}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and $\mathsf{RetrieveTags}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{B,4,j,2}$.

Program GenZero$_{B,4,j,2}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: punctured encryption key $\mathsf{EK}_1\{S_{i^*+1,j+1}\}$ of ACE, tag m_1^*. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{S_{i^*+1,j+1}\}}(0, m_1)$.

Program Increment$_{B,4,j,2}(l)$

Inputs: single-tag level l

Hardwired values: punctured encryption and decryption keys $\mathsf{EK}_1\{S_{i^*+1,j+1}\}$, $\mathsf{DK}_1\{S_{i^*+1,j}\}$ of ACE, tag m_1^*, set $S_{i^*,j}$, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,j}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{S_{i^*+1,j+1}\}}(i+1, m_1)$.

Program Transform$_{B,4,j,2}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{S_{i^*+1,j}\}$ of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,j}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
5. If $m_1 \neq m_1^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{B,4,j,2}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{S_{i^*+1,j}\}$ of ACE, punctured at the set $S_{i^*+1,j}$, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,j}\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Figure 52: Programs in $\mathsf{Hyb}_{B,4,j,2}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and $\mathsf{RetrieveTags}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{B,4,j,3}$.

Program GenZero$_{B,4,j,3}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: punctured encryption key $\mathsf{EK}_1\{S_{i^*+1,j}\}$ of ACE, tag m_1^*. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. If $m_1 = m_1^*$ then output 'fail';
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{S_{i^*+1,j}\}}(0, m_1)$.

Program Increment$_{B,4,j,3}(l)$

Inputs: single-tag level l

Hardwired values: punctured encryption and decryption keys $\mathsf{EK}_1\{S_{i^*+1,j}\}$, $\mathsf{DK}_1\{S_{i^*+1,j}\}$ of ACE, tag m_1^*, set $S_{i^*,j}$, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,j}\}}(l)$; if $\mathsf{out} =$ 'fail' then output 'fail'; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output 'fail';
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{S_{i^*+1,j}\}}(i+1, m_1)$.

Program Transform$_{B,4,j,3}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{S_{i^*+1,j}\}$ of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return 'fail';
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,j}\}}(l)$; if $\mathsf{out} =$ 'fail' then return 'fail'; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return 'fail';
4. If $m_1 = m_1^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
5. If $m_1 \neq m_1^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{B,4,j,3}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{S_{i^*+1,j}\}$ of ACE, punctured at the set $S_{i^*+1,j}$, upper bound T. Here $S_{a,b} = \{(a, m_1^*), (a+1, m_1^*), \ldots, (b, m_1^*)\}$ if $b \geq a$ and $\{\varnothing\}$ otherwise.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{S_{i^*+1,j}\}}(l)$; if $\mathsf{out} =$ 'fail' then output 'fail'; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output 'fail';
3. Output m_1.

Figure 53: Programs in $\mathsf{Hyb}_{B,4,j,3}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and RetrieveTags, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{B,5,1}$.

Program GenZero$_{B,5,1}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: punctured encryption key $\mathsf{EK}_1\{p_{i^*+1}\}$ of ACE, punctured at the point $p_{i^*+1} = (i^* + 1, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{i^*+1}\}}(0, m_1)$.

Program Increment$_{B,5,1}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_{i^*+1}\}$, DK_1 of ACE, punctured at the point $p_{i^*+1} = (i^* + 1, m_1^*)$, tag m_1^*, , upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_{i^*+1}\}}(i + 1, m_1)$.

Program Transform$_{B,5,1}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i - 1, m_1, m_2)$;
5. If $m_1 \neq m_1^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{B,5,1}(l)$

Inputs: single-tag level l

Hardwired values: decryption key DK_1 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Figure 54: Programs in $\mathsf{Hyb}_{B,5,1}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and $\mathsf{RetrieveTags}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{B,5,2}$.

Program GenZero$_{B,5,2}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key EK_1 of ACE, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1)$.

Program Increment$_{B,5,2}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys EK_1, DK_1 of ACE, tag m_1^*, OWF g, $I^* = g(i^*)$ for random i^*, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. If $m_1 = m_1^*$ and $g(i) = I^*$ then output $'\mathsf{fail}'$;
4. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(i+1, m_1)$.

Program Transform$_{B,5,2}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
5. If $m_1 \neq m_1^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{B,5,2}(l)$

Inputs: single-tag level l

Hardwired values: decryption key DK_1 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Figure 55: Programs in $\mathsf{Hyb}_{B,5,2}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and $\mathsf{RetrieveTags}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{B,5,3}$.

Program GenZero$_{B,5,3}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key EK_1 of ACE, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1)$.

Program Increment$_{B,5,3}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1, \mathsf{DK}_1$ of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(i + 1, m_1)$.

Program Transform$_{B,5,3}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i - 1, m_1, m_2)$;
5. If $m_1 \neq m_1^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program RetrieveTag$_{B,5,3}(l)$

Inputs: single-tag level l

Hardwired values: decryption key DK_1 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Figure 56: Programs in $\mathsf{Hyb}_{B,5,3}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and $\mathsf{RetrieveTags}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

7.4.3 Proof of lemma 4 (Restoring behavior of $\mathsf{Transform}$).

Starting from Hyb_C, we first change outputs of $\mathsf{Transform}$ from $[i-1, m_1^*, m_2]$ to $[i, m_1^*, m_2]$ for different $m_2 \neq m_2^*$ one by one, by considering the following sequence of hybrids for $q = 0, \ldots, \nu_2$, $q \neq m_2^*$, where $\nu_2 = 2^{|m_2|}$:

- $\mathsf{Hyb}_{C,1,q}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$, $\mathsf{Transform}_{C,1,q}[(l_1^*, m_2^*)]$, $\mathsf{isLess}_{C,1,q}$, $\mathsf{RetrieveTag}$, $\mathsf{RetrieveTags}_{C,1,q}$, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 58.

  That is, program $\mathsf{Transform}$ on input $([i, m_1^*], m_2)$ outputs $[i-1, m_1^*, m_2]$ for $m_2 \geq q$ or $m_2 = m_2^*$ and $[i, m_1^*, m_2]$ otherwise.

  Note that $\mathsf{Hyb}_C = \mathsf{Hyb}_{C,1,q}$ for $q = 0$.

In the following sequence of hybrids we change the output at $m_2 = q$ from $[i-1, m_1^*, q]$ to $[i, m_1^*, q]$:

- $\mathsf{Hyb}_{C,1,q,1,1}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$, $\mathsf{Transform}_{C,1,q,1,1}[(l_1^*, m_2^*)]$, $\mathsf{isLess}_{C,1,q,1,1}$, $\mathsf{RetrieveTag}$, $\mathsf{RetrieveTags}_{C,1,q,1,1}$, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 59.

  That is, in program $\mathsf{Transform}$ we puncture ACE encryption key EK_2 at the point $p_{T,q} = (T, m_1^*, q)$. Indistinguishability holds by iO, since $\mathsf{Transform}$ never encrypts this plaintext.

- $\mathsf{Hyb}_{C,1,q,1,2}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$, $\mathsf{Transform}_{C,1,q,1,2}[(l_1^*, m_2^*)]$, $\mathsf{isLess}_{C,1,q,1,2}$, $\mathsf{RetrieveTag}$, $\mathsf{RetrieveTags}_{C,1,q,1,2}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 60.

  That is, in programs isLess and $\mathsf{RetrieveTags}$ we puncture ACE decryption key DK_2 at the point $p_{T,q} = (T, m_1^*, q)$. Indistinguishability holds by security of constrained ACE key, since EK_2 is already punctured at the same point.

We consider the following hybrids for $j = T-1, \ldots, 0$, switching the output from $[j, m_1^*, q]$ to $[j+1, m_1^*, q]$:

- $\mathsf{Hyb}_{C,1,q,2,j,1}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$, $\mathsf{Transform}_{C,1,q,2,j,1}[(l_1^*, m_2^*)]$, $\mathsf{isLess}_{C,1,q,2,j,1}$, $\mathsf{RetrieveTag}$, $\mathsf{RetrieveTags}_{C,1,q,2,j,1}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 61.

  That is, in this hybrid $\mathsf{EK}_2, \mathsf{DK}_2$ are punctured at the point $p_{j+1,q} = (j+1, m_1^*, q)$.

  Note that $\mathsf{Hyb}_{C,1,q,1,2} = \mathsf{Hyb}_{C,1,q,2,j,1}$ for $j = T-1$.

- $\mathsf{Hyb}_{C,1,q,2,j,2}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$, $\mathsf{Transform}_{C,1,q,2,j,2}[(l_1^*, m_2^*)]$, $\mathsf{isLess}_{C,1,q,2,j,2}$, $\mathsf{RetrieveTag}$, $\mathsf{RetrieveTags}_{C,1,q,2,j,2}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 62.

  That is, we additionally puncture ACE keys $\mathsf{EK}_2, \mathsf{DK}_2$ at the point $p_{j,q} = (j, m_1^*, q)$ and hardwire $L_{j,q}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, q)$ to eliminate the need to encrypt or decrypt $p_{j,q}$ in programs $\mathsf{Transform}$, isLess, and RetrieveTags. Indistinguishability holds by iO.

Note that in program isLess we instruct the program to use the value $p_{j+1,q} = (j+1, m_1^*, q)$ on input $L^*_{j,q}$ (instead of correct value $p_{j,q} = (j, m_1^*, q)$). However, this doesn't change the overall functionality of the program: using $p_{j+1,q}$ instead of $p_{j,q}$ could change the result of comparison only if the other input was an encryption of $p_{j+1,q}$ (since comparison will result in true when $p_{j,q}$ is used and false when $p_{j+1,q}$ is used). However, DK_2 is punctured at a set which includes $p_{j+1,q}$, and thus no ciphertext is decrypted to $p_{j+1,q}$. Thus programs $\mathsf{isLess}_{12,q,2,j,1}$ and $\mathsf{isLess}_{12,q,2,j,0}$ have the same functionality.

- $\mathsf{Hyb}_{C,1,q,2,j,3}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*]$, Increment, $\mathsf{Transform}_{C,1,q,2,j,3}[(l_1^*, m_2^*)]$, $\mathsf{isLess}_{C,1,q,2,j,3}$, RetrieveTag, $\mathsf{RetrieveTags}_{C,1,q,2,j,3}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 63.

  That is, we replace $L^*_{j,q} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, q)$ with $L^*_{j+1,q} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j+1, m_1^*, q)$ in programs Transform, isLess and RetrieveTags. Indistinguishability holds by security of ACE for punctured points $p_{j,q}, p_{j+1,q}$.

- $\mathsf{Hyb}_{C,1,q,2,j,4}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*]$, Increment, $\mathsf{Transform}_{C,1,q,2,j,4}[(l_1^*, m_2^*)]$, $\mathsf{isLess}_{C,1,q,2,j,4}$, RetrieveTag, $\mathsf{RetrieveTags}_{C,1,q,2,j,4}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 64.

  That is, we unpuncture ACE keys $\mathsf{EK}_2, \mathsf{DK}_2$ at the point $p_{j+1,q} = (j+1, m_1^*, q)$ and remove hardwired $L^*_{j+1,q} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j+1, m_1^*, q)$ in programs Transform, isLess, and RetrieveTags. Indistinguishability holds by iO.

  Note that $\mathsf{Hyb}_{C,1,q,2,j,4} = \mathsf{Hyb}_{C,1,q,2,j-1,1}$ for $j = T-1, \ldots, 1$.

Next we separately consider the case $j = -1$, switching the output from $[-1, m_1^*, q]$ to $[0, m_1^*, q]$:

- $\mathsf{Hyb}_{C,1,q,2,-1,1}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*]$, Increment, $\mathsf{Transform}_{C,1,q,2,-1,1}[(l_1^*, m_2^*)]$, $\mathsf{isLess}_{C,1,q,2,-1,1}$, RetrieveTag, $\mathsf{RetrieveTags}_{C,1,q,2,-1,1}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 65.

  That is, in this hybrid $\mathsf{EK}_2, \mathsf{DK}_2$ are punctured at the point $p_{0,q} = (0, m_1^*, q)$.

  Note that $\mathsf{Hyb}_{C,1,q,2,-1,1} = \mathsf{Hyb}_{C,1,q,2,j,4}$ for $j = 0$.

- $\mathsf{Hyb}_{C,1,q,2,-1,2}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*]$, Increment, $\mathsf{Transform}_{C,1,q,2,-1,2}[(l_1^*, m_2^*)]$, $\mathsf{isLess}_{C,1,q,2,-1,2}$, RetrieveTag, $\mathsf{RetrieveTags}_{C,1,q,2,-1,2}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 66.

  That is, we additionally puncture ACE keys $\mathsf{EK}_2, \mathsf{DK}_2$ at the point $p_{-1,q} = (-1, m_1^*, q)$ and hardwire $L^*_{-1,q} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(-1, m_1^*, q)$ to eliminate the need to encrypt or decrypt $p_{-1,q}$ in programs Transform, isLess, and RetrieveTags. Indistinguishability holds by iO.

  Note that in programs isLess and RetrieveTags we instruct the program to output fail, given $L^*_{-1,q} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(-1, m_1^*, q)$ as input, since both programs treat levels with $i < 0$ as invalid.

- $\mathsf{Hyb}_{C,1,q,2,-1,3}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} =$

$\mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$, $\mathsf{Transform}_{C,1,q,2,-1,3}[(l_1^*, m_2^*)]$, $\mathsf{isLess}_{C,1,q,2,-1,3}$, $\mathsf{RetrieveTag}$, $\mathsf{RetrieveTags}_{C,1,q,2,-1,3}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 67.

That is, we replace $L_{-1,q}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(-1, m_1^*, q)$ with $L_{0,q}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, q)$ in programs Transform, isLess and RetrieveTags. Indistinguishability holds by security of ACE for punctured points $p_{-1,q}, p_{0,q}$.

Next we clean up punctured keys:

- $\mathsf{Hyb}_{C,1,q,3,1}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$, $\mathsf{Transform}_{C,1,q,3,1}[(l_1^*, m_2^*)]$, $\mathsf{isLess}_{C,1,q,3,1}$, $\mathsf{RetrieveTag}$, $\mathsf{RetrieveTags}_{C,1,q,3,1}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 68.

  That is, we unpuncture ACE keys $\mathsf{EK}_2, \mathsf{DK}_2$ at the point $p_{0,q} = (0, m_1^*, q)$ and remove hardwired $L_{0,q}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, q)$ in programs Transform, isLess, and RetrieveTags. Indistinguishability holds by iO.

- $\mathsf{Hyb}_{C,1,q,3,2}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$, $\mathsf{Transform}_{C,1,q,3,2}[(l_1^*, m_2^*)]$, $\mathsf{isLess}_{C,1,q,3,2}$, $\mathsf{RetrieveTag}$, $\mathsf{RetrieveTags}_{C,1,q,3,2}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 69.

  That is, we unpuncture ACE key DK_2 at the point $p_{-1,q} = (-1, m_1^*, q)$ in programs Transform, isLess, and RetrieveTags. Indistinguishability holds by security of a constrained ACE key, since EK_2 is punctured at $p_{-1,q}$.

- $\mathsf{Hyb}_{C,1,q,3,3}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$, $\mathsf{Transform}_{C,1,q,3,3}[(l_1^*, m_2^*)]$, $\mathsf{isLess}_{C,1,q,3,3}$, $\mathsf{RetrieveTag}$, $\mathsf{RetrieveTags}_{C,1,q,3,3}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 70.

  That is, we unpuncture ACE key EK_2 at the point $p_{-1,q} = (-1, m_1^*, q)$ in program Transform. Indistinguishability holds by iO, since Transform never encrypts this value.

Note that programs isLess and RetrieveTags now output 'fail' on input $[0, m_1^*, q]$. We fix this in the following hybrids:

- $\mathsf{Hyb}_{C,1,q,4,1}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{C,1,q,4,1}[m_1^*]$, $\mathsf{Increment}_{C,1,q,4,1}$, $\mathsf{Transform}_{C,1,q,4,1}[(l_1^*, m_2^*)]$, $\mathsf{isLess}_{C,1,q,4,1}$, $\mathsf{RetrieveTag}_{C,1,q,4,1}$, $\mathsf{RetrieveTags}_{C,1,q,4,1}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 71.

  That is, in this hybrid we puncture ACE encryption key EK_1 at $p_0 = (0, m_1^*)$ in programs GenZero and Increment. Indistinguishability holds by iO, since these programs never encrypt p_0.

- $\mathsf{Hyb}_{C,1,q,4,2}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{C,1,q,4,2}[m_1^*]$, $\mathsf{Increment}_{C,1,q,4,2}$, $\mathsf{Transform}_{C,1,q,4,2}[(l_1^*, m_2^*)]$, $\mathsf{isLess}_{C,1,q,4,2}$, $\mathsf{RetrieveTag}_{C,1,q,4,2}$, $\mathsf{RetrieveTags}_{C,1,q,4,2}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 72.

That is, in this hybrid we puncture ACE decryption key DK_1 at the same point $p_0 = (0, m_1^*)$ in programs Increment, Transform, and RetrieveTag. Indistinguishability holds by security of constrained decryption of ACE, since corresponding encryption key EK_1 is already punctured at p_0.

- $\mathsf{Hyb}_{C,1,q,4,3}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{C,1,q,4,3}[m_1^*], \mathsf{Increment}_{C,1,q,4,3}, \mathsf{Transform}_{C,1,q,4,3}[(l_1^*, m_2^*)], \mathsf{isLess}_{C,1,q,4,3}, \mathsf{RetrieveTag}_{C,1,q,4,3}, \mathsf{RetrieveTags}_{C,1,q,4,3}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 73.

  That is, in this hybrid we puncture ACE encryption key EK_2 at $p_{0,q} = (0, m_1^*, q)$ in program Transform. Indistinguishability holds by security of iO, since, due to punctured $\mathsf{DK}_1\{p_0\}$, this program always outputs 'fail' on input $([0, m_1^*], q)$ and thus never needs to encrypt $p_{0,q}$.

- $\mathsf{Hyb}_{C,1,q,4,4}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{C,1,q,4,4}[m_1^*], \mathsf{Increment}_{C,1,q,4,4}, \mathsf{Transform}_{C,1,q,4,4}[(l_1^*, m_2^*)], \mathsf{isLess}_{C,1,q,4,4}, \mathsf{RetrieveTag}_{C,1,q,4,4}, \mathsf{RetrieveTags}_{C,1,q,4,4}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 74.

  That is, in this hybrid we puncture ACE decryption key DK_2 at the same point $p_{0,q} = (0, m_1^*, q)$ in programs isLess and RetrieveTags. Indistinguishability holds by security of constrained decryption of ACE, since corresponding encryption key EK_2 is already punctured at $p_{0,q}$.

- $\mathsf{Hyb}_{C,1,q,4,5}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{C,1,q,4,5}[m_1^*], \mathsf{Increment}_{C,1,q,4,5}, \mathsf{Transform}_{C,1,q,4,5}[(l_1^*, m_2^*)], \mathsf{isLess}_{C,1,q,4,5}, \mathsf{RetrieveTag}_{C,1,q,4,5}, \mathsf{RetrieveTags}_{C,1,q,4,5}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 75.

  That is, we remove instructions to output 'fail' in programs isLess and RetrieveTags on input $[0, m_1^*, q]$. Indistinguishability holds by iO, since these instructions are never executed due to the fact that DK_2 is punctured at $p_{0,q} = (0, m_1^*, q)$ and thus the programs output 'fail' during decryption.

- $\mathsf{Hyb}_{C,1,q,4,6}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{C,1,q,4,6}[m_1^*], \mathsf{Increment}_{C,1,q,4,6}, \mathsf{Transform}_{C,1,q,4,6}[(l_1^*, m_2^*)], \mathsf{isLess}_{C,1,q,4,6}, \mathsf{RetrieveTag}_{C,1,q,4,6}, \mathsf{RetrieveTags}_{C,1,q,4,6}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 76.

  That is, in this hybrid we unpuncture ACE decryption key DK_2 at $p_{0,q} = (0, m_1^*, q)$ in programs isLess and RetrieveTags. Indistinguishability holds by security of constrained decryption of ACE, since corresponding encryption key EK_2 is punctured at $p_{0,q}$.

- $\mathsf{Hyb}_{C,1,q,4,7}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{C,1,q,4,7}[m_1^*], \mathsf{Increment}_{C,1,q,4,7}, \mathsf{Transform}_{C,1,q,4,7}[(l_1^*, m_2^*)], \mathsf{isLess}_{C,1,q,4,7}, \mathsf{RetrieveTag}_{C,1,q,4,7}, \mathsf{RetrieveTags}_{C,1,q,4,7}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 77.

That is, in this hybrid we unpuncture ACE encryption key EK_2 at $p_{0,q} = (0, m_1^*, q)$ in program Transform. Indistinguishability holds by security of iO, since, due to punctured $\mathsf{DK}_1\{p_0\}$, this program always outputs 'fail' on input $([0, m_1^*], q)$ and thus never needs to encrypt $p_{0,q}$.

- $\mathsf{Hyb}_{C,1,q,4,8}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{C,1,q,4,8}[m_1^*], \mathsf{Increment}_{C,1,q,4,8}, \mathsf{Transform}_{C,1,q,4,8}[(l_1^*, m_2^*)], \mathsf{isLess}_{C,1,q,4,8}, \mathsf{RetrieveTag}_{C,1,q,4,8}, \mathsf{RetrieveTags}_{C,1,q,4,8}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 78.

  That is, in this hybrid we unpuncture ACE decryption key DK_1 at $p_0 = (0, m_1^*)$ in programs Increment, Transform, and RetrieveTag. Indistinguishability holds by security of constrained decryption of ACE, since corresponding encryption key EK_1 is punctured at p_0.

- $\mathsf{Hyb}_{C,1,q,4,9}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{C,1,q,4,9}[m_1^*], \mathsf{Increment}_{C,1,q,4,9}, \mathsf{Transform}_{C,1,q,4,9}[(l_1^*, m_2^*)], \mathsf{isLess}_{C,1,q,4,9}, \mathsf{RetrieveTag}_{C,1,q,4,9}, \mathsf{RetrieveTags}_{C,1,q,4,9}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 79.

  That is, in this hybrid we unpuncture ACE encryption key EK_1 at $p_0 = (0, m_1^*)$ in programs GenZero and Increment. Indistinguishability holds by iO, since these programs never encrypt p_0.

This concludes fixing behavior of Transform for the case $m_2 \neq m_2^*$. Next we fix the case $m_2 = m_2^*$ in a similar manner, except that we need different hybrids for the case $j = -1, 0$ (to prevent switching L_0^* to L_1^*):

- $\mathsf{Hyb}_{C,2,1,1}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*], \mathsf{Increment}, \mathsf{Transform}_{C,2,1,1}[(l_1^*, m_2^*)], \mathsf{isLess}_{C,2,1,1}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}_{C,2,1,1}$, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 80.

  Note that $\mathsf{Hyb}_{C,1,q,4,9} = \mathsf{Hyb}_{C,2,1,1}$ for $q = 2^{|m_2|}$.

- $\mathsf{Hyb}_{C,2,1,2}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*], \mathsf{Increment}, \mathsf{Transform}_{C,2,1,2}[(l_1^*, m_2^*)], \mathsf{isLess}_{C,2,1,2}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}_{C,2,1,2}$, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 81.

  That is, in program Transform we puncture ACE encryption key EK_2 at the point $p_{T,m_2^*} = (T, m_1^*, m_2^*)$. Indistinguishability holds by iO, since Transform never encrypts this plaintext.

- $\mathsf{Hyb}_{C,2,1,3}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*], \mathsf{Increment}, \mathsf{Transform}_{C,2,1,3}[(l_1^*, m_2^*)], \mathsf{isLess}_{C,2,1,3}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}_{C,2,1,3}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 82.

  That is, in programs isLess and RetrieveTags we puncture ACE decryption key DK_2 at the point $p_{T,m_2^*} = (T, m_1^*, m_2^*)$. Indistinguishability holds by security of constrained ACE key, since EK_2 is already punctured at the same point.

We consider the following hybrids for $j = T-1, \dots, 1$, switching the output from $[j, m_1^*, m_2^*]$ to $[j+1, m_1^*, m_2^*]$:

- $\mathsf{Hyb}_{C,2,2,j,1}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*], \mathsf{Increment}, \mathsf{Transform}_{C,2,2,j,1}[(l_1^*, m_2^*)], \mathsf{isLess}_{C,2,2,j,1}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}_{C,2,2,j,1}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 83.

  That is, in this hybrid $\mathsf{EK}_2, \mathsf{DK}_2$ are punctured at the point $p_{j+1,m_2^*} = (j+1, m_1^*, m_2^*)$.

  Note that $\mathsf{Hyb}_{C,2,1,3} = \mathsf{Hyb}_{C,2,2,j,1}$ for $j = T-1$.

- $\mathsf{Hyb}_{C,2,2,j,2}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*], \mathsf{Increment}, \mathsf{Transform}_{C,2,2,j,2}[(l_1^*, m_2^*)], \mathsf{isLess}_{C,2,2,j,2}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}_{C,2,2,j,2}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 84.

  That is, we additionally puncture ACE keys $\mathsf{EK}_2, \mathsf{DK}_2$ at the point $p_{j,m_2^*} = (j, m_1^*, m_2^*)$ and hardwire $L_{j,m_2^*}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, m_2^*)$ to eliminate the need to encrypt or decrypt p_{j,m_2^*} in programs Transform, isLess, and RetrieveTags. Indistinguishability holds by iO.

  Note that in program isLess we instruct the program to use the value $p_{j+1,m_2^*} = (j+1, m_1^*, m_2^*)$ on input $L_{j,m_2^*}^*$ (instead of correct value $p_{j,m_2^*} = (j, m_1^*, m_2^*)$). However, this doesn't change the overall functionality of the program: using p_{j+1,m_2^*} instead of p_{j,m_2^*} could change the result of comparison only if the other input was an encryption of p_{j+1,m_2^*} (since comparison will result in true when p_{j,m_2^*} is used and false when p_{j+1,m_2^*} is used). However, DK_2 is punctured at a set which includes p_{j+1,m_2^*}, and thus no ciphertext is decrypted to p_{j+1,m_2^*}. Thus programs $\mathsf{isLess}_{C,2,2,j,1}$ and $\mathsf{isLess}_{C,2,2,j,2}$ have the same functionality.

- $\mathsf{Hyb}_{C,2,2,j,3}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*], \mathsf{Increment}, \mathsf{Transform}_{C,2,2,j,3}[(l_1^*, m_2^*)], \mathsf{isLess}_{C,2,2,j,3}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}_{C,2,2,j,3}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 85.

  That is, we replace $L_{j,m_2^*}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, m_2^*)$ with $L_{j+1,m_2^*}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j+1, m_1^*, m_2^*)$ in programs Transform, isLess and RetrieveTags. Indistinguishability holds by security of ACE for punctured points $p_{j,m_2^*}, p_{j+1,m_2^*}$.

- $\mathsf{Hyb}_{C,2,2,j,4}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*], \mathsf{Increment}, \mathsf{Transform}_{C,2,2,j,4}[(l_1^*, m_2^*)], \mathsf{isLess}_{C,2,2,j,4}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}_{C,2,2,j,4}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 86.

  That is, we unpuncture ACE keys $\mathsf{EK}_2, \mathsf{DK}_2$ at the point $p_{j+1,m_2^*} = (j+1, m_1^*, m_2^*)$ and remove hardwired $L_{j+1,m_2^*}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j+1, m_1^*, m_2^*)$ in programs Transform, isLess, and RetrieveTags. Indistinguishability holds by iO.

  Note that $\mathsf{Hyb}_{C,2,2,j,4} = \mathsf{Hyb}_{C,2,2,j-1,1}$ for $j = T-1, \dots, 2$.

Finally we consider the case $j = -1$, switching the output from $[-1, m_1^*, m_2^*]$ to $[0, m_1^*, m_2^*]$ and cleaning up any left puncturing:

- $\mathsf{Hyb}_{C,2,3,1}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$, $\mathsf{Transform}_{C,2,3,1}[(l_1^*, m_2^*)]$, $\mathsf{isLess}_{C,2,3,1}$, $\mathsf{RetrieveTag}$, $\mathsf{RetrieveTags}_{C,2,3,1}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 87.

  In this hybrid $\mathsf{EK}_2, \mathsf{DK}_2$ are punctured at the point $p_{1,m_2^*} = (1, m_1^*, m_2^*)$.

  Note that $\mathsf{Hyb}_{C,2,3,1} = \mathsf{Hyb}_{C,2,2,j,4}$ for $j = 1$.

- $\mathsf{Hyb}_{C,2,3,2}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$, $\mathsf{Transform}_{C,2,3,2}[(l_1^*, m_2^*)]$, $\mathsf{isLess}_{C,2,3,2}$, $\mathsf{RetrieveTag}$, $\mathsf{RetrieveTags}_{C,2,3,2}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 88.

  That is, we unpuncture ACE key DK_2 at the point $p_{1,m_2^*} = (1, m_1^*, m_2^*)$. in programs isLess and $\mathsf{RetrieveTags}$. Indistinguishability holds by security of a constrained ACE key, since EK_2 is punctured at p_{1,m_2^*}.

- $\mathsf{Hyb}_{C,2,3,3}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{C,2,3,3}[m_1^*]$, $\mathsf{Increment}_{C,2,3,3}$, $\mathsf{Transform}_{C,2,3,3}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{C,2,3,3}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 89.

  That is, we change the following: first, we puncture ACE key EK_1 at the point $p_0 = (0, m_1^*)$ in programs $\mathsf{GenZero}$ and $\mathsf{Increment}$: this is without changing the functionality of those programs, since then never need to encrypt p_0. Second, we unpuncture ACE key EK_2 at point $p_{1,m_2^*} = (1, m_1^*, m_2^*)$ in program $\mathsf{Transform}$, since this program never needs to encrypt p_{1,m_2^*} due to the first instruction (which tells the program to output $'\mathsf{fail}'$ if it gets $([1, m_1^*], m_2^*)$ as input)). Indistinguishability holds by iO.

- $\mathsf{Hyb}_{C,2,3,4}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{C,2,3,4}[m_1^*]$, $\mathsf{Increment}_{C,2,3,4}$, $\mathsf{Transform}_{C,2,3,4}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{C,2,3,4}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 90.

  That is, in programs $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$ we puncture ACE decryption key DK_1 at the point $p_0 = (0, m_1^*)$. Indistinguishability holds by security of constrained ACE key, since EK_1 is already punctured at the same point.

- $\mathsf{Hyb}_{C,2,3,5}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{C,2,3,5}[m_1^*]$, $\mathsf{Increment}_{C,2,3,5}$, $\mathsf{Transform}_{C,2,3,5}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{C,2,3,5}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 91.

  That is, we let program $\mathsf{Transform}$ output $[0, m_1^*, m_2^*]$ (instead of $[-1, m_1^*, m_2^*]$) on input $([0, m_1^*], m_2^*)$. This doesn't change the functionality of the program, since DK_1 is punctured the point $p_0 = (0, m_1^*)$, thus no valid encryption of $(0, m_1^*)$ exists, and $\mathsf{Transform}$ aborts on input $[0, m_1^*], m_2^*$. Indistinguishability holds by iO.

- $\mathsf{Hyb}_{C,2,3,6}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{C,2,3,6}[m_1^*]$, $\mathsf{Increment}_{C,2,3,6}$, $\mathsf{Transform}_{C,2,3,6}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{C,2,3,6}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 92.

That is, in programs $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$ we unpuncture ACE decryption key DK_1 at the point $p_0 = (0, m_1^*)$. Indistinguishability holds by security of constrained ACE key, since EK_1 is already punctured at the same point.

- $\mathsf{Hyb}_{C,2,3,7}$. We give the adversary $(\mathsf{PP}, l_1^*, L_0^*, m_1^*, m_2^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{GenZero}_{C,2,3,7}[m_1^*]$, $\mathsf{Increment}_{C,2,3,7}$, $\mathsf{Transform}_{C,2,3,7}[(l_1^*, m_2^*)]$, isLess, $\mathsf{RetrieveTag}_{C,2,3,7}$, $\mathsf{RetrieveTags}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}, $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. Description of the programs can be found on fig. 93.

  That is, we unpuncture ACE key EK_1 at the point $p_0 = (0, m_1^*)$ in programs $\mathsf{GenZero}$ and $\mathsf{Increment}$. Indistinguishability holds by iO, since neither program encrypts this value.

  Note that $\mathsf{Hyb}_{C,2,3,7} = \mathsf{Hyb}_D$.

Thus, the the advantage of the PPT adversary in distinguishing between Hyb_C and Hyb_D is at most

$$(2^{\tau(\lambda)} - 1)((2T+9) \cdot 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))} + (T+1) \cdot 2^{-\Omega(\nu_{\mathsf{ACE.Indist}}(\lambda))} + 6 \cdot 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}) +$$

$$(2(T-1)+4) \cdot 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))} + (T-1) \cdot 2^{-\Omega(\nu_{\mathsf{ACE.Indist}}(\lambda))} + 4 \cdot 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))} =$$

$$2^{\tau(\lambda)}(T \cdot 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))} + T \cdot 2^{-\Omega(\nu_{\mathsf{ACE.Indist}}(\lambda))} + 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}).$$

Programs in Hyb_C.

Program GenZero$_C[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key EK_1 of ACE, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1)$.

Program Increment$_C(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1, \mathsf{DK}_1$ of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(i+1, m_1)$.

Program Transform$_C[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
5. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program isLess$_C(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTag$_C(l)$

Inputs: single-tag level l

Hardwired values: decryption key DK_1 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Program RetrieveTags$_C(L)$

Inputs: double-tag level L

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1, m_2.

Figure 57: Programs in Hyb_C. In addition, in this hybrid the adversary gets $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q}$.

Program Transform$_{C,1,q}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 \geq q$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
6. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$;

Program isLess$_{C,1,q}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,1,q}(L)$

Inputs: double-tag level L

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1, m_2.

Figure 58: Programs in $\mathsf{Hyb}_{C,1,q}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,1,1}$.

Program Transform$_{C,1,q,1,1}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key $\mathsf{EK}_2\{p_{T,q}\}$ of ACE punctured at $p_{T,q} = (T, m_1^*, q)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{T,q}\}}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 \geq q$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{T,q}\}}(i-1, m_1, m_2)$;
6. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{T,q}\}}(i, m_1, m_2)$;

Program isLess$_{C,1,q,1,1}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,1,q,1,1}(L)$

Inputs: double-tag level L

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1, m_2.

Figure 59: Programs in $\mathsf{Hyb}_{C,1,q,1,1}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,1,2}$.

Program Transform$_{C,1,q,1,2}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key $\mathsf{EK}_2\{p_{T,q}\}$ of ACE punctured at $p_{T,q} = (T, m_1^*, q)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{T,q}\}}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 \geq q$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{T,q}\}}(i-1, m_1, m_2)$;
6. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{T,q}\}}(i, m_1, m_2)$;

Program isLess$_{C,1,q,1,2}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key $\mathsf{DK}_2\{p_{T,q}\}$ of ACE punctured at $p_{T,q} = (T, m_1^*, q)$, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{T,q}\}}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{T,q}\}}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,1,q,1,2}(L)$

Inputs: double-tag level L

Hardwired values: decryption key $\mathsf{DK}_2\{p_{T,q}\}$ of ACE punctured at $p_{T,q} = (T, m_1^*, q)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{T,q}\}}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1, m_2.

Figure 60: Programs in $\mathsf{Hyb}_{C,1,q,1,2}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,2,j,1}$.

Program Transform$_{C,1,q,2,j,1}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key $\mathsf{EK}_2\{p_{j+1,q}\}$ of ACE punctured at $p_{j+1,q} = (j+1, m_1^*, q)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j+1,q}\}}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 > q$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j+1,q}\}}(i-1, m_1, m_2)$;
6. If $m_1 = m_1^*$, $m_2 = q$, and $i \leq j+1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j+1,q}\}}(i-1, m_1, m_2)$;
7. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j+1,q}\}}(i, m_1, m_2)$.

Program isLess$_{C,1,q,2,j,1}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key $\mathsf{DK}_2\{p_{j+1,q}\}$ of ACE punctured at $p_{j+1,q} = (j+1, m_1^*, q)$, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j+1,q}\}}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j+1,q}\}}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,1,q,2,j,1}(L)$

Inputs: double-tag level L

Hardwired values: decryption key $\mathsf{DK}_2\{p_{j+1,q}\}$ of ACE punctured at $p_{j+1,q} = (j+1, m_1^*, q)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j+1,q}\}}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1, m_2.

Figure 61: Programs in $\mathsf{Hyb}_{C,1,q,2,j,1}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,2,j,2}$.

Program Transform$_{C,1,q,2,j,2}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key $\mathsf{EK}_2\{p_{j,q}, p_{j+1,q}\}$ of ACE punctured at $p_{j,q} = (j, m_1^*, q)$, $p_{j+1,q} = (j+1, m_1^*, q)$, double-tag level $L_{j,q}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, q)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return 'fail';
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} =$ 'fail' then return 'fail'; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return 'fail';
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j,q},p_{j+1,q}\}}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 > q$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j,q},p_{j+1,q}\}}(i-1, m_1, m_2)$;
6. If $m_1 = m_1^*$, $m_2 = q$, and $i = j+1$ return $L_{j,q}^*$;
7. If $m_1 = m_1^*$, $m_2 = q$, and $i < j+1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j,q},p_{j+1,q}\}}(i-1, m_1, m_2)$;
8. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j,q},p_{j+1,q}\}}(i, m_1, m_2)$.

Program isLess$_{C,1,q,2,j,2}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key $\mathsf{DK}_2\{p_{j,q}, p_{j+1,q}\}$ of ACE punctured at $p_{j,q} = (j, m_1^*, q)$, $p_{j+1,q} = (j+1, m_1^*, q)$, double-tag level $L_{j,q}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, q)$, upper bound T.

1. If $L' = L_{j,q}^*$ then set $(i', m_1', m_2') = (j+1, m_1^*, q)$, else $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j,q},p_{j+1,q}\}}(L')$; if $\mathsf{out}' =$ 'fail' then output 'fail'; else parse out' as (i', m_1', m_2').
2. If $L'' = L_{j,q}^*$ then set $(i'', m_1'', m_2'') = (j+1, m_1^*, q)$, else $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j,q},p_{j+1,q}\}}(L'')$; if $\mathsf{out}'' =$ 'fail' then output 'fail'; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output 'fail';
4. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,1,q,2,j,2}(L)$

Inputs: double-tag level L

Hardwired values: decryption key $\mathsf{DK}_2\{p_{j,q}, p_{j+1,q}\}$ of ACE punctured at $p_{j,q} = (j, m_1^*, q)$, $p_{j+1,q} = (j+1, m_1^*, q)$, double-tag level $L_{j,q}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, q)$, upper bound T.

1. If $L = L_{j,q}^*$ then return (m_1^*, q);
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j,q},p_{j+1,q}\}}(L)$; if $\mathsf{out} =$ 'fail' then output 'fail'; else parse out as (i, m_1, m_2).
3. If $i > T$ or $i < 0$ then output 'fail';
4. Return (m_1, m_2).

Figure 62: Programs in $\mathsf{Hyb}_{C,1,q,2,j,2}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,2,j,3}$.

Program Transform $_{C,1,q,2,j,3}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key $\mathsf{EK}_2\{p_{j,q}, p_{j+1,q}\}$ of ACE punctured at $p_{j,q} = (j, m_1^*, q)$, $p_{j+1,q} = (j+1, m_1^*, q)$, double-tag level $L_{j+1,q}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j+1, m_1^*, q)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j,q},p_{j+1,q}\}}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 > q$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j,q},p_{j+1,q}\}}(i-1, m_1, m_2)$;
6. If $m_1 = m_1^*$, $m_2 = q$, and $i = j+1$ return $L_{j+1,q}^*$;
7. If $m_1 = m_1^*$, $m_2 = q$, and $i < j+1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j,q},p_{j+1,q}\}}(i-1, m_1, m_2)$;
8. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j,q},p_{j+1,q}\}}(i, m_1, m_2)$.

Program isLess $_{C,1,q,2,j,3}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key $\mathsf{DK}_2\{p_{j,q}, p_{j+1,q}\}$ of ACE punctured at $p_{j,q} = (j, m_1^*, q)$, $p_{j+1,q} = (j+1, m_1^*, q)$, double-tag level $L_{j+1,q}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j+1, m_1^*, q)$, upper bound T.

1. If $L' = L_{j+1,q}^*$ then set $(i', m_1', m_2') = (j+1, m_1^*, q)$, else $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j,q},p_{j+1,q}\}}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. If $L'' = L_{j+1,q}^*$ then set $(i'', m_1'', m_2'') = (j+1, m_1^*, q)$, else $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j,q},p_{j+1,q}\}}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTags $_{C,1,q,2,j,3}(L)$

Inputs: double-tag level L

Hardwired values: decryption key $\mathsf{DK}_2\{p_{j,q}, p_{j+1,q}\}$ of ACE punctured at $p_{j,q} = (j, m_1^*, q)$, $p_{j+1,q} = (j+1, m_1^*, q)$, double-tag level $L_{j+1,q}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j+1, m_1^*, q)$, upper bound T.

1. If $L = L_{j+1,q}^*$ then return (m_1^*, q);
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j,q},p_{j+1,q}\}}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
3. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
4. Return (m_1, m_2).

Figure 63: Programs in $\mathsf{Hyb}_{C,1,q,2,j,3}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,2,j,4}$.

Program Transform$_{C,1,q,2,j,4}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key $\mathsf{EK}_2\{p_{j,q}\}$ of ACE punctured at $p_{j,q} = (j, m_1^*, q)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j,q}\}}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 > q$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j,q}\}}(i-1, m_1, m_2)$;
6. If $m_1 = m_1^*$, $m_2 = q$, and $i \leq j$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j,q}\}}(i-1, m_1, m_2)$;
7. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j,q}\}}(i, m_1, m_2)$.

Program isLess$_{C,1,q,2,j,4}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key $\mathsf{DK}_2\{p_{j,q}\}$ of ACE punctured at $p_{j,q} = (j, m_1^*, q)$, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j,q}\}}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j,q}\}}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,1,q,2,j,4}(L)$

Inputs: double-tag level L

Hardwired values: decryption key $\mathsf{DK}_2\{p_{j,q}\}$ of ACE punctured at $p_{j,q} = (j, m_1^*, q)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j,q}\}}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Return (m_1, m_2).

Figure 64: Programs in $\mathsf{Hyb}_{C,1,q,2,j,4}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,2,-1,1}$.

Program Transform$_{C,1,q,2,-1,1}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key $\mathsf{EK}_2\{p_{0,q}\}$ of ACE punctured at $p_{0,q} = (0, m_1^*, q)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{0,q}\}}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 > q$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{0,q}\}}(i-1, m_1, m_2)$;
6. If $m_1 = m_1^*$, $m_2 = q$, and $i \leq 0$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{0,q}\}}(i-1, m_1, m_2)$;
7. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{0,q}\}}(i, m_1, m_2)$.

Program isLess$_{C,1,q,2,-1,1}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key $\mathsf{DK}_2\{p_{0,q}\}$ of ACE punctured at $p_{0,q} = (0, m_1^*, q)$, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{0,q}\}}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{0,q}\}}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,1,q,2,-1,1}(L)$

Inputs: double-tag level L

Hardwired values: decryption key $\mathsf{DK}_2\{p_{0,q}\}$ of ACE punctured at $p_{0,q} = (0, m_1^*, q)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{0,q}\}}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1, m_2.

Figure 65: Programs in $\mathsf{Hyb}_{C,1,q,2,-1,1}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,2,-1,2}$.

Program Transform$_{C,1,q,2,-1,2}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key $\mathsf{EK}_2\{p_{-1,q}, p_{0,q}\}$ of ACE punctured at $p_{-1,q} = (-1, m_1^*, q)$, $p_{0,q} = (0, m_1^*, q)$, double-tag level $L^*_{-1,q} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(-1, m_1^*, q)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{-1,q}, p_{0,q}\}}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 > q$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{-1,q}, p_{0,q}\}}(i-1, m_1, m_2)$;
6. If $m_1 = m_1^*$, $m_2 = q$, and $i = 0$ return $L^*_{-1,q}$;
7. If $m_1 = m_1^*$, $m_2 = q$, and $i < 0$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{-1,q}, p_{0,q}\}}(i-1, m_1, m_2)$;
8. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{-1,q}, p_{0,q}\}}(i, m_1, m_2)$.

Program isLess$_{C,1,q,2,-1,2}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key $\mathsf{DK}_2\{p_{-1,q}, p_{0,q}\}$ of ACE punctured at $p_{-1,q} = (-1, m_1^*, q)$, $p_{0,q} = (0, m_1^*, q)$, double-tag level $L^*_{-1,q} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(-1, m_1^*, q)$, upper bound T.

1. If $L' = L^*_{-1,q}$ then output $'\mathsf{fail}'$;
   else $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{-1,q}, p_{0,q}\}}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. If $L'' = L^*_{-1,q}$ then output $'\mathsf{fail}'$;
   else $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{-1,q}, p_{0,q}\}}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,1,q,2,-1,2}(L)$

Inputs: double-tag level L

Hardwired values: decryption key $\mathsf{DK}_2\{p_{-1,q}, p_{0,q}\}$ of ACE punctured at $p_{-1,q} = (-1, m_1^*, q)$, $p_{0,q} = (0, m_1^*, q)$, double-tag level $L^*_{-1,q} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(-1, m_1^*, q)$, upper bound T.

1. If $L = L^*_{-1,q}$ then output $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{-1,q}, p_{0,q}\}}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
3. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
4. Return (m_1, m_2).

Figure 66: Programs in $\mathsf{Hyb}_{C,1,q,2,-1,2}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,2,-1,3}$.

Program Transform$_{C,1,q,2,-1,3}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key $\mathsf{EK}_2\{p_{-1,q}, p_{0,q}\}$ of ACE punctured at $p_{-1,q} = (-1, m_1^*, q)$, $p_{0,q} = (0, m_1^*, q)$, double-tag level $L_{0,q}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, q)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{-1,q},p_{0,q}\}}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 > q$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{-1,q},p_{0,q}\}}(i-1, m_1, m_2)$;
6. If $m_1 = m_1^*$, $m_2 = q$, and $i = 0$ return $L_{0,q}^*$;
7. If $m_1 = m_1^*$, $m_2 = q$, and $i < 0$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{-1,q},p_{0,q}\}}(i-1, m_1, m_2)$;
8. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{-1,q},p_{0,q}\}}(i, m_1, m_2)$.

Program isLess$_{C,1,q,2,-1,3}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key $\mathsf{DK}_2\{p_{-1,q}, p_{0,q}\}$ of ACE punctured at $p_{-1,q} = (-1, m_1^*, q)$, $p_{0,q} = (0, m_1^*, q)$, double-tag level $L_{0,q}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, q)$, upper bound T.

1. If $L' = L_{0,q}^*$ then output $'\mathsf{fail}'$;
   else $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{-1,q},p_{0,q}\}}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. If $L'' = L_{0,q}^*$ then output $'\mathsf{fail}'$;
   else $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{-1,q},p_{0,q}\}}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,1,q,2,-1,3}(L)$

Inputs: double-tag level L

Hardwired values: decryption key $\mathsf{DK}_2\{p_{-1,q}, p_{0,q}\}$ of ACE punctured at $p_{-1,q} = (-1, m_1^*, q)$, $p_{0,q} = (0, m_1^*, q)$, double-tag level $L_{0,q}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, q)$, upper bound T.

1. If $L = L_{0,q}^*$ then output $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{-1,q},p_{0,q}\}}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
3. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
4. Return (m_1, m_2).

Figure 67: Programs in $\mathsf{Hyb}_{C,1,q,2,-1,3}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,3,1}$.

Program Transform$_{C,1,q,3,1}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key $\mathsf{EK}_2\{p_{-1,q}\}$ of ACE punctured at $p_{-1,q} = (-1, m_1^*, q)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return 'fail';
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} =$ 'fail' then return 'fail'; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return 'fail';
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{-1,q}\}}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 \geq q+1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{-1,q}\}}(i-1, m_1, m_2)$;
6. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{-1,q}\}}(i, m_1, m_2)$.

Program isLess$_{C,1,q,3,1}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key $\mathsf{DK}_2\{p_{-1,q}\}$ of ACE punctured at $p_{-1,q} = (-1, m_1^*, q)$, message q, tag m_1^*, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{-1,q}\}}(L')$; if $\mathsf{out}' =$ 'fail' then output 'fail'; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{-1,q}\}}(L'')$; if $\mathsf{out}'' =$ 'fail' then output 'fail'; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output 'fail';
4. If $(i', m_1', m_2') = (0, m_1^*, q)$ then output 'fail';
5. If $(i'', m_1'', m_2'') = (0, m_1^*, q)$ then output 'fail';
6. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,1,q,3,1}(L)$

Inputs: double-tag level L

Hardwired values: decryption key $\mathsf{DK}_2\{p_{-1,q}\}$ of ACE punctured at $p_{-1,q} = (-1, m_1^*, q)$, message q, tag m_1^*, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{-1,q}\}}(L)$; if $\mathsf{out} =$ 'fail' then output 'fail'; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output 'fail';
3. If $(i, m_1, m_2) = (0, m_1^*, q)$ then output 'fail';
4. Return (m_1, m_2).

Figure 68: Programs in $\mathsf{Hyb}_{C,1,q,3,1}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,3,2}$.

Program Transform$_{C,1,q,3,2}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key $\mathsf{EK}_2\{p_{-1,q}\}$ of ACE punctured at $p_{-1,q} = (-1, m_1^*, q)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{-1,q}\}}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 \geq q+1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{-1,q}\}}(i-1, m_1, m_2)$;
6. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{-1,q}\}}(i, m_1, m_2)$.

Program isLess$_{C,1,q,3,2}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key DK_2 of ACE, message q, tag m_1^*, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $(i', m_1', m_2') = (0, m_1^*, q)$ then output $'\mathsf{fail}'$;
5. If $(i'', m_1'', m_2'') = (0, m_1^*, q)$ then output $'\mathsf{fail}'$;
6. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,1,q,3,2}(L)$

Inputs: double-tag level L

Hardwired values: decryption key DK_2 of ACE, message q, tag m_1^*, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. If $(i, m_1, m_2) = (0, m_1^*, q)$ then output $'\mathsf{fail}'$;
4. Return (m_1, m_2).

Figure 69: Programs in $\mathsf{Hyb}_{C,1,q,3,2}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,3,3}$.

Program Transform$_{C,1,q,3,3}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 \geq q+1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
6. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program isLess$_{C,1,q,3,3}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key DK_2 of ACE, message q, tag m_1^*, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $(i', m_1', m_2') = (0, m_1^*, q)$ then output $'\mathsf{fail}'$;
5. If $(i'', m_1'', m_2'') = (0, m_1^*, q)$ then output $'\mathsf{fail}'$;
6. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,1,q,3,3}(L)$

Inputs: double-tag level L

Hardwired values: decryption key DK_2 of ACE, message q, tag m_1^*, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. If $(i, m_1, m_2) = (0, m_1^*, q)$ then output $'\mathsf{fail}'$;
4. Return (m_1, m_2).

Figure 70: Programs in $\mathsf{Hyb}_{C,1,q,3,3}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,4,1}$.

Program GenZero$_{C,1,q,4,1}[m_1^*](m_1)$
Inputs: tag $m_1 \in M$.
Hardwired values: encryption key $\mathsf{EK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, tag m_1^*.
1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(0, m_1)$.

Program Increment$_{C,1,q,4,1}(l)$
Inputs: single-tag level l
Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_0\}, \mathsf{DK}_1$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.
1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(i+1, m_1)$.

Program Transform$_{C,1,q,4,1}[(l_1^*, m_2^*)](l, m_2)$
Inputs: single-tag level l, tag $m_2 \in M$
Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.
1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 \geq q+1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
6. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program isLess$_{C,1,q,4,1}(L', L'')$
Inputs: double-tag levels L', L''
Hardwired values: decryption key DK_2 of ACE, message q, tag m_1^*, upper bound T.
1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $(i', m_1', m_2') = (0, m_1^*, q)$ then output $'\mathsf{fail}'$;
5. If $(i'', m_1'', m_2'') = (0, m_1^*, q)$ then output $'\mathsf{fail}'$;
6. If $i' < i''$ then output true, else output false.

Program RetrieveTag$_{C,1,q,4,1}(l)$
Inputs: single-tag level l
Hardwired values: decryption key DK_1 of ACE, upper bound T.
1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Program RetrieveTags$_{C,1,q,4,1}(L)$
Inputs: double-tag level L
Hardwired values: decryption key DK_2 of ACE, message q, tag m_1^*, upper bound T.
1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. If $(i, m_1, m_2) = (0, m_1^*, q)$ then output $'\mathsf{fail}'$;
4. Return (m_1, m_2).

Figure 71: Programs in $\mathsf{Hyb}_{C,1,q,4,1}$. In addition, in this hybrid the adversary gets $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,4,2}$.

Program GenZero$_{C,1,q,4,2}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output 'fail';
2. output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(0, m_1)$.

Program Increment$_{C,1,q,4,2}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_0\}$, $\mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if out $=$ 'fail' then output 'fail'; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output 'fail';
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(i+1, m_1)$.

Program Transform$_{C,1,q,4,2}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return 'fail';
2. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if out $=$ 'fail' then return 'fail'; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return 'fail';
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 \geq q+1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
6. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program isLess$_{C,1,q,4,2}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key DK_2 of ACE, message q, tag m_1^*, upper bound T.

1. out$' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if out$'$ $=$ 'fail' then output 'fail'; else parse out$'$ as (i', m_1', m_2').
2. out$'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if out$''$ $=$ 'fail' then output 'fail'; else parse out$''$ as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output 'fail';
4. If $(i', m_1', m_2') = (0, m_1^*, q)$ then output 'fail';
5. If $(i'', m_1'', m_2'') = (0, m_1^*, q)$ then output 'fail';
6. If $i' < i''$ then output true, else output false.

Program RetrieveTag(l)

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if out $=$ 'fail' then output 'fail'; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output 'fail';
3. Output m_1.

Program RetrieveTags(L)

Inputs: double-tag level L

Hardwired values: decryption key DK_2 of ACE, message q, tag m_1^*, upper bound T.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if out $=$ 'fail' then output 'fail'; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output 'fail';
3. If $(i, m_1, m_2) = (0, m_1^*, q)$ then output 'fail';
4. Return (m_1, m_2).

Figure 72: Programs in $\mathsf{Hyb}_{C,1,q,4,2}$. In addition, in this hybrid the adversary gets $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,4,3}$.

Program GenZero$_{C,1,q,4,3}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(0, m_1)$.

Program Increment$_{C,1,q,4,3}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_0\}, \mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(i+1, m_1)$.

Program Transform$_{C,1,q,4,3}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, encryption key $\mathsf{EK}_2\{p_{0,q}\}$ of ACE punctured at $p_{0,q} = (0, m_1^*, q)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{0,q}\}}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 \geq q+1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{0,q}\}}(i-1, m_1, m_2)$;
6. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{0,q}\}}(i, m_1, m_2)$.

Program isLess$_{C,1,q,4,3}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key DK_2 of ACE, message q, tag m_1^*, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $(i', m_1', m_2') = (0, m_1^*, q)$ then output $'\mathsf{fail}'$;
5. If $(i'', m_1'', m_2'') = (0, m_1^*, q)$ then output $'\mathsf{fail}'$;
6. If $i' < i''$ then output true, else output false.

Program RetrieveTag$_{C,1,q,4,3}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Program RetrieveTags$_{C,1,q,4,3}(L)$

Inputs: double-tag level L

Hardwired values: decryption key DK_2 of ACE, message q, tag m_1^*, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. If $(i, m_1, m_2) = (0, m_1^*, q)$ then output $'\mathsf{fail}'$;
4. Return (m_1, m_2).

Figure 73: Programs in $\mathsf{Hyb}_{C,1,q,4,3}$. In addition, in this hybrid the adversary gets $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,4,4}$.

Program GenZero$_{C,1,q,4,4}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(0, m_1)$.

Program Increment$_{C,1,q,4,4}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_0\}, \mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(i+1, m_1)$.

Program Transform$_{C,1,q,4,4}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, encryption key $\mathsf{EK}_2\{p_{0,q}\}$ of ACE punctured at $p_{0,q} = (0, m_1^*, q)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{0,q}\}}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 \geq q+1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{0,q}\}}(i-1, m_1, m_2)$;
6. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{0,q}\}}(i, m_1, m_2)$.

Program isLess$_{C,1,q,4,4}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key $\mathsf{DK}_2\{p_{0,q}\}$ of ACE punctured at $p_{0,q} = (0, m_1^*, q)$, message q, tag m_1^*, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{0,q}\}}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{0,q}\}}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $(i', m_1', m_2') = (0, m_1^*, q)$ then output $'\mathsf{fail}'$;
5. If $(i'', m_1'', m_2'') = (0, m_1^*, q)$ then output $'\mathsf{fail}'$;
6. If $i' < i''$ then output true, else output false.

Program RetrieveTag$_{C,1,q,4,4}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Program RetrieveTags$_{C,1,q,4,4}(L)$

Inputs: double-tag level L

Hardwired values: decryption key $\mathsf{DK}_2\{p_{0,q}\}$ of ACE punctured at $p_{0,q} = (0, m_1^*, q)$, message q, tag m_1^*, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{0,q}\}}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. If $(i, m_1, m_2) = (0, m_1^*, q)$ then output $'\mathsf{fail}'$; 145
4. Return (m_1, m_2).

Figure 74: Programs in $\mathsf{Hyb}_{C,1,q,4,4}$. In addition, in this hybrid the adversary gets $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,4,5}$.

Program GenZero$_{C,1,q,4,5}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(0, m_1)$.

Program Increment$_{C,1,q,4,5}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_0\}, \mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(i+1, m_1)$.

Program Transform$_{C,1,q,4,5}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, encryption key $\mathsf{EK}_2\{p_{0,q}\}$ of ACE punctured at $p_{0,q} = (0, m_1^*, q)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{0,q}\}}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 \geq q+1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{0,q}\}}(i-1, m_1, m_2)$;
6. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{0,q}\}}(i, m_1, m_2)$.

Program isLess$_{C,1,q,4,5}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key $\mathsf{DK}_2\{p_{0,q}\}$ of ACE punctured at $p_{0,q} = (0, m_1^*, q)$, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{0,q}\}}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{0,q}\}}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTag$_{C,1,q,4,5}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Program RetrieveTags$_{C,1,q,4,5}(L)$

Inputs: double-tag level L

Hardwired values: decryption key $\mathsf{DK}_2\{p_{0,q}\}$ of ACE punctured at $p_{0,q} = (0, m_1^*, q)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{0,q}\}}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Return (m_1, m_2).

Figure 75: Programs in $\mathsf{Hyb}_{C,1,q,4,5}$. In addition, in this hybrid the adversary gets $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,4,6}$.

Program GenZero$_{C,1,q,4,6}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(0, m_1)$.

Program Increment$_{C,1,q,4,6}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_0\}, \mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(i+1, m_1)$.

Program Transform$_{C,1,q,4,6}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, encryption key $\mathsf{EK}_2\{p_{0,q}\}$ of ACE punctured at $p_{0,q} = (0, m_1^*, q)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{0,q}\}}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 \geq q+1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{0,q}\}}(i-1, m_1, m_2)$;
6. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{0,q}\}}(i, m_1, m_2)$.

Program isLess$_{C,1,q,4,6}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTag$_{C,1,q,4,6}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Program RetrieveTags$_{C,1,q,4,6}(L)$

Inputs: double-tag level L

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Return (m_1, m_2).

Figure 76: Programs in $\mathsf{Hyb}_{C,1,q,4,6}$. In addition, in this hybrid the adversary gets $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,4,7}$.

Program GenZero$_{C,1,q,4,7}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(0, m_1)$.

Program Increment$_{C,1,q,4,7}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_0\}, \mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(i + 1, m_1)$.

Program Transform$_{C,1,q,4,7}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i - 1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 \geq q + 1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i - 1, m_1, m_2)$;
6. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program isLess$_{C,1,q,4,7}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTag$_{C,1,q,4,7}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Program RetrieveTags$_{C,1,q,4,7}(L)$

Inputs: double-tag level L

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Return (m_1, m_2).

Figure 77: Programs in $\mathsf{Hyb}_{C,1,q,4,7}$. In addition, in this hybrid the adversary gets $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,4,8}$.

Program GenZero$_{C,1,q,4,8}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output 'fail';
2. output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(0, m_1)$.

Program Increment$_{C,1,q,4,8}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_0\}, \mathsf{DK}_1$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if out = 'fail' then output 'fail'; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output 'fail';
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(i+1, m_1)$.

Program Transform$_{C,1,q,4,8}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return 'fail';
2. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if out = 'fail' then return 'fail'; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return 'fail';
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 \geq q+1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
6. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program isLess$_{C,1,q,4,8}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. out$'$ $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if out$'$ = 'fail' then output 'fail'; else parse out$'$ as (i', m_1', m_2').
2. out$''$ $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if out$''$ = 'fail' then output 'fail'; else parse out$''$ as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output 'fail';
4. If $i' < i''$ then output true, else output false.

Program RetrieveTag$_{C,1,q,4,8}(l)$

Inputs: single-tag level l

Hardwired values: decryption key DK_1 of ACE, upper bound T.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if out = 'fail' then output 'fail'; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output 'fail';
3. Output m_1.

Program RetrieveTags$_{C,1,q,4,8}(L)$

Inputs: double-tag level L

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if out = 'fail' then output 'fail'; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output 'fail';
3. Return (m_1, m_2).

Figure 78: Programs in $\mathsf{Hyb}_{C,1,q,4,8}$. In addition, in this hybrid the adversary gets $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,1,q,4,9}$.

Program GenZero$_{C,1,q,4,9}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key EK_1 of ACE, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1)$.

Program Increment$_{C,1,q,4,9}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1, \mathsf{DK}_1$ of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(i+1, m_1)$.

Program Transform$_{C,1,q,4,9}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, message q, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*$ and $m_2 \geq q+1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
6. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program isLess$_{C,1,q,4,9}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTag$_{C,1,q,4,9}(l)$

Inputs: single-tag level l

Hardwired values: decryption key DK_1 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Program RetrieveTags$_{C,1,q,4,9}(L)$

Inputs: double-tag level L

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Return (m_1, m_2).

Figure 79: Programs in $\mathsf{Hyb}_{C,1,q,4,9}$. In addition, in this hybrid the adversary gets $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,2,1,1}$.

Program Transform$_{C,2,1,1}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
5. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$;

Program isLess$_{C,2,1,1}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,2,1,1}(L)$

Inputs: double-tag level L

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1, m_2.

Figure 80: Programs in $\mathsf{Hyb}_{C,2,1,1}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

;

Programs in $\mathsf{Hyb}_{C,2,1,2}$.

Program Transform$_{C,2,1,2}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key $\mathsf{EK}_2\{p_{T,m_2^*}\}$ of ACE punctured at $p_{T,m_2^*} = (T, m_1^*, m_2^*)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{T,m_2^*}\}}(i-1, m_1, m_2)$;
5. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{T,m_2^*}\}}(i, m_1, m_2)$;

Program isLess$_{C,2,1,2}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,2,1,2}(L)$

Inputs: double-tag level L

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1, m_2.

Figure 81: Programs in $\mathsf{Hyb}_{C,2,1,2}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,2,1,3}$.

Program Transform$_{C,2,1,3}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key $\mathsf{EK}_2\{p_{T,m_2^*}\}$ of ACE punctured at $p_{T,m_2^*} = (T, m_1^*, m_2^*)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return 'fail';
2. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if out = 'fail' then return 'fail'; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return 'fail';
4. If $m_1 = m_1^*$ and $m_2 = m_2^*$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{T,m_2^*}\}}(i-1, m_1, m_2)$;
5. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{T,m_2^*}\}}(i, m_1, m_2)$;

Program isLess$_{C,2,1,3}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key $\mathsf{DK}_2\{p_{T,m_2^*}\}$ of ACE punctured at $p_{T,m_2^*} = (T, m_1^*, m_2^*)$, upper bound T.

1. out$'$ $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{T,m_2^*}\}}(L')$; if out$'$ = 'fail' then output 'fail'; else parse out$'$ as (i', m_1', m_2').
2. out$''$ $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{T,m_2^*}\}}(L'')$; if out$''$ = 'fail' then output 'fail'; else parse out$''$ as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output 'fail';
4. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,2,1,3}(L)$

Inputs: double-tag level L

Hardwired values: decryption key $\mathsf{DK}_2\{p_{T,m_2^*}\}$ of ACE punctured at $p_{T,m_2^*} = (T, m_1^*, m_2^*)$, upper bound T.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{T,m_2^*}\}}(L)$; if out = 'fail' then output 'fail'; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output 'fail';
3. Output m_1, m_2.

Figure 82: Programs in $\mathsf{Hyb}_{C,2,1,3}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, Increment and RetrieveTag, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,2,2,j,1}$.

Program Transform$_{C,2,2,j,1}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key $\mathsf{EK}_2\{p_{j+1,m_2^*}\}$ of ACE punctured at $p_{j+1,m_2^*} = (j+1, m_1^*, m_2^*)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, index j, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*, m_2 = m_2^*$ and $i \le j+1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j+1,m_2^*}\}}(i-1, m_1, m_2)$;
5. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j+1,m_2^*}\}}(i, m_1, m_2)$.

Program isLess$_{C,2,2,j,1}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key $\mathsf{DK}_2\{p_{j+1,m_2^*}\}$ of ACE punctured at $p_{j+1,m_2^*} = (j+1, m_1^*, m_2^*)$, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j+1,m_2^*}\}}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j+1,m_2^*}\}}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \ne (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,2,2,j,1}(L)$

Inputs: double-tag level L

Hardwired values: decryption key $\mathsf{DK}_2\{p_{j+1,m_2^*}\}$ of ACE punctured at $p_{j+1,m_2^*} = (j+1, m_1^*, m_2^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j+1,m_2^*}\}}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1, m_2.

Figure 83: Programs in $\mathsf{Hyb}_{C,2,2,j,1}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,2,2,j,2}$.

Program Transform$_{C,2,2,j,2}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key $\mathsf{EK}_2\{p_{j,m_2^*}, p_{j+1,m_2^*}\}$ of ACE punctured at $p_{j,m_2^*} = (j, m_1^*, m_2^*)$, $p_{j+1,m_2^*} = (j+1, m_1^*, m_2^*)$, double-tag level $L^*_{j,m_2^*} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, m_2^*)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, index j, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*, m_2 = m_2^*$ and $i < j+1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j,m_2^*}, p_{j+1,m_2^*}\}}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*, m_2 = m_2^*$ and $i = j+1$ return L^*_{j,m_2^*};
6. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j,m_2^*}, p_{j+1,m_2^*}\}}(i, m_1, m_2)$.

Program isLess$_{C,2,2,j,2}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key $\mathsf{DK}_2\{p_{j,m_2^*}, p_{j+1,m_2^*}\}$ of ACE punctured at $p_{j,m_2^*} = (j, m_1^*, m_2^*)$, $p_{j+1,m_2^*} = (j+1, m_1^*, m_2^*)$, double-tag level $L^*_{j,m_2^*} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, m_2^*)$, upper bound T.

1. If $L' = L^*_{j,m_2^*}$ then set $(i', m_1', m_2') = (j+1, m_1^*, m_2^*)$,
   else $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j,m_2^*}, p_{j+1,m_2^*}\}}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. If $L'' = L^*_{j,m_2^*}$ then set $(i'', m_1'', m_2'') = (j+1, m_1^*, m_2^*)$,
   else $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j,m_2^*}, p_{j+1,m_2^*}\}}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,2,2,j,2}(L)$

Inputs: double-tag level L

Hardwired values: decryption key $\mathsf{DK}_2\{p_{j,m_2^*}, p_{j+1,m_2^*}\}$ of ACE punctured at $p_{j,m_2^*} = (j, m_1^*, m_2^*)$, $p_{j+1,m_2^*} = (j+1, m_1^*, m_2^*)$, double-tag level $L^*_{j,m_2^*} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, m_2^*)$, upper bound T.

1. If $L = L^*_{j,m_2^*}$ then return (m_1^*, m_2^*);
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j,m_2^*}, p_{j+1,m_2^*}\}}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
3. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
4. Return (m_1, m_2).

Figure 84: Programs in $\mathsf{Hyb}_{C,2,2,j,2}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,2,2,j,3}$.

Program Transform$_{C,2,2,j,3}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key $\mathsf{EK}_2\{p_{j,m_2^*}, p_{j+1,m_2^*}\}$ of ACE punctured at $p_{j,m_2^*} = (j, m_1^*, m_2^*)$, $p_{j+1,m_2^*} = (j+1, m_1^*, m_2^*)$, double-tag level $L^*_{j+1,m_2^*} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j+1, m_1^*, m_2^*)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, index j, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*, m_2 = m_2^*$ and $i < j+1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j+1,m_2^*}\}}(i-1, m_1, m_2)$;
5. If $m_1 = m_1^*, m_2 = m_2^*$ and $i = j+1$ return L^*_{j+1,m_2^*};
6. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j,m_2^*}, p_{j+1,m_2^*}\}}(i, m_1, m_2)$.

Program isLess$_{C,2,2,j,3}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key $\mathsf{DK}_2\{p_{j,m_2^*}, p_{j+1,m_2^*}\}$ of ACE punctured at $p_{j,m_2^*} = (j, m_1^*, m_2^*)$, $p_{j+1,m_2^*} = (j+1, m_1^*, m_2^*)$, double-tag level $L^*_{j+1,m_2^*} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j+1, m_1^*, m_2^*)$, upper bound T.

1. If $L' = L^*_{j+1,m_2^*}$ then set $(i', m_1', m_2') = (j+1, m_1^*, m_2^*)$, else $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j,m_2^*}, p_{j+1,m_2^*}\}}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. If $L'' = L^*_{j+1,m_2^*}$ then set $(i'', m_1'', m_2'') = (j+1, m_1^*, m_2^*)$, else $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j,m_2^*}, p_{j+1,m_2^*}\}}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,2,2,j,3}(L)$

Inputs: double-tag level L

Hardwired values: decryption key $\mathsf{DK}_2\{p_{j,m_2^*}, p_{j+1,m_2^*}\}$ of ACE punctured at $p_{j,m_2^*} = (j, m_1^*, m_2^*)$, $p_{j+1,m_2^*} = (j+1, m_1^*, m_2^*)$, double-tag level $L^*_{j+1,m_2^*} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j+1, m_1^*, m_2^*)$, upper bound T.

1. If $L = L^*_{j+1,m_2^*}$ then return (m_1^*, m_2^*);
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j,m_2^*}, p_{j+1,m_2^*}\}}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
3. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
4. Return (m_1, m_2).

Figure 85: Programs in $\mathsf{Hyb}_{C,2,2,j,3}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,2,2,j,4}$.

Program Transform$_{C,2,2,j,4}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key $\mathsf{EK}_2\{p_{j,m_2^*}\}$ of ACE punctured at $p_{j,m_2^*} = (j, m_1^*, m_2^*)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, index j, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*, m_2 = m_2^*$ and $i \leq j$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j,m_2^*}\}}(i-1, m_1, m_2)$;
5. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{j,m_2^*}\}}(i, m_1, m_2)$.

Program isLess$_{C,2,2,j,4}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key $\mathsf{DK}_2\{p_{j,m_2^*}\}$ of ACE punctured at $p_{j,m_2^*} = (j, m_1^*, m_2^*)$, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j,m_2^*}\}}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j,m_2^*}\}}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,2,2,j,4}(L)$

Inputs: double-tag level L

Hardwired values: decryption key $\mathsf{DK}_2\{p_{j,m_2^*}\}$ of ACE punctured at $p_{j,m_2^*} = (j, m_1^*, m_2^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{j,m_2^*}\}}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Return (m_1, m_2).

Figure 86: Programs in $\mathsf{Hyb}_{C,2,2,j,4}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,2,3,1}$.

Program Transform$_{C,2,3,1}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key $\mathsf{EK}_2\{p_{1,m_2^*}\}$ of ACE punctured at $p_{1,m_2^*} = (1, m_1^*, m_2^*)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*, m_2 = m_2^*$ and $i \leq 1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{1,m_2^*}\}}(i-1, m_1, m_2)$;
5. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{1,m_2^*}\}}(i, m_1, m_2)$.

Program isLess$_{C,2,3,1}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key $\mathsf{DK}_2\{p_{1,m_2^*}\}$ of ACE punctured at $p_{1,m_2^*} = (1, m_1^*, m_2^*)$, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{1,m_2^*}\}}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{1,m_2^*}\}}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,2,3,1}(L)$

Inputs: double-tag level L

Hardwired values: decryption key $\mathsf{DK}_2\{p_{1,m_2^*}\}$ of ACE punctured at $p_{1,m_2^*} = (1, m_1^*, m_2^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2\{p_{1,m_2^*}\}}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Return (m_1, m_2).

Figure 87: Programs in $\mathsf{Hyb}_{C,2,3,1}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,2,3,2}$.

Program Transform$_{C,2,3,2}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key $\mathsf{EK}_2\{p_{1,m_2^*}\}$ of ACE punctured at $p_{1,m_2^*} = (1, m_1^*, m_2^*)$, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*, m_2 = m_2^*$ and $i \leq 1$ return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{1,m_2^*}\}}(i-1, m_1, m_2)$;
5. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2\{p_{1,m_2^*}\}}(i, m_1, m_2)$.

Program isLess$_{C,2,3,2}(L', L'')$

Inputs: double-tag levels L', L''

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out}' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L')$; if $\mathsf{out}' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out' as (i', m_1', m_2').
2. $\mathsf{out}'' \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L'')$; if $\mathsf{out}'' = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out'' as (i'', m_1'', m_2'').
3. If $i' > T$ or $i'' > T$ or $i' < 0$ or $i'' < 0$ or $(m_1', m_2') \neq (m_1'', m_2'')$ then output $'\mathsf{fail}'$;
4. If $i' < i''$ then output true, else output false.

Program RetrieveTags$_{C,2,3,2}(L)$

Inputs: double-tag level L

Hardwired values: decryption key DK_2 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_2}(L)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1, m_2).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Return (m_1, m_2).

Figure 88: Programs in $\mathsf{Hyb}_{C,2,3,2}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs $\mathsf{GenZero}[m_1^*]$, $\mathsf{Increment}$ and $\mathsf{RetrieveTag}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,2,3,3}$.

Program Transform$_{C,2,3,3}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. If $m_1 = m_1^*$, $m_2 = m_2^*$, and $i \leq 0$, return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
5. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program GenZero$_{C,2,3,3}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(0, m_1)$.

Program Increment$_{C,2,3,3}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_0\}, \mathsf{DK}_1$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(i+1, m_1)$.

Program RetrieveTag$_{C,2,3,3}(l)$

Inputs: single-tag level l

Hardwired values: decryption key DK_1 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Figure 89: Programs in $\mathsf{Hyb}_{C,2,3,3}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and $\mathsf{RetrieveTags}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,2,3,4}$.

Program Transform$_{C,2,3,4}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_1^*, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return 'fail';
2. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if out = 'fail' then return 'fail'; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return 'fail';
4. If $m_1 = m_1^*$, $m_2 = m_2^*$, and $i \leq 0$, return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$;
5. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program GenZero$_{C,2,3,4}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output 'fail';
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(0, m_1)$.

Program Increment$_{C,2,3,4}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_0\}$, $\mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if out = 'fail' then output 'fail'; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output 'fail';
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(i+1, m_1)$.

Program RetrieveTag$_{C,2,3,4}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if out = 'fail' then output 'fail'; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output 'fail';
3. Output m_1.

Figure 90: Programs in $\mathsf{Hyb}_{C,2,3,4}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and RetrieveTags, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,2,3,5}$.

Program Transform$_{C,2,3,5}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key $\mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program GenZero$_{C,2,3,5}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(0, m_1)$.

Program Increment$_{C,2,3,5}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_0\}, \mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(i + 1, m_1)$.

Program RetrieveTag$_{C,2,3,5}(l)$

Inputs: single-tag level l

Hardwired values: decryption key $\mathsf{DK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1\{p_0\}}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Figure 91: Programs in $\mathsf{Hyb}_{C,2,3,5}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and $\mathsf{RetrieveTags}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,2,3,6}$.

Program Transform$_{C,2,3,6}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program GenZero$_{C,2,3,6}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key $\mathsf{EK}_1\{p_0\}$ of ACE punctured at $p_0 = (0, m_1^*)$, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(0, m_1)$.

Program Increment$_{C,2,3,6}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys $\mathsf{EK}_1\{p_0\}, \mathsf{DK}_1$ of ACE punctured at $p_0 = (0, m_1^*)$, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1\{p_0\}}(i + 1, m_1)$.

Program RetrieveTag$_{C,2,3,6}(l)$

Inputs: single-tag level l

Hardwired values: decryption key DK_1 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Figure 92: Programs in $\mathsf{Hyb}_{C,2,3,6}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and $\mathsf{RetrieveTags}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

Programs in $\mathsf{Hyb}_{C,2,3,7}$.

Program Transform$_{C,2,3,7}[(l_1^*, m_2^*)](l, m_2)$

Inputs: single-tag level l, tag $m_2 \in M$

Hardwired values: decryption key DK_1 of ACE, encryption key EK_2 of ACE, single-tag level $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, tag m_2^*, upper bound T.

1. If $(l, m_2) = (l_1^*, m_2^*)$ then return $'\mathsf{fail}'$;
2. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then return $'\mathsf{fail}'$; else parse out as (i, m_1).
3. If $i > T$ or $i < 0$ then return $'\mathsf{fail}'$;
4. Return $L \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$.

Program GenZero$_{C,2,3,7}[m_1^*](m_1)$

Inputs: tag $m_1 \in M$.

Hardwired values: encryption key EK_1 of ACE, tag m_1^*.

1. If $m_1 = m_1^*$ then output $'\mathsf{fail}'$;
2. Output $l \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1)$.

Program Increment$_{C,2,3,7}(l)$

Inputs: single-tag level l

Hardwired values: encryption and decryption keys EK_1, DK_1 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i \geq T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. output $l_{+1} \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_1}(i + 1, m_1)$.

Program RetrieveTag$_{C,2,3,7}(l)$

Inputs: single-tag level l

Hardwired values: decryption key DK_1 of ACE, upper bound T.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_1}(l)$; if $\mathsf{out} = '\mathsf{fail}'$ then output $'\mathsf{fail}'$; else parse out as (i, m_1).
2. If $i > T$ or $i < 0$ then output $'\mathsf{fail}'$;
3. Output m_1.

Figure 93: Programs in $\mathsf{Hyb}_{C,2,3,7}$. In addition, in this hybrid the adversary gets unmodified obfuscated programs isLess and $\mathsf{RetrieveTags}$, together with $l_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$.

7.5 Security reductions

7.5.1 Reductions in the proof of lemma 2 (Switching from ℓ_0^* to ℓ_1^*)

We show that for any PPT adversary,

$$\mathsf{adv}_{\mathsf{Hyb}_A,\mathsf{Hyb}_B}(\lambda) \le T \cdot 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))} + T \cdot 2^{-\Omega(\nu_{\mathsf{ACE.Indist}}(\lambda))} + 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}.$$

Lemma 5. $\mathsf{adv}_{\mathsf{Hyb}_A,\mathsf{Hyb}_{A,1,1}}(\lambda) \le 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$.

Proof. In programs $\mathsf{GenZero}_{A,1,1}$ and $\mathsf{Increment}_{A,1,1}$ encryption key EK_1 is punctured at $p_{T+1} = (T+1, m_1^*)$. This is without changing the functionality, since $\mathsf{GenZero}$ only encrypts plaintexts of the form $(0, m_1)$, and $\mathsf{Increment}$ outputs $'\mathsf{fail}'$ when $i = T$ and thus never encrypts $(T+1, m_1^*)$. □

Lemma 6. $\mathsf{adv}_{\mathsf{Hyb}_{A,1,1},\mathsf{Hyb}_{A,1,2}}(\lambda) \le 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}$.

Proof. Indistinguishability immediately follows from security of constrained decryption of ACE for the challenge set $\{p_{T+1}\} = \{(T+1, m_1^*)\}$ to puncture encryption key EK_1 and challenge sets $\{p_{T+1}\}, \varnothing$ to puncture decryption key DK_1. Indeed, given $\mathsf{EK}_1\{p_{T+1}\}$ and key which is either DK_1 or $\mathsf{DK}_1\{p_{T+1}\}$, it is easy to reconstruct the rest of the distribution. □

Lemma 7. $\mathsf{adv}_{\mathsf{Hyb}_{A,2,j,1},\mathsf{Hyb}_{A,2,j,2}}(\lambda) \le 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$ *for* $1 \le j \le T$.

Proof. In programs $\mathsf{GenZero}$, $\mathsf{Increment}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$ we puncture EK_1, DK_1 at $p_j = (j, m_1^*)$ and hardwire $\ell_j^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(j, m_1^*)$ when required, in order to preserve functionality.

In program $\mathsf{GenZero}_{A,2,j,2}$ we can puncture EK_1 at p_j without changing the functionality, since $\mathsf{GenZero}_{A,2,j,2}$ only encrypts plaintexts of the form $(0, m_1)$ (note that $j \ge 1$).

In program $\mathsf{Increment}_{A,2,j,2}$ we puncture DK_1 at p_j and, in order to preserve the functionality, instruct the program to output $\mathsf{ACE.Enc}_{\mathsf{EK}_1}(j+2, m_1^*)$ on input ℓ_j^* (note that this is what $\mathsf{Increment}_{A,2,j,1}$ outputs on input ℓ_j^*)[30]. Further, we puncture EK_1 at p_j and, in order to preserve the functionality, instruct the program to output ℓ_j^* on input $\mathsf{ACE.Enc}_{\mathsf{EK}_1}(j-1, m_1^*)$ (note that this is what $\mathsf{Increment}_{A,2,j,1}$ does).

In program $\mathsf{Transform}_{A,2,j,2}$ we puncture DK_1 at p_j and, in order to preserve the functionality, instruct the program to output $\mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, m_2)$ on input (ℓ_j^*, m_2) for any m_2 (note that this is what $\mathsf{Transform}_{A,2,j,1}$ does). Because of this instruction, we can also instruct $\mathsf{Transform}_{A,2,j,2}$ to output $'\mathsf{fail}'$ when $(i, m_1) = (j, m_1^*)$, since this line will never be reached.

In program $\mathsf{RetrieveTag}_{A,2,j,2}$ we puncture DK_1 at p_j and, in order to preserve the functionality, instruct the program to output m_1^* on input ℓ_j^* (note that this is what $\mathsf{RetrieveTag}_{A,2,j,1}$ does).

□

Lemma 8. $\mathsf{adv}_{\mathsf{Hyb}_{A,2,j,2},\mathsf{Hyb}_{A,2,j,3}}(\lambda) \le 2^{-\Omega(\nu_{\mathsf{ACE.Indist}}(\lambda))}$ *for* $1 \le j \le T$.

Proof. Indistinguishability immediately follows from indistinguishability of ACE ciphertexts for the challenge plaintexts $p_j = (j, m_1^*)$ and $p_{j+1} = (j+1, m_1^*)$. Indeed, given $\mathsf{EK}_1\{p_j, p_{j+1}\}$, $\mathsf{DK}_1\{p_j, p_{j+1}\}$, and either $\ell_j^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(j, m_1^*)$ or $\ell_{j+1}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(j+1, m_1^*)$, it is easy to reconstruct the rest of

[30] Except for the case $j = T$, when we instead instruct the program to output $'\mathsf{fail}'$.

the distribution. Note that indeed only one of the two ciphertexts is used in both hybrids (in particular, since $j \geq 1$, the key is never punctured at $p_0 = (0, m_1^*)$ and therefore we can always compute ℓ_0^* for the distribution). $\square$

Lemma 9. $\mathsf{adv}_{\mathsf{Hyb}_{A,2,j,3},\mathsf{Hyb}_{A,2,j,4}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$ *for* $1 \leq j \leq T$.

Proof. In programs GenZero, Increment, Transform, RetrieveTag we unpuncture EK_1, DK_1 at $p_{j+1} = (j+1, m_1^*)$ and remove hardwired $\ell_{j+1}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(j+1, m_1^*)$:

In program $\mathsf{GenZero}_{A,2,j,4}$ we can unpuncture EK_1 at p_{j+1} without changing the functionality, since $\mathsf{GenZero}_{A,2,j,3}$ only encrypts plaintexts of the form $(0, m_1)$ (note that $j \geq 1$).

In program $\mathsf{Increment}_{A,2,j,4}$ we unpuncture DK_1 at p_{j+1}, remove the instruction to output $\mathsf{ACE.Enc}_{\mathsf{EK}_1}(j+2, m_1^*)$ on input ℓ_{j+1}^* and, in order to preserve the functionality, instruct the program to output $\mathsf{ACE.Enc}_{\mathsf{EK}_1}(j+2, m_1^*)$ when $(i, m_1) = (j+1, m_1^*)$ (we don't put any separate instruction since this is normal behavior of Increment); [31]. Further, we unpuncture EK_1 at p_{j+1}, remove the instruction to output ℓ_{j+1}^* on input $\mathsf{ACE.Enc}_{\mathsf{EK}_1}(j-1, m_1^*)$ and, in order to preserve the functionality, instruct the program to output $\mathsf{ACE.Enc}_{\mathsf{EK}_1}(j+1, m_1^*)$ when $(i, m_1) = (j-1, m_1^*)$.

In program $\mathsf{Transform}_{A,2,j,4}$ we unpuncture DK_1 at p_j, remove the instruction to output $\mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, m_2)$ on input (ℓ_{j+1}^*, m_2) for any m_2 and, in order to preserve the functionality, instruct the program to output $\mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$ when $(i, m_1) = (j+1, m_1^*)$.

In program $\mathsf{RetrieveTag}_{A,2,j,4}$ we unpuncture DK_1 at p_{j+1} and remove the instruction to output m_1^* on input ℓ_{j+1}^*. No additional change is required since this is what RetrieveTag would normally output[32]. $\square$

Lemma 10. $\mathsf{adv}_{\mathsf{Hyb}_{A,2,0,1},\mathsf{Hyb}_{A,2,0,2}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$.

Proof. In programs GenZero, Increment, Transform, RetrieveTag we puncture EK_1, DK_1 at $p_0 = (0, m_1^*)$ and hardwire $\ell_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$ when required, in order to preserve functionality:

In program $\mathsf{GenZero}_{A,2,0,2}$ we can puncture EK_1 at p_0 without changing the functionality, since $\mathsf{GenZero}_{A,2,0,2}$ outputs 'fail' when $m_1 = m_1^*$.

In program $\mathsf{Increment}_{A,2,0,2}$ we puncture DK_1 at p_0 and, in order to preserve the functionality, instruct the program to output $\mathsf{ACE.Enc}_{\mathsf{EK}_1}(2, m_1^*)$ on input ℓ_0^* (note that this is what $\mathsf{Increment}_{A,2,0,1}$ outputs on input ℓ_0^*). Further, we puncture EK_1 at p_0: this is without changing the functionality, since this program never needs to encrypt plaintexts with value 0.

In program $\mathsf{Transform}_{A,2,0,2}$ we puncture DK_1 at p_0 and, in order to preserve the functionality, instruct the program to output $\mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2)$ on input (ℓ_0^*, m_2) for any m_2 (note that this is what $\mathsf{Transform}_{A,2,0,1}$ does). Because of this instruction, we can also instruct $\mathsf{Transform}_{A,2,0,2}$ to output 'fail' when $(i, m_1) = (0, m_1^*)$, since this line will never be reached.

In program $\mathsf{RetrieveTag}_{A,2,0,2}$ we puncture DK_1 at p_0 and, in order to preserve the functionality, instruct the program to output m_1^* on input ℓ_0^* (note that this is what $\mathsf{RetrieveTag}_{A,2,0,1}$ does).

[31] Except for the case $j = T$, when we instead remove the instruction to output 'fail'. Note that Increment outputs 'fail' when $i = T+1$ so no additional modification is required. The other exception is the case $j = T-1$, where $\mathsf{Increment}_{A,2,j,3}$ contains the instruction to output $\mathsf{ACE.Enc}_{\mathsf{EK}_1}(T+1, m_1^*)$ on input ℓ_T^*, and thus in $\mathsf{Increment}_{A,2,j,4}$ we change the upper bound from T to $T+1$ for the case $m_1 = m_1^*$ in order to preserve the functionality.

[32] Except for the case $j = T$, which instruct the program to output m_1^* on input ℓ_{T+1}^*. In this case we additionally change the upper bound to $T+1$, instead of T, for the case $m_1 = m_1^*$ in program $\mathsf{RetrieveTag}_{A,2,j,4}$

□

Lemma 11. $\mathsf{adv}_{\mathsf{Hyb}_{A,2,0,2},\mathsf{Hyb}_{A,2,0,3}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{ACE.Indist}}(\lambda))}$.

Proof. Indistinguishability immediately follows from indistinguishability of ACE ciphertexts for the challenge plaintexts $p_0 = (0, m_1^*)$ and $p_1 = (1, m_1^*)$. Indeed, given $\mathsf{EK}_1\{p_0, p_1\}$, $\mathsf{DK}_1\{p_0, p_1\}$, and either $\ell_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(0, m_1^*)$ or $\ell_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$, it is easy to reconstruct the rest of the distribution. Note that indeed only one of the two ciphertexts is used in both hybrids (in particular, a single-tag level we give to the adversary is either ℓ_0^* or ℓ_1^*). □

Lemma 12. $\mathsf{adv}_{\mathsf{Hyb}_{A,2,0,3},\mathsf{Hyb}_{A,3,1}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$.

Proof. In programs $\mathsf{GenZero}$, $\mathsf{Increment}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$ we unpuncture EK_1, DK_1 at $p_1 = (1, m_1^*)$ and remove hardwired $\ell_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(j+1, m_1^*)$:

In program $\mathsf{GenZero}_{A,3,2}$ we can unpuncture EK_1 at p_1 without changing the functionality, since $\mathsf{GenZero}_{A,2,0,3}$ only encrypts plaintexts of the form $(0, m_1)$.

In program $\mathsf{Increment}_{A,3,1}$ we unpuncture DK_1 at p_1 and remove the additional instruction to output $\mathsf{ACE.Enc}_{\mathsf{EK}_1}(2, m_1^*)$ on input ℓ_1^* (this is without changing the functionality, since this is what the program normally does). Further, we unpuncture EK_1 at p_1 without changing the functionality: indeed, the program could possibly encrypt p_1 only given an encryption of p_0 as input. However, DK_1 is punctured at p_0 and thus the program would instead output $'\mathsf{fail}'$ on such input.

In program $\mathsf{Transform}_{A,3,1}$ we instruct the program to output $\mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$, given an encryption of (i, m_1^*) and m_2 as input, in the whole range of i from 0 to T. In contrast, program $\mathsf{Transform}_{A,2,0,3}$ does this only for $2 \leq i \leq T$. However, this is without changing the functionality: first, $\mathsf{Transform}_{A,2,0,3}$ outputs $\mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2)$, given ℓ_1^* and m_2 as input, thus we didn't change the case $i = 1$. Second, DK_1 is punctured at p_0, and thus we can arbitrary change behaviour for the case $i = 0$ since the program never reaches that line when $i = 0$, outputting $'\mathsf{fail}'$ during decryption.

With this modification, we can remove the instruction to output $\mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2)$ on input (ℓ_1^*, m_2) and then unpuncture DK_1 at point p_1.

In program $\mathsf{RetrieveTag}_{A,3,1}$ we unpuncture DK_1 at p_1 and remove the instruction to output m_1^* on input ℓ_1^*. No additional change is required since this is what $\mathsf{RetrieveTag}$ would normally output. □

Lemma 13. $\mathsf{adv}_{\mathsf{Hyb}_{A,3,1},\mathsf{Hyb}_{A,3,2}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}$.

Proof. Indistinguishability immediately follows from security of constrained decryption of ACE for the challenge set $\{p_0\} = \{(0, m_1^*)\}$ to puncture encryption key EK_1 and challenge sets $\{p_0\}, \varnothing$ to puncture decryption key DK_1. Indeed, given $\mathsf{EK}_1\{p_0\}$ and key which is either DK_1 or $\mathsf{DK}_1\{p_0\}$, it is easy to reconstruct the rest of the distribution. □

Lemma 14. $\mathsf{adv}_{\mathsf{Hyb}_{A,3,2},\mathsf{Hyb}_{A,3,3}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$.

Proof. In programs $\mathsf{GenZero}$ and $\mathsf{Increment}$ we unpuncture EK_1 at $p_0 = (0, m_1^*)$. This doesn't change the functionality, since $\mathsf{GenZero}$ outputs $'\mathsf{fail}'$ when $m_1 = m_1^*$, and $\mathsf{Increment}$ never encrypts a plaintext with value 0. □

7.5.2 Reductions in the proof of lemma 3 (Changing the upper bound from $T+1$ to T)

We show that

$$\mathsf{adv}_{\mathsf{Hyb}_B,\mathsf{Hyb}_C}(\lambda) \leq 2^{-\Omega(\gamma(\lambda))} + \frac{1}{T} + T \cdot 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))} + T \cdot 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}.$$

($1/T$ term comes from the fact that the reduction works only when $i^* \neq 0$, where i^* is chosen randomly between 0 and T).

Lemma 15. $\mathsf{adv}_{\mathsf{Hyb}_B,\mathsf{Hyb}_{B,1,1}}(\lambda) \leq 2^{-\Omega(\gamma(\lambda))}$.

Proof. Assume there is a poly-time distinguisher D which distinguishes between these two hybrids with probability $\eta \geq 2^{-o(\gamma(\lambda))}$ (for infinitely many λ_i). Then, since:

- programs $\mathsf{Increment}_B$ and $\mathsf{Increment}_{B,1,1}$ differ only at one point (due to the fact that g is injective);
- $\eta \geq 2^{-o(\gamma(\lambda))} \geq 2^{-o(\nu_{\mathsf{iO}}(\lambda))}$ (from the condition $\gamma(\lambda) \leq O(\nu_{\mathsf{iO}}(\lambda))$ in the theorem statement),

it follows from lemma 1 that there exists an inverter which runs in time at most $O(1/\eta)\log T = 2^{o(\gamma(\lambda))}\log T$, which by the condition in the theorem statement is at most $O(2^{\nu_{\mathsf{OWF}}(\log T)})$. This inverter inverts the one way function with probability at least $(1-2^{-\Omega(\lambda)})\eta$, which contradicts the fact that g is $2^{O(\nu_{\mathsf{OWF}}(\lambda \log T))}, 2^{-\Omega(\nu_{\mathsf{OWF}}(\lambda \log T))}$-secure OWF.

□

Lemma 16. $\mathsf{adv}_{\mathsf{Hyb}_{B,1,1},\mathsf{Hyb}_{B,1,2}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$.

Proof. First, note that both programs $\mathsf{GenZero}_{B,1,1}$ and $\mathsf{GenZero}_{B,1,2}$ are functionally equivalent: since $i^*+1 \neq 0$, and $\mathsf{GenZero}$ only needs to encrypt value 0, we can safely puncture EK_1 at (i^*+1, m_1^*).

Second, programs $\mathsf{Increment}_{B,1,1}$ and $\mathsf{Increment}_{B,1,2}$ are functionally equivalent as well: the only difference in the code is that the first outputs $'\mathsf{fail}'$ when $(m_1, i) = (m_1^*, i^*)$ (on input $\mathsf{ACE.Enc}_{\mathsf{EK}_1}(i^*, m_1^*)$), and the second instead outputs $'\mathsf{fail}'$ when it tries to encrypt a punctured point (i^*+1, m_1^*), which happens on the same input $\mathsf{ACE.Enc}_{\mathsf{EK}_1}(i^*, m_1^*)$. □

Lemma 17. *If* $i^* \neq 0$, $\mathsf{adv}_{\mathsf{Hyb}_{B,2,j,1},\mathsf{Hyb}_{B,2,j,2}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}$ *for* $i^* \leq j \leq T$.

Proof. Indistinguishability immediately follows from security of constrained decryption of ACE for the challenge set $S_{i^*+1,j+1}$ to puncture encryption key EK_1 and challenge sets $S_{i^*+1,j}, S_{i^*+1,j+1}$ to puncture decryption key DK_1 (here $S_{a,b} = \{(m_1^*, a), (m_1^*, a+1), \ldots, (m_1^*, b)\}$ if $b \geq a$ and $\varnothing$ otherwise). Indeed, given $\mathsf{EK}_1\{S_{i^*+1,j+1}\}$ and key which is either $\mathsf{DK}_1\{S_{i^*+1,j}\}$ or $\mathsf{DK}_1\{S_{i^*+1,j+1}\}$, it is easy to reconstruct the rest of the distribution, as long as $i^* \neq 0$. That is, we can sample remaining keys, obfuscate all programs, and compute $\ell_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$ (using the challenge encryption key $\mathsf{EK}_1\{S_{i^*+1,j+1}\}$ which is not punctured at $(1, m_1^*)$ since $i^* \neq 0$) and $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. □

Lemma 18. $\mathsf{adv}_{\mathsf{Hyb}_{B,2,j,2},\mathsf{Hyb}_{B,2,j,3}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$ *for* $i^* \leq j \leq T$.

Proof. In programs $\mathsf{GenZero}$, $\mathsf{Increment}$ we additionally puncture EK_1 at $p_{j+2} = (j+2, m_1^*)$.

In program $\mathsf{GenZero}$ we can puncture EK_1 at p_{j+2} without changing the functionality, since $\mathsf{GenZero}$ only encrypts plaintexts of the form $(0, m_1)$, but $j+2 \neq 0$.

In program $\mathsf{Increment}$ we can puncture EK_1 at p_{j+2} without changing the functionality, since DK_1 is punctured at the point p_{j+1}, thus $\mathsf{Increment}$ never needs to encrypt p_{j+2} since on input $[j+1, m_1^*]$ it instead outputs $'\mathsf{fail}'$ during decryption. $\square$

Lemma 19. $\mathsf{adv}_{\mathsf{Hyb}_{B,3,1},\mathsf{Hyb}_{B,3,2}}(\lambda) \le 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$.

Proof. In programs $\mathsf{Increment}$, $\mathsf{Transform}$, and $\mathsf{RetrieveTag}$ we change the upper bound from $T+1$ back to T.

In particular, in program $\mathsf{Increment}_{B,3,2}$ we now additionally output $'\mathsf{fail}'$ when $i = T$ and $m_1 = m_1^*$. This is without changing the functionality, since this line is never reached: both programs $\mathsf{Increment}_{B,3,1}$ and $\mathsf{Increment}_{B,3,2}$ anyway output $'\mathsf{fail}'$ on input $[T, m_1^*]$, since DK_1 is punctured at (T, m_1^*).

In program $\mathsf{Transform}$ we now additionally output $'\mathsf{fail}'$ when $i = T+1$ and $m_1 = m_1^*$. This is without changing the functionality, since this line is never reached: both programs $\mathsf{Transform}_{B,3,1}$ and $\mathsf{Transform}_{B,3,2}$ anyway output $'\mathsf{fail}'$ on input $[T+1, m_1^*]$ and any m_2, since DK_1 is punctured at $(T+1, m_1^*)$.

In program $\mathsf{RetrieveTag}$ we now additionally output $'\mathsf{fail}'$ when $i = T+1$ and $m_1 = m_1^*$. This is without changing the functionality, since this line is never reached: both programs $\mathsf{RetrieveTag}_{B,3,1}$ and $\mathsf{RetrieveTag}_{B,3,2}$ anyway output $'\mathsf{fail}'$ on input $[T+1, m_1^*]$, since DK_1 is punctured at $(T+1, m_1^*)$. $\square$

Lemma 20. *If* $i^* \neq 0$*,* $\mathsf{adv}_{\mathsf{Hyb}_{B,4,j,1},\mathsf{Hyb}_{B,4,j,2}}(\lambda) \le 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}$ *for* $i^* \le j \le T$.

Proof. Indistinguishability immediately follows from security of constrained decryption of ACE for the challenge set $S_{i^*+1,j+1}$ to puncture encryption key EK_1 and challenge sets $S_{i^*+1,j}, S_{i^*+1,j+1}$ to puncture decryption key DK_1 (here $S_{a,b} = \{(m_1^*, a), (m_1^*, a+1), \ldots, (m_1^*, b)\}$ if $b \ge a$ and $\varnothing$ otherwise). Indeed, given $\mathsf{EK}_1\{S_{i^*+1,j+1}\}$ and key which is either $\mathsf{DK}_1\{S_{i^*+1,j}\}$ or $\mathsf{DK}_1\{S_{i^*+1,j+1}\}$, it is easy to reconstruct the rest of the distribution, as long as $i^* \neq 0$. That is, we can sample remaining keys, obfuscate all programs, and compute $\ell_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$ (using the challenge encryption key $\mathsf{EK}_1\{S_{i^*+1,j+1}\}$ which is not punctured at $(1, m_1^*)$ since $i^* \neq 0$) and $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. $\square$

Lemma 21. $\mathsf{adv}_{\mathsf{Hyb}_{B,4,j,2},\mathsf{Hyb}_{B,4,j,3}}(\lambda) \le 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$ *for* $i^* \le j \le T$.

Proof. In programs $\mathsf{GenZero}$, $\mathsf{Increment}$ we unpuncture EK_1 at $p_{j+1} = (j+1, m_1^*)$.

In program $\mathsf{GenZero}$ we can unpuncture EK_1 at p_{j+1} without changing the functionality, since $\mathsf{GenZero}$ only encrypts plaintexts of the form $(0, m_1)$, but $j+1 \neq 0$.

In program $\mathsf{Increment}$ we can unpuncture EK_1 at p_{j+1} without changing the functionality, since DK_1 is punctured at the point p_j, thus $\mathsf{Increment}$ never needs to encrypt p_{j+1} since on input $[j, m_1^*]$ it instead outputs $'\mathsf{fail}'$ during decryption. $\square$

Lemma 22. $\mathsf{adv}_{\mathsf{Hyb}_{B,5,1},\mathsf{Hyb}_{B,5,2}}(\lambda) \le 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$.

Proof. First, note that both programs $\mathsf{GenZero}_{B,5,1}$ and $\mathsf{GenZero}_{B,5,2}$ are functionally equivalent: since $i^*+1 \neq 0$, and $\mathsf{GenZero}$ only needs to encrypt value 0, we can safely unpuncture EK_1 at (i^*+1, m_1^*).

Second, programs $\mathsf{Increment}_{B,5,1}$ and $\mathsf{Increment}_{B,5,2}$ are functionally equivalent as well: the only difference in the code is that the first outputs $'\mathsf{fail}'$ when it tries to encrypt a punctured point (i^*+1, m_1^*) (which happens on input $\mathsf{ACE.Enc}_{\mathsf{EK}_1}(i^*, m_1^*)$), and the second outputs $'\mathsf{fail}'$ when $(m_1, i) = (m_1^*, i^*)$, which happens on the same input $\mathsf{ACE.Enc}_{\mathsf{EK}_1}(i^*, m_1^*)$. $\square$

Lemma 23. $\mathsf{adv}_{\mathsf{Hyb}_{B,5,2},\mathsf{Hyb}_{B,5,3}}(\lambda) \le 2^{-\Omega(\gamma(\lambda))}$.

Proof. Assume there is a poly-time distinguisher D which distinguishes between these two hybrids with probability $\eta \ge 2^{-o(\gamma(\lambda))}$ (for infinitely many λ_i). Then, since:

- programs $\mathsf{Increment}_B$ and $\mathsf{Increment}_{B,1,1}$ differ only at one point (due to the fact that g is injective);
- $\eta \ge 2^{-o(\gamma(\lambda))} \ge 2^{-o(\nu_{\mathsf{iO}}(\lambda))}$ (from the condition $\gamma(\lambda) \le O(\nu_{\mathsf{iO}}(\lambda)))$ in the theorem statement),

it follows from lemma 1 that there exists an inverter which runs in time at most $O(1/\eta)\log T = 2^{o(\gamma(\lambda))}\log T$, which by the condition in the theorem statement is at most $O(2^{\nu_{\mathsf{OWF}}(\log T)})$. This inverter inverts the one way function with probability at least $(1 - 2^{-\Omega(\lambda)})\eta$, which contradicts the fact that g is $2^{O(\nu_{\mathsf{OWF}}(\lambda \log T))}, 2^{-\Omega(\nu_{\mathsf{OWF}}(\lambda \log T))}$-secure OWF. ☐

7.5.3 Reductions in the proof of lemma 4 (Restoring behavior of Transform)

We show that

$$\mathsf{adv}_{\mathsf{Hyb}_C,\mathsf{Hyb}_D}(\lambda) \le 2^{\tau(\lambda)}(T \cdot 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))} + T \cdot 2^{-\Omega(\nu_{\mathsf{ACE.Indist}}(\lambda))} + 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}).$$

Lemma 24. $\mathsf{adv}_{\mathsf{Hyb}_{C,1,q},\mathsf{Hyb}_{C,1,q,1,1}}(\lambda) \le 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$.

Proof. In program Transform we puncture encryption key EK_2 at $p_{T,q} = (T, m_1^*, q)$. This is without changing the functionality, since Transform never encrypts this point: indeed, it encrypts $(i-1, m_1, m_2)$ when $m_2 = q$, but will abort instead if $i = T+1$. ☐

Lemma 25. $\mathsf{adv}_{\mathsf{Hyb}_{C,1,q,1,1},\mathsf{Hyb}_{C,1,q,1,2}}(\lambda) \le 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}$ *for* $q \ne m_2^*$.

Proof. Indistinguishability immediately follows from security of constrained decryption of ACE for the challenge set $\{p_{T,q}\} = \{(T, m_1^*, q)\}$ to puncture encryption key EK_2 and challenge sets $\{p_{T,q}\}, \varnothing$ to puncture decryption key DK_2. Indeed, given $\mathsf{EK}_2\{p_{T,q}\}$ and key which is either $\mathsf{DK}_2\{p_{T,q}\}$ or DK_2, it is easy to reconstruct the rest of the distribution. That is, we can sample remaining keys, obfuscate all programs, and compute $\ell_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$ and $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$ (using the challenge encryption key $\mathsf{EK}_2\{p_{T,q}\}$ which is not punctured at $(0, m_1^*, m_2^*)$. ☐

Lemma 26. $\mathsf{adv}_{\mathsf{Hyb}_{C,1,q,2,j,1},\mathsf{Hyb}_{C,1,q,2,j,2}}(\lambda) \le 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$, *for* $q \ne m_2^*, 0 \le j \le T-1$.

Proof. We puncture ACE keys $\mathsf{EK}_2, \mathsf{DK}_2$ at the point $p_{j,q} = (j, m_1^*, q)$ and hardwire $L_{j,q}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, q)$ to eliminate the need to encrypt or decrypt $p_{j,q}$ in programs Transform, isLess, and RetrieveTags, without changing their functionality.

More specifically, in program Transform we puncture EK_2 at $p_{j,q} = (j, m_1^*, q)$ and, in order to preserve the functionality, add an instruction to output $L_{j,q}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, q)$ when $(i, m_1, m_2) = (j+1, m_1^*, q)$.

In program isLess we puncture decryption key DK_2 at $p_{j,q} = (j, m_1^*, q)$ and, in order to preserve the functionality, instruct the program not to decrypt $L_{j,q}^*$, but to use $(j+1, m_1^*, q)$ as the result of decryption instead. Note that this is different from what $L_{j,q}^*$ would normally decrypt to, which is (j, m_1^*, q). However, we argue that this doesn't change the functionality of the program. Indeed:

- The set of inputs on which isLess outputs $'\mathsf{fail}'$ isn't changed; in particular, since $0 \le j \le T-1$, both (j, m_1^*, q) and $(j+1, m_1^*, q)$ are within 0 to T limits and thus are both valid.
- The result of the comparison on inputs $[i', m_1, m_2]$ and $[i'', m_1, m_2]$, where $(m_1, m_2) \neq (m_1^*, q)$, remains the same;
- The result of the comparison on inputs $[i', m_1^*, q]$ and $[i'', m_1^*, q]$, where $i', i'' \neq j$ and $i', i'' \neq j+1$, remains the same;
- The output of the program on inputs $([i', m_1^*, q], [i'', m_1^*, q])$, where $i' = j+1$ or $i'' = j+1$, is $'\mathsf{fail}'$ for both the original and modified programs, since DK_2 is punctured at $p_{j+1,q} = (j+1, m_1^*, q)$ and thus decryption returns $'\mathsf{fail}'$;
- The result of the comparison on inputs $([i', m_1^*, q], [j, m_1^*, q] = L_{j,q}^*)$, remains the same, since for both programs the output is:
  - true for $0 \le i' < j$;
  - false for $i' = j$ (indeed, in the original program in this case $i' = i'' = j$, and in the modified program $i' = i'' = j+1$, since $[i', m_1^*, q] = L_{j,q}^*$ when $i' = j$ and the program uses $j+1$ as the decryption result);
  - $'\mathsf{fail}'$ for $i' = j+1$, since DK_2 is punctured at $p_{j+1,q} = (j+1, m_1^*, q)$ and thus decryption returns $'\mathsf{fail}'$;
  - false for $j+2 \le i' \le T$.
- Similarly, the result of the comparison on inputs $([j, m_1^*, q] = L_{j,q}^*, [i', m_1^*, q])$ remains the same for the original program and modified program (with the difference that the result is false for $0 \le i' < j$ and true for $j+2 \le i' \le T$).

In program $\mathsf{RetrieveTags}$ we puncture decryption key DK_2 at $p_{j,q} = (j, m_1^*, q)$ and, in order to preserve the functionality, instruct the program to output (m_1^*, q) on input $L_{j,q}^*$. $\square$

Lemma 27. $\mathsf{adv}_{\mathsf{Hyb}_{C,1,q,2,j,2}, \mathsf{Hyb}_{C,1,q,2,j,3}}(\lambda) \le 2^{-\Omega(\nu_{\mathsf{ACE.Indist}}(\lambda))}$*, for* $q \neq m_2^*, 0 \le j \le T-1$.

Proof. Indistinguishability immediately follows from indistinguishability of ACE ciphertexts for the challenge plaintexts $p_{j,q} = (j, m_1^*, q)$ and $p_{j+1,q} = (j+1, m_1^*, q)$. Indeed, given $\mathsf{EK}_2\{p_{j,q}, p_{j+1,q}\}$, $\mathsf{DK}_2\{p_{j,q}, p_{j+1,q}\}$, and either $L_{j,q}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, q)$ or $L_{j+1,q}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j+1, m_1^*, q)$, it is easy to reconstruct the rest of the distribution. That is, we can sample remaining keys, obfuscate all program (note that indeed at most one of two ciphertexts $L_{j,q}^*$, $L_{j+1,q}^*$ is used in programs of $\mathsf{Hyb}_{C,1,q,2,j,2}$ and $\mathsf{Hyb}_{C,1,q,2,j,3}$), and compute $\ell_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$ and $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$ (using the challenge encryption key $\mathsf{EK}_2\{p_{j,q}, p_{j+1,q}\}$ which is not punctured at $(0, m_1^*, m_2^*)$ since $q \neq m_2^*$). $\square$

Lemma 28. $\mathsf{adv}_{\mathsf{Hyb}_{C,1,q,2,j,3}, \mathsf{Hyb}_{C,1,q,2,j,4}}(\lambda) \le 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$*, for* $q \neq m_2^*, 0 \le j \le T-1$.

Proof. We unpuncture ACE keys $\mathsf{EK}_2, \mathsf{DK}_2$ at the point $p_{j+1,q} = (j+1, m_1^*, q)$ and remove hardwired $L_{j+1,q}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j+1, m_1^*, q)$ in programs $\mathsf{Transform}$, isLess, and $\mathsf{RetrieveTags}$, without changing their functionality.

More specifically, in program $\mathsf{Transform}$ we unpuncture EK_2 at $p_{j+1,q} = (j+1, m_1^*, q)$ and remove an instruction to output $L_{j+1,q}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j+1, m_1^*, q)$ when $(i, m_1, m_2) = (j+1, m_1^*, q)$. This is without changing the functionality, since now the program will run an encryption $\mathsf{ACE.Enc}_{\mathsf{EK}_2}(j+1, m_1^*, q)$ when $(i, m_1, m_2) = (j+1, m_1^*, q)$, instead of directly outputting hardwired $L^*_{j+1,q}$.

In program isLess we unpuncture decryption key DK_2 at $p_{j+1,q} = (j+1, m_1^*, q)$ and remove an instruction to use $(j+1, m_1^*, q)$ as a result of decrypting $L^*_{j+1,q}$, thus making the program decrypt $L^*_{j+1,q}$ instead. This is without changing the functionality, since $(j+1, m_1^*, q)$ is what $L^*_{j+1,q}$ decrypts to.

In program $\mathsf{RetrieveTags}$ we unpuncture decryption key DK_2 at $p_{j+1,q} = (j+1, m_1^*, q)$ and remove an instruction to output (m_1^*, q) on input $L^*_{j+1,q}$. This is without changing the functionality, since (m_1^*, q) is what the program outputs when decrypting $L^*_{j+1,q}$. □

Lemma 29. $\mathsf{adv}_{\mathsf{Hyb}_{C,1,q,2,-1,1},\mathsf{Hyb}_{C,1,q,2,-1,2}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$*, for* $q \neq m_2^*$.

Proof. We puncture ACE keys $\mathsf{EK}_2, \mathsf{DK}_2$ at the point $p_{-1,q} = (-1, m_1^*, q)$ and hardwire $L^*_{-1,q} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(-1, m_1^*, q)$ to eliminate the need to encrypt or decrypt $p_{-1,q}$ in programs $\mathsf{Transform}$, isLess, and $\mathsf{RetrieveTags}$, without changing their functionality.

More specifically, in program $\mathsf{Transform}$ we puncture EK_2 at $p_{-1,q} = (-1, m_1^*, q)$ and, in order to preserve the functionality, add an instruction to output $L^*_{-1,q} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(-1, m_1^*, q)$ when $(i, m_1, m_2) = (0, m_1^*, q)$.

In program isLess we puncture decryption key DK_2 at $p_{-1,q} = (-1, m_1^*, q)$ and instruct the program to output $'\mathsf{fail}'$, given $L^*_{-1,q}$. This is without changing the functionality, since $[-1, m_1^*, q]$ is treated by the program as an invalid input, since the value i should be between 0 and T.

In program $\mathsf{RetrieveTags}$ we puncture decryption key DK_2 at $p_{-1,q} = (-1, m_1^*, q)$ and instruct the program to output $'\mathsf{fail}'$, given $L^*_{-1,q}$. This is without changing the functionality, since $[-1, m_1^*, q]$ is treated by the program as an invalid input, since the value i should be between 0 and T. □

Lemma 30. $\mathsf{adv}_{\mathsf{Hyb}_{C,1,q,2,-1,2},\mathsf{Hyb}_{C,1,q,2,-1,3}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{ACE.Indist}}(\lambda))}$*, for* $q \neq m_2^*$.

Proof. Indistinguishability immediately follows from indistinguishability of ACE ciphertexts for the challenge plaintexts $p_{-1,q} = (-1, m_1^*, q)$ and $p_{0,q} = (0, m_1^*, q)$. Indeed, given $\mathsf{EK}_2\{p_{-1,q}, p_{0,q}\}$, $\mathsf{DK}_2\{p_{-1,q}, p_{0,q}\}$, and either $L^*_{-1,q} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(-1, m_1^*, q)$ or $L^*_{0,q} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, q)$, it is easy to reconstruct the rest of the distribution. That is, we can sample remaining keys, obfuscate all program (note that indeed at most one of two ciphertexts $L^*_{-1,q}$, $L^*_{0,q}$ is used in programs of $\mathsf{Hyb}_{C,1,q,2,-1,2}$ and $\mathsf{Hyb}_{C,1,q,2,-1,3}$), and compute $\ell_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$ and $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$ (using the challenge encryption key $\mathsf{EK}_2\{p_{-1,q}, p_{0,q}\}$ which is not punctured at $(0, m_1^*, m_2^*)$ since $q \neq m_2^*$).

□

Lemma 31. $\mathsf{adv}_{\mathsf{Hyb}_{C,1,q,2,-1,3},\mathsf{Hyb}_{C,1,q,3,1}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$*, for* $q \neq m_2^*$.

Proof. We unpuncture ACE keys $\mathsf{EK}_2, \mathsf{DK}_2$ at the point $p_{0,q} = (0, m_1^*, q)$ and remove hardwired $L^*_{0,q} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, q)$ in programs $\mathsf{Transform}$, isLess, and $\mathsf{RetrieveTags}$, without changing their functionality.

More specifically, in program $\mathsf{Transform}$ we unpuncture EK_2 at $p_{0,q} = (0, m_1^*, q)$ and remove an instruction to output $L^*_{0,q} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, q)$ when $(i, m_1, m_2) = (0, m_1^*, q)$. This is without changing the functionality, since now the program will run an encryption $\mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, q)$ when $(i, m_1, m_2) = (0, m_1^*, q)$, instead of directly outputting hardwired $L^*_{0,q}$.

In program isLess we unpuncture decryption key DK_2 at $p_{0,q} = (0, m_1^*, q)$ and remove an instruction to output $'\mathsf{fail}'$ given $L_{0,q}^*$; to preserve the functionality, we instruct the program to output $'\mathsf{fail}'$ when (i', m_1', m_2') or (i'', m_1'', m_2'') is equal to $(0, m_1^*, q)$.

In program $\mathsf{RetrieveTags}$ we unpuncture decryption key DK_2 at $p_{0,q} = (0, m_1^*, q)$ and remove an instruction to output $'\mathsf{fail}'$ given $L_{0,q}^*$; to preserve the functionality, we instruct the program to output $'\mathsf{fail}'$ when $(i, m_1, m_2) = (0, m_1^*, q)$. □

Lemma 32. $\mathsf{adv}_{\mathsf{Hyb}_{C,1,q,3,1}, \mathsf{Hyb}_{C,1,q,3,2}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}$ *for* $q \neq m_2^*$.

Proof. Indistinguishability immediately follows from security of constrained decryption of ACE for the challenge set $\{p_{-1,q}\} = \{(-1, m_1^*, q)\}$ to puncture encryption key EK_2 and challenge sets $\{p_{-1,q}\}, \varnothing$ to puncture decryption key DK_2. Indeed, given $\mathsf{EK}_2\{p_{-1,q}\}$ and key which is either $\mathsf{DK}_2\{p_{-1,q}\}$ or DK_2, it is easy to reconstruct the rest of the distribution. That is, we can sample remaining keys, obfuscate all programs, and compute $\ell_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$ and $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$ (using the challenge encryption key $\mathsf{EK}_2\{p_{-1,q}\}$ which is not punctured at $(0, m_1^*, m_2^*)$ since $q \neq m_2^*$). □

Lemma 33. $\mathsf{adv}_{\mathsf{Hyb}_{C,1,q,3,2}, \mathsf{Hyb}_{C,1,q,3,3}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$*, for* $q \neq m_2^*$.

Proof. We unpuncture ACE key EK_2 at the point $p_{-1,q} = (-1, m_1^*, q)$ in program $\mathsf{Transform}$. This is without changing the functionality, since this program never needs to encrypt $p_{-1,q}$: indeed, when $(m_1, m_2) = (m_1^*, q)$, the program only encrypts (i, m_1, m_2), where $0 \leq i \leq T$. □

Lemma 34. $\mathsf{adv}_{\mathsf{Hyb}_{C,1,q,3,3}, \mathsf{Hyb}_{C,1,q,4,1}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$*, for* $q \neq m_2^*$.

Proof. In programs $\mathsf{GenZero}$ and $\mathsf{Increment}$ we puncture encryption key EK_1 at $p_0 = (0, m_1^*)$. This is without changing the functionality, since neither program needs to encrypt this point. □

Lemma 35. $\mathsf{adv}_{\mathsf{Hyb}_{C,1,q,4,1}, \mathsf{Hyb}_{C,1,q,4,2}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}$.

Proof. Indistinguishability immediately follows from security of constrained decryption of ACE for the challenge set $\{p_0\} = \{(0, m_1^*)\}$ to puncture encryption key EK_1 and challenge sets $\{p_0\}, \varnothing$ to puncture decryption key DK_1. Indeed, given $\mathsf{EK}_1\{p_0\}$ and key which is either DK_1 or $\mathsf{DK}_1\{p_0\}$, it is easy to reconstruct the rest of the distribution. That is, we can sample remaining keys, obfuscate all programs, and compute $\ell_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$ (using the challenge encryption key $\mathsf{EK}_1\{p_0\}$ which is not punctured at $(1, m_1^*)$) and $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. □

Lemma 36. $\mathsf{adv}_{\mathsf{Hyb}_{C,1,q,4,2}, \mathsf{Hyb}_{C,1,q,4,3}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$*, for* $q \neq m_2^*$.

Proof. In program $\mathsf{Transform}$ we puncture encryption key EK_2 at $p_{0,q} = (0, m_1^*, q)$. This is without changing the functionality: indeed, in order to encrypt $p_{0,q}$, the program should get $([0, m_1^*], q)$ as input, but on this input $\mathsf{Transform}$ instead outputs $'\mathsf{fail}'$, since decryption key DK_1 is punctured at $p_0 = (0, m_1^*)$. □

Lemma 37. $\mathsf{adv}_{\mathsf{Hyb}_{C,1,q,4,3}, \mathsf{Hyb}_{C,1,q,4,4}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}$ *for* $q \neq m_2^*$.

Proof. Indistinguishability immediately follows from security of constrained decryption of ACE for the challenge set $\{p_{0,q}\} = \{(0, m_1^*, q)\}$ to puncture encryption key EK_2 and challenge sets $\{p_{0,q}\}, \varnothing$ to puncture decryption key DK_2. Indeed, given $\mathsf{EK}_2\{p_{0,q}\}$ and key which is either $\mathsf{DK}_2\{p_{0,q}\}$ or DK_2, it is easy to reconstruct the rest of the distribution. That is, we can sample remaining keys, obfuscate all programs, and compute $\ell_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$ and $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$ (using the challenge encryption key $\mathsf{EK}_2\{p_{0,q}\}$ which is not punctured at $(0, m_1^*, m_2^*)$ since $q \neq m_2^*$). $\square$

Lemma 38. $\mathsf{adv}_{\mathsf{Hyb}_{C,1,q,4,4},\mathsf{Hyb}_{C,1,q,4,5}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$*, for* $q \neq m_2^*$.

Proof. In programs isLess and $\mathsf{RetrieveTags}$ we remove an instruction to output $'\mathsf{fail}'$, given $[0, m_1^*, q]$. This is without changing the functionality, since in both programs DK_2 is punctured at $p_{0,q} = (0, m_1^*, q)$, thus making the programs output $'\mathsf{fail}'$ during decryption; thus the instructions which we are removing are never reached anyway, and we can safely remove them. $\square$

Lemma 39. $\mathsf{adv}_{\mathsf{Hyb}_{C,1,q,4,5},\mathsf{Hyb}_{C,1,q,4,6}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}$ *for* $q \neq m_2^*$.

Proof. Indistinguishability immediately follows from security of constrained decryption of ACE for the challenge set $\{p_{0,q}\} = \{(0, m_1^*, q)\}$ to puncture encryption key EK_2 and challenge sets $\{p_{0,q}\}, \varnothing$ to puncture decryption key DK_2. Indeed, given $\mathsf{EK}_2\{p_{0,q}\}$ and key which is either $\mathsf{DK}_2\{p_{0,q}\}$ or DK_2, it is easy to reconstruct the rest of the distribution. That is, we can sample remaining keys, obfuscate all programs, and compute $\ell_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$ and $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$ (using the challenge encryption key $\mathsf{EK}_2\{p_{0,q}\}$ which is not punctured at $(0, m_1^*, m_2^*)$ since $q \neq m_2^*$). $\square$

Lemma 40. $\mathsf{adv}_{\mathsf{Hyb}_{C,1,q,4,6},\mathsf{Hyb}_{C,1,q,4,7}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$*, for* $q \neq m_2^*$.

Proof. In program $\mathsf{Transform}$ we unpuncture encryption key EK_2 at $p_{0,q} = (0, m_1^*, q)$. This is without changing the functionality: indeed, in order to encrypt $p_{0,q}$, the program should get $([0, m_1^*], q)$ as input, but on this input $\mathsf{Transform}$ instead outputs $'\mathsf{fail}'$, since decryption key DK_1 is punctured at $p_0 = (0, m_1^*)$. $\square$

Lemma 41. $\mathsf{adv}_{\mathsf{Hyb}_{C,1,q,4,7},\mathsf{Hyb}_{C,1,q,4,8}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}$.

Proof. Indistinguishability immediately follows from security of constrained decryption of ACE for the challenge set $\{p_0\} = \{(0, m_1^*)\}$ to puncture encryption key EK_1 and challenge sets $\{p_0\}, \varnothing$ to puncture decryption key DK_1. Indeed, given $\mathsf{EK}_1\{p_0\}$ and key which is either DK_1 or $\mathsf{DK}_1\{p_0\}$, it is easy to reconstruct the rest of the distribution. That is, we can sample remaining keys, obfuscate all programs, and compute $\ell_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$ (using the challenge encryption key $\mathsf{EK}_1\{p_0\}$ which is not punctured at $(1, m_1^*)$) and $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. $\square$

Lemma 42. $\mathsf{adv}_{\mathsf{Hyb}_{C,1,q,4,8},\mathsf{Hyb}_{C,1,q,4,9}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$*, for* $q \neq m_2^*$.

Proof. In programs $\mathsf{GenZero}$ and $\mathsf{Increment}$ we unpuncture encryption key EK_1 at $p_0 = (0, m_1^*)$. This is without changing the functionality, since neither program needs to encrypt this point. $\square$

Lemma 43. $\mathsf{adv}_{\mathsf{Hyb}_{C,2,1,1},\mathsf{Hyb}_{C,2,1,2}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$.

Proof. In program $\mathsf{Transform}$ we puncture encryption key EK_2 at $p_{T,m_2^*} = (T, m_1^*, m_2^*)$. This is without changing the functionality, since $\mathsf{Transform}$ never encrypts this point: indeed, when $(m_1, m_2) = (m_1^*, m_2^*)$ the largest value it encrypts is $(T-1, m_1, m_2)$. $\square$

Lemma 44. $\mathsf{adv}_{\mathsf{Hyb}_{C,2,1,2},\mathsf{Hyb}_{C,2,1,3}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}$.

Proof. Indistinguishability immediately follows from security of constrained decryption of ACE for the challenge set $\{p_{T,m_2^*}\} = \{(T, m_1^*, m_2^*)\}$ to puncture encryption key EK_2 and challenge sets $\{p_{T,m_2^*}\}, \varnothing$ to puncture decryption key DK_2. Indeed, given $\mathsf{EK}_2\{p_{T,m_2^*}\}$ and key which is either $\mathsf{DK}_2\{p_{T,m_2^*}\}$ or DK_2, it is easy to reconstruct the rest of the distribution. That is, we can sample remaining keys, obfuscate all programs, and compute $\ell_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$ and $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$ (using the challenge encryption key $\mathsf{EK}_2\{p_{T,m_2^*}\}$ which is not punctured at $(0, m_1^*, m_2^*)$). $\square$

Lemma 45. $\mathsf{adv}_{\mathsf{Hyb}_{C,2,2,j,1},\mathsf{Hyb}_{C,2,2,j,2}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$*, for* $1 \leq j \leq T-1$.

Proof. We puncture ACE keys $\mathsf{EK}_2, \mathsf{DK}_2$ at the point $p_{j,m_2^*} = (j, m_1^*, m_2^*)$ and hardwire $L_{j,m_2^*}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, m_2^*)$ to eliminate the need to encrypt or decrypt p_{j,m_2^*} in programs $\mathsf{Transform}$, isLess, and $\mathsf{RetrieveTags}$, without changing their functionality.

More specifically, in program $\mathsf{Transform}$ we puncture EK_2 at $p_{j,m_2^*} = (j, m_1^*, m_2^*)$ and, in order to preserve the functionality, add an instruction to output $L_{j,m_2^*}^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, m_2^*)$ when $(i, m_1, m_2) = (j+1, m_1^*, m_2^*)$.

In program isLess we puncture decryption key DK_2 at $p_{j,m_2^*} = (j, m_1^*, m_2^*)$ and, in order to preserve the functionality, instruct the program not to decrypt $L_{j,m_2^*}^*$, but to use $(j+1, m_1^*, m_2^*)$ as the result of decryption instead. Note that this is different from what $L_{j,m_2^*}^*$ would normally decrypt to, which is (j, m_1^*, m_2^*). However, we argue that this doesn't change the functionality of the program. Indeed:

- The set of inputs on which isLess outputs $'\mathsf{fail}'$ isn't changed; in particular, since $0 \leq j \leq T-1$, both (j, m_1^*, m_2^*) and $(j+1, m_1^*, m_2^*)$ are within 0 to T limits and thus are both valid.
- The result of the comparison on inputs $[i', m_1, m_2]$ and $[i'', m_1, m_2]$, where $(m_1, m_2) \neq (m_1^*, m_2^*)$, remains the same, for all i', i'';
- The result of the comparison on inputs $[i', m_1^*, m_2^*]$ and $[i'', m_1^*, m_2^*]$, where $i', i'' \neq j$ and $i', i'' \neq j+1$, remains the same;
- The output of the program on inputs $([i', m_1^*, m_2^*], [i'', m_1^*, m_2^*])$, where $i' = j+1$ or $i'' = j+1$, is $'\mathsf{fail}'$ for both the original and modified programs, since DK_2 is punctured at $p_{j+1,m_2^*} = (j+1, m_1^*, m_2^*)$ and thus decryption returns $'\mathsf{fail}'$;
- The result of the comparison on inputs $([i', m_1^*, m_2^*], [j, m_1^*, m_2^*] = L_{j,m_2^*}^*)$, remains the same, since for both programs the output is:
  - true for $0 \leq i' < j$;
  - false for $i' = j$ (indeed, in the original program in this case $i' = i'' = j$, and in the modified program $i' = i'' = j+1$, since $[i', m_1^*, m_2^*] = L_{j,m_2^*}^*$ when $i' = j$ and the program uses $j+1$ as the decryption result);
  - $'\mathsf{fail}'$ for $i' = j+1$, since DK_2 is punctured at $p_{j+1,m_2^*} = (j+1, m_1^*, m_2^*)$ and thus decryption returns $'\mathsf{fail}'$;
  - false for $j+2 \leq i' \leq T$.
- Similarly, the result of the comparison on inputs $([j, m_1^*, m_2^*] = L_{j,m_2^*}^*, [i', m_1^*, m_2^*])$ remains the same for the original program and modified program (with the difference that the result is false for $0 \leq i' < j$ and true for $j+2 \leq i' \leq T$).

In program $\mathsf{RetrieveTags}$ we puncture decryption key DK_2 at $p_{j,m_2^*} = (j, m_1^*, m_2^*)$ and, in order to preserve the functionality, instruct the program to output (m_1^*, m_2^*) on input L^*_{j,m_2^*}. ☐

Lemma 46. $\mathsf{adv}_{\mathsf{Hyb}_{C,2,2,j,2},\mathsf{Hyb}_{C,2,2,j,3}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{ACE.Indist}}(\lambda))}$*, for* $1 \leq j \leq T-1$.

Proof. Indistinguishability immediately follows from indistinguishability of ACE ciphertexts for the challenge plaintexts $p_{j,m_2^*} = (j, m_1^*, m_2^*)$ and $p_{j+1,m_2^*} = (j+1, m_1^*, m_2^*)$. Indeed, given $\mathsf{EK}_2\{p_{j,m_2^*}, p_{j+1,m_2^*}\}$, $\mathsf{DK}_2\{p_{j,m_2^*}, p_{j+1,m_2^*}\}$, and either $L^*_{j,m_2^*} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j, m_1^*, m_2^*)$ or $L^*_{j+1,m_2^*} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j+1, m_1^*, m_2^*)$, it is easy to reconstruct the rest of the distribution. That is, we can sample remaining keys, obfuscate all programs (note that creating the programs in each of the hybrids $\mathsf{Hyb}_{C,2,2,j,2}$, $\mathsf{Hyb}_{C,2,2,j,3}$ requires to know exactly *one* of the two ciphertexts L^*_{j,m_2^*}, L^*_{j+1,m_2^*}), and compute $\ell_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$ and $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$ (using the challenge encryption key $\mathsf{EK}_2\{p_{j,m_2^*}, p_{j+1,m_2^*}\}$ which is not punctured at $(0, m_1^*, m_2^*)$ since $j \geq 1$). ☐

Lemma 47. $\mathsf{adv}_{\mathsf{Hyb}_{C,2,2,j,3},\mathsf{Hyb}_{C,2,2,j,4}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$*, for* $1 \leq j \leq T-1$.

Proof. We unpuncture ACE keys $\mathsf{EK}_2, \mathsf{DK}_2$ at the point $p_{j+1,m_2^*} = (j+1, m_1^*, m_2^*)$ and remove hardwired $L^*_{j+1,m_2^*} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j+1, m_1^*, m_2^*)$ in programs $\mathsf{Transform}$, isLess, and $\mathsf{RetrieveTags}$, without changing their functionality.

More specifically, in program $\mathsf{Transform}$ we unpuncture EK_2 at $p_{j+1,m_2^*} = (j+1, m_1^*, m_2^*)$ and remove an instruction to output $L^*_{j+1,m_2^*} = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(j+1, m_1^*, m_2^*)$ when $(i, m_1, m_2) = (j+1, m_1^*, m_2^*)$. This is without changing the functionality, since now the program will run an encryption $\mathsf{ACE.Enc}_{\mathsf{EK}_2}(j+1, m_1^*, m_2^*)$ when $(i, m_1, m_2) = (j+1, m_1^*, m_2^*)$, instead of directly outputting hardwired L^*_{j+1,m_2^*}.

In program isLess we unpuncture decryption key DK_2 at $p_{j+1,m_2^*} = (j+1, m_1^*, m_2^*)$ and remove an instruction to use $(j+1, m_1^*, m_2^*)$ as a result of decrypting L^*_{j+1,m_2^*}, thus making the program decrypt L^*_{j+1,m_2^*} instead. This is without changing the functionality, since $(j+1, m_1^*, m_2^*)$ is what L^*_{j+1,m_2^*} decrypts to.

In program $\mathsf{RetrieveTags}$ we unpuncture decryption key DK_2 at $p_{j+1,m_2^*} = (j+1, m_1^*, m_2^*)$ and remove an instruction to output (m_1^*, m_2^*) on input L^*_{j+1,m_2^*}. This is without changing the functionality, since (m_1^*, m_2^*) is what the program outputs when decrypting L^*_{j+1,m_2^*}. ☐

Lemma 48. $\mathsf{adv}_{\mathsf{Hyb}_{C,2,3,1},\mathsf{Hyb}_{C,2,3,2}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}$.

Proof. Indistinguishability immediately follows from security of constrained decryption of ACE for the challenge set $\{p_{1,m_2^*}\} = \{(1, m_1^*, m_2^*)\}$ to puncture encryption key EK_2 and challenge sets $\{p_{1,m_2^*}\}, \varnothing$ to puncture decryption key DK_2. Indeed, given $\mathsf{EK}_2\{p_{1,m_2^*}\}$ and *key* which is either $\mathsf{DK}_2\{p_{1,m_2^*}\}$ or DK_2, it is easy to reconstruct the rest of the distribution. That is, we can sample remaining keys, obfuscate all programs, and compute $\ell_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$ and $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$ (using the challenge encryption key $\mathsf{EK}_2\{p_{1,m_2^*}\}$ which is not punctured at $(0, m_1^*, m_2^*)$). ☐

Lemma 49. $\mathsf{adv}_{\mathsf{Hyb}_{C,2,3,2},\mathsf{Hyb}_{C,2,3,3}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$.

Proof. In program $\mathsf{Transform}$ we do the following changes. First, we change the condition for when to encrypt $i-1$ from $i \leq 1$ to $i \leq 0$. This is without changing the functionality, since the case $(i, m_1, m_2) = (1, m_1^*, m_2^*)$ corresponds to the input $([1, m_1^*], m_2^*)$, in which case the program outputs $'\mathsf{fail}'$ at the very beginning, thus the line with the condition is not reached on this input anyway. For the same reason we can unpuncture EK_2 at $p_{1,m_2^*} = (1, m_1^*, m_2^*)$.

Next, in programs $\mathsf{GenZero}$ and $\mathsf{Increment}$ we puncture encryption key EK_1 at $p_0 = (0, m_1^*)$. This is without changing the functionality, since neither program needs to encrypt this point. $\square$

Lemma 50. $\mathsf{adv}_{\mathsf{Hyb}_{C,2,3,3},\mathsf{Hyb}_{C,2,3,4}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}$.

Proof. Indistinguishability immediately follows from security of constrained decryption of ACE for the challenge set $\{p_0\} = \{(0, m_1^*)\}$ to puncture encryption key EK_1 and challenge sets $\{p_0\}, \varnothing$ to puncture decryption key DK_1. Indeed, given $\mathsf{EK}_1\{p_0\}$ and key which is either DK_1 or $\mathsf{DK}_1\{p_0\}$, it is easy to reconstruct the rest of the distribution. That is, we can sample remaining keys, obfuscate all programs, and compute $\ell_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$ (using the challenge encryption key $\mathsf{EK}_1\{p_0\}$ which is not punctured at $(1, m_1^*)$) and $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. $\square$

Lemma 51. $\mathsf{adv}_{\mathsf{Hyb}_{C,2,3,4},\mathsf{Hyb}_{C,2,3,5}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$.

Proof. In program $\mathsf{Transform}$ we make the program output $\mathsf{ACE.Enc}_{\mathsf{EK}_2}(i, m_1, m_2)$ instead of $\mathsf{ACE.Enc}_{\mathsf{EK}_2}(i-1, m_1, m_2)$ for the case $(i, m_1, m_2) = (0, m_1^*, m_2^*)$; this is without changing the funcitonality, since encryption is never reached in the case. Indeed, on input $([0, m_1^*], m_2^*)$ $\mathsf{Transform}$ outputs $'\mathsf{fail}'$ during decryption, since DK_1 is punctured at $p_0 = (0, m_1^*)$. $\square$

Lemma 52. $\mathsf{adv}_{\mathsf{Hyb}_{C,2,3,5},\mathsf{Hyb}_{C,2,3,6}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{ACE.ConstrDec}}(\lambda))}$.

Proof. Indistinguishability immediately follows from security of constrained decryption of ACE for the challenge set $\{p_0\} = \{(0, m_1^*)\}$ to puncture encryption key EK_1 and challenge sets $\{p_0\}, \varnothing$ to puncture decryption key DK_1. Indeed, given $\mathsf{EK}_1\{p_0\}$ and key which is either DK_1 or $\mathsf{DK}_1\{p_0\}$, it is easy to reconstruct the rest of the distribution. That is, we can sample remaining keys, obfuscate all programs, and compute $\ell_1^* = \mathsf{ACE.Enc}_{\mathsf{EK}_1}(1, m_1^*)$ (using the challenge encryption key $\mathsf{EK}_1\{p_0\}$ which is not punctured at $(1, m_1^*)$) and $L_0^* = \mathsf{ACE.Enc}_{\mathsf{EK}_2}(0, m_1^*, m_2^*)$. $\square$

Lemma 53. $\mathsf{adv}_{\mathsf{Hyb}_{C,2,3,6},\mathsf{Hyb}_{C,2,3,7}}(\lambda) \leq 2^{-\Omega(\nu_{\mathsf{iO}}(\lambda))}$.

Proof. In programs $\mathsf{GenZero}$ and $\mathsf{Increment}$ we unpuncture encryption key EK_1 at $p_0 = (0, m_1^*)$. This is without changing the functionality, since neither program needs to encrypt this point. $\square$

8 Proof of bideniability of our encryption protocol

8.1 List of hybrids

In this section we present a list of hybrids with brief explanation of why indistinguishability holds. Formal security reductions can be found in section 8.2.

We note that we repeat some hybrids in order to get 4 clean steps (e.g. hybrids $\mathsf{Hyb}_{B,3,3} - \mathsf{Hyb}_{B,3,5}$ at the very end of the proof of lemma 55 are immediately undone at the very beginning of the proof of lemma 56).

Lemma 54. [Indistinguishability of explanations of the sender] *Assuming $(t(\lambda), \varepsilon(\lambda))$ security of relaxed ACE, iO and sparse extracting PRFs, the distiributions in $\mathsf{Hyb}_A, \mathsf{Hyb}_B$ are $(t(\lambda), O(\varepsilon(\lambda)))$-close.*

Lemma 55. [Indistinguishability of explanations of the receiver] *Assuming $(t(\lambda), \varepsilon(\lambda))$ security of ACE, relaxed ACE,* iO*,* prg *and sparse extracting PRFs, the distiributions in* $\mathsf{Hyb}_B, \mathsf{Hyb}_C$ *are* $(t(\lambda), O(\varepsilon(\lambda)) + 2^{-\tau(\lambda)})$*-close.*

Lemma 56. [Semantic security] *Assuming $(t(\lambda), \varepsilon(\lambda))$ security of ACE, relaxed ACE,* iO*, and sparse extracting PRFs, the distiributions in* $\mathsf{Hyb}_C, \mathsf{Hyb}_D$ *are* $(t(\lambda), O(\varepsilon(\lambda)) + O(2^{-\tau(\lambda)}))$*-close.*

Lemma 57. [Indistinguishability of levels] *Assuming $(t(\lambda), \varepsilon(\lambda))$ security of relaxed ACE,* iO*, and sparse extracting PRFs, and assuming $(t(\lambda), \varepsilon_1(\lambda, T, \tau))$-secure level system, the distiributions in* $\mathsf{Hyb}_D, \mathsf{Hyb}_E$ *are* $(t(\lambda), O(\varepsilon(\lambda)) + \varepsilon_1(\lambda, T, \tau))$*-close.*

8.1.1 Proof of lemma 54 (Indistinguishability of explanation of the sender)

- $\mathsf{Hyb}_{A,1}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s^*, r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, $\mu_1{}^* = \mathsf{SG}(s^*, m_0^*)$, $\mu_2{}^* = \mathsf{P2}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$. Programs are presented on fig. 94.

  Note that $\mathsf{Hyb}_{A,1} = \mathsf{Hyb}_A$, conditioned on the fact that s^* is outside of the image of ACE.

- $\mathsf{Hyb}_{A,2}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s^*, r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{A,1}, \mathsf{P2}, \mathsf{P3}_{A,1}, \mathsf{Dec}, \mathsf{SFake}_{A,1}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, $\mu_1{}^* = \mathsf{SG}(s^*, m_0^*)$, $\mu_2{}^* = \mathsf{P2}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc}_{EK}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$. Programs are presented on fig. 95.

  That is, we modify programs of the sender by puncturing encryption key of sender-fake ACE $\mathsf{EK}_S\{S_{\ell_0^*}\}$ at the set $S_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$, decryption key of sender-fake ACE $\mathsf{DK}_S\{s^*, s'\}$ at s^* and s' (where $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$), and the key k_S of extracting PRF SG of the sender at the points (s^*, m_0^*) and (s', m_0^*). In addition, we hardwire certain outputs inside programs of the sender to make sure that functionality of the programs doesn't change. Indistinguishability holds by iO.

- $\mathsf{Hyb}_{A,3}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s^*, r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{A,1}, \mathsf{P2}, \mathsf{P3}_{A,1}, \mathsf{Dec}, \mathsf{SFake}_{A,1}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, $\mu_1{}^*$ is chosen at random, $\mu_2{}^* = \mathsf{P2}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc}_{EK}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$. Programs are presented on fig. 95.

  That is, we choose $\mu_1{}^*$ at random instead of computing it as $\mu_1{}^* = \mathsf{SG}_{k_S}(s^*, m_0^*)$. Indistinguishability holds by pseudorandomness of the PRF SG at the punctured point (s^*, m_0^*).

- $\mathsf{Hyb}_{A,4}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{A,1}, \mathsf{P2}, \mathsf{P3}_{A,1}, \mathsf{Dec}, \mathsf{SFake}_{A,1}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, $\mu_1{}^*$ is chosen at random, $\mu_2{}^* = \mathsf{P2}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc}_{EK}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. Programs are presented on fig. 95.

  That is, we switch the roles of s^* and s' everywhere in the distribution: namely, we give s' (instead of s^*) to the adversary as randomness of the sender, and we change s^* to s' and s' to s^* everywhere in the programs. Note that this doesn't change the code of the programs since programs use s^* and s' in the same way. Indistinguishability holds by the symmetry of sender-fake ACE, which says that $(s^*, s', \mathsf{EK}_S\{S_{\ell_0^*}\}, \mathsf{DK}_S\{s^*, s'\})$ is indistinguishable from $(s', s^*, \mathsf{EK}_S\{S_{\ell_0^*}\}, \mathsf{DK}_S\{s^*, s'\})$, where $p = (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, s^* is randomly chosen, $s' = \mathsf{ACE.Enc}_{EK_S}(p)$. Note that $\mathsf{DK}_S\{s^*, s'\}$ is first punctured at one of the points s^*, s' which is lexicographically smaller, and then at the other.

- $\mathsf{Hyb}_{A,5}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{A,1}, \mathsf{P2}, \mathsf{P3}_{A,1}, \mathsf{Dec}, \mathsf{SFake}_{A,1}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, $\mu_1{}^* = \mathsf{SG}(s^*, m_0^*)$, $\mu_2{}^* = \mathsf{P2}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc}_{EK}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. Programs are presented on fig. 95.

  That is, we generate $\mu_1{}^*$ as $\mu_1{}^* = \mathsf{SG}_{k_S}(s^*, m_0^*)$ instead of choosing it at random. Indistinguishability holds by pseudorandomness of the PRF SG at the punctured point (s^*, m_0^*).

- $\mathsf{Hyb}_{A,6}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, $\mu_1{}^* = \mathsf{SG}(s^*, m_0^*)$, $\mu_2{}^* = \mathsf{P2}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc}_{EK}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. Programs are presented on fig. 94.

  That is, we revert all changes we made to programs and thus use original programs of our deniable encryption scheme in this hybrid. Indistinguishability holds by iO, since we remove puncturing without changing the functionality of the programs.

  Note that $\mathsf{Hyb}_{A,6} = \mathsf{Hyb}_B$, conditioned on the fact that s^* is outside of the image of ACE.

Programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$**.**

Program $\mathsf{P1}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: decryption key DK_S of sender-fake ACE, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = '\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m = m'$ then return ${\mu_1}'$;
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; decryption key DK_S of sender-fake ACE, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = '\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', {\mu_1}', {\mu_2}'$ then return ${\mu_3}'$;
   (c) If $m, \mu_1 = m', {\mu_1}'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; encryption and decryption keys $\mathsf{EK}_S, \mathsf{DK}_S$ of sender-fake ACE.

1. **Validity check:** if $\mathsf{P1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = '\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1 = m', {\mu_1}'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = '\mathsf{fail}'$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 94: Programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$.

Programs $\mathsf{P1}_{A,1}, \mathsf{P3}_{A,1}, \mathsf{SFake}_{A,1}$.

Program $\mathsf{P1}_{A,1}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: punctured decryption key $\mathsf{DK}_S\{s^*, s'\}$ of sender-fake ACE, punctured key $k_S\{(s^*, m_0^*), (s', m_0^*)\}$ of an extracting PRF SG, variables $s^*, s', m_0^*, {\mu_1}^*$.

1. **Trapdoor step:**
   (a) If $(s, m) = (s^*, m_0^*)$ or $(s, m) = (s', m_0^*)$ then return ${\mu_1}^*$;
   (b) If $s = s^*$ or $s = s'$ then goto main step;
   (c) out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{s^*,s'\}}(s)$; if out $=$ 'fail' goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (d) If $m = m'$ then return ${\mu_1}'$;
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S\{(s^*,m_0^*),(s',m_0^*)\}}(s, m)$.

Program $\mathsf{P3}_{A,1}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{A,1}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; punctured decryption key $\mathsf{DK}_S\{s^*, s'\}$ of sender-fake ACE, encryption key EK of main ACE, variables $s^*, s', m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*$.

1. **Validity check:** if $\mathsf{P1}_{A,1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) If $(s, m, \mu_1, \mu_2) = (s^*, m_0^*, {\mu_1}^*, {\mu_2}^*)$ or $(s, m, \mu_1, \mu_2) = (s', m_0^*, {\mu_1}^*, {\mu_2}^*)$ then return ${\mu_3}^*$;
   (b) If $(s, m, \mu_1) = (s^*, m_0^*, {\mu_1}^*)$ or $(s, m, \mu_1) = (s', m_0^*, {\mu_1}^*)$ then return $\mu_3 \leftarrow \mathsf{Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, \mu_2, \mathsf{Transform}(\ell_0^*, \mu_2))$;
   (c) If $s = s^*$ or $s = s'$ then goto main step;
   (d) out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{s^*,s'\}}(s)$; if out $=$ 'fail' goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (e) If $m, \mu_1, \mu_2 = m', {\mu_1}', {\mu_2}'$ then return ${\mu_3}'$;
   (f) If $m, \mu_1 = m', {\mu_1}'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}_{A,1}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{A,1}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; punctured encryption key $\mathsf{EK}_S\{S_{\ell_0^*}\}$ (where $S_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$) and punctured decryption key $\mathsf{DK}_S\{s^*, s'\}$, variables $s^*, s', m_0^*, {\mu_1}^*, \ell_0^*$.

1. **Validity check:** if $\mathsf{P1}_{A,1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) If $(s, m, \mu_1) = (s^*, m_0^*, {\mu_1}^*)$ or $(s, m, \mu_1) = (s', m_0^*, {\mu_1}^*)$ then return $\mathsf{Enc}_{\mathsf{EK}_S\{p\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \mathsf{Increment}(\ell_0^*))$;
   (b) If $s = s^*$ or $s = s'$ then goto main step;
   (c) out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{s^*,s'\}}(s)$; if out $=$ 'fail' goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (d) If $m, \mu_1 = m', {\mu_1}'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} =$ 'fail' then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{S_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{S_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 95: Programs $\mathsf{P1}_{A,1}, \mathsf{P3}_{A,1}, \mathsf{SFake}_{A,1}$, used in the proof of lemma 54 (indistinguishability of explanations of the sender).

8.1.2 Proof of lemma 55 (Indistinguishability of explanation of the receiver)

First in a sequence of hybrids we "eliminate" complementary ciphertext $\overline{\mu_3{}^*} = \mathsf{ACE.Enc_{EK}}(1 \oplus m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, i.e. make programs Dec and SFake reject it:

- $\mathsf{Hyb}_{B,1,1}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, $\mu_1{}^* = \mathsf{SG}(s^*, m_0^*)$, $\mu_2{}^* = \mathsf{RG}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc_{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. Programs can be found in fig. 96 (programs of the sender) and fig. 100 (programs of the receiver).

  Note that this distribution is exactly the distribution from Hyb_B, conditioned on the fact that s^*, r^* are outside of images of their ACE.

- $\mathsf{Hyb}_{B,1,2}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,1}, \mathsf{P2}, \mathsf{P3}_{B,1}, \mathsf{Dec}, \mathsf{SFake}_{B,1}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, $\mu_1{}^* = \mathsf{SG}(s^*, m_0^*)$, $\mu_2{}^* = \mathsf{RG}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc_{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. Programs can be found in fig. 97 (programs of the sender) and fig. 100 (programs of the receiver).

  That is, in program SFake we puncture encryption key EK_S of the sender-fake ACE at the set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. Indistinguishability holds by iO, since this modification doesn't change the functionality of SFake due to the fact that SFake never encrypts plaintexts with level ℓ_0^*.

- $\mathsf{Hyb}_{B,1,3}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,2}, \mathsf{P2}, \mathsf{P3}_{B,2}, \mathsf{Dec}, \mathsf{SFake}_{B,2}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, $\mu_1{}^* = \mathsf{SG}(s^*, m_0^*)$, $\mu_2{}^* = \mathsf{RG}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc_{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. Programs can be found in fig. 98 (programs of the sender) and fig. 100 (programs of the receiver).

  That is, in programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$ we puncture decryption key DK_S of the sender-fake ACE at the same set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. Indistinguishability holds by security of constrained decryption of ACE, since the corresponding encryption key EK_S is already punctured at the same set.

- $\mathsf{Hyb}_{B,1,4}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,2}, \mathsf{P2}, \mathsf{P3}_{B,2}, \mathsf{Dec}, \mathsf{SFake}_{B,2}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, $\mu_1{}^*$ is chosen at random, $\mu_2{}^* = \mathsf{RG}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc_{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. Programs can be found in fig. 98 (programs of the sender) and fig. 100 (programs of the receiver).

  That is, we choose $\mu_1{}^*$ at random instead of computing it as $\mu_1{}^* = \mathsf{SG}_{k_S}(s^*, m_0^*)$. Indistinguishability holds by the strong extracting property of the sender PRF SG (note that s^* was not used anywhere else in the distribution).

- $\mathsf{Hyb}_{B,1,5}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,3}, \mathsf{P2}, \mathsf{P3}_{B,3}, \mathsf{Dec}, \mathsf{SFake}_{B,3}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, $\mu_1{}^*$ is chosen at random, $\mu_2{}^* = \mathsf{RG}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc_{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. Programs can be found in fig. 99 (programs of the sender) and fig. 100 (programs of the receiver).

That is, in program P3 we puncture encryption key EK of the main ACE at the point $\overline{p} = (1 \oplus m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$. Indistinguishability holds by iO, since P3 never needs to encrypt this point. Roughly, this is because of the following: since $\mu_1{}^*$ is random and outside of the image of a PRF SG, P3 never encrypts $\overline{p}$ in the main step. In order to encrypt it in trapdoor step, P3 needs to take as input some fake s encoding level ℓ_0^*. However, due to the fact that DK_S is punctured at the set $P_{\ell_0^*}$ which contains all but one strings with ℓ_0^*, the only valid fake s with ℓ_0^* is s'. However, running P3 on s' cannot result in encrypting $\overline{p}$ in the trapdoor step since $\overline{p}$ contains the wrong plaintext $1 \oplus m_0^*$ (instead of m_0^*).

- $\mathsf{Hyb}_{B,1,6}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,3}, \mathsf{P2}_{B,1}, \mathsf{P3}_{B,3}, \mathsf{Dec}_{B,1}, \mathsf{SFake}_{B,3}, \mathsf{RFake}_{B,1}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, $\mu_1{}^*$ is chosen at random, $\mu_2{}^* = \mathsf{RG}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. Programs can be found in fig. 99 (programs of the sender) and fig. 101 (programs of the receiver).

  That is, in programs Dec, RFake we puncture decryption key DK of the main ACE at the same point $\overline{p} = (1 \oplus m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$. Indistinguishability holds by security of constrained decryption of ACE, since the corresponding encryption key EK is already punctured at this point.

Now $\overline{\mu_3{}^*} = \mathsf{ACE.Enc}_{\mathsf{EK}}(1 \oplus m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$ is rejected by Dec and RFake. In the following hybrids, similarly to previous lemma, we switch the roles of r^* and r', using the fact that programs treat them similarly, once $\overline{\mu_3{}^*}$ is eliminated[33].

- $\mathsf{Hyb}_{B,2,1}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,3}, \mathsf{P2}_{B,2}, \mathsf{P3}_{B,3}, \mathsf{Dec}_{B,2}, \mathsf{SFake}_{B,3}, \mathsf{RFake}_{B,2}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, $\mu_1{}^*$ is chosen at random, $\mu_2{}^* = \mathsf{RG}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. Programs can be found in fig. 99 (programs of the sender) and fig. 102 (programs of the receiver).

  That is, we modify programs of the receiver (P2, Dec, RFake) by puncturing encryption key of receiver-fake ACE $\mathsf{EK}_R\{S_{\hat{\rho}^*}\}$ at $S_{\hat{\rho}^*} = \{(*, *, *, *, *, \hat{\rho}^*)\}$ for randomly chosen $\hat{\rho}^*$. Next, we puncture decryption key of receiver-fake ACE $\mathsf{DK}_R\{r^*, r'\}$ at r^* and r' (where $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(p)$, $p = (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \hat{\rho}^*)$), and the key k_R of extracting PRF RG of the receiver at the points $(r^*, \mu_1{}^*)$ and $(r', \mu_1{}^*)$. In addition, we hardwire certain outputs inside programs of the receiver to make sure that functionality of the programs doesn't change. Indistinguishability holds by iO.

- $\mathsf{Hyb}_{B,2,2}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,3}, \mathsf{P2}_{B,2}, \mathsf{P3}_{B,3}, \mathsf{Dec}_{B,2}, \mathsf{SFake}_{B,3}, \mathsf{RFake}_{B,2}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, $\mu_1{}^*$ is chosen at random, $\mu_2{}^*$ is chosen at random, $\mu_3{}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. Programs can be found in fig. 99 (programs of the sender) and fig. 102 (programs of the receiver).

  That is, we choose $\mu_2{}^*$ at random instead of computing it as $\mu_2{}^* = \mathsf{RG}_{k_S}(r^*, \mu_1{}^*)$. Indistinguishability holds by pseudorandomness of the PRF SG at the punctured point $(r^*, \mu_1{}^*)$.

[33] The problem with $\overline{\mu_3{}^*}$ is that unmodified Dec on input $(r^*, \mu_1{}^*, \mu_2{}^*, \overline{\mu_3{}^*})$ outputs $1 \oplus m_0^*$ (via main step), and on input $(r', \mu_1{}^*, \mu_2{}^*, \overline{\mu_3{}^*})$ it outputs 'fail' (via trapdoor step, since levels in r' and $\overline{\mu_3{}^*}$ are both 0 and "isLess = true" check fails. Because of this difference, in $\mathsf{Hyb}_{B,2,1}$ we wouldn't be able to modify program Dec such that the code treats r^* and r' in the same way. However, after $\mathsf{Hyb}_{B,1,6}$ $\overline{\mu_3{}^*}$ is not a valid ciphertext anymore and thus in $\mathsf{Hyb}_{B,2,1}$ we can instruct Dec to output 'fail' on both r^* and r'.

- $\mathsf{Hyb}_{B,2,3}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,3}, \mathsf{P2}_{B,2}, \mathsf{P3}_{B,3}, \mathsf{Dec}_{B,2}, \mathsf{SFake}_{B,3}, \mathsf{RFake}_{B,2}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^*$ is chosen at random, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \hat{\rho}^*)$ for randomly chosen $\hat{\rho}^*$. Programs can be found in fig. 99 (programs of the sender) and fig. 102 (programs of the receiver).

  That is, we switch the roles of r^* and r' everywhere in the distribution: namely, we give r' (instead of r^*) to the adversary as randomness of the receiver, and we change r^* to r' and r' to r^* everywhere in the programs. Note that this doesn't change the code of the programs since programs use r^* and r' in the same way. Indistinguishability holds by the symmetry of receiver-fake ACE, which says that $(r^*, r', \mathsf{EK}_R\{S_{\hat{\rho}^*}\}, \mathsf{DK}_R\{r^*, r'\})$ is indistinguishable from $(r', r^*, \mathsf{EK}_R\{S_{\hat{\rho}^*}\}, \mathsf{DK}_R\{r', r^*\})$, where $S_{\hat{\rho}^*} = \{(*, *, *, *, *, \hat{\rho}^*)\}$, $p = (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \hat{\rho}^*)$, r^* is randomly chosen, $r' = \mathsf{ACE.Enc}_{EK_R}(p)$.

- $\mathsf{Hyb}_{B,2,4}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,3}, \mathsf{P2}_{B,2}, \mathsf{P3}_{B,3}, \mathsf{Dec}_{B,2}, \mathsf{SFake}_{B,3}, \mathsf{RFake}_{B,2}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \hat{\rho}^*)$ for randomly chosen $\hat{\rho}^*$. Programs can be found in fig. 99 (programs of the sender) and fig. 102 (programs of the receiver).

  That is, we compute ${\mu_2}^*$ as ${\mu_2}^* = \mathsf{RG}_{k_R}(r^*, {\mu_1}^*)$ instead of choosing it at random. Indistinguishability holds by pseudorandomness of the PRF RG at the punctured point $(r^*, {\mu_1}^*)$.

- $\mathsf{Hyb}_{B,2,5}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,3}, \mathsf{P2}_{B,1}, \mathsf{P3}_{B,3}, \mathsf{Dec}_{B,1}, \mathsf{SFake}_{B,3}, \mathsf{RFake}_{B,1}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \hat{\rho}^*)$ for randomly chosen $\hat{\rho}^*$. Programs can be found in fig. 99 (programs of the sender) and fig. 101 (programs of the receiver).

  That is, we revert all changes we made to programs in $\mathsf{Hyb}_{B,2,1}$ and thus use original programs $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$, except that DK remains punctured at the point $\overline{p} = (1 \oplus m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$. Indistinguishability holds by iO, since we remove puncturing without changing the functionality of the programs.

- $\mathsf{Hyb}_{B,2,6}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,3}, \mathsf{P2}_{B,1}, \mathsf{P3}_{B,3}, \mathsf{Dec}_{B,1}, \mathsf{SFake}_{B,3}, \mathsf{RFake}_{B,1}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, ${\mu_1}^* = \mathsf{SG}_{k_S}(s^*, m_0^*)$, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 99 (programs of the sender) and fig. 101 (programs of the receiver).

  That is, we replace randomly chosen $\hat{\rho}^*$ with $\mathsf{prg}(\rho^*)$ for randomly chosen ρ^*, when generating r'. Indistinguishability holds by security of a prg.

Finally, in the following hybrids we revert all changes we made in hybrids $\mathsf{Hyb}_{B,1,1}$ - $\mathsf{Hyb}_{B,1,6}$, thus restoring all programs (and making $\overline{{\mu_3}^*}$ a valid ciphertext):

- $\mathsf{Hyb}_{B,3,1}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,3}, \mathsf{P2}, \mathsf{P3}_{B,3}, \mathsf{Dec}, \mathsf{SFake}_{B,3}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 99 (programs of the sender) and fig. 100 (programs of the receiver).

  That is, in programs $\mathsf{Dec}, \mathsf{RFake}$ we unpuncture decryption key DK of the main ACE at the point $\overline{p} = (1 \oplus m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$. Indistinguishability holds by security of constrained decryption of ACE, since the corresponding encryption key EK is punctured at this point.

- $\mathsf{Hyb}_{B,3,2}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,2}, \mathsf{P2}, \mathsf{P3}_{B,2}, \mathsf{Dec}, \mathsf{SFake}_{B,2}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 98 (programs of the sender) and fig. 100 (programs of the receiver).

  That is, in program $\mathsf{P3}$ we unpuncture encryption key EK of the main ACE at the point $\overline{p} = (1 \oplus m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$. Indistinguishability holds by iO, because of the same reason as in $\mathsf{Hyb}_{B,1,5}$.

- $\mathsf{Hyb}_{B,3,3}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,2}, \mathsf{P2}, \mathsf{P3}_{B,2}, \mathsf{Dec}, \mathsf{SFake}_{B,2}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, ${\mu_1}^* = \mathsf{SG}(s^*, m_0^*)$, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 98 (programs of the sender) and fig. 100 (programs of the receiver).

  That is, we choose ${\mu_1}^*$ as ${\mu_1}^* = \mathsf{SG}_{k_S}(s^*, m_0^*)$ instead of computing it at random. Indistinguishability holds by the strong extracting property of the sender PRF SG (note that s^* is not used anywhere else in the distribution).

- $\mathsf{Hyb}_{B,3,4}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,1}, \mathsf{P2}, \mathsf{P3}_{B,1}, \mathsf{Dec}, \mathsf{SFake}_{B,1}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, ${\mu_1}^* = \mathsf{SG}(s^*, m_0^*)$, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 97 (programs of the sender) and fig. 100 (programs of the receiver).

  That is, in programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$ we unpuncture decryption key DK_S of the sender-fake ACE at the same set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$. Indistinguishability holds by security of constrained decryption of ACE, since the corresponding encryption key EK_S is already punctured at the same set.

- $\mathsf{Hyb}_{B,3,5}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, ${\mu_1}^* = \mathsf{SG}(s^*, m_0^*)$, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 96 (programs of the sender) and fig. 100 (programs of the receiver).

That is, in program SFake we unpuncture encryption key EK_S of the sender-fake ACE at the set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$. Indistinguishability holds by iO, since this modification doesn't change the functionality of SFake due to the fact that SFake never encrypts plaintexts with level ℓ_0^*.

Note that $\mathsf{Hyb}_{B,3,5}$ is the same as Hyb_C, conditioned on the fact that s^*, r^* are outside of image of ACE.

Programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$**.**

Program $\mathsf{P1}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: decryption key DK_S of sender-fake ACE, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m = m'$ then return μ_1';
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; decryption key DK_S of sender-fake ACE, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', \mu_1', \mu_2'$ then return μ_3';
   (c) If $m, \mu_1 = m', \mu_1'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; encryption and decryption keys $\mathsf{EK}_S, \mathsf{DK}_S$ of sender-fake ACE.

1. **Validity check:** if $\mathsf{P1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m, \mu_1 = m', \mu_1'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = \mathsf{'fail'}$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 96: Programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$.

Programs $\mathsf{P1}_{B,1}, \mathsf{P3}_{B,1}, \mathsf{SFake}_{B,1}$**.**

Program $\mathsf{P1}_{B,1}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: decryption key DK_S of sender-fake ACE, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m = m'$ then return ${\mu_1}'$;
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}_{B,1}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{B,1}$, GenZero, Transform, RetrieveTag; decryption key DK_S of sender-fake ACE, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}_{B,1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', {\mu_1}', {\mu_2}'$ then return ${\mu_3}'$;
   (c) If $m, \mu_1 = m', {\mu_1}'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}_{B,1}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{B,1}$, GenZero, Increment; punctured encryption key $\mathsf{EK}_S\{P_{\ell_0^*}\}$ and decryption key DK_S of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$.

1. **Validity check:** if $\mathsf{P1}_{B,1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1 = m', {\mu_1}'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = \mathsf{'fail'}$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 97: Programs $\mathsf{P1}_{B,1}, \mathsf{P3}_{B,1}, \mathsf{SFake}_{B,1}$, used in the proof of lemma 55 (indistinguishability of explanations of the receiver).

Programs $\mathsf{P1}_{B,2}, \mathsf{P3}_{B,2}, \mathsf{SFake}_{B,2}$.

Program $\mathsf{P1}_{B,2}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: punctured decryption key $\mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) out $\leftarrow$ $\mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if out $=$ $'\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m = m'$ then return ${\mu_1}'$;
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}_{B,2}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{B,2}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; punctured decryption key $\mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}_{B,2}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) out $\leftarrow$ $\mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if out $=$ $'\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', {\mu_1}', {\mu_2}'$ then return ${\mu_3}'$;
   (c) If $m, \mu_1 = m', {\mu_1}'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}_{B,2}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{B,2}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; punctured encryption and decryption keys $\mathsf{EK}_S\{P_{\ell_0^*}\}, \mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$.

1. **Validity check:** if $\mathsf{P1}_{B,2}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) out $\leftarrow$ $\mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if out $=$ $'\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1 = m', {\mu_1}'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = '\mathsf{fail}'$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 98: Programs $\mathsf{P1}_{B,2}, \mathsf{P3}_{B,2}, \mathsf{SFake}_{B,2}$, used in the proof of lemma 55 (indistinguishability of explanations of the receiver).

Programs $\mathsf{P1}_{B,3}, \mathsf{P3}_{B,3}, \mathsf{SFake}_{B,3}$.

Program $\mathsf{P1}_{B,3}(s,m)$

Inputs: sender randomness s, message m.

Hardwired values: punctured decryption key $\mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*,*,*,*,\ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m = m'$ then return ${\mu_1}'$;
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s,m)$.

Program $\mathsf{P3}_{B,3}(s,m,\mu_1,\mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{B,3}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; punctured decryption key $\mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*,*,*,*,\ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, punctured encryption key $\mathsf{EK}\{\overline{p}\}$ of main ACE, where $\overline{p} = (1 \oplus m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$.

1. **Validity check:** if $\mathsf{P1}_{B,3}(s,m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', {\mu_1}', {\mu_2}'$ then return ${\mu_3}'$;
   (c) If $m, \mu_1 = m', {\mu_1}'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}\{\overline{p}\}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}\{\overline{p}\}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}_{B,3}(s,m,\hat{m},\mu_1,\mu_2,\mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{B,3}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; punctured encryption and decryption keys $\mathsf{EK}_S\{P_{\ell_0^*}\}, \mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*,*,*,*,\ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$.

1. **Validity check:** if $\mathsf{P1}_{B,3}(s,m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1 = m', {\mu_1}'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = \mathsf{'fail'}$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 99: Programs $\mathsf{P1}_{B,3}, \mathsf{P3}_{B,3}, \mathsf{SFake}_{B,3}$, used in the proof of lemma 55 (indistinguishability of explanations of the receiver).

Programs $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$.

Program $\mathsf{P2}(r, \mu_1)$

Inputs: receiver randomness r, the first message μ_1 in the protocol.

Hardwired values: decryption key DK_R of receiver-fake ACE, key k_R of an extracting PRF RG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_R}(r)$; if $\mathsf{out} = \mathsf{'fail'}$ then goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', L', \hat{\rho})$;
   (b) If $\mu_1 = {\mu_1}'$ then return ${\mu_2}'$;
2. **Main step:**
   (a) Return $\mu_2 \leftarrow \mathsf{RG}_{k_R}(r, \mu_1)$.

Program $\mathsf{Dec}(r, \mu_1, \mu_2, \mu_3)$

Inputs: receiver randomness r, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P2}$, isLess, $\mathsf{RetrieveTags}$; decryption key DK_R of receiver-fake ACE, decryption key DK of the main ACE.

1. **Validity check:** if $\mathsf{P2}(r, \mu_1) \neq \mu_2$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_R}(r)$; if $\mathsf{out}' = \mathsf{'fail'}$ then goto main step; else parse out' as $(m', {\mu_1}', {\mu_2}', {\mu_3}', L', \hat{\rho})$;
   (b) if $\mu_1, \mu_2, \mu_3 = {\mu_1}', {\mu_2}', {\mu_3}'$ then return m';
   (c) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}}(\mu_3)$; if $\mathsf{out}'' = \mathsf{'fail'}$ then abort, else parse out'' as $(m'', {\mu_1}'', {\mu_2}'', L'')$;
   (d) If $\mu_1, \mu_2 = {\mu_1}', {\mu_2}'$ then
      i. If $({\mu_1}', {\mu_2}') = ({\mu_1}'', {\mu_2}'') = \mathsf{RetrieveTags}(L'')$ and $\mathsf{isLess}(L', L'') = \mathsf{true}$ then return m'';
      ii. Else abort.
3. **Main step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}}(\mu_3)$; if $\mathsf{out} = \mathsf{'fail'}$ then abort, else parse out as $(m'', {\mu_1}'', {\mu_2}'', L'')$;
   (b) If $(\mu_1, \mu_2) = ({\mu_1}'', {\mu_2}'') = \mathsf{RetrieveTags}(L'')$ then return m'';
   (c) Else abort.

Program $\mathsf{RFake}(\hat{m}, \mu_1, \mu_2, \mu_3; \rho)$

Inputs: fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3, random coins ρ.

Hardwired values: encryption key EK_R of receiver-fake ACE, decryption key DK of the main ACE.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}}(\mu_3)$; if $\mathsf{out} = \mathsf{'fail'}$ then abort, else parse out as $(m'', {\mu_1}'', {\mu_2}'', L'')$;
2. Return $r' \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_R}(\hat{m}, \mu_1, \mu_2, \mu_3, L'', \mathsf{prg}(\rho))$.

Figure 100: Programs $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$.

Programs $\mathsf{P2}_{B,1}, \mathsf{Dec}_{B,1}, \mathsf{RFake}_{B,1}$.

Program $\mathsf{P2}_{B,1}(r, \mu_1)$

Inputs: receiver randomness r, the first message μ_1 in the protocol.

Hardwired values: decryption key DK_R of receiver-fake ACE, key k_R of an extracting PRF RG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_R}(r)$; if $\mathsf{out} = \text{'fail'}$ then goto main step, else parse out as $(m', \mu_1{}', \mu_2{}', \mu_3{}', L', \hat{\rho})$;
   (b) If $\mu_1 = \mu_1{}'$ then return $\mu_2{}'$;
2. **Main step:**
   (a) Return $\mu_2 \leftarrow \mathsf{RG}_{k_R}(r, \mu_1)$.

Program $\mathsf{Dec}_{B,1}(r, \mu_1, \mu_2, \mu_3)$

Inputs: receiver randomness r, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P2}$, isLess, $\mathsf{RetrieveTags}$; decryption key DK_R of receiver-fake ACE, punctured decryption key $\mathsf{DK}\{\overline{p}\}$ of the main ACE, where $\overline{p} = (1 \oplus m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$.

1. **Validity check:** if $\mathsf{P2}(r, \mu_1) \neq \mu_2$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_R}(r)$; if $\mathsf{out}' = \text{'fail'}$ then goto main step; else parse out' as $(m', \mu_1{}', \mu_2{}', \mu_3{}', L', \hat{\rho})$;
   (b) if $\mu_1, \mu_2, \mu_3 = \mu_1{}', \mu_2{}', \mu_3{}'$ then return m';
   (c) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{\overline{p}\}}(\mu_3)$; if $\mathsf{out}'' = \text{'fail'}$ then abort, else parse out'' as $(m'', \mu_1{}'', \mu_2{}'', L'')$;
   (d) If $\mu_1, \mu_2 = \mu_1{}', \mu_2{}'$ then
      i. If $(\mu_1{}', \mu_2{}') = (\mu_1{}'', \mu_2{}'') = \mathsf{RetrieveTags}(L'')$ and $\mathsf{isLess}(L', L'') = \mathsf{true}$ then return m'';
      ii. Else abort.
3. **Main step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{\overline{p}\}}(\mu_3)$; if $\mathsf{out} = \text{'fail'}$ then abort, else parse out as $(m'', \mu_1{}'', \mu_2{}'', L'')$;
   (b) If $(\mu_1, \mu_2) = (\mu_1{}'', \mu_2{}'') = \mathsf{RetrieveTags}(L'')$ then return m'';
   (c) Else abort.

Program $\mathsf{RFake}_{B,1}(\hat{m}, \mu_1, \mu_2, \mu_3; \rho)$

Inputs: fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3, random coins ρ.

Hardwired values: encryption key EK_R of receiver-fake ACE, punctured decryption key $\mathsf{DK}\{\overline{p}\}$ of the main ACE, where $\overline{p} = (1 \oplus m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{\overline{p}\}}(\mu_3)$; if $\mathsf{out} = \text{'fail'}$ then abort, else parse out as $(m'', \mu_1{}'', \mu_2{}'', L'')$;
2. Return $r' \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_R}(\hat{m}, \mu_1, \mu_2, \mu_3, L'', \mathsf{prg}(\rho))$.

Figure 101: Programs $\mathsf{P2}_{B,1}, \mathsf{Dec}_{B,1}, \mathsf{RFake}_{B,1}$, used in the proof of lemma 55 (indistinguishability of explanations of the receiver).

Programs $\mathsf{P2}_{B,2}, \mathsf{Dec}_{B,2}, \mathsf{RFake}_{B,2}$.

Program $\mathsf{P2}_{B,2}(r, \mu_1)$

Inputs: receiver randomness r, the first message μ_1 in the protocol.

Hardwired values: punctured decryption key $\mathsf{DK}_R\{r^*, r'\}$ of receiver-fake ACE, punctured key $k_R\{(r^*, {\mu_1}^*), (r', {\mu_1}^*)\}$ of an extracting PRF RG, variables $r^*, r', {\mu_1}^*, {\mu_2}^*$.

1. **Trapdoor step:**
   (a) If $(r, \mu_1) = (r^*, {\mu_1}^*)$ or $(r, \mu_1) = (r', {\mu_1}^*)$ then return ${\mu_2}^*$;
   (b) If $r = r^*$ or $r = r'$ then goto main step;
   (c) out $\leftarrow$ $\mathsf{ACE.Dec}_{\mathsf{DK}_R\{r^*,r'\}}(r)$; if out $=$ 'fail' then goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', L', \hat{\rho})$;
   (d) If $\mu_1 = {\mu_1}'$ then return ${\mu_2}'$;
2. **Main step:**
   (a) Return $\mu_2 \leftarrow \mathsf{RG}_{k_R\{(r^*,{\mu_1}^*),(r',{\mu_1}^*)\}}(r, \mu_1)$.

Program $\mathsf{Dec}_{B,2}(r, \mu_1, \mu_2, \mu_3)$

Inputs: receiver randomness r, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P2}_{B,2}$, isLess, $\mathsf{RetrieveTags}$; punctured decryption key $\mathsf{DK}_R\{r^*, r'\}$ of receiver-fake ACE, punctured decryption key $\mathsf{DK}\{\overline{p}\}$ of the main ACE, where $\overline{p} = (1 \oplus m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, variables $r^*, r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, m_0^*$.

1. **Validity check:** if $\mathsf{P2}_{B,2}(r, \mu_1) \neq \mu_2$ then abort;
2. **Trapdoor step:**
   (a) If $(r, \mu_1, \mu_2, \mu_3) = (r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$ or $(r, \mu_1, \mu_2, \mu_3) = (r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$ then return m_0^*;
   (b) If $(r, \mu_1, \mu_2) = (r^*, {\mu_1}^*, {\mu_2}^*)$ or $(r, \mu_1, \mu_2) = (r', {\mu_1}^*, {\mu_2}^*)$ then then goto main step;
   (c) If $r = r^*$ or $r = r'$ then goto main step;
   (d) out $\leftarrow$ $\mathsf{ACE.Dec}_{\mathsf{DK}_R\{r^*,r'\}}(r)$; if out' $=$ 'fail' then goto main step; else parse out' as $(m', {\mu_1}', {\mu_2}', {\mu_3}', L', \hat{\rho})$;
   (e) if $\mu_1, \mu_2, \mu_3 = {\mu_1}', {\mu_2}', {\mu_3}'$ then return m';
   (f) out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{\overline{p}\}}(\mu_3)$; if out'' $=$ 'fail' then abort, else parse out'' as $(m'', {\mu_1}'', {\mu_2}'', L'')$;
   (g) If $\mu_1, \mu_2 = {\mu_1}', {\mu_2}'$ then
      i. If $({\mu_1}', {\mu_2}') = ({\mu_1}'', {\mu_2}'') = \mathsf{RetrieveTags}(L'')$ and $\mathsf{isLess}(L', L'') = \mathsf{true}$ then return m'';
      ii. Else abort.
3. **Main step:**
   (a) out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{\overline{p}\}}(\mu_3)$; if out $=$ 'fail' then abort, else parse out as $(m'', {\mu_1}'', {\mu_2}'', L'')$;
   (b) If $(\mu_1, \mu_2) = ({\mu_1}'', {\mu_2}'') = \mathsf{RetrieveTags}(L'')$ then return m'';
   (c) Else abort.

Program $\mathsf{RFake}_{B,2}(\hat{m}, \mu_1, \mu_2, \mu_3; \rho)$

Inputs: fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3, random coins ρ.

Hardwired values: punctured encryption key $\mathsf{EK}_R\{S_{\hat{\rho}^*}\}$ of receiver-fake ACE, where $S_{\hat{\rho}^*} = \{(*, *, *, *, *, \hat{\rho}^*)\}$ for randomly chosen $\hat{\rho}^*$, punctured decryption key $\mathsf{DK}\{\overline{p}\}$ of the main ACE, where $\overline{p} = (1 \oplus m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{\overline{p}\}}(\mu_3)$; if out $=$ 'fail' then abort, else parse out as $(m'', {\mu_1}'', {\mu_2}'', L'')$;
2. Return $r' \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_R\{p\}}(\hat{m}, \mu_1, \mu_2, \mu_3, L'', \mathsf{prg}(\rho))$.

Figure 102: Programs $\mathsf{P2}_{B,2}, \mathsf{Dec}_{B,2}, \mathsf{RFake}_{B,2}$, used in the proof of lemma 55 (indistinguishability of explanations of the receiver).

8.1.3 Proof of lemma 56 (Semantic security)

- $\mathsf{Hyb}_{C,1,1}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, $\mu_1{}^* = \mathsf{SG}(s^*, m_0^*)$, $\mu_2{}^* = \mathsf{RG}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 103 (programs of the sender) and fig. 107 (programs of the receiver).

  Note that this distribution is exactly the distribution from Hyb_C, conditioned on the fact that s^*, r^* are outside of image of ACE.

- $\mathsf{Hyb}_{C,1,2}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{C,1}, \mathsf{P2}, \mathsf{P3}_{C,1}, \mathsf{Dec}, \mathsf{SFake}_{C,1}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, $\mu_1{}^* = \mathsf{SG}(s^*, m_0^*)$, $\mu_2{}^* = \mathsf{RG}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 104 (programs of the sender) and fig. 107 (programs of the receiver).

  That is, in program SFake we puncture encryption key EK_S of the sender-fake ACE at the set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. Indistinguishability holds by iO, since this modification doesn't change the functionality of SFake due to the fact that SFake never encrypts plaintexts with level ℓ_0^*.

- $\mathsf{Hyb}_{C,1,3}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{C,2}, \mathsf{P2}, \mathsf{P3}_{C,2}, \mathsf{Dec}, \mathsf{SFake}_{C,2}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, $\mu_1{}^* = \mathsf{SG}(s^*, m_0^*)$, $\mu_2{}^* = \mathsf{RG}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 105 (programs of the sender) and fig. 107 (programs of the receiver).

  That is, in programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$ we puncture decryption key DK_S of the sender-fake ACE at the same set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. Indistinguishability holds by security of constrained decryption of ACE, since the corresponding encryption key EK_S is already punctured at the same set.

- $\mathsf{Hyb}_{C,1,4}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{C,2}, \mathsf{P2}, \mathsf{P3}_{C,2}, \mathsf{Dec}, \mathsf{SFake}_{C,2}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, $\mu_1{}^*$ is chosen at random, $\mu_2{}^* = \mathsf{RG}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 105 (programs of the sender) and fig. 107 (programs of the receiver).

  That is, we choose $\mu_1{}^*$ at random instead of computing it as $\mu_1{}^* = \mathsf{SG}_{k_S}(s^*, m_0^*)$. Indistinguishability holds by the strong extracting property of the sender PRF SG (note that s^* was not used anywhere else in the distribution).

- $\mathsf{Hyb}_{C,1,5}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{C,2}, \mathsf{P2}, \mathsf{P3}_{C,2}, \mathsf{Dec}, \mathsf{SFake}_{C,2}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; $\mu_1{}^*$

is chosen at random, $\mu_2{}^*$ is chosen at random, $\mu_3{}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 105 (programs of the sender) and fig. 107 (programs of the receiver).

That is, we choose $\mu_2{}^*$ at random instead of computing it as $\mu_2{}^* = \mathsf{RG}_{k_R}(r^*, \mu_1{}^*)$. Indistinguishability holds by the strong extracting property of the receiver PRF RG (note that r^* was not used anywhere else in the distribution).

- $\mathsf{Hyb}_{C,2,1}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{C,3}, \mathsf{P2}, \mathsf{P3}_{C,3}, \mathsf{Dec}, \mathsf{SFake}_{C,3}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; $\mu_1{}^*$ is chosen at random, $\mu_2{}^*$ is chosen at random, $\mu_3{}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 106 (programs of the sender) and fig. 107 (programs of the receiver).

  That is, in program P3 we puncture encryption key EK of the main ACE at the points $p_0 = (m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*), p_1 = (m_1^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$. Indistinguishability holds by iO, since P3 never needs to encrypt these points. Roughly, this is because of the following: since $\mu_1{}^*$ is random and outside of the image of a PRF SG, P3 never encrypts p_0, p_1 in the main step. In order to encrypt it in trapdoor step, P3 needs to take as input some fake s encoding level ℓ_0^*. However, due to the fact that DK_S is punctured at the set $P_{\ell_0^*}$ which contains all but one strings with ℓ_0^*, the only valid fake s with ℓ_0^* is s'. However, running P3 on s' cannot result in encrypting p_0 or p_1 in the trapdoor step: in order to hit the trapdoor step with s', the input to P3 should be $(s', m_0^*, \mu_1{}^*, \mu_2{}^*)$; however, in this case the program immediately outputs $\mu_3{}'$ without running an encryption algorithm.

- $\mathsf{Hyb}_{C,2,2}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{C,3}, \mathsf{P2}_{C,1}, \mathsf{P3}_{C,3}, \mathsf{Dec}_{C,1}, \mathsf{SFake}_{C,3}, \mathsf{RFake}_{C,1}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; $\mu_1{}^*$ is chosen at random, $\mu_2{}^*$ is chosen at random, $\mu_3{}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 106 (programs of the sender) and fig. 108 (programs of the receiver).

  That is, in programs $\mathsf{Dec}, \mathsf{RFake}$ we puncture decryption key DK of the main ACE at the point $p_1 = (m_1^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$. Indistinguishability holds by security of constrained decryption of ACE, since the corresponding encryption key EK is already punctured at this point (and encryption of p_1 is not used anywhere in the distribution).

- $\mathsf{Hyb}_{C,2,3}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{C,3}, \mathsf{P2}_{C,2}, \mathsf{P3}_{C,3}, \mathsf{Dec}_{C,2}, \mathsf{SFake}_{C,3}, \mathsf{RFake}_{C,2}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; $\mu_1{}^*$ is chosen at random, $\mu_2{}^*$ is chosen at random, $\mu_3{}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 106 (programs of the sender) and fig. 109 (programs of the receiver).

  That is, we modify programs Dec and RFake by additionally puncturing decryption key of main ACE DK at the point $p_0 = (m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$. In addition, we hardwire certain outputs inside program RFake to make sure that its functionality doesn't change. (Note that in program Dec we only puncture keys, without hardwiring anything. However, this doesn't change the functionality of Dec. This is because Dec would output $\perp$ when trying to decrypt an encryption of p_0 anyway: roughly, this is because the main step cannot be reached because ${\mu_2}^*$ doesn't have a preimage, and trapdoor step would output $\perp$ because there doesn't exist fake randomness with level smaller than 0.) Indistinguishability holds by iO.

- $\mathsf{Hyb}_{C,2,4}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{C,3}, \mathsf{P2}_{C,2}, \mathsf{P3}_{C,3}, \mathsf{Dec}_{C,2}, \mathsf{SFake}_{C,3}, \mathsf{RFake}_{C,2}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; ${\mu_1}^*$ is chosen at random, ${\mu_2}^*$ is chosen at random, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 106 (programs of the sender) and fig. 109 (programs of the receiver).

  That is, we generate ${\mu_3}^*$ as an encryption of $p_1 = (m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$ instead of $p_0 = (m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$. Indistinguishability holds by security of the main ACE, since encryption and decryption keys EK, DK are punctured at both p_0, p_1.

- $\mathsf{Hyb}_{C,2,5}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{C,3}, \mathsf{P2}_{C,3}, \mathsf{P3}_{C,3}, \mathsf{Dec}_{C,3}, \mathsf{SFake}_{C,3}, \mathsf{RFake}_{C,3}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; ${\mu_1}^*$ is chosen at random, ${\mu_2}^*$ is chosen at random, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 106 (programs of the sender) and fig. 110 (programs of the receiver).

  That is, we modify programs Dec and RFake by unpuncturing decryption key of main ACE DK at the point $p_1 = (m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$ (note that DK remains punctured at $p_0 = (m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$). We also remove additional instructions introduced in $\mathsf{Hyb}_{C,2,3}$. Indistinguishability holds by iO, since we don't change functionality of the programs.

- $\mathsf{Hyb}_{C,2,6}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{C,3}, \mathsf{P2}, \mathsf{P3}_{C,3}, \mathsf{Dec}, \mathsf{SFake}_{C,3}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; ${\mu_1}^*$ is chosen at random, ${\mu_2}^*$ is chosen at random, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 106 (programs of the sender) and fig. 107 (programs of the receiver).

  That is, in programs $\mathsf{Dec}, \mathsf{RFake}$ we unpuncture decryption key DK of the main ACE at the point $p_0 = (m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$. Indistinguishability holds by security of constrained decryption of ACE, since the corresponding encryption key EK is punctured at this point (and encryption of p_0 is not used anywhere in the distribution).

- $\mathsf{Hyb}_{C,2,7}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{C,2}, \mathsf{P2}, \mathsf{P3}_{C,2}, \mathsf{Dec}, \mathsf{SFake}_{C,2}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; ${\mu_1}^*$ is chosen at random, ${\mu_2}^*$ is chosen at random, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 105 (programs of the sender) and fig. 107 (programs of the receiver).

  That is, in program P3 we unpuncture encryption key EK of the main ACE at the points $p_0 = (m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $p_1 = (m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$. Indistinguishability holds by iO, since this doesn't change functionality of P3 for the same reason as in $\mathsf{Hyb}_{C,2,1}$.

- $\mathsf{Hyb}_{C,3,1}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{C,2}, \mathsf{P2}, \mathsf{P3}_{C,2}, \mathsf{Dec}, \mathsf{SFake}_{C,2}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 105 (programs of the sender) and fig. 107 (programs of the receiver).

  That is, we compute ${\mu_2}^*$ as ${\mu_2}^* = \mathsf{RG}_{k_R}(r^*, {\mu_1}^*)$ instead of choosing it at random. Indistinguishability holds by the strong extracting property of the receiver PRF RG (note that r^* is not used anywhere else in the distribution).

- $\mathsf{Hyb}_{C,3,2}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{C,2}, \mathsf{P2}, \mathsf{P3}_{C,2}, \mathsf{Dec}, \mathsf{SFake}_{C,2}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^* is chosen at random, r^* is chosen at random, ${\mu_1}^* = \mathsf{SG}_{k_S}(s^*, m_1^*)$, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 105 (programs of the sender) and fig. 107 (programs of the receiver).

  That is, we compute ${\mu_1}^*$ as ${\mu_1}^* = \mathsf{SG}_{k_S}(s^*, m_1^*)$ instead of choosing it at random. Indistinguishability holds by the strong extracting property of the sender PRF SG (note that s^* is not used anywhere else in the distribution).

- $\mathsf{Hyb}_{C,3,3}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{C,1}, \mathsf{P2}, \mathsf{P3}_{C,1}, \mathsf{Dec}, \mathsf{SFake}_{C,1}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^* is chosen at random, r^* is chosen at random, ${\mu_1}^* = \mathsf{SG}_{k_S}(s^*, m_1^*)$, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 104 (programs of the sender) and fig. 107 (programs of the receiver).

  That is, in programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$ we unpuncture decryption key DK_S of the sender-fake ACE at the set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$. Indistinguishability holds by security of constrained decryption of ACE, since the corresponding encryption key EK_S is punctured at the same set.

- $\mathsf{Hyb}_{C,3,4}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^* is chosen at random, r^* is chosen at random, ${\mu_1}^* = \mathsf{SG}_{k_S}(s^*, m_1^*)$, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 103 (programs of the sender) and fig. 107 (programs of the receiver).

  That is, in program SFake we unpuncture encryption key EK_S of the sender-fake ACE at the set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$. Indistinguishability holds by iO, since this modification doesn't change the functionality of SFake due to the fact that SFake never encrypts plaintexts with level ℓ_0^*.

  Note that $\mathsf{Hyb}_{C,3,4} = \mathsf{Hyb}_D$, conditioned on the fact that s^*, r^* are outisde of image of ACE.

Programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$**.**

Program $\mathsf{P1}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: decryption key DK_S of sender-fake ACE, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = {}'\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m = m'$ then return ${\mu_1}'$;
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; decryption key DK_S of sender-fake ACE, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = {}'\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', {\mu_1}', {\mu_2}'$ then return ${\mu_3}'$;
   (c) If $m, \mu_1 = m', {\mu_1}'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; encryption and decryption keys $\mathsf{EK}_S, \mathsf{DK}_S$ of sender-fake ACE.

1. **Validity check:** if $\mathsf{P1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = {}'\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1 = m', {\mu_1}'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = {}'\mathsf{fail}'$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 103: Programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$.

Programs $\mathsf{P1}_{C,1}, \mathsf{P3}_{C,1}, \mathsf{SFake}_{C,1}$**.**

Program $\mathsf{P1}_{C,1}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: decryption key DK_S of sender-fake ACE, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = \text{'}\mathsf{fail}\text{'}$ goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m = m'$ then return μ_1';
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}_{C,1}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{C,1}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; decryption key DK_S of sender-fake ACE, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}_{C,1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = \text{'}\mathsf{fail}\text{'}$ goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', \mu_1', \mu_2'$ then return μ_3';
   (c) If $m, \mu_1 = m', \mu_1'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}_{C,1}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{C,1}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; punctured encryption key $\mathsf{EK}_S\{P_{\ell_0^*}\}$ and decryption key DK_S of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$.

1. **Validity check:** if $\mathsf{P1}_{C,1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = \text{'}\mathsf{fail}\text{'}$ goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m, \mu_1 = m', \mu_1'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = \text{'}\mathsf{fail}\text{'}$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 104: Programs $\mathsf{P1}_{C,1}, \mathsf{P3}_{C,1}, \mathsf{SFake}_{C,1}$, used in the proof of lemma 56 (semantic security).

Programs $\mathsf{P1}_{C,2}, \mathsf{P3}_{C,2}, \mathsf{SFake}_{C,2}$**.**

Program $\mathsf{P1}_{C,2}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: punctured decryption key $\mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = {}'\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m = m'$ then return ${\mu_1}'$;
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}_{C,2}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{C,2}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; punctured decryption key $\mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}_{C,2}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = {}'\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', {\mu_1}', {\mu_2}'$ then return ${\mu_3}'$;
   (c) If $m, \mu_1 = m', {\mu_1}'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}_{C,2}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{C,2}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; punctured encryption and decryption keys $\mathsf{EK}_S\{P_{\ell_0^*}\}, \mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$.

1. **Validity check:** if $\mathsf{P1}_{C,2}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = {}'\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1 = m', {\mu_1}'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = {}'\mathsf{fail}'$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 105: Programs $\mathsf{P1}_{C,2}, \mathsf{P3}_{C,2}, \mathsf{SFake}_{C,2}$, used in the proof of lemma 56 (semantic security).

Programs $\mathsf{P1}_{C,3}, \mathsf{P3}_{C,3}, \mathsf{SFake}_{C,3}$**.**

Program $\mathsf{P1}_{C,3}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: punctured decryption key $\mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = '\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m = m'$ then return ${\mu_1}'$;
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}_{C,3}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{C,3}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; punctured decryption key $\mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, punctured encryption key $\mathsf{EK}\{p_0, p_1\}$ of main ACE, where $p_0 = (m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $p_1 = (m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$.

1. **Validity check:** if $\mathsf{P1}_{C,3}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = '\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', {\mu_1}', {\mu_2}'$ then return ${\mu_3}'$;
   (c) If $m, \mu_1 = m', {\mu_1}'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}\{p_0, p_1\}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}\{p_0, p_1\}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}_{C,3}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{C,3}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; punctured encryption and decryption keys $\mathsf{EK}_S\{P_{\ell_0^*}\}, \mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$.

1. **Validity check:** if $\mathsf{P1}_{C,3}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = '\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1 = m', {\mu_1}'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = '\mathsf{fail}'$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 106: Programs $\mathsf{P1}_{C,3}, \mathsf{P3}_{C,3}, \mathsf{SFake}_{C,3}$, used in the proof of lemma 56 (semantic security).

Programs $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$.

Program $\mathsf{P2}(r, \mu_1)$

Inputs: receiver randomness r, the first message μ_1 in the protocol.

Hardwired values: decryption key DK_R of receiver-fake ACE, key k_R of an extracting PRF RG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_R}(r)$; if $\mathsf{out} = '\mathsf{fail}'$ then goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', L', \hat{\rho})$;
   (b) If $\mu_1 = \mu_1'$ then return μ_2';
2. **Main step:**
   (a) Return $\mu_2 \leftarrow \mathsf{RG}_{k_R}(r, \mu_1)$.

Program $\mathsf{Dec}(r, \mu_1, \mu_2, \mu_3)$

Inputs: receiver randomness r, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P2}$, isLess, $\mathsf{RetrieveTags}$; decryption key DK_R of receiver-fake ACE, decryption key DK of the main ACE.

1. **Validity check:** if $\mathsf{P2}(r, \mu_1) \neq \mu_2$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_R}(r)$; if $\mathsf{out}' = '\mathsf{fail}'$ then goto main step; else parse out' as $(m', \mu_1', \mu_2', \mu_3', L', \hat{\rho})$;
   (b) if $\mu_1, \mu_2, \mu_3 = \mu_1', \mu_2', \mu_3'$ then return m';
   (c) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}}(\mu_3)$; if $\mathsf{out}'' = '\mathsf{fail}'$ then abort, else parse out'' as $(m'', \mu_1'', \mu_2'', L'')$;
   (d) If $\mu_1, \mu_2 = \mu_1', \mu_2'$ then
      i. If $(\mu_1', \mu_2') = (\mu_1'', \mu_2'') = \mathsf{RetrieveTags}(L'')$ and $\mathsf{isLess}(L', L'') = \mathsf{true}$ then return m'';
      ii. Else abort.
3. **Main step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}}(\mu_3)$; if $\mathsf{out} = '\mathsf{fail}'$ then abort, else parse out as $(m'', \mu_1'', \mu_2'', L'')$;
   (b) If $(\mu_1, \mu_2) = (\mu_1'', \mu_2'') = \mathsf{RetrieveTags}(L'')$ then return m'';
   (c) Else abort.

Program $\mathsf{RFake}(\hat{m}, \mu_1, \mu_2, \mu_3; \rho)$

Inputs: fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3, random coins ρ.

Hardwired values: encryption key EK_R of receiver-fake ACE, decryption key DK of the main ACE.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}}(\mu_3)$; if $\mathsf{out} = '\mathsf{fail}'$ then abort, else parse out as $(m'', \mu_1'', \mu_2'', L'')$;
2. Return $r' \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_R}(\hat{m}, \mu_1, \mu_2, \mu_3, L'', \mathsf{prg}(\rho))$.

Figure 107: Programs $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$.

Programs $\mathsf{P2}_{C,1}, \mathsf{Dec}_{C,1}, \mathsf{RFake}_{C,1}$.

Program $\mathsf{P2}_{C,1}(r, \mu_1)$

Inputs: receiver randomness r, the first message μ_1 in the protocol.

Hardwired values: decryption key DK_R of receiver-fake ACE, key k_R of an extracting PRF RG.

1. **Trapdoor step:**
   (a) out $\leftarrow$ $\mathsf{ACE.Dec}_{\mathsf{DK}_R}(r)$; if out $=$ $'$fail$'$ then goto main step, else parse out as $(m', \mu_1{}', \mu_2{}', \mu_3{}', L', \hat{\rho})$;
   (b) If $\mu_1 = \mu_1{}'$ then return $\mu_2{}'$;
2. **Main step:**
   (a) Return $\mu_2 \leftarrow \mathsf{RG}_{k_R}(r, \mu_1)$.

Program $\mathsf{Dec}_{C,1}(r, \mu_1, \mu_2, \mu_3)$

Inputs: receiver randomness r, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P2}_{C,1}$, isLess, RetrieveTags; decryption key DK_R of receiver-fake ACE, punctured decryption key $\mathsf{DK}\{p_1\}$ of the main ACE, where $p_1 = (m_1^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$.

1. **Validity check:** if $\mathsf{P2}_{C,1}(r, \mu_1) \neq \mu_2$ then abort;
2. **Trapdoor step:**
   (a) out $\leftarrow$ $\mathsf{ACE.Dec}_{\mathsf{DK}_R}(r)$; if out$'$ $=$ $'$fail$'$ then goto main step; else parse out$'$ as $(m', \mu_1{}', \mu_2{}', \mu_3{}', L', \hat{\rho})$;
   (b) if $\mu_1, \mu_2, \mu_3 = \mu_1{}', \mu_2{}', \mu_3{}'$ then return m';
   (c) out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{p_1\}}(\mu_3)$; if out$''$ $=$ $'$fail$'$ then abort, else parse out$''$ as $(m'', \mu_1{}'', \mu_2{}'', L'')$;
   (d) If $\mu_1, \mu_2 = \mu_1{}', \mu_2{}'$ then
      i. If $(\mu_1{}', \mu_2{}') = (\mu_1{}'', \mu_2{}'') = \mathsf{RetrieveTags}(L'')$ and $\mathsf{isLess}(L', L'') = \mathsf{true}$ then return m'';
      ii. Else abort.
3. **Main step:**
   (a) out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{p_1\}}(\mu_3)$; if out $=$ $'$fail$'$ then abort, else parse out as $(m'', \mu_1{}'', \mu_2{}'', L'')$;
   (b) If $(\mu_1, \mu_2) = (\mu_1{}'', \mu_2{}'') = \mathsf{RetrieveTags}(L'')$ then return m'';
   (c) Else abort.

Program $\mathsf{RFake}_{C,1}(\hat{m}, \mu_1, \mu_2, \mu_3; \rho)$

Inputs: fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3, random coins ρ.

Hardwired values: encryption key EK_R of receiver-fake ACE, punctured decryption key $\mathsf{DK}\{p_1\}$ of the main ACE, where $p_1 = (m_1^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{p_1\}}(\mu_3)$; if out $=$ $'$fail$'$ then abort, else parse out as $(m'', \mu_1{}'', \mu_2{}'', L'')$;
2. Return $r' \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_R}(\hat{m}, \mu_1, \mu_2, \mu_3, L'', \mathsf{prg}(\rho))$.

Figure 108: Programs $\mathsf{P2}_{C,1}, \mathsf{Dec}_{C,1}, \mathsf{RFake}_{C,1}$, used in the proof of lemma 56 (semantic security).

Programs $\mathsf{P2}_{C,2}, \mathsf{Dec}_{C,2}, \mathsf{RFake}_{C,2}$.

Program $\mathsf{P2}_{C,2}(r, \mu_1)$

Inputs: receiver randomness r, the first message μ_1 in the protocol.

Hardwired values: decryption key DK_R of receiver-fake ACE, key k_R of an extracting PRF RG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_R}(r)$; if $\mathsf{out} = \mathsf{'fail'}$ then goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', L', \hat{\rho})$;
   (b) If $\mu_1 = \mu_1'$ then return μ_2';
2. **Main step:**
   (a) Return $\mu_2 \leftarrow \mathsf{RG}_{k_R}(r, \mu_1)$.

Program $\mathsf{Dec}_{C,2}(r, \mu_1, \mu_2, \mu_3)$

Inputs: receiver randomness r, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P2}_{C,2}$, isLess, $\mathsf{RetrieveTags}$; decryption key DK_R of receiver-fake ACE, punctured decryption key $\mathsf{DK}\{p_0, p_1\}$ of the main ACE, where $p_0 = (m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $p_1 = (m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$.

1. **Validity check:** if $\mathsf{P2}_{C,2}(r, \mu_1) \neq \mu_2$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_R}(r)$; if $\mathsf{out}' = \mathsf{'fail'}$ then goto main step; else parse out' as $(m', \mu_1', \mu_2', \mu_3', L', \hat{\rho})$;
   (b) if $\mu_1, \mu_2, \mu_3 = \mu_1', \mu_2', \mu_3'$ then return m';
   (c) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{p_0,p_1\}}(\mu_3)$; if $\mathsf{out}'' = \mathsf{'fail'}$ then abort, else parse out'' as $(m'', \mu_1'', \mu_2'', L'')$;
   (d) If $\mu_1, \mu_2 = \mu_1', \mu_2'$ then
      i. If $(\mu_1', \mu_2') = (\mu_1'', \mu_2'') = \mathsf{RetrieveTags}(L'')$ and $\mathsf{isLess}(L', L'') = \mathsf{true}$ then return m'';
      ii. Else abort.
3. **Main step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{p_0,p_1\}}(\mu_3)$; if $\mathsf{out} = \mathsf{'fail'}$ then abort, else parse out as $(m'', \mu_1'', \mu_2'', L'')$;
   (b) If $(\mu_1, \mu_2) = (\mu_1'', \mu_2'') = \mathsf{RetrieveTags}(L'')$ then return m'';
   (c) Else abort.

Program $\mathsf{RFake}_{C,2}(\hat{m}, \mu_1, \mu_2, \mu_3; \rho)$

Inputs: fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3, random coins ρ.

Hardwired values: encryption key EK_R of receiver-fake ACE, punctured decryption key $\mathsf{DK}\{p_0, p_1\}$ of the main ACE, where $p_0 = (m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $p_1 = (m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, variables ${\mu_3}^*, L_0^*$.

1. If $\mu_3 = {\mu_3}^*$ then set $L'' = L_0^*$;
   else $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{p_0,p_1\}}(\mu_3)$; if $\mathsf{out} = \mathsf{'fail'}$ then abort, else parse out as $(m'', \mu_1'', \mu_2'', L'')$;
2. Return $r' \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_R}(\hat{m}, \mu_1, \mu_2, \mu_3, L'', \mathsf{prg}(\rho))$.

Figure 109: Programs $\mathsf{P2}_{C,2}, \mathsf{Dec}_{C,2}, \mathsf{RFake}_{C,2}$, used in the proof of lemma 56 (semantic security).

Programs $\mathsf{P2}_{C,3}, \mathsf{Dec}_{C,3}, \mathsf{RFake}_{C,3}$**.**

Program $\mathsf{P2}_{C,3}(r, \mu_1)$

Inputs: receiver randomness r, the first message μ_1 in the protocol.

Hardwired values: decryption key DK_R of receiver-fake ACE, key k_R of an extracting PRF RG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_R}(r)$; if $\mathsf{out} = \mathsf{'fail'}$ then goto main step, else parse out as $(m', \mu_1{}', \mu_2{}', \mu_3{}', L', \hat{\rho})$;
   (b) If $\mu_1 = \mu_1{}'$ then return $\mu_2{}'$;
2. **Main step:**
   (a) Return $\mu_2 \leftarrow \mathsf{RG}_{k_R}(r, \mu_1)$.

Program $\mathsf{Dec}_{C,3}(r, \mu_1, \mu_2, \mu_3)$

Inputs: receiver randomness r, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P2}_{C,3}$, isLess, $\mathsf{RetrieveTags}$; decryption key DK_R of receiver-fake ACE, punctured decryption key $\mathsf{DK}\{p_0\}$ of the main ACE, where $p_0 = (m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$.

1. **Validity check:** if $\mathsf{P2}_{C,3}(r, \mu_1) \neq \mu_2$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_R}(r)$; if $\mathsf{out}' = \mathsf{'fail'}$ then goto main step; else parse out' as $(m', \mu_1{}', \mu_2{}', \mu_3{}', L', \hat{\rho})$;
   (b) if $\mu_1, \mu_2, \mu_3 = \mu_1{}', \mu_2{}', \mu_3{}'$ then return m';
   (c) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{p_0\}}(\mu_3)$; if $\mathsf{out}'' = \mathsf{'fail'}$ then abort, else parse out'' as $(m'', \mu_1{}'', \mu_2{}'', L'')$;
   (d) If $\mu_1, \mu_2 = \mu_1{}', \mu_2{}'$ then
      i. If $(\mu_1{}', \mu_2{}') = (\mu_1{}'', \mu_2{}'') = \mathsf{RetrieveTags}(L'')$ and $\mathsf{isLess}(L', L'') = \mathsf{true}$ then return m'';
      ii. Else abort.
3. **Main step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{p_0\}}(\mu_3)$; if $\mathsf{out} = \mathsf{'fail'}$ then abort, else parse out as $(m'', \mu_1{}'', \mu_2{}'', L'')$;
   (b) If $(\mu_1, \mu_2) = (\mu_1{}'', \mu_2{}'') = \mathsf{RetrieveTags}(L'')$ then return m'';
   (c) Else abort.

Program $\mathsf{RFake}_{C,3}(\hat{m}, \mu_1, \mu_2, \mu_3; \rho)$

Inputs: fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3, random coins ρ.

Hardwired values: encryption key EK_R of receiver-fake ACE, punctured decryption key $\mathsf{DK}\{p_0\}$ of the main ACE, where $p_0 = (m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{p_0\}}(\mu_3)$; if $\mathsf{out} = \mathsf{'fail'}$ then abort, else parse out as $(m'', \mu_1{}'', \mu_2{}'', L'')$;
2. Return $r' \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_R}(\hat{m}, \mu_1, \mu_2, \mu_3, L'', \mathsf{prg}(\rho))$.

Figure 110: Programs $\mathsf{P2}_{C,3}, \mathsf{Dec}_{C,3}, \mathsf{RFake}_{C,3}$, used in the proof of lemma 56 (semantic security).

8.1.4 Proof of lemma 57 (Indistinguishability of levels)

- $\mathsf{Hyb}_{D,1,1}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, $\mu_1{}^* = \mathsf{SG}(s^*, m_1^*)$, $\mu_2{}^* = \mathsf{RG}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs of the sender can be found in fig. 111.

  Note that this distribution is exactly the distribution from Hyb_C, conditioned on the fact that s^*, r^* are outside of image of ACE.

- $\mathsf{Hyb}_{D,1,2}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{D,1}, \mathsf{P2}, \mathsf{P3}_{D,1}, \mathsf{Dec}, \mathsf{SFake}_{D,1}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, $\mu_1{}^* = \mathsf{SG}(s^*, m_1^*)$, $\mu_2{}^* = \mathsf{RG}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs of the sender can be found in fig. 112.

  That is, in program SFake we puncture encryption key EK_S of the sender-fake ACE at the set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. Indistinguishability holds by iO, since this modification doesn't change the functionality of SFake due to the fact that SFake never encrypts plaintexts with level ℓ_0^*.

- $\mathsf{Hyb}_{D,1,3}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{D,2}, \mathsf{P2}, \mathsf{P3}_{D,2}, \mathsf{Dec}, \mathsf{SFake}_{D,2}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, $\mu_1{}^* = \mathsf{SG}(s^*, m_1^*)$, $\mu_2{}^* = \mathsf{RG}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs of the sender can be found in fig. 113.

  That is, in programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$ we puncture decryption key DK_S of the sender-fake ACE at the same set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. Indistinguishability holds by security of constrained decryption of ACE, since the corresponding encryption key EK_S is already punctured at the same set.

- $\mathsf{Hyb}_{D,1,4}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{D,2}, \mathsf{P2}, \mathsf{P3}_{D,2}, \mathsf{Dec}, \mathsf{SFake}_{D,2}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, $\mu_1{}^*$ is chosen at random, $\mu_2{}^* = \mathsf{RG}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs of the sender can be found in fig. 113.

  That is, we choose $\mu_1{}^*$ at random instead of computing it as $\mu_1{}^* = \mathsf{SG}_{k_S}(s^*, m_1^*)$. Indistinguishability holds by the strong extracting property of the sender PRF SG (note that s^* was not used anywhere else in the distribution).

- $\mathsf{Hyb}_{D,2,1}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{D,3}, \mathsf{P2}, \mathsf{P3}_{D,3}, \mathsf{Dec}, \mathsf{SFake}_{D,3}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, $\mu_1{}^*$ is chosen at random, $\mu_2{}^* = \mathsf{RG}(r^*, \mu_1{}^*)$, $\mu_3{}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs of the sender can be found in fig. 114.

  That is, in programs $\mathsf{P3}$ and SFake we use punctured programs $\mathsf{GenZero}[\mu_1{}^*]$, $\mathsf{Transform}[\ell_0^*, \mu_2{}^*]$.

Indistinguishability holds by iO, since this doesn't change functionality of $\mathsf{P3}$ and SFake. Roughly, this is because of the following:

Since ${\mu_1}^*$ is random and outside of the image of a PRF SG, programs $\mathsf{P3}$ and SFake never call $\mathsf{GenZero}({\mu_1}^*)$ in the main step, and program $\mathsf{P3}$ never calls $\mathsf{Transform}(\ell_0^*, {\mu_2}^*)$ in the main step.

In order to call $\mathsf{Transform}(\ell_0^*, {\mu_2}^*)$ in trapdoor step, $\mathsf{P3}$ needs to take as input some fake s encoding level ℓ_0^*. However, due to the fact that DK_S is punctured at the set $P_{\ell_0^*}$ which contains all but one strings with ℓ_0^*, the only valid fake s with ℓ_0^* is s'. However, running $\mathsf{P3}$ on s' cannot result in calling $\mathsf{Transform}(\ell_0^*, {\mu_2}^*)$ in the trapdoor step: in order to hit the trapdoor step with s' and run $\mathsf{Transform}$ with $\mu_2 = {\mu_2}^*$, the input to $\mathsf{P3}$ should be $(s', m_0^*, {\mu_1}^*, {\mu_2}^*)$; however, in this case the program immediately outputs ${\mu_3}'$ without running $\mathsf{Transform}$.

- $\mathsf{Hyb}_{D,2,2}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{D,4}, \mathsf{P2}, \mathsf{P3}_{D,4}, \mathsf{Dec}, \mathsf{SFake}_{D,4}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs of the sender can be found in fig. 115.

  That is, we switch the single-tag level used in generation of s' from $\ell_0^* = [0, {\mu_1}^*]$ to $\ell_1^* = [1, {\mu_1}^*]$. Indistinguishability holds by security of level system: recall that it guarantees that ℓ_0^* is indistinguishable from ℓ_1^*, even given $L_0^* = [0, {\mu_1}^*, {\mu_2}^*]$ and punctured programs of the level system.

  Note that now keys $\mathsf{EK}_S, \mathsf{DK}_S$ of the sender-fake ACE become punctured at the set $P_{\ell_1^*} = \{(*, *, *, *, \ell_1^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$ instead of $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, and program $\mathsf{Transform}$ becomes punctured at the point $(\ell_1^*, {\mu_2}^*)$ instead of $(\ell_0^*, {\mu_2}^*)$.

- $\mathsf{Hyb}_{D,2,3}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{D,5}, \mathsf{P2}, \mathsf{P3}_{D,5}, \mathsf{Dec}, \mathsf{SFake}_{D,5}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs of the sender can be found in fig. 116.

  That is, in programs $\mathsf{P3}$ and SFake we use original programs $\mathsf{GenZero}$, $\mathsf{Transform}$ instead of punctured programs $\mathsf{GenZero}[{\mu_1}^*]$, $\mathsf{Transform}[\ell_1^*, {\mu_2}^*]$. Indistinguishability holds by iO, since this doesn't change functionality of $\mathsf{P3}$ and SFake. Roughly, this is because of similar reasoning as in Hyb_D, except for ℓ_1^* instead of ℓ_0^*:

  Since ${\mu_1}^*$ is random and outside of the image of a PRF SG, programs $\mathsf{P3}$ and SFake never call $\mathsf{GenZero}({\mu_1}^*)$ in the main step, and program $\mathsf{P3}$ never calls $\mathsf{Transform}(\ell_1^*, {\mu_2}^*)$ in the main step.

  In order to call $\mathsf{Transform}(\ell_1^*, {\mu_2}^*)$ in trapdoor step, $\mathsf{P3}$ needs to take as input some fake s encoding level ℓ_1^*. However, due to the fact that DK_S is punctured at the set $P_{\ell_1^*}$ which contains all but one strings with ℓ_1^*, the only valid fake s with ℓ_1^* is s'. However, running $\mathsf{P3}$ on s' cannot result in calling $\mathsf{Transform}(\ell_1^*, {\mu_2}^*)$ in the trapdoor step: in order to hit the trapdoor step with s' and run $\mathsf{Transform}$ with $\mu_2 = {\mu_2}^*$, the input to $\mathsf{P3}$ should be $(s', m_0^*, {\mu_1}^*, {\mu_2}^*)$; however, in this case the program immediately outputs ${\mu_3}'$ without running $\mathsf{Transform}$.

- $\mathsf{Hyb}_{D,3,1}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{D,6}, \mathsf{P2}, \mathsf{P3}_{D,6}, \mathsf{Dec}, \mathsf{SFake}_{D,6}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$,

$s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs of the sender can be found in fig. 117.

That is, in program SFake we additionally puncture encryption key EK_S of the sender-fake ACE at the set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$ (recall that it is already punctured at the set $P_{\ell_1^*} = \{(*, *, *, *, \ell_1^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$). Indistinguishability holds by security of iO, since this modification doesn't change the functionality of SFake due to the fact that SFake never encrypts plaintexts with level ℓ_0^*.

- $\mathsf{Hyb}_{D,3,2}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{D,7}, \mathsf{P2}, \mathsf{P3}_{D,7}, \mathsf{Dec}, \mathsf{SFake}_{D,7}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs of the sender can be found in fig. 118.

  That is, in programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$ we additionally puncture decryption key DK_S of the sender-fake ACE at the set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$ (recall that it is already punctured at the set $P_{\ell_1^*} = \{(*, *, *, *, \ell_1^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$). Indistinguishability holds by security of constrained decryption of ACE, since the corresponding encryption key EK is already punctured at $P_{\ell_0^*} \cup P_{\ell_1^*}$.

- $\mathsf{Hyb}_{D,3,3}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{D,8}, \mathsf{P2}, \mathsf{P3}_{D,8}, \mathsf{Dec}, \mathsf{SFake}_{D,8}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs of the sender can be found in fig. 119.

  That is, in programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$ we unpuncture decryption key DK_S of the sender-fake ACE at the set $P_{\ell_1^*} = \{(*, *, *, *, \ell_1^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$ (but this key still remains punctured at the set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$). Indistinguishability holds by security of constrained decryption of ACE, since the corresponding encryption key EK is already punctured at $P_{\ell_0^*} \cup P_{\ell_1^*}$.

- $\mathsf{Hyb}_{D,3,4}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{D,9}, \mathsf{P2}, \mathsf{P3}_{D,9}, \mathsf{Dec}, \mathsf{SFake}_{D,9}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs of the sender can be found in fig. 120.

  That is, in program SFake we unpuncture encryption key EK_S of the sender-fake ACE at the set $P_{\ell_1^*} = \{(*, *, *, *, \ell_1^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$ (but this key still remains punctured at the set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$). Indistinguishability holds by security of iO, since this doesn't change the functionality of SFake. Indeed, the program never needs to encrypt any plaintext containing ℓ_1^* because of the following. Since ${\mu_1}^*$ is random and outside of the image of a PRF SG, program SFake never calls $\mathsf{GenZero}({\mu_1}^*)$ in the main step and thus never needs to encrypt $\ell_1^* = \mathsf{Increment}(\mathsf{GenZero}({\mu_1}^*))$. In order to encrypt a plaintext containing ℓ_1^* in the trapdoor step, SFake needs to get as input fake s which contains ℓ_0^*. However, since DK_S is punctured at $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$, there do not exist valid fake s with ℓ_0^*, thus the program never needs to encrypt plaintexts with ℓ_1^*.

- $\mathsf{Hyb}_{D,3,5}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{D,10}, \mathsf{P2}, \mathsf{P3}_{D,10}, \mathsf{Dec}, \mathsf{SFake}_{D,10}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$,

$s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs of the sender can be found in fig. 121.

That is, in programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$ we unpuncture decryption key DK_S of the sender-fake ACE at the set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$. Indistinguishability holds by security of constrained decryption of ACE, since the corresponding encryption key EK is already punctured at $P_{\ell_0^*}$.

- $\mathsf{Hyb}_{D,3,6}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs of the sender can be found in fig. 122.

  That is, in program SFake we unpuncture encryption key EK_S of the sender-fake ACE at the set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$. Indistinguishability holds by security of iO, since this doesn't change the functionality of SFake since SFake never needs to encrypt plaintexts with ℓ_0^*.

- $\mathsf{Hyb}_{D,3,7}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^* is chosen at random, r^* is chosen at random, ${\mu_1}^* = \mathsf{SG}_{k_S}(s^*, m_1^*)$, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs of the sender can be found in fig. 122.

  That is, we compute ${\mu_1}^*$ as ${\mu_1}^* = \mathsf{SG}_{k_S}(s^*, m_1^*)$ instead of choosing it at random. Indistinguishability holds by the strong extracting property of the sender PRF SG (note that s^* is not used anywhere else in the distribution).

  Note that $\mathsf{Hyb}_{D,3,7} = \mathsf{Hyb}_E$.

Programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$**.**

Program $\mathsf{P1}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: decryption key DK_S of sender-fake ACE, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = '\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m = m'$ then return ${\mu_1}'$;
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms P1, GenZero, Transform, RetrieveTag; decryption key DK_S of sender-fake ACE, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = '\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', {\mu_1}', {\mu_2}'$ then return ${\mu_3}'$;
   (c) If $m, \mu_1 = m', {\mu_1}'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms P1, GenZero, Increment; encryption and decryption keys $\mathsf{EK}_S, \mathsf{DK}_S$ of sender-fake ACE.

1. **Validity check:** if $\mathsf{P1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = '\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1 = m', {\mu_1}'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = '\mathsf{fail}'$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 111: Programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$.

Programs $\mathsf{P1}_{D,1}, \mathsf{P3}_{D,1}, \mathsf{SFake}_{D,1}$.

Program $\mathsf{P1}_{D,1}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: decryption key DK_S of sender-fake ACE, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = '\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m = m'$ then return ${\mu_1}'$;
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}_{D,1}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,1}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; decryption key DK_S of sender-fake ACE, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}_{D,1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = '\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', {\mu_1}', {\mu_2}'$ then return ${\mu_3}'$;
   (c) If $m, \mu_1 = m', {\mu_1}'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}_{D,1}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,1}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; punctured encryption key $\mathsf{EK}_S\{P_{\ell_0^*}\}$ and decryption key DK_S of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$.

1. **Validity check:** if $\mathsf{P1}_{D,1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = '\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1 = m', {\mu_1}'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = '\mathsf{fail}'$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 112: Programs $\mathsf{P1}_{D,1}, \mathsf{P3}_{D,1}, \mathsf{SFake}_{D,1}$, used in the proof of lemma 57 (security of levels).

Programs $\mathsf{P1}_{D,2}, \mathsf{P3}_{D,2}, \mathsf{SFake}_{D,2}$.

Program $\mathsf{P1}_{D,2}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: punctured decryption key $\mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*,*,*,*,\ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m = m'$ then return ${\mu_1}'$;
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}_{D,2}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,2}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; punctured decryption key $\mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*,*,*,*,\ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}_{D,2}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', {\mu_1}', {\mu_2}'$ then return ${\mu_3}'$;
   (c) If $m, \mu_1 = m', {\mu_1}'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}_{D,2}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,2}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; punctured encryption and decryption keys $\mathsf{EK}_S\{P_{\ell_0^*}\}, \mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*,*,*,*,\ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$.

1. **Validity check:** if $\mathsf{P1}_{D,2}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1 = m', {\mu_1}'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = \mathsf{'fail'}$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 113: Programs $\mathsf{P1}_{D,2}, \mathsf{P3}_{D,2}, \mathsf{SFake}_{D,2}$, used in the proof of lemma 57 (security of levels).

Programs $\mathsf{P1}_{D,3}, \mathsf{P3}_{D,3}, \mathsf{SFake}_{D,3}$**.**

Program $\mathsf{P1}_{D,3}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: punctured decryption key $\mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', \mu_1{}', \mu_2{}', \mu_3{}', \ell')$;
   (b) If $m = m'$ then return $\mu_1{}'$;
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}_{D,3}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,3}$, punctured $\mathsf{GenZero}[\mu_1{}^*]$, punctured $\mathsf{Transform}[(\ell_0^*, \mu_2{}^*)]$, $\mathsf{RetrieveTag}$; punctured decryption key $\mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}_{D,3}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', \mu_1{}', \mu_2{}', \mu_3{}', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', \mu_1{}', \mu_2{}'$ then return $\mu_3{}'$;
   (c) If $m, \mu_1 = m', \mu_1{}'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}[(\ell_0^*, \mu_2{}^*)](\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}[(\ell_0^*, \mu_2{}^*)](\mathsf{GenZero}[\mu_1{}^*](\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}_{D,3}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,3}$, punctured $\mathsf{GenZero}[\mu_1{}^*]$, $\mathsf{Increment}$; punctured encryption and decryption keys $\mathsf{EK}_S\{P_{\ell_0^*}\}, \mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$.

1. **Validity check:** if $\mathsf{P1}_{D,3}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', \mu_1{}', \mu_2{}', \mu_3{}', \ell')$;
   (b) If $m, \mu_1 = m', \mu_1{}'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = \mathsf{'fail'}$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}[\mu_1{}^*](\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 114: Programs $\mathsf{P1}_{D,3}, \mathsf{P3}_{D,3}, \mathsf{SFake}_{D,3}$, used in the proof of lemma 57 (security of levels).

Programs $\mathsf{P1}_{D,4}, \mathsf{P3}_{D,4}, \mathsf{SFake}_{D,4}$.

Program $\mathsf{P1}_{D,4}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: punctured decryption key $\mathsf{DK}_S\{P_{\ell_1^*}\}$ of sender-fake ACE, where $P_{\ell_1^*} = \{(*, *, *, *, \ell_1^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_1^*)$, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_1^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', \mu_1{}', \mu_2{}', \mu_3{}', \ell')$;
   (b) If $m = m'$ then return $\mu_1{}'$;
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}_{D,4}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,4}$, punctured $\mathsf{GenZero}[\mu_1{}^*]$, punctured $\mathsf{Transform}[(\ell_1^*, \mu_2{}^*)]$, $\mathsf{RetrieveTag}$; punctured decryption key $\mathsf{DK}_S\{P_{\ell_1^*}\}$ of sender-fake ACE, where $P_{\ell_1^*} = \{(*, *, *, *, \ell_1^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_1^*)$, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}_{D,4}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_1^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', \mu_1{}', \mu_2{}', \mu_3{}', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', \mu_1{}', \mu_2{}'$ then return $\mu_3{}'$;
   (c) If $m, \mu_1 = m', \mu_1{}'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}[(\ell_1^*, \mu_2{}^*)](\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}[(\ell_1^*, \mu_2{}^*)](\mathsf{GenZero}[\mu_1{}^*](\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}_{D,4}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,4}$, punctured $\mathsf{GenZero}[\mu_1{}^*]$, $\mathsf{Increment}$; punctured encryption and decryption keys $\mathsf{EK}_S\{P_{\ell_1^*}\}, \mathsf{DK}_S\{P_{\ell_1^*}\}$ of sender-fake ACE, where $P_{\ell_1^*} = \{(*, *, *, *, \ell_1^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_1^*)$.

1. **Validity check:** if $\mathsf{P1}_{D,4}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_1^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', \mu_1{}', \mu_2{}', \mu_3{}', \ell')$;
   (b) If $m, \mu_1 = m', \mu_1{}'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = \mathsf{'fail'}$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_1^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}[\mu_1{}^*](\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_1^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 115: Programs $\mathsf{P1}_{D,4}, \mathsf{P3}_{D,4}, \mathsf{SFake}_{D,4}$, used in the proof of lemma 57 (security of levels).

Programs $\mathsf{P1}_{D,5}, \mathsf{P3}_{D,5}, \mathsf{SFake}_{D,5}$.

Program $\mathsf{P1}_{D,5}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: punctured decryption key $\mathsf{DK}_S\{P_{\ell_1^*}\}$ of sender-fake ACE, where $P_{\ell_1^*} = \{(*, *, *, *, \ell_1^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_1^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m = m'$ then return ${\mu_1}'$;
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}_{D,5}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,5}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; punctured decryption key $\mathsf{DK}_S\{P_{\ell_1^*}\}$ of sender-fake ACE, where $P_{\ell_1^*} = \{(*, *, *, *, \ell_1^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}_{D,5}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_1^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', {\mu_1}', {\mu_2}'$ then return ${\mu_3}'$;
   (c) If $m, \mu_1 = m', {\mu_1}'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}_{D,5}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,5}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; punctured encryption and decryption keys $\mathsf{EK}_S\{P_{\ell_1^*}\}, \mathsf{DK}_S\{P_{\ell_1^*}\}$ of sender-fake ACE, where $P_{\ell_1^*} = \{(*, *, *, *, \ell_1^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$.

1. **Validity check:** if $\mathsf{P1}_{D,5}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_1^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1 = m', {\mu_1}'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = \mathsf{'fail'}$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_1^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_1^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 116: Programs $\mathsf{P1}_{D,5}, \mathsf{P3}_{D,5}, \mathsf{SFake}_{D,5}$, used in the proof of lemma 57 (security of levels).

Programs $\mathsf{P1}_{D,6}, \mathsf{P3}_{D,6}, \mathsf{SFake}_{D,6}$**.**

Program $\mathsf{P1}_{D,6}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: punctured decryption key $\mathsf{DK}_S\{P_{\ell_1^*}\}$ of sender-fake ACE, where $P_{\ell_1^*} = \{(*, *, *, *, \ell_1^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_1^*}\}}(s)$; if $\mathsf{out} = {}'\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m = m'$ then return ${\mu_1}'$;
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}_{D,6}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,6}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; punctured decryption key $\mathsf{DK}_S\{P_{\ell_1^*}\}$ of sender-fake ACE, where $P_{\ell_1^*} = \{(*, *, *, *, \ell_1^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}_{D,6}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_1^*}\}}(s)$; if $\mathsf{out} = {}'\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', {\mu_1}', {\mu_2}'$ then return ${\mu_3}'$;
   (c) If $m, \mu_1 = m', {\mu_1}'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}_{D,6}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,6}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; punctured encryption and decryption keys $\mathsf{EK}_S\{P_{\ell_1^*} \cup P_{\ell_0^*}\}, \mathsf{DK}_S\{P_{\ell_1^*}\}$ of sender-fake ACE, where $P_{\ell_1^*} = \{(*, *, *, *, \ell_1^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$.

1. **Validity check:** if $\mathsf{P1}_{D,6}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_1^*}\}}(s)$; if $\mathsf{out} = {}'\mathsf{fail}'$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1 = m', {\mu_1}'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = {}'\mathsf{fail}'$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_1^*} \cup P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_1^*} \cup P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 117: Programs $\mathsf{P1}_{D,6}, \mathsf{P3}_{D,6}, \mathsf{SFake}_{D,6}$, used in the proof of lemma 57 (security of levels).

Programs $\mathsf{P1}_{D,7}, \mathsf{P3}_{D,7}, \mathsf{SFake}_{D,7}$.

Program $\mathsf{P1}_{D,7}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: punctured decryption key $\mathsf{DK}_S\{P_{\ell_1^*} \cup P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_1^*} = \{(*, *, *, *, \ell_1^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_1^*} \cup P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m = m'$ then return ${\mu_1}'$;
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}_{D,7}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,7}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; punctured decryption key $\mathsf{DK}_S\{P_{\ell_1^*} \cup P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_1^*} = \{(*, *, *, *, \ell_1^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}_{D,7}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_1^*} \cup P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', {\mu_1}', {\mu_2}'$ then return ${\mu_3}'$;
   (c) If $m, \mu_1 = m', {\mu_1}'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}_{D,7}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,7}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; punctured encryption and decryption keys $\mathsf{EK}_S\{P_{\ell_1^*} \cup P_{\ell_0^*}\}$, $\mathsf{DK}_S\{P_{\ell_1^*} \cup P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_1^*} = \{(*, *, *, *, \ell_1^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$.

1. **Validity check:** if $\mathsf{P1}_{D,7}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_1^*} \cup P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1 = m', {\mu_1}'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = \mathsf{'fail'}$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_1^*} \cup P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_1^*} \cup P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 118: Programs $\mathsf{P1}_{D,7}, \mathsf{P3}_{D,7}, \mathsf{SFake}_{D,7}$, used in the proof of lemma 57 (security of levels).

Programs $\mathsf{P1}_{D,8}, \mathsf{P3}_{D,8}, \mathsf{SFake}_{D,8}$.

Program $\mathsf{P1}_{D,8}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: punctured decryption key $\mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = \text{'fail'}$ goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m = m'$ then return μ_1';
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}_{D,8}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,8}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; punctured decryption key $\mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}_{D,8}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = \text{'fail'}$ goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', \mu_1', \mu_2'$ then return μ_3';
   (c) If $m, \mu_1 = m', \mu_1'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}_{D,8}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,8}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; punctured encryption and decryption keys $\mathsf{EK}_S\{P_{\ell_1^*} \cup P_{\ell_0^*}\}, \mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_1^*} = \{(*, *, *, *, \ell_1^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_1^*)$, $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$.

1. **Validity check:** if $\mathsf{P1}_{D,8}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = \text{'fail'}$ goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m, \mu_1 = m', \mu_1'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = \text{'fail'}$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_1^*} \cup P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_1^*} \cup P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 119: Programs $\mathsf{P1}_{D,8}, \mathsf{P3}_{D,8}, \mathsf{SFake}_{D,8}$, used in the proof of lemma 57 (security of levels).

Programs $\mathsf{P1}_{D,9}, \mathsf{P3}_{D,9}, \mathsf{SFake}_{D,9}$.

Program $\mathsf{P1}_{D,9}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: punctured decryption key $\mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) out $\leftarrow$ $\mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if out $=$ $'\mathsf{fail}'$ goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m = m'$ then return μ_1';
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}_{D,9}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,9}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; punctured decryption key $\mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}_{D,9}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) out $\leftarrow$ $\mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if out $=$ $'\mathsf{fail}'$ goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', \mu_1', \mu_2'$ then return μ_3';
   (c) If $m, \mu_1 = m', \mu_1'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}_{D,9}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,9}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; punctured encryption and decryption keys $\mathsf{EK}_S\{P_{\ell_0^*}\}$, $\mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$.

1. **Validity check:** if $\mathsf{P1}_{D,9}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) out $\leftarrow$ $\mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if out $=$ $'\mathsf{fail}'$ goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m, \mu_1 = m', \mu_1'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = '\mathsf{fail}'$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 120: Programs $\mathsf{P1}_{D,9}, \mathsf{P3}_{D,9}, \mathsf{SFake}_{D,9}$, used in the proof of lemma 57 (security of levels).

Programs $\mathsf{P1}_{D,10}, \mathsf{P3}_{D,10}, \mathsf{SFake}_{D,10}$.

Program $\mathsf{P1}_{D,10}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: decryption key DK_S of sender-fake ACE, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = '\mathsf{fail}'$ goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m = m'$ then return μ_1';
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}_{D,10}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,10}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; decryption key DK_S of sender-fake ACE, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}_{D,10}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = '\mathsf{fail}'$ goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', \mu_1', \mu_2'$ then return μ_3';
   (c) If $m, \mu_1 = m', \mu_1'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}_{D,10}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{D,10}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; punctured encryption key $\mathsf{EK}_S\{P_{\ell_0^*}\}$ and decryption key DK_S of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$.

1. **Validity check:** if $\mathsf{P1}_{D,10}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = '\mathsf{fail}'$ goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m, \mu_1 = m', \mu_1'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = '\mathsf{fail}'$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 121: Programs $\mathsf{P1}_{D,10}, \mathsf{P3}_{D,10}, \mathsf{SFake}_{D,10}$, used in the proof of lemma 57 (security of levels).

Programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$.

Program $\mathsf{P1}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: decryption key DK_S of sender-fake ACE, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = \text{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m = m'$ then return ${\mu_1}'$;
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; decryption key DK_S of sender-fake ACE, encryption key EK of main ACE.

1. **Validity check:** if $\mathsf{P1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = \text{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', {\mu_1}', {\mu_2}'$ then return ${\mu_3}'$;
   (c) If $m, \mu_1 = m', {\mu_1}'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; encryption and decryption keys $\mathsf{EK}_S, \mathsf{DK}_S$ of sender-fake ACE.

1. **Validity check:** if $\mathsf{P1}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S}(s)$; if $\mathsf{out} = \text{'fail'}$ goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', \ell')$;
   (b) If $m, \mu_1 = m', {\mu_1}'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = \text{'fail'}$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 122: Programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$.

8.2 Detailed proof of security

In this section we present formal security reductions for each hybrid described in section 8.1.

We denote by σ' the maximum size of programs of deniable encryption in the construction and the proof. Since our construction uses multiple layers of obfuscation, σ' is some polynomial of λ. As we note in appendix B, we could instead use only one layer of obfuscation, and the resulting code would have size $\sigma = O(\lambda^3)$.

8.2.1 Reductions in the proof of lemma 54 (Indistinguishability of explanation of the sender)

Let $t(\lambda)$ be any function in $\Omega(\mathsf{poly}(\lambda))$, and let $\varepsilon(\lambda)$ be a negligible function in $w(2^{-\lambda})$. Assuming the sender-fake relaxed ACE, sparse extracting puncturable PRF, and iO for program size σ' is $(t(\lambda), \varepsilon(\lambda))$-secure, we show that no time-$t(\lambda)$ adversary can distinguish between Hyb_A and Hyb_B with more than $O(\varepsilon(\lambda))$ advantage.

Note that conditioning on s^* begin not in the image of ACE incurs only $2^{-\lambda}$ loss and therefore we omit it.

Lemma 58. *Statistical distance between distributions* $\mathsf{Hyb}_A, \mathsf{Hyb}_{A,1}$ *is at most* $2^{-\lambda}$.

Proof. Since randomly chosen s^* is a valid ciphertext of sender ACE with probability at most $2^{-\lambda}$, with all but this probability both P1 and P3 will fail do decrypt s^* under DK_S and therefore will run main step, outputting ${\mu_1}^* = \mathsf{SG}_{k_S}(s^*, m_0^*)$ and ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, respectively. □

Lemma 59. *Assume s^* is outside of the image of sender ACE. Then, if there exists an adversary which can distinguish* $\mathsf{Hyb}_{A,1}$ *and* $\mathsf{Hyb}_{A,2}$ *in time $t(\lambda)$ with advantage $\varepsilon(\lambda)$, then there exists an adversary which can break* iO *for programs of size σ' in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\frac{1}{3} \cdot \varepsilon(\lambda)$.*

Proof. Below we analyze all three pairs of programs assuming that s^* is outside the image of sender ACE, and thus $\mathsf{ACE.Dec}_{\mathsf{DK}_S}(s^*) = \mathsf{'fail'}$. We show that programs have the same functionality. We use the fact that all underlying primitives satisfy correctness.

Program P1. We present case analysis to show that the behavior of programs P1 and $\mathsf{P1}_{A,1}$ on each input is the same:

- Case $s = s^*$:
  - Case $m = m_0^*$: P1 outputs ${\mu_1}^*$ via main step since s^* is outside of image of ACE. $\mathsf{P1}_{A,1}$ outputs ${\mu_1}^*$ due to hardwired instruction.
  - Case $m \neq m_0^*$: P1 executes main step and outputs $\mathsf{SG}_{k_S}(s^*, m)$ since s^* is outside of image of ACE. $\mathsf{P1}_{A,1}$ executes main step and outputs $\mathsf{SG}_{k_S}(s^*, m)$ due to hardwired instruction.
- Case $s = s'$:
  - Case $m = m_0^*$: P1 outputs ${\mu_1}^*$ via trapdoor step. $\mathsf{P1}_{A,1}$ outputs ${\mu_1}^*$ due to hardwired instruction.
  - Case $m \neq m_0^*$: P1 skips the trapdoor step since s' contains the wrong $m_0^* \neq m$, and outputs $\mathsf{SG}_{k_S}(s', m)$. $\mathsf{P1}_{A,1}$ executes main step and output $\mathsf{SG}_{k_S}(s', m)$ due to hardwired instruction.

- Case $s \neq s', s^*$: $\mathsf{P1}$ and $\mathsf{P1}_{A,1}$ execute the same code, since punctured keys preserve functionality on all inputs which are not punctured (note that when $s \neq s', s^*$ keys are indeed never used at punctured points).

Program P3. Next we compare programs $\mathsf{P3}$ and $\mathsf{P3}_{A,1}$. Note that validity check passes on the same set of inputs in programs $\mathsf{P3}$ and $\mathsf{P3}_{A,1}$, since programs $\mathsf{P1}$ and $\mathsf{P1}_{A,1}$ are functionally equivalent. We present the analysis assuming inputs passed the validity check.

- Case $s = s^*$:
  - Case $(m, \mu_1) = (m_0^*, {\mu_1}^*)$:
    - Case $\mu_2 = {\mu_2}^*$: $\mathsf{P3}$ outputs ${\mu_3}^*$ via main step since s^* is outside of image of ACE. $\mathsf{P3}_{A,1}$ outputs ${\mu_3}^*$ due to hardwired instruction.
    - Case $\mu_2 \neq {\mu_2}^*$: $\mathsf{P3}$ outputs $\mathsf{Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, \mu_2, \mathsf{Transform}(\mathsf{GenZero}({\mu_1}^*), \mu_2))$ via main step since s^* is outside of image of ACE. $\mathsf{P3}_{A,1}$ outputs $\mathsf{Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, \mu_2, \mathsf{Transform}(\ell_0^*, \mu_2))$ due to hardwired instruction. Note that $\mathsf{GenZero}({\mu_1}^*) = \ell_0^*$ and thus both outputs are the same.
  - Case $(m, \mu_1) \neq (m_0^*, {\mu_1}^*)$: $\mathsf{P3}$ executes main step and outputs $\mathsf{Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2))$ since s^* is outside of image of ACE. $\mathsf{P3}_{A,1}$ executes main step due to hardwired instruction and outputs $\mathsf{Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2))$.
- Case $s = s'$:
  - Case $(m, \mu_1) = (m_0^*, {\mu_1}^*)$:
    - Case $\mu_2 = {\mu_2}^*$: $\mathsf{P3}$ outputs ${\mu_1}^*$ via trapdoor step. $\mathsf{P3}_{A,1}$ outputs ${\mu_1}^*$ due to hardwired instruction.
    - Case $\mu_2 \neq {\mu_2}^*$: $\mathsf{P3}$ gets level ℓ_0^* from s' and outputs $\mathsf{Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, \mu_2, \mathsf{Transform}(\ell_0^*, \mu_2))$ via trapdoor step. $\mathsf{P3}_{A,1}$ outputs $\mathsf{Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, \mu_2, \mathsf{Transform}(\ell_0^*, \mu_2))$ due to hardwired instruction.
  - Case $(m, \mu_1) \neq (m_0^*, {\mu_1}^*)$: $\mathsf{P3}$ skips the trapdoor step since s' contains the wrong $(m_0^*, {\mu_1}^*) \neq (m, \mu_1)$, and outputs $\mathsf{Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2))$ via main step. $\mathsf{P3}_{A,1}$ executes main step due to hardwired instruction and outputs $\mathsf{Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2))$.
- Case $s \neq s', s^*$: $\mathsf{P3}$ and $\mathsf{P3}_{A,1}$ execute the same code, since punctured keys preserve functionality on all inputs which are not punctured. Note that in this case these keys are never used at punctured points.

Program SFake. Next we compare programs SFake and $\mathsf{SFake}_{A,1}$. Note that validity check passes on the same set of inputs in programs SFake and $\mathsf{SFake}_{A,1}$, since programs $\mathsf{P1}$ and $\mathsf{P1}_{A,1}$ are functionally equivalent. We present the analysis assuming inputs passed the validity check.

- Case $s = s^*$:
  - Case $(m, \mu_1) = (m_0^*, {\mu_1}^*)$ (for arbitrary $(\hat{m}, \mu_2, \mu_3)$): SFake outputs $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(\hat{m}, {\mu_1}^*, \mu_2, \mu_3, \mathsf{Increment}(\mathsf{GenZero}({\mu_1}^*)))$ via main step since s^* is outside of image of ACE. $\mathsf{SFake}_{A,1}$ outputs $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(\hat{m}, {\mu_1}^*, \mu_2, \mu_3, \mathsf{Increment}(\ell_0^*))$ due to

hardwired instruction. Note that $\mathsf{GenZero}(\mu_1{}^*) = \ell_0^*$ and thus both outputs are the same.

– Case $(m, \mu_1) \neq (m_0^*, \mu_1{}^*)$ (for arbitrary $(\hat{m}, \mu_2, \mu_3)$): SFake executes main step since s^* is outside of image of ACE and outputs $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \mathsf{Increment}(\mathsf{GenZero}(\mu_1)))$. $\mathsf{SFake}_{A,1}$ skips the trapdoor step due to hardwired instruction and outputs the same value via main step.

- Case $s = s'$:

  – Case $(m, \mu_1) = (m_0^*, \mu_1{}^*)$ (for arbitrary $(\hat{m}, \mu_2, \mu_3)$): SFake gets ℓ_0^* from s', increments it and outputs $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(\hat{m}, \mu_1{}^*, \mu_2, \mu_3, \mathsf{Increment}(\ell_0^*))$. $\mathsf{SFake}_{A,1}$ outputs the same value due to hardwired instruction.

  – Case $(m, \mu_1) \neq (m_0^*, \mu_1{}^*)$ (for arbitrary $(\hat{m}, \mu_2, \mu_3)$): SFake skips the trapdoor step since s' contains the wrong $(m_0^*, \mu_1{}^*) \neq (m, \mu_1)$, and outputs $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \mathsf{Increment}(\mathsf{GenZero}(\mu_1)))$ via main step. $\mathsf{SFake}_{A,1}$ skips the trapdoor step due to hardwired instruction and outputs $\mathsf{Enc}_{\mathsf{EK}}(m, \mu_1, \mu_2, \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2))$ via main step.

- Case $s \neq s', s^*$: SFake and $\mathsf{SFake}_{A,1}$ execute the same code, since punctured keys preserve functionality on all inputs which are not punctured. Note that keys are never used at punctured points (in particular, the program never needs to encrypt a plaintext containing ℓ_0^*, and thus the key can be punctured at $S_{\ell_0^*} = \{*, *, *, *, \ell_0^*\}$).

□

Lemma 60. *Assume s^* is outside of the image of sender ACE. Then, if there exists an adversary which can distinguish $\mathsf{Hyb}_{A,2}$ and $\mathsf{Hyb}_{A,3}$ in time $t(\lambda)$ with advantage $\varepsilon(\lambda)$, then there exists an adversary which can break security of a puncturable PRF SG_{k_S} in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. We give a reduction from indistinguishability of hybrids $\mathsf{Hyb}_{A,2}$ and $\mathsf{Hyb}_{A,3}$ to security of a puncturable PRF SG_{k_S} at the punctured point (s^*, m_0^*).

The reduction first takes plaintexts m_0^*, m_1^* from the adversary. Next it chooses random s^* and sends the point (s^*, m_0^*) to the challenger of puncturable PRF game. The reduction gets back from the challenger the punctured key $k_S\{(s^*, m_0^*)\}$ and the value $\mu_1{}^*$, which is either $\mathsf{SG}_{k_S}(s^*, m_0^*)$ or randomly chosen.

Next the reduction reconstructs the rest of the distribution as follows. It samples all keys used in programs (except $k_S\{(s^*, m_0^*)\}$), namely keys EK, DK of the main ACE, keys $\mathsf{EK}_S, \mathsf{DK}_S$ of the sender ACE, keys $\mathsf{EK}_R, \mathsf{DK}_R$ of the receiver ACE, key k_R of the sparse extracting PRF RG of the receiver. It also runs setup of the level system to create the code of $\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}$.

It chooses random r^* and sets $\mu_2{}^* = \mathsf{RG}_{k_R}(r^*, \mu_1{}^*)$. It computes levels $\ell_0^* = \mathsf{GenZero}(\mu_1{}^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, \mu_2{}^*)$. It sets $\mu_3{}^* = \mathsf{Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$ and $s' = \mathsf{Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$.

Next it computes punctured keys $\mathsf{DK}_S\{s^*, s'\}$, $k_S\{(s^*, m_0^*), (s', m_0^*)\}$ (by additionally puncturing challenge $k_S\{(s^*, m_0^*)\}$ at (s', m_0^*)), and $\mathsf{EK}_S\{S_{\ell_0^*}\}$, $S_{\ell_0^*} = \{*, *, *, *, \ell_0^*\}$.

Then the reduction uses variables and code created above to construct and obfuscate programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$, (fig. 95) and $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$ (fig. 19). It gives obfuscated programs to the adversary, together with $s^*, r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*$. If challenge $\mu_1{}^*$ was $\mathsf{SG}_{k_S}(s^*, m_0^*)$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{A,2}$. If $\mu_1{}^*$ was randomly chosen, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{A,3}$.

Note that this reduction is using the fact that an adversary who holds the punctured key can additionally puncture it at another point. We note that the construction of an extracting PRF [SW14] is based on GGM PRF and satisfies this property. □

Lemma 61. *Assume s^* is outside of the image of sender ACE. Then, if there exists an adversary which can distinguish $\mathsf{Hyb}_{A,3}$ and $\mathsf{Hyb}_{A,4}$ in time $t(\lambda)$ with advantage $\varepsilon(\lambda)$, then there exists an adversary which can break symmetry of a sender-fake relaxed ACE scheme in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. We give a reduction from indistinguishability of hybrids $\mathsf{Hyb}_{A,3}$ and $\mathsf{Hyb}_{A,4}$ to symmetry of sender ACE.

The reduction first takes plaintexts m_0^*, m_1^* from the adversary. Next it samples all keys used in programs (except $\mathsf{EK}_S, \mathsf{DK}_S$), namely keys EK, DK of the main ACE, keys $\mathsf{EK}_R, \mathsf{DK}_R$ of the receiver ACE, key k_S of the sparse extracting PRF SG of the sender, key k_R of the sparse extracting PRF RG of the receiver. It also runs setup of the level system to create the code of $\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}$.

It chooses random r^* and sets $\mu_1{}^*$ to be randomly chosen, $\mu_2{}^* = \mathsf{RG}_{k_R}(r^*, \mu_1{}^*)$. It computes levels $\ell_0^* = \mathsf{GenZero}(\mu_1{}^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, \mu_2{}^*)$. It sets $\mu_3{}^* = \mathsf{Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$.

Next the reduction sends $p = (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$ as the challenge point to the challenger of the symmetry of ACE. The challenger chooses random s^*, samples keys $\mathsf{EK}_S, \mathsf{DK}_S$ of ACE and computes $s' = \mathsf{Enc}_{\mathsf{EK}_S}(p)$, and punctures EK_S at $S_{\ell_0^*} = \{*, *, *, *, \ell_0^*\}$ and DK_S at s^*, s' (DK_S is first punctured at one of the strings s^*, s' which is lexicographically smaller, and then at the other). The reduction gets back from the challenger $(s_1, s_2, \mathsf{EK}_S\{S_{\ell_0^*}\}, \mathsf{DK}_S\{s^*, s'\})$, where $s_1 = s^*, s_2 = s'$ or $s_1 = s', s_2 = s^*$.

Next the reduction computes punctured key $k_S\{(s_1, m_0^*), (s_2, m_0^*)\}$. Then it uses variables and code created above to construct and obfuscate programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$, (fig. 95) and $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$ (fig. 19). In particular, in every place where s^*, s' appear, e.g. in code of programs, or as a punctured point, the reduction first uses one of the strings s_1, s_2 which is lexicographically smaller, and then the other (note that s^*, s' always appear together in the distribution, except for the value given to the adversary as randomness of the sender).

Next the reduction gives obfuscated programs to the adversary, together with $s_1, r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*$. If challenge s_1, s_2 are s^*, s', then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{A,3}$. If s_1, s_2 are s', s^*, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{A,4}$. □

Lemma 62. *Assume s^* is outside of the image of sender ACE. Then, if there exists an adversary which can distinguish $\mathsf{Hyb}_{A,4}$ and $\mathsf{Hyb}_{A,5}$ in time $t(\lambda)$ with advantage $\varepsilon(\lambda)$, then there exists an adversary which can break security of a puncturable PRF SG_{k_S} in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is very similar to the proof of indistinguishability of hybrids $\mathsf{Hyb}_{A,2}, \mathsf{Hyb}_{A,3}$, except that the reduction gives s' instead of s^* as randomness of the sender to the adversary.

We give a reduction from indistinguishability of hybrids $\mathsf{Hyb}_{A,4}$ and $\mathsf{Hyb}_{A,5}$ to security of a puncturable PRF SG_{k_S} at the punctured point (s^*, m_0^*).

The reduction first takes plaintexts m_0^*, m_1^* from the adversary. Next it chooses random s^* and sends the point (s^*, m_0^*) to the challenger of puncturable PRF game. The reduction gets back from the challenger the punctured key $k_S\{(s^*, m_0^*)\}$ and the value $\mu_1{}^*$, which is either $\mathsf{SG}_{k_S}(s^*, m_0^*)$ or randomly chosen.

Next the reduction reconstructs the rest of the distribution as follows. It samples all keys used in programs (except $k_S\{(s^*, m_0^*)\}$), namely keys EK, DK of the main ACE, keys $\mathsf{EK}_S, \mathsf{DK}_S$ of the sender ACE, keys $\mathsf{EK}_R, \mathsf{DK}_R$ of the receiver ACE, key k_R of the sparse extracting PRF RG of the receiver. It also runs setup of the level system to create the code of $\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}$.

It chooses random r^* and sets $\mu_2{}^* = \mathsf{RG}_{k_R}(r^*, \mu_1{}^*)$. It computes levels $\ell_0^* = \mathsf{GenZero}(\mu_1{}^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, \mu_2{}^*)$. It sets $\mu_3{}^* = \mathsf{Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$ and $s' = \mathsf{Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$.

Next it computes punctured keys $\mathsf{DK}_S\{s^*, s'\}$, $k_S\{(s^*, m_0^*), (s', m_0^*)\}$ (by additionally puncturing challenge $k_S\{(s^*, m_0^*)\}$ at (s', m_0^*)), and $\mathsf{EK}_S\{S_{\ell_0^*}\}$, $S_{\ell_0^*} = \{*, *, *, *, \ell_0^*\}$

Then the reduction uses variables and code created above to construct and obfuscate programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$, (fig. 95) and $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$ (fig. 19). It gives obfuscated programs to the adversary, together with $s', r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*$. If challenge $\mu_1{}^*$ was $\mathsf{SG}_{k_S}(s^*, m_0^*)$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{A,5}$. If $\mu_1{}^*$ was randomly chosen, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{A,4}$.

Note that this reduction is using the fact that an adversary who holds the punctured key can additionally puncture it at another point. We note that the construction of an extracting PRF [SW14] is based on GGM PRF and satisfies this property. □

Lemma 63. *Assume s^* is outside of the image of sender ACE. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{A,5}$ and $\mathsf{Hyb}_{A,6}$, then there exists an adversary which can break iO for programs of size σ' in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\frac{1}{3} \cdot \varepsilon(\lambda)$.*

Proof. The proof is identical to the proof of lemma 59, except that we give s', and not s^*, as randomness of the sender to the adversary. □

Finally, we note that the distributions in $\mathsf{Hyb}_{A,6}$ and Hyb_B are $2^{-\lambda}$-close (the reasoning is similar to distributions $\mathsf{Hyb}_A, \mathsf{Hyb}_{A,1}$).

8.2.2 Reductions in the proof of lemma 55 (Indistinguishability of explanation of the receiver)

Let $t(\lambda)$ be any function in $\Omega(\mathsf{poly}(\lambda))$, and let $\varepsilon(\lambda)$ be a negligible function in $w(2^{-\lambda})$. Assuming the prg, the sender-fake relaxed ACE, receiver-fake relaxed ACE, main ACE, sparse extracting puncturable PRF, and iO for program size σ' are $(t(\lambda), \varepsilon(\lambda))$-secure, we show that no time-$t(\lambda)$ adversary can distinguish between Hyb_B and Hyb_C with more than $O(\varepsilon(\lambda)) + 2^{-\tau(\lambda)}$ advantage.

(Note that security loss $2^{-\tau(\lambda)}$ comes from conditioning on the fact that $\mu_1{}^*$ is outside of the image of the PRF SG. Conditioning on $s^*, r^*, \hat{\rho}^*$ incurs only $2^{-\lambda}$ loss and therefore we omit it.).

Lemma 64. *Statistical distance between distributions $\mathsf{Hyb}_B, \mathsf{Hyb}_{B,1,1}$ is at most $2 \cdot 2^{-\lambda}$.*

Proof. Since randomly chosen s^* is a valid ciphertext of sender ACE with probability $\mathsf{senderACE.sparsity}(\lambda)$, with all but this probability both $\mathsf{P1}$ and $\mathsf{P3}$ will fail do decrypt s^* under DK_S and therefore will run main step, outputting $\mu_1{}^* = \mathsf{SG}_{k_S}(s^*, m_0^*)$ and $\mu_3{}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, respectively.

Similarly, randomly chosen r^* is a valid ciphertext of receiver ACE with probability $\text{receiverACE.sparsity}(\lambda)$, and thus with all but this probability P2 will fail do decrypt r^* under DK_R and therefore will run main step, outputting $\mu_2{}^* = \mathsf{RG}_{k_R}(r^*, \mu_1{}^*)$. □

Lemma 65. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{B,1,1}$ and $\mathsf{Hyb}_{B,1,2}$, then there exists an adversary which can break* iO *for programs of size σ' in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The only difference between programs SFake and $\mathsf{SFake}_{B,1}$ is that $\mathsf{SFake}_{B,1}$ uses a punctured key $\mathsf{EK}_S\{P_{\ell_0^*}\}$, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. This is without changing functionality, since SFake never needs to encrypt a plaintext with level ℓ_0^*, since $\ell_0^* = [0, \mu_1{}^*]$ and SFake encrypts levels with value at least 1. □

Lemma 66. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{B,1,2}$ and $\mathsf{Hyb}_{B,1,3}$, then there exists an adversary which can break security of constrained decryption of sender-fake relaxed ACE in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. We give a reduction from indistinguishability of hybrids $\mathsf{Hyb}_{B,1,2}$ and $\mathsf{Hyb}_{B,1,3}$ to security of constrained decryption of sender ACE.

The reduction first takes plaintexts m_0^*, m_1^* from the adversary. It samples all keys used in programs (except $\mathsf{EK}_S, \mathsf{DK}_S$), namely keys EK, DK of the main ACE, keys $\mathsf{EK}_R, \mathsf{DK}_R$ of the receiver ACE, key k_S of the sparse extracting PRF SG of the sender, key k_R of the sparse extracting PRF RG of the receiver. It also runs setup of the level system to create the code of GenZero, Increment, Transform, isLess, RetrieveTag, RetrieveTags.

It chooses random s^*, r^* and sets $\mu_1{}^* = \mathsf{SG}_{k_S}(s^*, m_0^*)$, $\mu_2{}^* = \mathsf{RG}_{k_R}(r^*, \mu_1{}^*)$. It computes levels $\ell_0^* = \mathsf{GenZero}(\mu_1{}^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, \mu_2{}^*)$. It sets $\mu_3{}^* = \mathsf{Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$.

Next the reduction sends the set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$ to puncture encryption key and sets $P_{\ell_0^*}, \varnothing$ to puncture decryption key to the challenger of constrained decryption game. The challenger samples keys EK_S, DK_S and it sends back to the reduction $\mathsf{EK}_S\{P_{\ell_0^*}\}$ and key which is either $\mathsf{DK}_S\{P_{\ell_0^*}\}$ or $\mathsf{DK}_S\{\varnothing\}$.

Next the reduction computes $s' = \mathsf{Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$ (note that this point is not punctured).

Then the reduction uses variables and code created above to construct and obfuscate programs P1, P3, SFake, (fig. 97, fig. 98) and P2, Dec, RFake (fig. 19). It gives obfuscated programs to the adversary, together with $s', r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*$. If challenge key was $\mathsf{DK}_S\{\varnothing\}$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{B,1,2}$. If key was $\mathsf{DK}_S\{P_{\ell_0^*}\}$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{B,1,3}$. □

Lemma 67. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{B,1,3}$ and $\mathsf{Hyb}_{B,1,4}$, then there exists an adversary which can break the strong computational extractor property of the PRF* SG *in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. We give a reduction from indistinguishability of hybrids $\mathsf{Hyb}_{B,1,3}$ and $\mathsf{Hyb}_{B,1,4}$ to strong computationally extracting PRF SG_{k_S}.

The reduction first takes plaintexts m_0^*, m_1^* from the adversary. It sends the point m_0^* to the challenger of strong extractor game. The challenger samples the key k_S for SG and either chooses $\mu_1{}^*$ at random or computes it as $\mu_1{}^* = \mathsf{SG}_{k_S}(s^*, m_0^*)$ for randomly chosen s^*. The reduction gets back from the challenger the key k_S and the value $\mu_1{}^*$.

Next the reduction reconstructs the rest of the distribution as follows. It samples all keys used in programs (except k_S), namely keys EK, DK of the main ACE, keys $\mathsf{EK}_S, \mathsf{DK}_S$ of the sender ACE, keys $\mathsf{EK}_R, \mathsf{DK}_R$ of the receiver ACE, key k_R of the sparse extracting PRF RG of the receiver. It also runs setup of the level system to create the code of $\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}$.

It chooses random r^* and sets $\mu_2{}^* = \mathsf{RG}_{k_R}(r^*, \mu_1{}^*)$. It computes levels $\ell_0^* = \mathsf{GenZero}(\mu_1{}^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, \mu_2{}^*)$. It sets $\mu_3{}^* = \mathsf{Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$ and $s' = \mathsf{Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$.

Next it computes punctured keys $\mathsf{EK}_S\{P_{\ell_0^*}\}$, $\mathsf{DK}_S\{P_{\ell_0^*}\}$, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$.

Then the reduction uses variables and code created above to construct and obfuscate programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$, (fig. 98) and $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$ (fig. 19). It gives obfuscated programs to the adversary, together with $s', r^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*$. If challenge $\mu_1{}^*$ was $\mathsf{SG}_{k_S}(s^*, m_0^*)$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{B,1,3}$. If $\mu_1{}^*$ was randomly chosen, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{B,1,4}$.

□

Lemma 68. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that $\mu_1{}^*$ is outside the image of the PRF SG. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{B,1,4}$ and $\mathsf{Hyb}_{B,1,5}$, then there exists an adversary which can break iO for programs of size σ' in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The only difference between programs $\mathsf{P3}_{B,2}$ and $\mathsf{P3}_{B,3}$ is that $\mathsf{P3}_{B,3}$ uses a punctured key $\mathsf{EK}\{\overline{p}\}$, where $\overline{p} = (1 \oplus m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$. We argue that the program never needs to encrypt any plaintext of the form $(*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, and therefore puncturing this point doesn't change the functionality:

Note that, since $\mu_1{}^*$ is random, it is outside of the image of a PRF SG with overwhelming probability, and thus validity check can pass only if $\mathsf{P3}$ is run on some $(s, m, \mu_1{}^*, \mu_2{}^*)$, where s encodes $m, \mu_1{}^*$ (and other values). However, note that $\mathsf{P3}_{B,2}$ on such input can only execute trapdoor step (and not the main step); thus the key in the main step can be safely punctured. Further, in order for the program to run encryption algorithm in the trapdoor step on any plaintext of the form $(*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, fake s should encode level ℓ_0^*. However, note that DK_S is punctured at the set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, and thus $\mathsf{P3}_{B,2}$ rejects all fake s with ℓ_0^* inside except s which encodes $(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, that is, s'. Finally, note that running $\mathsf{P3}_{B,2}$ on $(s', m, \mu_1{}^*, \mu_2{}^*)$ will pass validity check only if $m = m_0^*$ (again, since $\mu_1{}^*$ is outside of the image of PRF SG). Thus $(s', m_0^*, \mu_1{}^*, \mu_2{}^*)$ is the only potentially problematic input. However, running $\mathsf{P3}_{B,2}$ on $(s', m_0^*, \mu_1{}^*, \mu_2{}^*)$ will not trigger encryption algorithm, since the program directly outputs the value $\mu_3{}^*$ encoded in s'. Thus $\mathsf{P3}_{B,2}$ never encrypts any plaintext of the form $(*, \mu_1{}^*, \mu_2{}^*, L_0^*)$ in the trapdoor step. □

Lemma 69. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that $\mu_1{}^*$ is outside the image of the PRF SG. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{B,1,5}$ and $\mathsf{Hyb}_{B,1,6}$, then there exists an adversary which can break* *security of constrained decryption of the main ACE in time* $t(\lambda) + \mathsf{poly}(\lambda)$ *with distinguishing advantage* $\varepsilon(\lambda)$.

Proof. We give a reduction from indistinguishability of hybrids $\mathsf{Hyb}_{B,1,5}$ and $\mathsf{Hyb}_{B,1,6}$ to security of constrained decryption of main ACE.

The reduction first takes plaintexts m_0^*, m_1^* from the adversary. It samples all keys used in programs (except EK, DK), namely keys $\mathsf{EK}_S, \mathsf{DK}_S$ of sender ACE, keys $\mathsf{EK}_R, \mathsf{DK}_R$ of the receiver ACE, key k_S of the sparse extracting PRF SG of the sender, key k_R of the sparse extracting PRF RG of the receiver. It also runs setup of the level system to create the code of $\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}$.

It chooses random r^* and sets ${\mu_1}^*$ at random, ${\mu_2}^* = \mathsf{RG}_{k_R}(r^*, {\mu_1}^*)$. It computes levels $\ell_0^* = \mathsf{GenZero}({\mu_1}^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, {\mu_2}^*)$.

Next the reduction sends the set consisting of a single point $\overline{p} = (1 \oplus m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$ to puncture encryption key and sets $\overline{p}, \varnothing$ to puncture decryption key to the challenger of constrained decryption game. The challenger samples keys EK, DK and it sends back to the reduction $\mathsf{EK}\{\overline{p}\}$ and key which is either $\mathsf{DK}\{\overline{p}\}$ or $\mathsf{DK}\{\varnothing\}$.

Next the reduction computes ${\mu_3}^* = \mathsf{Enc}_{\mathsf{EK}\{\overline{p}\}}(m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$ (note that this point isn't punctured, thus the reduction can indeed encrypt it).

It punctured keys $\mathsf{EK}_S\{P_{\ell_0^*}\}$, $\mathsf{DK}_S\{P_{\ell_0^*}\}$, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, and sets $s' = \mathsf{Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$.

Then the reduction uses variables and code created above to construct and obfuscate programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$ (fig. 99) and $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$ (fig. 100, fig. 101). It gives obfuscated programs to the adversary, together with $s', r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*$. If challenge key was $\mathsf{DK}\{\varnothing\}$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{B,1,5}$. If key was $\mathsf{DK}_S\{\overline{p}\}$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{B,1,6}$. □

Lemma 70. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that ${\mu_1}^*$ is outside the image of the PRF* SG*, and $\hat{\rho}^*$ is outside of the image of the* prg*. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish* $\mathsf{Hyb}_{B,1,6}$ *and* $\mathsf{Hyb}_{B,2,1}$*, then there exists an adversary which can break security of* iO *for σ'-sized programs in time* $t(\lambda) + \mathsf{poly}(\lambda)$ *with distinguishing advantage* $\frac{1}{3} \cdot \varepsilon(\lambda)$.

Proof. In this analysis we assume that r^* is outside the image of receiver ACE, and thus $\mathsf{ACE.Dec}_{\mathsf{DK}_R}(r^*) =$ $'\mathsf{fail}'$.

Programs $\mathsf{P2}$ **and** $\mathsf{P2}_{B,2}$**.** We present case analysis to show that the behavior of programs $\mathsf{P2}$ and $\mathsf{P2}_{B,2}$ on each input is the same:

- Case $r = r^*$:
  - Case $\mu_1 = {\mu_1}^*$: $\mathsf{P2}$ outputs ${\mu_2}^*$ via main step since r^* is outside of image of ACE. $\mathsf{P2}_{B,2}$ outputs ${\mu_2}^*$ due to hardwired instruction.
  - Case $\mu_1 \neq {\mu_1}^*$: $\mathsf{P2}$ executes main step and outputs $\mathsf{RG}_{k_R}(r^*, \mu_1)$ since r^* is outside of image of ACE. $\mathsf{P2}_{B,2}$ executes main step and outputs $\mathsf{RG}_{k_R}(r^*, \mu_1)$ due to hardwired instruction.
- Case $r = r'$:

- Case $\mu_1 = {\mu_1}^*$: $\mathsf{P2}$ outputs ${\mu_2}^*$ via trapdoor step. $\mathsf{P2}_{B,2}$ outputs ${\mu_2}^*$ due to hardwired instruction.
- Case $\mu_1 \neq {\mu_1}^*$: $\mathsf{P2}$ skips the trapdoor step since r' contains the wrong ${\mu_1}^* \neq \mu_1$, and outputs $\mathsf{RG}_{k_R}(r', \mu_1)$. $\mathsf{P2}_{B,2}$ executes main step due to hardwired instruction and outputs $\mathsf{RG}_{k_R}(r', \mu_1)$.

- Case $r \neq r', r^*$: $\mathsf{P2}$ and $\mathsf{P2}_{B,2}$ execute the same code, since punctured keys preserve functionality on all inputs which are not punctured. Note that keys are never used at punctured points.

Programs Dec **and** $\mathsf{Dec}_{B,2}$**.** Next we compare programs Dec and $\mathsf{Dec}_{B,2}$. Note that validity check passes on the same set of inputs in programs Dec and $\mathsf{Dec}_{B,2}$, since programs $\mathsf{P2}$ and $\mathsf{P2}_{B,1}$ are functionally equivalent. We present the analysis assuming inputs passed the validity check.

- Case $r = r^*$:
  - Case $(\mu_1, \mu_2) = ({\mu_1}^*, {\mu_2}^*)$:
    * Case $\mu_3 = {\mu_3}^*$: Dec outputs m_0^* via main step since r^* is outside of image of ACE. $\mathsf{Dec}_{B,1}$ outputs m_0^* due to hardwired instruction.
    * Case $\mu_3 \neq {\mu_3}^*$: since r^* is outside of image of ACE, Dec executes the main step. $\mathsf{Dec}_{B,2}$ skips the trapdoor step due to hardwired instruction and performes exactly the same actions in the main step.
  - Case $(\mu_1, \mu_2) \neq ({\mu_1}^*, {\mu_2}^*)$: Dec executes main step since r^* is outside of image of ACE. $\mathsf{Dec}_{B,2}$ skips the trapdoor step due to hardwired instruction and performes exactly the same actions in the main step.
- Case $r = r'$:
  - Case $(\mu_1, \mu_2) = ({\mu_1}^*, {\mu_2}^*)$:
    * Case $\mu_3 = {\mu_3}^*$: Dec outputs m_0^* via trapdoor step. $\mathsf{Dec}_{B,2}$ outputs m_0^* due to hardwired instruction.
    * Case $\mu_3 \neq {\mu_3}^*$: Dec executes trapdoor step. That is, it tries to decrypt μ_3 and either outputs its plaintext or $'\mathsf{fail}'$. In order for Dec to outputs a plaintext (and not $'\mathsf{fail}'$), μ_1, μ_2 should be the same in the input, in μ_3, in r', and in L'', and moreover, $\mathsf{isLess}(L', L'')$ should be true. Since r' has level $L' = L_0^*$, isLess is true for all L'' of the form $[i, {\mu_1}^*, {\mu_2}^*]$, where $i > 0$. In other words, μ_3 should be an encryption of $(m, {\mu_1}^*, {\mu_2}^*, L'')$, where $L'' = [i, {\mu_1}^*, {\mu_2}^*]$, $i > 0$, and m is arbitrary. We call it condition 1.

      $\mathsf{Dec}_{B,2}$ is instructed to skip the trapdoor step and execute the main step. That is, it decrypts μ_3 and either outputs its plaintext or $'\mathsf{fail}'$. In order for $\mathsf{Dec}_{B,1}$ to outputs a plaintext (and not $'\mathsf{fail}'$), μ_1, μ_2 should be the same in the input, in μ_3, and in L'' (however, unlike Dec, there is no "$\mathsf{isLess}(L', L'') = \mathsf{true}$" condition). In other words, μ_3 should be an encryption of $(m, {\mu_1}^*, {\mu_2}^*, L'')$, where $L'' = [i, {\mu_1}^*, {\mu_2}^*]$, $i \geq 0$, and m is arbitrary. We call it condition 2.

      Thus, the only difference in these conditions for Dec and $\mathsf{Dec}_{B,2}$ is that, given an encryption of $(m, {\mu_1}^*, {\mu_2}^*, [0, {\mu_1}^*, {\mu_2}^*])$ for any m (that is, ${\mu_3}^*$ or $\overline{{\mu_3}^*}$), condition 1 instructs to output $'\mathsf{fail}'$ and condition 2 instructs to output m. However, we claim that both programs Dec and $\mathsf{Dec}_{B,2}$ still behave the same on inputs ${\mu_3}^*$ or $\overline{{\mu_3}^*}$. Indeed, recall that if the input was $(r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, both programs would output m_0^* as analysed in the previous case. If the input was $(r', \mu_1{}^*, \mu_2{}^*, \overline{\mu_3{}^*})$, both programs would output $'\mathsf{fail}'$, since decryption key DK of the main ACE is punctured at the point $\overline{p} = (1 \oplus m_0^*, \mu_1{}^*, \mu_2{}^*, [0, \mu_1{}^*, \mu_2{}^*])$.

Thus, in this case both programs have the same functionality.

– Case $(\mu_1, \mu_2) \neq (\mu_1{}^*, \mu_2{}^*)$: Dec skips the trapdoor step since r' contains the wrong $(\mu_1{}^*, \mu_2{}^*) \neq (\mu_1, \mu_2)$, and executes the main step. $\mathsf{Dec}_{B,2}$ skips the trapdoor step due to hardwired instruction and executes the main step.

- Case $r \neq r', r^*$: Dec and $\mathsf{Dec}_{B,2}$ execute the same code, since punctured keys preserve functionality on all inputs which are not punctured. Note that $\mathsf{Dec}_{B,2}$ never uses key DK_R at the punctured points, thus puncturing it doesn't change the functionality of the program. Note that the key $\mathsf{DK}_{\overline{p}}$ can be used by Dec and $\mathsf{Dec}_{B,2}$ to decrypt an encryption of $\overline{p}$, however it is punctured at both programs and thus functionality of both programs is the same in this case.

Programs RFake **and** $\mathsf{RFake}_{B,2}$**.** Next we compare programs RFake and $\mathsf{RFake}_{B,2}$. Note that the only difference is that $\mathsf{RFake}_{B,2}$ uses a punctured key $\mathsf{EK}_R\{S_{\hat{\rho}^*}\}$, where $S_{\hat{\rho}^*} = \{(*, *, *, *, *, \hat{\rho}^*)\}$ for randomly chosen $\hat{\rho}^*$. By assumption of the lemma, $\hat{\rho}^*$ is outside of the image of this prg, and thus $\mathsf{RFake}_{B,2}$ never needs to encrypt any of points ending with $\hat{\rho}^*$. Therefore puncturing the key doesn't change the functionality of the program. □

Lemma 71. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that $\mu_1{}^*$ is outside the image of the PRF* SG*, and $\hat{\rho}^*$ is outside of the image of the* prg*. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish* $\mathsf{Hyb}_{B,2,1}$ *and* $\mathsf{Hyb}_{B,2,2}$*, then there exists an adversary which can break security of of a puncturable PRF* RG_{k_R} *in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is very similar to the proof of indistinguishability of hybrids $\mathsf{Hyb}_{A,2}, \mathsf{Hyb}_{A,3}$, except that the reduction is for PRF of the receiver, not the PRF of the sender.

We give a reduction to security of a puncturable PRF RG_{k_R} at the punctured point $(r^*, \mu_1{}^*)$.

The reduction first takes plaintexts m_0^*, m_1^* from the adversary. Next it chooses random $r^*, \mu_1{}^*$ and sends the point $(r^*, \mu_1{}^*)$ to the challenger of puncturable PRF game. The reduction gets back from the challenger the punctured key $k_R\{(r^*, \mu_1{}^*)\}$ and the value $\mu_2{}^*$, which is either $\mathsf{RG}_{k_R}(r^*, \mu_1{}^*)$ or randomly chosen.

Next the reduction reconstructs the rest of the distribution as follows. It samples all keys used in programs (except $k_R\{(r^*, \mu_1{}^*)\}$), namely keys EK, DK of the main ACE, keys $\mathsf{EK}_S, \mathsf{DK}_S$ of the sender ACE, keys $\mathsf{EK}_R, \mathsf{DK}_R$ of the receiver ACE, key k_S of the sparse extracting PRF SG of the sender. It also runs setup of the level system to create the code of $\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}$.

It computes levels $\ell_0^* = \mathsf{GenZero}(\mu_1{}^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, \mu_2{}^*)$. It sets $\mu_3{}^* = \mathsf{Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, $r' = \mathsf{Enc}_{\mathsf{EK}_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \hat{\rho}^*)$ for randomly chosen $\hat{\rho}^*$.

Next it computes punctured keys $\mathsf{DK}_R\{r^*, r'\}$, $k_R\{(r^*, \mu_1{}^*), (r', \mu_1{}^*)\}$ (by additionally puncturing challenge $k_R\{(r^*, \mu_1{}^*)\}$ at $(r', \mu_1{}^*)$), and $\mathsf{EK}_R\{S_{\hat{\rho}^*}\}$, $S_{\hat{\rho}^*} = \{(*, *, *, *, *, \hat{\rho}^*)\}$ for randomly chosen $\hat{\rho}^*$. It also punctures keys $\mathsf{EK}_S\{P_{\ell_0^*}\}$, $\mathsf{DK}_S\{P_{\ell_0^*}\}$, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, and $\mathsf{EK}\{\overline{p}\}$, $\mathsf{DK}\{\overline{p}\}$ at $\overline{p} = (1 \oplus m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$.

Then the reduction uses variables and code created above to construct and obfuscate programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$, (fig. 99) and $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$ (fig. 102). It gives obfuscated programs to the adversary, together with

$s', r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*$. If challenge ${\mu_2}^*$ was $\mathsf{RG}_{k_R}(r^*, {\mu_1}^*)$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{B,2,1}$. If ${\mu_2}^*$ was randomly chosen, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{B,2,2}$.

Note that this reduction is using the fact that an adversary who holds the punctured key can additionally puncture it at another point. We note that the construction of an extracting puncturable PRF of [SW14] is based on GGM PRF and satisfies this property. □

Lemma 72. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that ${\mu_1}^*$ is outside the image of the PRF SG, and $\hat{\rho}^*$ is outside of the image of the* prg. *Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{B,2,2}$ and $\mathsf{Hyb}_{B,2,3}$, then there exists an adversary which can break the symmetry of a receiver-fake relaxed ACE in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is very similar to the proof of indistinguishability of hybrids $\mathsf{Hyb}_{A,3}, \mathsf{Hyb}_{A,4}$, except that the reduction is to the ACE of the receiver, not ACE of the sender.

We give a reduction to symmetry of receiver ACE.

The reduction first takes plaintexts m_0^*, m_1^* from the adversary. Next it samples all keys used in programs (except $\mathsf{EK}_R, \mathsf{DK}_R$), namely keys EK, DK of the main ACE, keys $\mathsf{EK}_S, \mathsf{DK}_S$ of the sender ACE, key k_S of the sparse extracting PRF SG of the sender, key k_R of the sparse extracting PRF RG of the receiver. It also runs setup of the level system to create the code of $\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}$.

It chooses random ${\mu_1}^*, {\mu_2}^*$. It computes levels $\ell_0^* = \mathsf{GenZero}({\mu_1}^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, {\mu_2}^*)$. It sets ${\mu_3}^* = \mathsf{Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$.

Next the reduction chooses $\hat{\rho}^*$ at random and sends $p = (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \hat{\rho}^*)$ as the challenge point to the challenger of the symmetry of ACE. The challenger chooses random r^*, samples keys $\mathsf{EK}_R, \mathsf{DK}_R$ of ACE and computes $r' = \mathsf{Enc}_{\mathsf{EK}_R}(p)$, and punctures EK_R at $S_{\hat{\rho}^*} = \{(*, *, *, *, *, \hat{\rho}^*)\}$ and DK_R at r^*, r' (DK_R is first punctured at one of the strings r^*, r' which is lexicographically smaller, and then at the other). The reduction gets back from the challenger $(r_1, r_2, \mathsf{EK}_R\{S_{\hat{\rho}^*}\}, \mathsf{DK}_R\{r^*, r'\})$, where $r_1 = r^*, r_2 = r'$ or $r_1 = r', r_2 = r^*$.

Next it computes punctured keys $\mathsf{EK}_S\{P_{\ell_0^*}\}$, $\mathsf{DK}_S\{P_{\ell_0^*}\}$, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, and $\mathsf{EK}\{\overline{p}\}, \mathsf{DK}\{\overline{p}\}$ where $\overline{p} = (1 \oplus m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$.

Next the reduction computes punctured key $k_R\{(r_1, {\mu_1}^*), (r_2, {\mu_1}^*)\}$. Then it uses variables and code created above to construct and obfuscate programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$, (fig. 99) and $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$ (fig. 102). In particular, in every place where r^*, r' appear, e.g. in code of programs, or as a punctured point, the reduction first uses one of the strings r_1, r_2 which is lexicographically smaller, and then the other (note that r^*, r' always appear together in the distribution, except for the value given to the adversary as randomness of the receiver).

Next the reduction gives obfuscated programs to the adversary, together with $s', r_1, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*$. If challenge r_1, r_2 are r^*, r', then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{B,2,2}$. If r_1, r_2 are r', r^*, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{B,2,3}$. □

Lemma 73. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that ${\mu_1}^*$ is outside the image of the PRF SG, and $\hat{\rho}^*$ is outside of the image of* *the* prg. *Then, if there exists an adversary which can* $(t(\lambda), \varepsilon(\lambda))$*-distinguish* $\mathsf{Hyb}_{B,2,3}$ *and* $\mathsf{Hyb}_{B,2,4}$*, then there exists an adversary which can break security of a puncturable PRF* RG_{k_R} *in time* $t(\lambda) + \mathsf{poly}(\lambda)$ *with distinguishing advantage* $\varepsilon(\lambda)$*.*

Proof. The proof is very similar to the proof of indistinguishability of hybrids $\mathsf{Hyb}_{B,2,1}, \mathsf{Hyb}_{B,2,2}$, except that r' and not r^* is given to the adversary as randomness of the receiever.

We give a reduction to security of a puncturable PRF RG_{k_R} at the punctured point $(r^*, {\mu_1}^*)$.

The reduction first takes plaintexts m_0^*, m_1^* from the adversary. Next it chooses random $r^*, {\mu_1}^*$ and sends the point $(r^*, {\mu_1}^*)$ to the challenger of puncturable PRF game. The reduction gets back from the challenger the punctured key $k_R\{(r^*, {\mu_1}^*)\}$ and the value ${\mu_2}^*$, which is either $\mathsf{RG}_{k_R}(r^*, {\mu_1}^*)$ or randomly chosen.

Next the reduction reconstructs the rest of the distribution as follows. It samples all keys used in programs (except $k_R\{(r^*, {\mu_1}^*)\}$), namely keys EK, DK of the main ACE, keys $\mathsf{EK}_S, \mathsf{DK}_S$ of the sender ACE, keys $\mathsf{EK}_R, \mathsf{DK}_R$ of the receiver ACE, key k_S of the sparse extracting PRF SG of the sender. It also runs setup of the level system to create the code of $\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}$.

It computes levels $\ell_0^* = \mathsf{GenZero}({\mu_1}^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, {\mu_2}^*)$. It sets ${\mu_3}^* = \mathsf{Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \hat{\rho}^*)$ for randomly chosen $\hat{\rho}^*$.

Next it computes punctured keys $\mathsf{DK}_R\{r^*, r'\}$, $k_R\{(r^*, {\mu_1}^*), (r', {\mu_1}^*)\}$ (by additionally puncturing challenge $k_R\{(r^*, {\mu_1}^*)\}$ at $(r', {\mu_1}^*)$), and $\mathsf{EK}_R\{S_{\hat{\rho}^*}\}$, $S_{\hat{\rho}^*} = \{(*, *, *, *, *, \hat{\rho}^*)\}$ for randomly chosen $\hat{\rho}^*$. It also punctures keys $\mathsf{EK}_S\{P_{\ell_0^*}\}$, $\mathsf{DK}_S\{P_{\ell_0^*}\}$, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, and $\mathsf{EK}\{\overline{p}\}$, $\mathsf{DK}\{\overline{p}\}$ at $\overline{p} = (1 \oplus m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$.

Then the reduction uses variables and code created above to construct and obfuscate programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$, (fig. 99) and $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$ (fig. 102). It gives obfuscated programs to the adversary, together with $s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*$. If challenge ${\mu_2}^*$ was $\mathsf{RG}_{k_R}(r^*, {\mu_1}^*)$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{B,2,4}$. If ${\mu_2}^*$ was randomly chosen, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{B,2,3}$.

Note that this reduction is using the fact that an adversary who holds the punctured key can additionally puncture it at another point. We note that the construction of an extracting puncturable PRF of [SW14] is based on GGM PRF and satisfies this property. □

Lemma 74. *Assume* s^*, r^* *are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that* ${\mu_1}^*$ *is outside the image of the PRF* SG*, and* $\hat{\rho}^*$ *is outside of the image of the* prg*. Then, if there exists an adversary which can* $(t(\lambda), \varepsilon(\lambda))$*-distinguish* $\mathsf{Hyb}_{B,2,4}$ *and* $\mathsf{Hyb}_{B,2,5}$*, then there exists an adversary which can break security of* iO *for* σ'*-sized programs in time* $t(\lambda) + \mathsf{poly}(\lambda)$ *with distinguishing advantage* $\frac{1}{3} \cdot \varepsilon(\lambda)$*.*

Proof. The proof is identical to the proof of lemma 70. □

Lemma 75. *Assume* s^*, r^* *are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that* ${\mu_1}^*$ *is outside the image of the PRF* SG*. Then, if there exists an adversary which can* $(t(\lambda), \varepsilon(\lambda))$*-distinguish* $\mathsf{Hyb}_{B,2,5}$ *and* $\mathsf{Hyb}_{B,2,6}$*, then there exists an adversary which can break security of a* prg *in time* $t(\lambda) + \mathsf{poly}(\lambda)$ *with distinguishing advantage* $\varepsilon(\lambda)$*.*

Proof. We give a reduction to security of a prg.

The reduction first takes plaintexts m_0^*, m_1^* from the adversary.

It samples all keys used in programs, namely keys EK, DK of the main ACE, keys $\mathsf{EK}_S, \mathsf{DK}_S$ of the sender ACE, keys $\mathsf{EK}_R, \mathsf{DK}_R$ of the receiver ACE, key k_S of the sparse extracting PRF SG of the sender, key k_R of the sparse extracting PRF RG of the receiver. It also runs setup of the level system to create the code of $\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}$.

Next it chooses random $r^*, {\mu_1}^*$ and computes ${\mu_2}^* = \mathsf{RG}_{k_R}(r^*, {\mu_1}^*)$. It computes levels $\ell_0^* = \mathsf{GenZero}({\mu_1}^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, {\mu_2}^*)$. It sets ${\mu_3}^* = \mathsf{Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$.

It receives $\hat{\rho}^*$ from a challenger of a prg game which is either randomly chosen or $\mathsf{prg}(\rho^*)$ for randomly chosen ρ^*. Then the reduction sets $r' = \mathsf{Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \hat{\rho}^*)$.

Next it punctures keys $\mathsf{EK}_S\{P_{\ell_0^*}\}, \mathsf{DK}_S\{P_{\ell_0^*}\}$, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, and $\mathsf{EK}\{\overline{p}\}, \mathsf{DK}\{\overline{p}\}$ at $\overline{p} = (1 \oplus m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$.

Then the reduction uses variables and code created above to construct and obfuscate programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$, (fig. 99) and $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$ (fig. 101). It gives obfuscated programs to the adversary, together with $s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*$. If challenge $\hat{\rho}^*$ was an image of a prg, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{B,2,6}$. If $\hat{\rho}^*$ was randomly chosen, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{B,2,5}$. □

Lemma 76. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that ${\mu_1}^*$ is outside the image of the PRF SG. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{B,2,6}$ and $\mathsf{Hyb}_{B,3,1}$, then there exists an adversary which can break security of constrained decryption of the main ACE in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is very similar to the proof of indistinguishability of hybrids $\mathsf{Hyb}_{B,1,5}, \mathsf{Hyb}_{B,1,6}$, except that r' and not r^* is given to the adversary as randomness of the receiver. □

Lemma 77. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that ${\mu_1}^*$ is outside the image of the PRF SG. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{B,3,1}$ and $\mathsf{Hyb}_{B,3,2}$, then there exists an adversary which can break security of iO for σ'-sized programs in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is identical to the proof of lemma 68. □

Lemma 78. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{B,3,2}$ and $\mathsf{Hyb}_{B,3,3}$, then there exists an adversary which can break the strong computational extractor property of a PRF SG_{k_S} in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is very similar to the proof of indistinguishability of hybrids $\mathsf{Hyb}_{B,1,3}, \mathsf{Hyb}_{B,1,4}$, except that r' and not r^* is given to the adversary as randomness of the receiver. □

Lemma 79. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{B,3,3}$ and $\mathsf{Hyb}_{B,3,4}$, then there exists an adversary which can break security of contrained decryption of a sender-fake relaxed ACE in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is very similar to the proof of indistinguishability of hybrids $\mathsf{Hyb}_{B,1,2}, \mathsf{Hyb}_{B,1,3}$, except that r' and not r^* is given to the adversary as randomness of the receiver. □

Lemma 80. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that ${\mu_1}^*$ is outside the image of the PRF SG. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{B,3,4}$ and $\mathsf{Hyb}_{B,3,5}$, then there exists an adversary which can break security of iO for σ'-sized programs in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is identical to the proof of lemma 65. □

Finally, we note that the distributions in $\mathsf{Hyb}_{B,3,5}$ and Hyb_C are $2^{-\lambda}$-close (the reasoning is similar to distributions $\mathsf{Hyb}_B, \mathsf{Hyb}_{B,1,1}$).

8.2.3 Reductions in the proof of lemma 56 (Semantic Security)

Let $t(\lambda)$ be any function in $\Omega(\mathsf{poly}(\lambda))$, and let $\varepsilon(\lambda)$ be a negligible function in $w(2^{-\lambda})$. Assuming the sender-fake relaxed ACE, receiver-fake relaxed ACE, main ACE, sparse extracting puncturable PRF, and iO for program size σ' are $(t(\lambda), \varepsilon(\lambda))$-secure, we show that no time-$t(\lambda)$ adversary can distinguish between Hyb_C and Hyb_D with more than $O(\varepsilon(\lambda)) + O(2^{-\tau(\lambda)})$ advantage.

(Note that security loss $O(2^{-\tau(\lambda)})$ comes from conditioning on the fact that ${\mu_1}^*, {\mu_2}^*$ are outside of the image of the corresponding PRFs. Conditioning on s^*, r^* incurs only $2^{-\lambda}$ loss and therefore we omit it.).

Lemma 81. *Statistical distance between distributions $\mathsf{Hyb}_C, \mathsf{Hyb}_{C,1,1}$ is at most $2 \cdot 2^{-\lambda}$.*

Proof. Same as indistinguishability between hybrids $\mathsf{Hyb}_B, \mathsf{Hyb}_{B,1,1}$. □

Lemma 82. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{C,1,1}$ and $\mathsf{Hyb}_{C,1,2}$, then there exists an adversary which can break security of iO for σ'-sized programs in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The only difference between programs SFake and $\mathsf{SFake}_{C,1}$ is that $\mathsf{SFake}_{C,1}$ uses a punctured key $\mathsf{EK}_S\{P_{\ell_0^*}\}$, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$. This is without changing functionality, since SFake never needs to encrypt a plaintext with level ℓ_0^*, since $\ell_0^* = [0, {\mu_1}^*]$ and SFake encrypts levels with value at least 1. □

Lemma 83. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{C,1,2}$ and $\mathsf{Hyb}_{C,1,3}$, then there exists an adversary which can break security of contrained decryption of a sender-fake relaxed ACE in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. Same as indistinguishability between hybrids $\mathsf{Hyb}_{B,3,3}, \mathsf{Hyb}_{B,3,4}$. □

Lemma 84. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{C,1,3}$ and $\mathsf{Hyb}_{C,1,4}$, then there exists an adversary which can break the strong computational extractor property of a PRF SG_{k_S} in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. Same as indistinguishability between hybrids $\mathsf{Hyb}_{B,3,2}, \mathsf{Hyb}_{B,3,3}$. □

Lemma 85. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that ${\mu_1}^*$ is outside the image of the PRF* SG. *Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish* $\mathsf{Hyb}_{C,1,4}$ *and* $\mathsf{Hyb}_{C,1,5}$, *then there exists an adversary which can break the strong computational extractor property of a PRF* RG_{k_R} *in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is similar to the proof of indistinguishability of hybrids $\mathsf{Hyb}_{B,1,3}, \mathsf{Hyb}_{B,1,4}$, except that the reduction is to the strong extracting PRF of the receiver, not the sender.

We give a reduction to strong computationally extracting PRF RG_{k_R}.

The reduction first takes plaintexts m_0^*, m_1^* from the adversary. It chooses ${\mu_1}^*$ at random and sends the point ${\mu_1}^*$ to the challenger of strong extractor game. The challenger samples the key k_R for RG and either chooses ${\mu_2}^*$ at random or computes it as ${\mu_2}^* = \mathsf{RG}_{k_R}(r^*, {\mu_1}^*)$ for randomly chosen r^*. The reduction gets back from the challenger the key k_R and the value ${\mu_2}^*$.

Next the reduction reconstructs the rest of the distribution as follows. It samples all keys used in programs (except k_R), namely keys EK, DK of the main ACE, keys $\mathsf{EK}_S, \mathsf{DK}_S$ of the sender ACE, keys $\mathsf{EK}_R, \mathsf{DK}_R$ of the receiver ACE, key k_S of the sparse extracting PRF SG of the sender. It also runs setup of the level system to create the code of $\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}$.

It computes levels $\ell_0^* = \mathsf{GenZero}({\mu_1}^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, {\mu_2}^*)$. It sets ${\mu_3}^* = \mathsf{Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*.

Next it computes punctured keys $\mathsf{EK}_S\{P_{\ell_0^*}\}$, $\mathsf{DK}_S\{P_{\ell_0^*}\}$, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$.

Then the reduction uses variables and code created above to construct and obfuscate programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$, (fig. 105) and $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$ (fig. 107). It gives obfuscated programs to the adversary, together with $s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*$. If challenge ${\mu_2}^*$ was $\mathsf{RG}_{k_R}(r^*, {\mu_1}^*)$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{C,1,4}$. If ${\mu_2}^*$ was randomly chosen, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{C,1,5}$.

□

Lemma 86. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that ${\mu_1}^*$ is outside the image of the PRF* SG, *and ${\mu_2}^*$ is outside the image of the PRF* RG. *Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish* $\mathsf{Hyb}_{C,1,5}$ *and* $\mathsf{Hyb}_{C,2,1}$, *then there exists an adversary which can break security of* iO *for σ'-sized programs in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The only difference between programs $\mathsf{P3}_{B,2}$ and $\mathsf{P3}_{B,3}$ is that $\mathsf{P3}_{B,3}$ uses a punctured key $\mathsf{EK}\{p_0, p_1\}$, where $p_0 = (m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $p_1 = (m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$. We argue that the program never needs to encrypt p_0, p_1, and therefore puncturing these points doesn't change the functionality:

Since we assumed that ${\mu_1}^*$ is outside of the image of a PRF SG, validity check can pass only if P3 is run on some $(s, m, {\mu_1}^*, {\mu_2}^*)$, where s encodes $m, {\mu_1}^*$ (and other values). However, note that $\mathsf{P3}_{C,2}$ on such input can only execute trapdoor step (and not the main step); thus the key in the main step can be safely punctured. Further, in order for the program to run encryption algorithm in the trapdoor step on input p_0 or p_1, fake s should encode level ℓ_0^*. However, note that DK_S is punctured at the set $P_{\ell_0^*} = \{(*,*,*,*,\ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, and thus $\mathsf{P3}_{C,2}$ rejects all fake s with ℓ_0^* inside except s which encodes $(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, that is, s'. Finally, note that running $\mathsf{P3}_{C,2}$ on $(s', m, {\mu_1}^*, {\mu_2}^*)$ will pass validity check only if $m = m_0^*$ (again, since ${\mu_1}^*$ is outside of the image of PRF SG). Thus $(s', m_0^*, {\mu_1}^*, {\mu_2}^*)$ is the only potentially problematic input (in particular, the key is never used to encrypt p_1). However, running $\mathsf{P3}_{C,2}$ on $(s', m_0^*, {\mu_1}^*, {\mu_2}^*)$ will not trigger encryption algorithm, since the program directly outputs the value ${\mu_3}^*$ encoded in s'. Thus $\mathsf{P3}_{C,2}$ never encrypts p_0 or p_1 in the trapdoor step. $\square$

Lemma 87. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that ${\mu_1}^*$ is outside the image of the PRF SG, and ${\mu_2}^*$ is outside the image of the PRF RG. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{C,2,1}$ and $\mathsf{Hyb}_{C,2,2}$, then there exists an adversary which can break security of constrained decryption of main ACE in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is similar to the proof of indistinguishability of hybrids $\mathsf{Hyb}_{B,3,3}, \mathsf{Hyb}_{B,3,4}$, except that EK is additionally punctured at another point, and ${\mu_2}^*$ is randomly chosen.

We give a reduction to security of constrained decryption of main ACE.

The reduction first takes plaintexts m_0^*, m_1^* from the adversary. It samples all keys used in programs (except EK, DK), namely keys $\mathsf{EK}_S, \mathsf{DK}_S$ of sender ACE, keys $\mathsf{EK}_R, \mathsf{DK}_R$ of the receiver ACE, key k_S of the sparse extracting PRF SG of the sender, key k_R of the sparse extracting PRF RG of the receiver. It also runs setup of the level system to create the code of $\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}$.

It chooses random ${\mu_1}^*, {\mu_2}^*$. It computes levels $\ell_0^* = \mathsf{GenZero}({\mu_1}^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, {\mu_2}^*)$.

Next the reduction sends the set consisting of two points $p_0 = (m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $p_1 = (m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$ to puncture encryption key, sets $p_1, \varnothing$ to puncture decryption key, and plaintext p_0 to the challenger of constrained decryption game (note that plaintext p_0 doesn't belong to the set $\{p_1\}$ for puncturing DK and thus this is a valid query to the challenger of constrained decryption game). The challenger samples keys EK, DK and it sends back to the reduction $\mathsf{EK}\{p_0, p_1\}$, key which is either $\mathsf{DK}\{p_1\}$ or $\mathsf{DK}\{\varnothing\}$, and ${\mu_3}^* = \mathsf{Enc}_{\mathsf{EK}}(p_0)$.

Next the reduction punctures keys $\mathsf{EK}_S\{P_{\ell_0^*}\}$, $\mathsf{DK}_S\{P_{\ell_0^*}\}$, where $P_{\ell_0^*} = \{(*,*,*,*,\ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, and sets $s' = \mathsf{Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*.

Then the reduction uses variables and code created above to construct and obfuscate programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$ (fig. 106) and $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$ (fig. 107, fig. 108). It gives obfuscated programs to the adversary, together with $s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*$. If challenge key was $\mathsf{DK}\{\varnothing\}$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{C,2,1}$. If key was $\mathsf{DK}_S\{p_1\}$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{C,2,2}$. $\square$

Lemma 88. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that ${\mu_1}^*$ is outside the image of the PRF SG, and ${\mu_2}^*$ is outside the image of the PRF RG. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{C,2,2}$ and $\mathsf{Hyb}_{C,2,3}$, then there exists an adversary which can break security of iO for σ'-sized circuits in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\frac{1}{2}\varepsilon(\lambda)$.*

Proof. We start with analyzing program Dec: The only difference between programs $\mathsf{Dec}_{C,1}$ and $\mathsf{Dec}_{C,2}$

is that $\mathsf{Dec}_{C,1}$ uses key $\mathsf{DK}\{p_1\}$ and $\mathsf{Dec}_{C,2}$ uses $\mathsf{DK}\{p_0, p_1\}$, i.e. the key is additionally punctured at p_0 (here $p_0 = (m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $p_1 = (m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$). We will argue that if $\mathsf{Dec}_{C,1}$ on input ${\mu_3}^* = \mathsf{ACE.Enc_{EK}}(p_0)$ reaches the line where it needs to decrypt ${\mu_3}^*$, then it always outputs $'\mathsf{fail}'$. Therefore puncturing this point (and thus forcing $\mathsf{Dec}_{C,2}$ to output $'\mathsf{fail}'$ when attempt to decrypt ${\mu_3}^*$) doesn't change the functionality:

First, note that if input $\mu_3 = {\mu_3}^*$, but $(\mu_1, \mu_2) \neq ({\mu_1}^*, {\mu_2}^*)$ and the program reached decryption of ${\mu_3}^*$, then the program outputs $'\mathsf{fail}'$: indeed, ${\mu_3}^*$ encrypts ${\mu_1}^*, {\mu_2}^*$ and thus the check $(\mu_1, \mu_2) = ({\mu_1}^*, {\mu_2}^*)$ will not pass.

Second, by assumption ${\mu_2}^*$ is outside of the image of a PRF RG, and thus validity check can pass only if $\mathsf{Dec}_{C,1}$ is run on some $(r, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where r encodes ${\mu_1}^*, {\mu_2}^*$ (and other values). However, note that $\mathsf{Dec}_{C,1}$ on such input can only execute the trapdoor step (and not the main step); thus the key in the main step can be safely punctured. Further, in order for the program to output m after decryption in the trapdoor step, the condition "$\mathsf{isLess}(L', L'')$" should hold. However, when input $\mu_3 = {\mu_3}^*$, L'' is equal to $[0, {\mu_1}^*, {\mu_2}^*]$, which is the smallest possible level and therefore there doesn't exist L' such that $\mathsf{isLess}(L', L'') = \mathsf{true}$. Thus, if $\mathsf{Dec}_{C,1}$ reached decryption in the trapdoor step on input ${\mu_3}^*$, it will anyway output $'\mathsf{fail}'$ due to failed "isLess" check and therefore we can puncture DK at p_0 such that an attempt to decrypt ${\mu_3}^*$ would cause Dec to output $'\mathsf{fail}'$ immediately.

Next we analyze program RFake. The difference between $\mathsf{RFake}_{C,1}$ and $\mathsf{RFake}_{C,2}$ is that the key DK, which is already punctured at p_1, is additionally punctured at p_0. In order to preserve the functionality of RFake on input ${\mu_3}^*$, we additionally instruct RFake to use level $L_0^* = [0, {\mu_1}^*, {\mu_2}^*]$ on input ${\mu_3}^*$ (without actually decrypting ${\mu_3}^*$). Note that this is what $\mathsf{RFake}_{C,1}$ would do on input ${\mu_3}^*$; thus this doesn't change the functionality. □

Lemma 89. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that ${\mu_1}^*$ is outside the image of the PRF* SG*, and ${\mu_2}^*$ is outside the image of the PRF* RG*. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish* $\mathsf{Hyb}_{C,2,3}$ *and* $\mathsf{Hyb}_{C,2,4}$*, then there exists an adversary which can break indistinguishability of ciphertexts of main ACE in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\frac{1}{2}\varepsilon(\lambda)$.*

Proof. We give a reduction to indistinguishability of ciphertexts of main ACE.

The reduction first takes plaintexts m_0^*, m_1^* from the adversary. It samples all keys used in programs (except EK, DK), namely keys $\mathsf{EK}_S, \mathsf{DK}_S$ of sender ACE, keys $\mathsf{EK}_R, \mathsf{DK}_R$ of the receiver ACE, key k_S of the sparse extracting PRF SG of the sender, key k_R of the sparse extracting PRF RG of the receiver. It also runs setup of the level system to create the code of $\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}$.

It chooses random ${\mu_1}^*, {\mu_2}^*$. It computes levels $\ell_0^* = \mathsf{GenZero}({\mu_1}^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, {\mu_2}^*)$.

Next the reduction sends the set consisting of two points $p_0 = (m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $p_1 = (m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$ to puncture encryption key, the same set $\{p_0, p_1\}$ to puncture decryption key, and plaintexts p_0, p_1 to the challenger of indistinguishability of ciphertexts game (note that plaintexts belong to both punctured sets and thus this is a valid query to the challenger of indistinguishability of ciphertexts game). The challenger samples keys EK, DK and it sends back to the reduction $\mathsf{EK}\{p_0, p_1\}$, $\mathsf{DK}\{p_0, p_1\}$, and ${\mu_3}^*$ which is either $\mathsf{Enc_{EK}}(p_0)$ or $\mathsf{Enc_{EK}}(p_1)$.

Next the reduction punctures keys $\mathsf{EK}_S\{P_{\ell_0^*}\}$, $\mathsf{DK}_S\{P_{\ell_0^*}\}$, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, and sets $s' = \mathsf{Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' =$

$\mathsf{Enc}_{\mathsf{EK}_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*.

Then the reduction uses variables and code created above to construct and obfuscate programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$ (fig. 106) and $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$ (fig. 109). It gives obfuscated programs to the adversary, together with $s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*$. If challenge ${\mu_3}^*$ was $\mathsf{Enc}_{\mathsf{EK}}(p_0)$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{C,2,3}$. If ${\mu_3}^*$ was $\mathsf{Enc}_{\mathsf{EK}}(p_1)$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{C,2,4}$. □

Lemma 90. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that ${\mu_1}^*$ is outside the image of the PRF SG, and ${\mu_2}^*$ is outside the image of the PRF RG. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{C,2,4}$ and $\mathsf{Hyb}_{C,2,5}$, then there exists an adversary which can break security of iO for σ'-sized circuits in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\frac{1}{2}\varepsilon(\lambda)$.*

Proof. The proof is very similar to the proof of lemma 88, except that in this hybrid ${\mu_3}^* = \mathsf{Enc}_{\mathsf{EK}}(p_1)$ instead of p_0, and we unpuncture DK at p_1 instead of p_0.

We start with analyzing program Dec: The only difference between programs $\mathsf{Dec}_{C,3}$ and $\mathsf{Dec}_{C,2}$ is that $\mathsf{Dec}_{C,3}$ uses key $\mathsf{DK}\{p_0\}$ and $\mathsf{Dec}_{C,2}$ uses $\mathsf{DK}\{p_0, p_1\}$, i.e. the key is additionally punctured at p_1 (here $p_0 = (m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $p_1 = (m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$). We will argue that if $\mathsf{Dec}_{C,3}$ on input ${\mu_3}^* = \mathsf{Enc}_{\mathsf{EK}}(p_1)$ reaches the line where it needs to decrypt ${\mu_3}^*$, then it always outputs $'\mathsf{fail}'$. Therefore puncturing this point (and thus forcing $\mathsf{Dec}_{C,2}$ to output $'\mathsf{fail}'$ when attempt to decrypt ${\mu_3}^*$) doesn't change the functionality:

First, note that if input $\mu_3 = {\mu_3}^*$, but $(\mu_1, \mu_2) \neq ({\mu_1}^*, {\mu_2}^*)$ and the program reached decryption of ${\mu_3}^*$, then the program outputs $'\mathsf{fail}'$: indeed, ${\mu_3}^*$ encrypts ${\mu_1}^*, {\mu_2}^*$ and thus the check $(\mu_1, \mu_2) = ({\mu_1}^*, {\mu_2}^*)$ will not pass.

Second, since ${\mu_2}^*$ is random, it is outside of the image of a PRF RG with overwhelming probability, and thus validity check can pass only if $\mathsf{Dec}_{C,3}$ is run on some $(r, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where r encodes ${\mu_1}^*, {\mu_2}^*$ (and other values). However, note that $\mathsf{Dec}_{C,3}$ on such input can only execute trapdoor step (and not the main step); thus the key in the main step can be safely punctured. Further, in order for the program to output m after decryption in the trapdoor step, the condition "$\mathsf{isLess}(L', L'')$" should hold. However, when input $\mu_3 = {\mu_3}^*$, L'' is equal to $[0, {\mu_1}^*, {\mu_2}^*]$, which is the smallest possible level and therefore there doesn't exist L' such that $\mathsf{isLess}(L', L'') = \mathsf{true}$. Thus, if $\mathsf{Dec}_{C,3}$ reached decryption in the trapdoor step on input ${\mu_3}^*$, it will anyway output $'\mathsf{fail}'$ due to failed "isLess" check and therefore we can puncture DK at p_1 such that an attempt to decrypt ${\mu_3}^*$ would cause Dec to output $'\mathsf{fail}'$ immediately.

Next we analyze program RFake. The difference between $\mathsf{RFake}_{C,3}$ and $\mathsf{RFake}_{C,2}$ is that the key DK, which is already punctured at p_0, is additionally punctured at p_1. In order to preserve the functionality of RFake on input ${\mu_3}^*$, we additionally instruct RFake to use level $L_0^* = [0, {\mu_1}^*, {\mu_2}^*]$ on input ${\mu_3}^*$ (without actually decrypting ${\mu_3}^*$). Note that this is what $\mathsf{RFake}_{C,3}$ would do on input ${\mu_3}^*$; thus this doesn't change the functionality. □

Lemma 91. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that ${\mu_1}^*$ is outside the image of the PRF SG, and ${\mu_2}^*$ is outside the image of the PRF RG. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{C,2,5}$ and $\mathsf{Hyb}_{C,2,6}$, then there exists an adversary which can break security of constrained decryption of main ACE in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is similar to the proof of indistinguishability of hybrids $\mathsf{Hyb}_{C,2,1}, \mathsf{Hyb}_{C,2,2}$, except that we unpuncture DK at p_0 instead of p_1, and our third message is $\mu_3{}^* = \mathsf{Enc}_{\mathsf{EK}}(p_1)$ instead of $\mu_3{}^* = \mathsf{Enc}_{\mathsf{EK}}(p_0)$.

We give a reduction to security of constrained decryption of main ACE.

The reduction first takes plaintexts m_0^*, m_1^* from the adversary. It samples all keys used in programs (except EK, DK), namely keys $\mathsf{EK}_S, \mathsf{DK}_S$ of sender ACE, keys $\mathsf{EK}_R, \mathsf{DK}_R$ of the receiver ACE, key k_S of the sparse extracting PRF SG of the sender, key k_R of the sparse extracting PRF RG of the receiver. It also runs setup of the level system to create the code of $\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}$.

It chooses random $\mu_1{}^*, \mu_2{}^*$. It computes levels $\ell_0^* = \mathsf{GenZero}(\mu_1{}^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, \mu_2{}^*)$.

Next the reduction sends the set consisting of two points $p_0 = (m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $p_1 = (m_1^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$ to puncture encryption key, sets $p_0, \varnothing$ to puncture decryption key, and plaintext p_1 to the challenger of constrained decryption game (note that plaintext p_1 doesn't belong to the set $\{p_0\}$ for puncturing DK and thus this is a valid query to the challenger of constrained decryption game). The challenger samples keys EK, DK and it sends back to the reduction $\mathsf{EK}\{p_0, p_1\}$, key which is either $\mathsf{DK}\{p_0\}$ or $\mathsf{DK}\{\varnothing\}$, and $\mu_3{}^* = \mathsf{Enc}_{\mathsf{EK}}(p_1)$.

Next the reduction punctures keys $\mathsf{EK}_S\{P_{\ell_0^*}\}$, $\mathsf{DK}_S\{P_{\ell_0^*}\}$, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, and sets $s' = \mathsf{Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$, $r' = \mathsf{Enc}_{\mathsf{EK}_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*.

Then the reduction uses variables and code created above to construct and obfuscate programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$ (fig. 106) and $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$ (fig. 110, fig. 107). It gives obfuscated programs to the adversary, together with $s', r', \mu_1{}^*, \mu_2{}^*, \mu_3{}^*$. If challenge key was $\mathsf{DK}\{\varnothing\}$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{C,2,6}$. If key was $\mathsf{DK}_S\{p_0\}$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{C,2,5}$. □

Lemma 92. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that $\mu_1{}^*$ is outside the image of the PRF SG, and $\mu_2{}^*$ is outside the image of the PRF RG. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{C,2,6}$ and $\mathsf{Hyb}_{C,2,7}$, then there exists an adversary which can break security of iO for σ'-sized programs in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is identical to the proof of lemma 86. □

Lemma 93. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that $\mu_1{}^*$ is outside the image of the PRF SG. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{C,2,7}$ and $\mathsf{Hyb}_{C,3,1}$, then there exists an adversary which can break the strong computational extractor property of a PRF RG_{k_R} in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is identical to the proof of indistinguishability of hybrids $\mathsf{Hyb}_{C,1,4}, \mathsf{Hyb}_{C,1,5}$, except that $\mu_3{}^* = \mathsf{Enc}_{\mathsf{EK}}(m_1^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$ instead of $\mu_3{}^* = \mathsf{Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$. □

Lemma 94. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{C,3,1}$ and $\mathsf{Hyb}_{C,3,2}$, then there exists an adversary which can break the strong computational extractor property of a PRF SG_{k_S} in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is similar to the proof of indistinguishability of hybrids $\mathsf{Hyb}_{C,1,3}, \mathsf{Hyb}_{C,1,4}$ (with the difference that $\mu_3{}^* = \mathsf{Enc}_{\mathsf{EK}}(m_1^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$ instead of $\mu_3{}^* = \mathsf{Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, and the reduction is made for the point (s^*, m_1^*) instead of (s^*, m_0^*)). □

Lemma 95. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{C,3,2}$ and $\mathsf{Hyb}_{C,3,3}$, then there exists an adversary which can break security of contrained decryption of a sender-fake relaxed ACE in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is similar to the proof of indistinguishability of hybrids $\mathsf{Hyb}_{C,1,2}, \mathsf{Hyb}_{C,1,3}$ (with the difference that $\mu_3{}^* = \mathsf{Enc}_{\mathsf{EK}}(m_1^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$ instead of $\mu_3{}^* = \mathsf{Enc}_{\mathsf{EK}}(m_0^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, and $\mu_1{}^* = \mathsf{SG}(s^*, m_1^*)$ instead of $\mu_1{}^* = \mathsf{SG}(s^*, m_0^*)$). □

Lemma 96. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{C,3,3}$ and $\mathsf{Hyb}_{C,3,4}$, then there exists an adversary which can break security of iO for σ'-sized programs in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The only difference between programs SFake and $\mathsf{SFake}_{C,1}$ is that $\mathsf{SFake}_{C,1}$ uses a punctured key $\mathsf{EK}_S\{P_{\ell_0^*}\}$, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. This is without changing functionality, since SFake never needs to encrypt a plaintext with level ℓ_0^*, since $\ell_0^* = [0, \mu_1{}^*]$ and SFake encrypts levels with value at least 1. □

Finally, we note that the distributions in $\mathsf{Hyb}_{C,3,4}$ and Hyb_D are $O(2^{-\lambda})$-close (the reasoning is similar to distributions $\mathsf{Hyb}_B, \mathsf{Hyb}_{B,1,1}$).

8.2.4 Reductions in the proof of lemma 57 (Indistinguishability of Levels)

Let $t(\lambda)$ be any function in $\Omega(\mathsf{poly}(\lambda))$, and let $\varepsilon(\lambda)$ be a negligible function in $w(2^{-\lambda})$. Assuming the sender-fake relaxed ACE, sparse extracting puncturable PRF, and iO for program size σ' are $(t(\lambda), \varepsilon(\lambda))$-secure, and assuming the level system is $(t(\lambda), \varepsilon_1(\lambda, T, \tau))$-secure, we show that no time-$t(\lambda)$ adversary can distinguish between Hyb_D and Hyb_E with more than $O(\varepsilon(\lambda)) + \varepsilon_1(\lambda, T, \tau)$ advantage.

(Note that security loss $O(2^{-\tau(\lambda)})$ comes from conditioning on the fact that $\mu_1{}^*$ is outside of the image of the corresponding PRF. Conditioning on s^*, r^* incurs only $2^{-\lambda}$ loss and therefore we omit it.).

Lemma 97. *Statistical distance between distributions $\mathsf{Hyb}_D, \mathsf{Hyb}_{D,1,1}$ is at most $2 \cdot 2^{-\lambda}$.*

Proof. Same as indistinguishability between hybrids $\mathsf{Hyb}_B, \mathsf{Hyb}_{B,1,1}$. □

Lemma 98. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{D,1,1}$ and $\mathsf{Hyb}_{D,1,2}$, then there exists an adversary which can break security of iO for σ'-sized programs in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The only difference between programs SFake and $\mathsf{SFake}_{D,1}$ is that $\mathsf{SFake}_{D,1}$ uses a punctured key $\mathsf{EK}_S\{P_{\ell_0^*}\}$, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_0^*)$. This is without changing functionality, since SFake never needs to encrypt a plaintext with level ℓ_0^*, since $\ell_0^* = [0, \mu_1{}^*]$ and SFake encrypts levels with value at least 1. □

Lemma 99. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{D,1,2}$ and $\mathsf{Hyb}_{D,1,3}$, then there exists an adversary which can break security of constrained decryption of sender-fake relaxed ACE in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is similar to the proof of indistinguishability of hybrids $\mathsf{Hyb}_{B,1,2}, \mathsf{Hyb}_{B,1,3}$ (with the difference that r' instead of r^* is given to the adversary, and ${\mu_3}^* = \mathsf{Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$ instead of ${\mu_3}^* = \mathsf{Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, ${\mu_1}^* = \mathsf{SG}(s^*, m_1^*)$ instead of ${\mu_1}^* = \mathsf{SG}(s^*, m_0^*)$). □

Lemma 100. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{D,1,3}$ and $\mathsf{Hyb}_{D,1,4}$, then there exists an adversary which can break computational strong extractor property of the PRF SG in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is similar to the proof of indistinguishability of hybrids $\mathsf{Hyb}_{B,1,3}, \mathsf{Hyb}_{B,1,4}$ (with the difference that r' (instead of r^*) is given to the adversary, ${\mu_3}^* = \mathsf{Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$ instead of ${\mu_3}^* = \mathsf{Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, and the reduction is made for the point (s^*, m_1^*) instead of (s^*, m_0^*)). □

Lemma 101. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that ${\mu_1}^*$ is outside the image of the PRF SG. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{D,1,4}$ and $\mathsf{Hyb}_{D,2,1}$, then there exists an adversary which can break security of iO for σ'-sized programs in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\frac{1}{2}\varepsilon(\lambda)$.*

Proof. The difference between programs in the two hybrids is that in $\mathsf{Hyb}_{D,2,1}$ programs use only punctured versions of programs of the level system. We argue that this doesn't change the functionality of the programs of deniable encryption, since these programs never need to call programs of the level system on punctured inputs.

We start with analyzing program $\mathsf{P3}_{D,2}$. By assumption, ${\mu_1}^*$ is outside of the image of a PRF SG, and thus when $\mu_1 = {\mu_1}^*$ validity check can pass only if $\mathsf{P3}$ is run on some $(s, m, {\mu_1}^*, \mu_2)$, where s encodes $m, {\mu_1}^*$ (and other values). However, note that $\mathsf{P3}_{D,2}$ on such input can only execute trapdoor step (and not the main step); thus in the main step we can use $\mathsf{GenZero}[{\mu_1}^*]$ which is punctured at ${\mu_1}^*$. Moreover, since $\mathsf{GenZero}[{\mu_1}^*]$ never outputs ℓ_0^*, we can also use $\mathsf{Transform}[(\ell_0^*, {\mu_2}^*)]$ which is punctured at the input $(\ell_0^*, {\mu_2}^*)$.

It remains to argue that we can puncture $\mathsf{Transform}[(\ell_0^*, {\mu_2}^*)]$ at the input $(\ell_0^*, {\mu_2}^*)$ in the trapdoor step as well. Note that in order to run $\mathsf{Transform}$ on this input in the trapdoor step, $\mathsf{P3}$ should take as input fake s which encodes ℓ_0^* (among other things). However, since DK_S is punctured at $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, the only such fake s is $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, that is, s'. Further, in order for "$(m, \mu_1) = (m', {\mu_1}')$" check to pass, inputs to $\mathsf{P3}$ should be $m = m_0^*$ and $\mu_1 = {\mu_1}^*$. Finally, in order to call $\mathsf{Transform}$ on $(\ell_0^*, {\mu_2}^*)$, the input μ_2 to $\mathsf{P3}$ should be ${\mu_2}^*$. In other words, the only input on which $\mathsf{P3}$ could potentially run $\mathsf{Transform}$ at the punctured point is $(s', m_0^*, {\mu_1}^*, {\mu_2}^*)$; however, in this case $\mathsf{P3}$ simply outputs ${\mu_3}^*$, which is encoded in s', without running $\mathsf{Transform}$ at all. Thus we can puncture $\mathsf{Transform}$ safely.

Next we analyze program $\mathsf{SFake}_{D,2}$. By assumption, ${\mu_1}^*$ is is outside of the image of a PRF SG, and thus validity check can pass only if SFake is run on some $(s, m, \hat{m}, {\mu_1}^*, \mu_2, \mu_3)$, where s encodes $m, {\mu_1}^*$ (and other values). However, note that $\mathsf{SFake}_{D,2}$ on such input can only execute trapdoor step (and not the main step); thus in the main step we can use $\mathsf{GenZero}[\mu_1{}^*]$ which is punctured at $\mu_1{}^*$. $\square$

Lemma 102. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that $\mu_1{}^*$ is outside the image of the PRF SG. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{D,2,1}$ and $\mathsf{Hyb}_{D,2,2}$, then there exists an adversary which can break security of the level system with an upper bound T and tag size τ in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. We give a reduction to security of the level system.

The reduction first takes plaintexts m_0^*, m_1^* from the adversary. It samples all keys used in programs, namely keys EK, DK of the main ACE, keys $\mathsf{EK}_S, \mathsf{DK}_S$ of the sender ACE, keys $\mathsf{EK}_R, \mathsf{DK}_R$ of the receiver ACE, key k_S of the sparse extracting PRF SG of the sender, key k_R of the sparse extracting PRF RG of the receiver.

It chooses random r^* and $\mu_1{}^*$ and computes $\mu_2{}^* = \mathsf{RG}_{k_R}(r^*, \mu_1{}^*)$. It sends $\mu_1{}^*, \mu_2{}^*$ as the first and the second tag to the challenger of the level system. The challenger chooses bit b at random and runs setup of the level system to obtain programs $\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}$. Then it computes $\ell_0^* = \mathsf{GenZero}(\mu_1{}^*)$, $\ell_1^* = \mathsf{Increment}(\ell_0^*)$, and $L_0^* = \mathsf{Transform}(\ell_0^*, \mu_2{}^*)$. It also obfuscates punctured programs $\mathsf{GenZero}[\mu_1{}^*], \mathsf{Increment}, \mathsf{Transform}[(\ell_b^*, \mu_2{}^*)], \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}$. It sends these obfuscated punctured programs to the reduction, together with ℓ_b^* and L_0^*.

The reduction computes $\mu_3{}^* = \mathsf{Enc}_{\mathsf{EK}}(m_1^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$, $s' = \mathsf{Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_b^*)$, and $r' = \mathsf{Enc}_{\mathsf{EK}_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*.

Next the reduction punctures keys $\mathsf{EK}_S\{P_{\ell_b^*}\}$, $\mathsf{DK}_S\{P_{\ell_b^*}\}$ at the set $P_{\ell_b^*} = \{(*, *, *, *, \ell_b^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_b^*)$.

Then the reduction uses variables and code obtained from the challenger to construct and obfuscate programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$, (fig. 114, fig. 115) and $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$ (fig. 19). It gives obfuscated programs to the adversary, together with $s', r', \mu_1{}^*, \mu_2{}^*, \mu_3{}^*$. If challenge bit b is 0, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{D,2,1}$. If b is 1, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{D,2,2}$. $\square$

Lemma 103. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that $\mu_1{}^*$ is outside the image of the PRF SG. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{D,2,2}$ and $\mathsf{Hyb}_{D,2,3}$, then there exists an adversary which can break security of iO for σ'-sized programs in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\frac{1}{2}\varepsilon(\lambda)$.*

Proof. This proof is very similar to the proof of lemma 101, except that $\mathsf{Transform}$ is punctured at $(\ell_1^*, \mu_2{}^*)$ instead of $(\ell_0^*, \mu_2{}^*)$.

The difference between programs in $\mathsf{Hyb}_{D,2,2}, \mathsf{Hyb}_{D,2,3}$ is that in $\mathsf{Hyb}_{D,2,2}$ programs use only punctured versions of programs of the level system. We argue that this doesn't change the functionality of the programs of deniable encryption, since these programs never need to call programs of the level system on punctured inputs.

We start with analyzing program $\mathsf{P3}_{D,4}$. By assumption $\mu_1{}^*$ is outside of the image of a PRF SG, and thus when $\mu_1 = \mu_1{}^*$ validity check can pass only if $\mathsf{P3}$ is run on some $(s, m, \mu_1{}^*, \mu_2)$, where s encodes $m, \mu_1{}^*$ (and other values). However, note that $\mathsf{P3}_{D,4}$ on such input can only execute trapdoor step (and not the main step); thus in the main step we can use $\mathsf{GenZero}[{\mu_1}^*]$ which is punctured at ${\mu_1}^*$. Moreover, since $\mathsf{GenZero}[{\mu_1}^*]$ never outputs ℓ_1^*, we can also use $\mathsf{Transform}[(\ell_1^*, {\mu_2}^*)]$ which is punctured at the input $(\ell_1^*, {\mu_2}^*)$.

It remains to argue that we can puncture $\mathsf{Transform}[(\ell_1^*, {\mu_2}^*)]$ at the input $(\ell_1^*, {\mu_2}^*)$ in the trapdoor step as well. Note that in order to run Transform on this input in the trapdoor step, $\mathsf{P3}_{D,5}$ should take as input fake s which encodes ℓ_1^* (among other things). However, since DK_S is punctured at $P_{\ell_1^*} = \{(*,*,*,*,\ell_1^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, the only such fake s is $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, that is, s'. Further, in order for "$(m, \mu_1) = (m', {\mu_1}')$" check to pass, inputs to P3 should be $m = m_0^*$ and $\mu_1 = {\mu_1}^*$. Finally, in order to call Transform on $(\ell_1^*, {\mu_2}^*)$, the input μ_2 to P3 should be ${\mu_2}^*$. In other words, the only input on which P3 could potentially run Transform at the punctured point is $(s', m_0^*, {\mu_1}^*, {\mu_2}^*)$; however, in this case P3 simply outputs ${\mu_3}^*$, which is encoded in s', without running Transform at all. Thus we can puncture Transform safely.

Next we analyze program $\mathsf{SFake}_{D,4}$. Since ${\mu_1}^*$ is outside of the image of a PRF SG, and thus validity check can pass only if SFake is run on some $(s, m, \hat{m}, {\mu_1}^*, \mu_2, \mu_3)$, where s encodes $m, {\mu_1}^*$ (and other values). However, note that $\mathsf{SFake}_{D,4}$ on such input can only execute trapdoor step (and not the main step); thus in the main step we can use $\mathsf{GenZero}[{\mu_1}^*]$ which is punctured at ${\mu_1}^*$. □

Lemma 104. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that ${\mu_1}^*$ is outside the image of the PRF* SG. *Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish* $\mathsf{Hyb}_{D,2,3}$ *and* $\mathsf{Hyb}_{D,3,1}$, *then there exists an adversary which can break security of* iO *for σ'-sized programs in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The only difference between programs $\mathsf{SFake}_{D,5}$ and $\mathsf{SFake}_{D,6}$ is that in $\mathsf{SFake}_{D,6}$ the key EK_S is also punctured at $P_{\ell_0^*}$, where $P_{\ell_0^*} = \{(*,*,*,*,\ell_0^*)\}$ (in addition to being punctured at $P_{\ell_1^*} = \{(*,*,*,*,\ell_1^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$). This is without changing functionality, since SFake never needs to encrypt a plaintext with level ℓ_0^*, since $\ell_0^* = [0, {\mu_1}^*]$ and SFake encrypts levels with value at least 1. □

Lemma 105. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that ${\mu_1}^*$ is outside the image of the PRF* SG. *Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish* $\mathsf{Hyb}_{D,3,1}$ *and* $\mathsf{Hyb}_{D,3,2}$, *then there exists an adversary which can break security of constrained decryption of sender-fake relaxed ACE in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is similar to the proof of indistinguishability of hybrids $\mathsf{Hyb}_{D,1,2}, \mathsf{Hyb}_{D,1,3}$, except that ℓ_1^* instead of ℓ_0^* is used in the distribution, and keys EK, DK are additionally punctured at the set $P_{\ell_1^*} = \{(*,*,*,*,\ell_1^*)\} \setminus (m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$.

We give a reduction to security of constrained decryption of sender ACE.

The reduction first takes plaintexts m_0^*, m_1^* from the adversary. It samples all keys used in programs (except $\mathsf{EK}_S, \mathsf{DK}_S$), namely keys EK, DK of the main ACE, keys $\mathsf{EK}_R, \mathsf{DK}_R$ of the receiver ACE, key k_S of the sparse extracting PRF SG of the sender, key k_R of the sparse extracting PRF RG of the receiver. It also runs setup of the level system to create the code of GenZero, Increment, Transform, isLess, RetrieveTag, RetrieveTags.

It chooses random r^* and ${\mu_1}^*$ and computes ${\mu_2}^* = \mathsf{RG}_{k_R}(r^*, {\mu_1}^*)$. It computes levels $\ell_0^* = \mathsf{GenZero}({\mu_1}^*)$, $\ell_1^* = \mathsf{Increment}(\ell_0^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, {\mu_2}^*)$. It sets ${\mu_3}^* = \mathsf{Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$.

Next the reduction sends the set $P_{\ell_0^*} \cup P_{\ell_1^*}$ as a set to puncture encryption key (where $P_{\ell_0^*} = \{(*,*,*,*,\ell_0^*)\}$, $P_{\ell_1^*} = \{(*,*,*,*,\ell_1^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_1^*)$), and sends sets $P_{\ell_1^*}$ and $P_{\ell_0^*} \cup P_{\ell_1^*}$ as sets to puncture decryption key to the challenger of constrained decryption game. The challenger samples keys $\mathsf{EK}_S, \mathsf{DK}_S$ and it sends back to the reduction $\mathsf{EK}_S\{P_{\ell_0^*} \cup P_{\ell_1^*}\}$ and key which is either $\mathsf{DK}_S\{P_{\ell_1^*}\}$ or $\mathsf{DK}_S\{P_{\ell_0^*} \cup P_{\ell_1^*}\}$.

Next the reduction computes $s' = \mathsf{Enc}_{\mathsf{EK}_S\{P_{\ell_0^*} \cup P_{\ell_1^*}\}}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_1^*)$ (note that this point is not punctured) and $r' = \mathsf{Enc}_{\mathsf{EK}_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*.

Then the reduction uses variables and code created above to construct and obfuscate programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$, (fig. 117, fig. 118) and $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$ (fig. 19). It gives obfuscated programs to the adversary, together with $s', r', \mu_1{}^*, \mu_2{}^*, \mu_3{}^*$. If challenge key is $\mathsf{DK}_S\{P_{\ell_0^*} \cup P_{\ell_1^*}\}$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{D,3,2}$. If key is $\mathsf{DK}_S\{P_{\ell_1^*}\}$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{D,3,1}$. □

Lemma 106. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that $\mu_1{}^*$ is outside the image of the PRF* SG*. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish* $\mathsf{Hyb}_{D,3,2}$ *and* $\mathsf{Hyb}_{D,3,3}$*, then there exists an adversary which can break security of constrained decryption of sender-fake relaxed ACE in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is similar to the proof of indistinguishability of hybrids $\mathsf{Hyb}_{D,3,1}, \mathsf{Hyb}_{D,3,2}$, except that we unpuncture DK at the set $P_{\ell_1^*} = \{(*,*,*,*,\ell_1^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_1^*)$ instead of $P_{\ell_0^*} = \{(*,*,*,*,\ell_0^*)\}$. We give a reduction to security of constrained decryption of sender ACE.

The reduction first takes plaintexts m_0^*, m_1^* from the adversary. It samples all keys used in programs (except $\mathsf{EK}_S, \mathsf{DK}_S$), namely keys EK, DK of the main ACE, keys $\mathsf{EK}_R, \mathsf{DK}_R$ of the receiver ACE, key k_S of the sparse extracting PRF SG of the sender, key k_R of the sparse extracting PRF RG of the receiver. It also runs setup of the level system to create the code of $\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}$.

It chooses random r^* and $\mu_1{}^*$ and computes $\mu_2{}^* = \mathsf{RG}_{k_R}(r^*, \mu_1{}^*)$. It computes levels $\ell_0^* = \mathsf{GenZero}(\mu_1{}^*)$, $\ell_1^* = \mathsf{Increment}(\ell_0^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, \mu_2{}^*)$. It sets $\mu_3{}^* = \mathsf{Enc}_{\mathsf{EK}}(m_1^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$.

Next the reduction sends the set $P_{\ell_0^*} \cup P_{\ell_1^*}$ as a set to puncture encryption key (where $P_{\ell_0^*} = \{(*,*,*,*,\ell_0^*)\}$, $P_{\ell_1^*} = \{(*,*,*,*,\ell_1^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_1^*)$), and sends sets $P_{\ell_0^*}$ and $P_{\ell_0^*} \cup P_{\ell_1^*}$ as sets to puncture decryption key to the challenger of constrained decryption game. The challenger samples keys $\mathsf{EK}_S, \mathsf{DK}_S$ and it sends back to the reduction $\mathsf{EK}_S\{P_{\ell_0^*} \cup P_{\ell_1^*}\}$ and key which is either $\mathsf{DK}_S\{P_{\ell_0^*}\}$ or $\mathsf{DK}_S\{P_{\ell_0^*} \cup P_{\ell_1^*}\}$.

Next the reduction computes $s' = \mathsf{Enc}_{\mathsf{EK}_S\{P_{\ell_0^*} \cup P_{\ell_1^*}\}}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_1^*)$ (note that this point is not punctured) and $r' = \mathsf{Enc}_{\mathsf{EK}_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*.

Then the reduction uses variables and code created above to construct and obfuscate programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$, (fig. 118, fig. 119) and $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$ (fig. 19). It gives obfuscated programs to the adversary, together with $s', r', \mu_1{}^*, \mu_2{}^*, \mu_3{}^*$. If challenge key is $\mathsf{DK}_S\{P_{\ell_0^*} \cup P_{\ell_1^*}\}$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{D,3,2}$. If key is $\mathsf{DK}_S\{P_{\ell_0^*}\}$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{D,3,3}$. □

Lemma 107. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that $\mu_1{}^*$ is outside the image of the PRF* SG*. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish* $\mathsf{Hyb}_{D,3,3}$ *and* $\mathsf{Hyb}_{D,3,4}$*, then there exists an adversary which can break security of* iO *for σ'-sized programs in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The only difference between programs $\mathsf{SFake}_{D,8}$ and $\mathsf{SFake}_{D,9}$ is that in $\mathsf{SFake}_{D,8}$ the key EK_S is also punctured at $P_{\ell_1^*}$, where $P_{\ell_1^*} = \{(*, *, *, *, \ell_1^*)\} \setminus (m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_1^*)$ (in addition to being punctured at $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$). We argue that this is without changing functionality:

First, note that the trapdoor step never needs to encrypt the plaintext with ℓ_1^*: for that SFake would need to get as input some fake s which encodes ℓ_0^*, but such fake s doesn't exist since DK_S is punctured on the whole set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$.

Second, in order to encrypt ℓ_1^* in the main step, $\mathsf{SFake}_{D,9}$ should get $\mu_1{}^*$ as input. However, in order to pass validity check with $\mu_1{}^*$ (which is outside of the image of PRF SG), $\mathsf{SFake}_{D,9}$ should get as input some $(s, m, \hat{m}, \mu_1{}^*, \mu_2, \mu_3)$, where s is fake and encodes $(m, \mu_1{}^*)$ (among other things). But on such input $\mathsf{SFake}_{D,9}$ never executes the main step - it executes the trapdoor step. Thus we can additionally puncture EK at $P_{\ell_1^*}$ in the main step. □

Lemma 108. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that $\mu_1{}^*$ is outside the image of the PRF SG. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{D,3,4}$ and $\mathsf{Hyb}_{D,3,5}$, then there exists an adversary which can break security of constrained decryption of sender-fake relaxed ACE in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is similar to the proof of indistinguishability of hybrids $\mathsf{Hyb}_{D,3,2}, \mathsf{Hyb}_{D,3,3}$, except that EK_S, DK_S are punctured at different sets. We give a reduction to security of constrained decryption of sender ACE.

The reduction first takes plaintexts m_0^*, m_1^* from the adversary. It samples all keys used in programs (except $\mathsf{EK}_S, \mathsf{DK}_S$), namely keys EK, DK of the main ACE, keys $\mathsf{EK}_R, \mathsf{DK}_R$ of the receiver ACE, key k_S of the sparse extracting PRF SG of the sender, key k_R of the sparse extracting PRF RG of the receiver. It also runs setup of the level system to create the code of $\mathsf{GenZero}, \mathsf{Increment}, \mathsf{Transform}, \mathsf{isLess}, \mathsf{RetrieveTag}, \mathsf{RetrieveTags}$.

It chooses random r^* and $\mu_1{}^*$ and computes $\mu_2{}^* = \mathsf{RG}_{k_R}(r^*, \mu_1{}^*)$. It computes levels $\ell_0^* = \mathsf{GenZero}(\mu_1{}^*)$, $\ell_1^* = \mathsf{Increment}(\ell_0^*)$, $L_0^* = \mathsf{Transform}(\ell_0^*, \mu_2{}^*)$. It sets $\mu_3{}^* = \mathsf{Enc}_{\mathsf{EK}}(m_1^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$.

Next the reduction sends the set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$ as a set to puncture encryption key, and sends sets $P_{\ell_0^*}$ and $\varnothing$ as sets to puncture decryption key to the challenger of constrained decryption game. The challenger samples keys $\mathsf{EK}_S, \mathsf{DK}_S$ and it sends back to the reduction $\mathsf{EK}_S\{P_{\ell_0^*}\}$ and key which is either $\mathsf{DK}_S\{P_{\ell_0^*}\}$ or $\mathsf{DK}_S\{\varnothing\}$.

Next the reduction computes $s' = \mathsf{Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_1^*)$ (note that this point is not punctured) and $r' = \mathsf{Enc}_{\mathsf{EK}_R}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*.

Then the reduction uses variables and code created above to construct and obfuscate programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$, (fig. 118, fig. 119) and $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$ (fig. 19). It gives obfuscated programs to the adversary, together with $s', r', \mu_1{}^*, \mu_2{}^*, \mu_3{}^*$. If challenge key is $\mathsf{DK}_S\{P_{\ell_0^*}\}$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{D,3,4}$. If key is $\mathsf{DK}_S\{\varnothing\}$, then the resulting distribution is exactly the distribution from $\mathsf{Hyb}_{D,3,5}$. □

Lemma 109. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Further, assume that $\mu_1{}^*$ is outside the image of the PRF SG. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{D,3,5}$ and $\mathsf{Hyb}_{D,3,6}$, then there exists an adversary which can break security of iO for σ'-sized programs in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The only difference between programs $\mathsf{SFake}_{D,9}$ and $\mathsf{SFake}_{D,10}$ is that in $\mathsf{SFake}_{D,9}$ the key EK_S is punctured at $P_{\ell_0^*}$, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$. This is without changing functionality, since SFake never needs to encrypt a plaintext with level ℓ_0^*, since $\ell_0^* = [0, \mu_1{}^*]$ and SFake encrypts levels with value at least 1. □

Lemma 110. *Assume s^*, r^* are outside of the image of the sender-fake ACE and the receiver-fake ACE, respectively. Then, if there exists an adversary which can $(t(\lambda), \varepsilon(\lambda))$-distinguish $\mathsf{Hyb}_{D,3,6}$ and $\mathsf{Hyb}_{D,3,7}$, then there exists an adversary which can break computational strong extractor property of the PRF SG in time $t(\lambda) + \mathsf{poly}(\lambda)$ with distinguishing advantage $\varepsilon(\lambda)$.*

Proof. The proof is identical to the proof of indistinguishability of hybrids $\mathsf{Hyb}_{D,1,3}, \mathsf{Hyb}_{D,1,4}$, except that fake s' is computed using level ℓ_1^* instead of ℓ_0^*. □

Finally, we note that the distributions in $\mathsf{Hyb}_{D,3,7}$ and Hyb_E are $O(2^{-\lambda})$-close (the reasoning is similar to distributions $\mathsf{Hyb}_B, \mathsf{Hyb}_{B,1,1}$).

9 Proof of off-the-record deniability of our encryption protocol

In this section we show that our scheme also satisfies off-the-record property, which says that the adversary who gets contradicting claims from parties (that is, the sender claims that the plaintext was m_0^* and shows consistent randomness, but the receiver claims that the plaintext was m_1^* and also shows consistent randomness) cannot tell which party is lying (if not both) and which plaintext was actually sent. In other words, neither party can prove which plaintext was used in the protocol. We underline however that this property only holds as long as parties act honestly during the protocol: indeed, a malicious party can always choose its randomness as a result of a prg and provide the seed of this prg as a proof that its randomness is genuine.

Recall the definition of off-the-record deniability states that the following three distributions are computationally indistinguishable:

- **the sender claims m_0^* was sent, the receiver claims m_1^* was sent, the plaintext was m_0^* :** $(\mathsf{PP}, m_0^*, m_1^*, m_2^*, s^*, r', \mathsf{tr}(s^*, r^*, m_0^*))$, where s^*, r^* are randomly chosen, $r' = \mathsf{RFake}(m_1^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*; \rho^*)$ for randomly chosen ρ^*, and $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}.
- **the sender claims m_0^* was sent, the receiver claims m_1^* was sent, the plaintext was m_1^* :** $(\mathsf{PP}, m_0^*, m_1^*, m_2^*, s', r^*, \mathsf{tr}(s^*, r^*, m_1^*))$, where s^*, r^* are randomly chosen, $s' = \mathsf{SFake}(s^*, m_1^*, m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, and $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}.
- **the sender claims m_0^* was sent, the receiver claims m_1^* was sent, the plaintext was m_2^* :** $(\mathsf{PP}, m_0^*, m_1^*, m_2^*, s', r', \mathsf{tr}(s^*, r^*, m_2^*))$, where s^*, r^* are randomly chosen, $s' = \mathsf{SFake}(s^*, m_2^*, m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$, $r' = \mathsf{RFake}(m_1^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*; \rho^*)$ for randomly chosen ρ^*, and $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}.

Note that the first distribution is the same as the following distribution, since $\mathsf{RFake}(m_1^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*; \rho^*)$ outputs $\mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_1^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$:

$\mathsf{Hyb}_A = (\mathsf{PP}, m_0^*, m_1^*, s^*, r', \mathsf{tr}(s^*, r^*, m_0^*))$, where s^*, r^* are randomly chosen, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_1^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*, and $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}.

Further, note that the second distribution is statistically close to the following distribution, since $\mathsf{SFake}(s^*, m_1^*, m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*)$ with overwhelming probability over the choice of s^* outputs $\mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_1^*)$:

$\mathsf{Hyb}_E = (\mathsf{PP}, m_0^*, m_1^*, s', r^*, \mathsf{tr}(s^*, r^*, m_1^*))$, where s^*, r^* are randomly chosen, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_1^*)$, and $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}.

Finally, note that the third distribution is statistically close to the following distribution:

$\mathsf{Hyb}_{D'} = (\mathsf{PP}, m_0^*, m_1^*, m_2^*, s', r', \mathsf{tr}(s^*, r^*, m_2^*))$, where s^*, r^* are randomly chosen, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_1^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*, and $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}.

Thus to prove off-the-record deniability it suffices to show indistinguishability between hybrids Hyb_A, Hyb_E, and $\mathsf{Hyb}_{D'}$. The proof of this statement consists of the same main components as the proof of deniability, albeit in a different order and with slight changes. Below we describe the structure of the proof and comment on the differences with the proof of deniability. Conscretely, we show that $\mathsf{Hyb}_A \approx \mathsf{Hyb}_B \approx \mathsf{Hyb}_C \approx \mathsf{Hyb}_D \approx \mathsf{Hyb}_E$ and that $\mathsf{Hyb}_C \approx \mathsf{Hyb}_{D'}$, where hybrids are as follows:

1. **Indistinguishability of explanations of the sender**: starting from Hyb_A, we switch real s^* to fake s', which encodes plaintext m_0^*, transcript ${\mu_1}^*, {\mu_2}^*, {\mu_3}^*$, and level $\ell^* = [0, {\mu_1}^*]$, moving to the following distribution:

   $\mathsf{Hyb}_B = (\mathsf{PP}, m_0^*, m_1^*, m_2^*, s', r', \mathsf{tr}(s^*, r^*, m_0^*))$, where s^*, r^* are randomly chosen, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*, and $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}.

   The proof of this step is identical to the proof of lemma 54, except that everywhere (in all hybrids and reductions) we additionally generate $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^* and give r' (instead of r^*) to the adversary.

2. **Indistinguishability of levels**: we switch the level encoded in s' from $\ell_0^* = [0, {\mu_1}^*]$ to $\ell_1^* = [1, {\mu_1}^*]$ (while keeping $L_0^* = [0, {\mu_1}^*, {\mu_2}^*]$ the same), moving to the following distribution:

   $\mathsf{Hyb}_C = (\mathsf{PP}, m_0^*, m_1^*, m_2^*, s', r', \mathsf{tr}(s^*, r^*, m_0^*))$, where s^*, r^* are randomly chosen, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^* and $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}.

   The proof of this step is identical to the proof of lemma 57, except that in all hybrids and reductions we generate $r' = \mathsf{RFake}(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*; \rho^*)$ instead of $r' = \mathsf{RFake}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*; \rho^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$ instead of ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, and ${\mu_1}^* = \mathsf{SG}(s^*, m_0^*)$ instead of ${\mu_1}^* = \mathsf{SG}(s^*, m_1^*)$ (except when ${\mu_1}^*$ is randomly chosen).

3. **Semantic security**: we switch the transcript from encrypting m_0^* to encrypting m_1^*, moving to the following distribution:

   $\mathsf{Hyb}_D = (\mathsf{PP}, m_0^*, m_1^*, m_2^*, s', r', \mathsf{tr}(s^*, r^*, m_1^*))$, where s^*, r^* are randomly chosen, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*, and $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}.

   The proof of this step is identical to the proof of lemma 56, except that in all hybrids and reductions we generate $r' = \mathsf{RFake}(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*; \rho^*)$ instead of $r' = \mathsf{RFake}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*; \rho^*)$, for randomly chosen ρ^*, and $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$ instead of $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$.

4. **Indistinguishability of explanations of the receiver**: we switch fake r', which encodes plaintext m_1^*, transcript ${\mu_1}^*, {\mu_2}^*, {\mu_3}^*$, and level $L^* = [0, {\mu_1}^*, {\mu_2}^*]$, to real (randomly chosen) r^*, thus moving to the following distribution:

   $\mathsf{Hyb}_E = (\mathsf{PP}, m_0^*, m_1^*, m_2^*, s', r^*, \mathsf{tr}(s^*, r^*, m_1^*))$, where s^*, r^* are randomly chosen, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, and $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}.

   The proof of this step is very close to the proof of lemma 55, except for a couple of changes. First, we switch the role of m_0^*, m_1^* everywhere (in hybrids and reductions), and we generate s' using level ℓ_1^* instead of ℓ_0^*. However, we still generate $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$ (as opposed to $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$), and we use the set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$ (isntead of

$P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\} \setminus (m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*))$.

For the ease of verification, in the paragraph below we present the list of hybrids proving indistinguishability of Hyb_D and Hyb_E.

Semantic security for plaintext m_2^*: besides showing indistinguishability between Hyb_C and Hyb_D, we also show indistinguishability between Hyb_C and $\mathsf{Hyb}_{D'}$, i.e. we switch the transcript from encrypting m_0^* to encrypting m_2^*, moving from Hyb_C to the following distribution:

$\mathsf{Hyb}_{D'} = (\mathsf{PP}, m_0^*, m_1^*, m_2^*, s', r', \mathsf{tr}(s^*, r^*, m_2^*))$, where s^*, r^* are randomly chosen, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*, and $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}.

The proof of this step is identical to the proof of lemma 56, except that in all hybrids and reductions we generate $r' = \mathsf{RFake}(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*; \rho^*)$ instead of $r' = \mathsf{RFake}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*; \rho^*)$, for randomly chosen ρ^*, and $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$ instead of $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_0^*)$. Also, everywhere in hybrids and reductions we use $p_0 = (m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $p_2 = (m_2^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$ instead of $p_0 = (m_0^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $p_1 = (m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$.

List of hybrids for the proof of indistinguishability of Hyb_D and Hyb_E

Now we present the list of hybrids for the proof of indistinguishability of receiver explanation of off-the-record deniability. We do not present the reductions since they are very similar to the corresponding reductions (section 8.2.2), used for hybrids in section 8.1.2 in the proof of lemma 55. For a more convenient reference to security reductions, we do not change enumeration of hybrids from section 8.1.2, and we keep hybrids in the same order as there (starting from randomly chosen r^*, and moving to fake r').

We also present programs (those which require changes compared to their version in the proof of lemma 55).

List of hybrids. First in a sequence of hybrids we "eliminate" complementary ciphertext $\overline{{\mu_3}^*} = \mathsf{ACE.Enc}_{\mathsf{EK}}(1 \oplus m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, i.e. make programs Dec and SFake reject it:

- $\mathsf{Hyb}_{B,1,1}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, ${\mu_1}^* = \mathsf{SG}(s^*, m_1^*)$, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$. Programs can be found in fig. 96 (programs of the sender) and fig. 100 (programs of the receiver).

  Note that this distribution is exactly the distribution from Hyb_D, conditioned on the fact that s^*, r^* are outside of images of their ACE.

- $\mathsf{Hyb}_{B,1,2}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,1}, \mathsf{P2}, \mathsf{P3}_{B,1}, \mathsf{Dec}, \mathsf{SFake}_{B,1}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, ${\mu_1}^* = \mathsf{SG}(s^*, m_1^*)$, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$. Programs can be found in fig. 97 (programs of the sender) and fig. 100 (programs of the receiver).

  That is, in program SFake we puncture encryption key EK_S of the sender-fake ACE at the set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$. Indistinguishability holds by iO, since this modification doesn't change the functionality of SFake due to the fact that SFake never encrypts plaintexts with level ℓ_0^*.

- $\mathsf{Hyb}_{B,1,3}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} =$

$\mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,2}, \mathsf{P2}, \mathsf{P3}_{B,2}, \mathsf{Dec}, \mathsf{SFake}_{B,2}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, ${\mu_1}^* = \mathsf{SG}(s^*, m_1^*)$, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$. Programs can be found in fig. 98 (programs of the sender) and fig. 100 (programs of the receiver).

That is, in programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$ we puncture decryption key DK_S of the sender-fake ACE at the same set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$. Indistinguishability holds by security of constrained key of ACE, since the corresponding encryption key EK_S is already punctured at the same set.

- $\mathsf{Hyb}_{B,1,4}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,2}, \mathsf{P2}, \mathsf{P3}_{B,2}, \mathsf{Dec}, \mathsf{SFake}_{B,2}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$. Programs can be found in fig. 98 (programs of the sender) and fig. 100 (programs of the receiver).

  That is, we choose ${\mu_1}^*$ at random instead of computing it as ${\mu_1}^* = \mathsf{SG}_{k_S}(s^*, m_1^*)$. Indistinguishability holds by the strong extracting property of the sender PRF SG (note that s^* was not used anywhere else in the distribution).

- $\mathsf{Hyb}_{B,1,5}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,3}, \mathsf{P2}, \mathsf{P3}_{B,3}, \mathsf{Dec}, \mathsf{SFake}_{B,3}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$. Programs can be found in fig. 123 (programs of the sender) and fig. 100 (programs of the receiver).

  That is, in program $\mathsf{P3}$ we puncture encryption key EK of the main ACE at the point $\overline{p} = (1 \oplus m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$. Indistinguishability holds by iO, since $\mathsf{P3}$ never needs to encrypt this point. Roughly, this is because of the following: since ${\mu_1}^*$ is random and outside of the image of a PRF SG, $\mathsf{P3}$ never encrypts $\overline{p}$ in the main step. In order to encrypt it in trapdoor step, $\mathsf{P3}$ needs to take as input some fake s encoding level ℓ_0^*, which doesn't exist due to the fact that DK_S is punctured at the set $P_{\ell_0^*}$.

- $\mathsf{Hyb}_{B,1,6}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,3}, \mathsf{P2}_{B,1}, \mathsf{P3}_{B,3}, \mathsf{Dec}_{B,1}, \mathsf{SFake}_{B,3}, \mathsf{RFake}_{B,1}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$. Programs can be found in fig. 123 (programs of the sender) and fig. 124 (programs of the receiver).

  That is, in programs $\mathsf{Dec}, \mathsf{RFake}$ we puncture decryption key DK of the main ACE at the same point $\overline{p} = (1 \oplus m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$. Indistinguishability holds by security of constrained key of ACE, since the corresponding encryption key EK is already punctured at this point.

Now $\overline{{\mu_3}^*} = \mathsf{ACE.Enc}_{\mathsf{EK}}(1 \oplus m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$ is rejected by Dec and RFake. In the following hybrids, similarly to previous lemma, we switch the roles of r^* and r', using the fact that programs treat them similarly, once $\overline{{\mu_3}^*}$ is eliminated[34].

[34] The problem with $\overline{{\mu_3}^*}$ is that unmodified Dec on input $(r^*, {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*})$ outputs $1 \oplus m_1^*$ (via main step), and on input $(r', {\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*})$ it outputs $'\mathsf{fail}'$ (via trapdoor step, since levels in r' and $\overline{{\mu_3}^*}$ are both 0 and "$\mathsf{isLess} = \mathtt{true}$" check fails. Because of this difference, in $\mathsf{Hyb}_{B,2,1}$ we wouldn't be able to modify program Dec such that the code treats r^* and r' in the same way. However, after $\mathsf{Hyb}_{B,1,6}$ $\overline{{\mu_3}^*}$ is not a valid ciphertext anymore and thus in $\mathsf{Hyb}_{B,2,1}$ we can instruct Dec to output $'\mathsf{fail}'$ on both r^* and r'.

- $\mathsf{Hyb}_{B,2,1}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,3}, \mathsf{P2}_{B,2}, \mathsf{P3}_{B,3}, \mathsf{Dec}_{B,2}, \mathsf{SFake}_{B,3}, \mathsf{RFake}_{B,2}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$. Programs can be found in fig. 123 (programs of the sender) and fig. 125 (programs of the receiver).

  That is, we modify programs of the receiver ($\mathsf{P2}$, Dec, RFake) by puncturing encryption key of receiver-fake ACE $\mathsf{EK}_R\{p\}$ at the point $p = (m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \hat{\rho}^*)$, decryption key of receiver-fake ACE $\mathsf{DK}_R\{r^*, r'\}$ at r^* and r' (where $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(p)$), and the key k_R of extracting PRF RG of the receiver at the points $(r^*, {\mu_1}^*)$ and $(r', {\mu_1}^*)$. In addition, we hardwire certain outputs inside programs of the receiver to make sure that functionality of the programs doesn't change. Indistinguishability holds by iO.

- $\mathsf{Hyb}_{B,2,2}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,3}, \mathsf{P2}_{B,2}, \mathsf{P3}_{B,3}, \mathsf{Dec}_{B,2}, \mathsf{SFake}_{B,3}, \mathsf{RFake}_{B,2}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^*$ is chosen at random, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$. Programs can be found in fig. 123 (programs of the sender) and fig. 125 (programs of the receiver).

  That is, we choose ${\mu_2}^*$ at random instead of computing it as ${\mu_2}^* = \mathsf{RG}_{k_S}(r^*, {\mu_1}^*)$. Indistinguishability holds by pseudorandomness of the PRF SG at the punctured point $(r^*, {\mu_1}^*)$.

- $\mathsf{Hyb}_{B,2,3}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,3}, \mathsf{P2}_{B,2}, \mathsf{P3}_{B,3}, \mathsf{Dec}_{B,2}, \mathsf{SFake}_{B,3}, \mathsf{RFake}_{B,2}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^*$ is chosen at random, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \hat{\rho}^*)$ for randomly chosen $\hat{\rho}^*$. Programs can be found in fig. 123 (programs of the sender) and fig. 125 (programs of the receiver).

  That is, we switch the roles of r^* and r' everywhere in the distribution: namely, we give r' (instead of r^*) to the adversary as randomness of the receiver, and we change r^* to r' and r' to r^* everywhere in the programs. Note that this doesn't change the code of the programs since programs use r^* and r' in the same way. Indistinguishability holds by the symmetry of receiver-fake ACE, which says that $(r^*, r', \mathsf{EK}_R\{p\}, \mathsf{DK}_R\{r^*, r'\})$ is indistinguishable from $(r', r^*, \mathsf{EK}_R\{p\}, \mathsf{DK}_R\{r', r^*\})$, where $p = (m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \hat{\rho}^*)$, r^* is randomly chosen, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(p)$.

- $\mathsf{Hyb}_{B,2,4}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,3}, \mathsf{P2}_{B,2}, \mathsf{P3}_{B,3}, \mathsf{Dec}_{B,2}, \mathsf{SFake}_{B,3}, \mathsf{RFake}_{B,2}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \hat{\rho}^*)$ for randomly chosen $\hat{\rho}^*$. Programs can be found in fig. 123 (programs of the sender) and fig. 125 (programs of the receiver).

  That is, we compute ${\mu_2}^*$ as ${\mu_2}^* = \mathsf{RG}_{k_R}(r^*, {\mu_1}^*)$ instead of choosing it at random. Indistinguishability holds by pseudorandomness of the PRF RG at the punctured point $(r^*, {\mu_1}^*)$.

- $\mathsf{Hyb}_{B,2,5}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,3}, \mathsf{P2}_{B,1}, \mathsf{P3}_{B,3}, \mathsf{Dec}_{B,1}, \mathsf{SFake}_{B,3}, \mathsf{RFake}_{B,1}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$,

$s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \hat{\rho}^*)$ for randomly chosen $\hat{\rho}^*$. Programs can be found in fig. 123 (programs of the sender) and fig. 124 (programs of the receiver).

That is, we revert all changes we made to programs in $\mathsf{Hyb}_{B,2,1}$ and thus use original programs $\mathsf{P2}, \mathsf{Dec}, \mathsf{RFake}$, except that DK remains punctured at the point $\overline{p} = (1 \oplus m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$. Indistinguishability holds by iO, since we remove puncturing without changing the functionality of the programs.

- $\mathsf{Hyb}_{B,2,6}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,3}, \mathsf{P2}_{B,1}, \mathsf{P3}_{B,3}, \mathsf{Dec}_{B,1}, \mathsf{SFake}_{B,3}, \mathsf{RFake}_{B,1}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, ${\mu_1}^* = \mathsf{SG}_{k_S}(s^*, m_1^*)$, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 123 (programs of the sender) and fig. 124 (programs of the receiver).

  That is, we replace randomly chosen $\hat{\rho}^*$ with $\mathsf{prg}(\rho^*)$ for randomly chosen ρ^*, when generating r'. Indistinguishability holds by security of a prg.

Finally, in the following hybrids we revert all changes we made in hybrids $\mathsf{Hyb}_{B,1,1}$ - $\mathsf{Hyb}_{B,1,6}$, thus restoring all programs (and making $\overline{{\mu_3}^*}$ a valid ciphertext):

- $\mathsf{Hyb}_{B,3,1}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,3}, \mathsf{P2}, \mathsf{P3}_{B,3}, \mathsf{Dec}, \mathsf{SFake}_{B,3}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 123 (programs of the sender) and fig. 100 (programs of the receiver).

  That is, in programs $\mathsf{Dec}, \mathsf{RFake}$ we unpuncture decryption key DK of the main ACE at the point $\overline{p} = (1 \oplus m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$. Indistinguishability holds by security of constrained key of ACE, since the corresponding encryption key EK is punctured at this point.

- $\mathsf{Hyb}_{B,3,2}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,2}, \mathsf{P2}, \mathsf{P3}_{B,2}, \mathsf{Dec}, \mathsf{SFake}_{B,2}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; r^* is chosen at random, ${\mu_1}^*$ is chosen at random, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 98 (programs of the sender) and fig. 100 (programs of the receiver).

  That is, in program P3 we unpuncture encryption key EK of the main ACE at the point $\overline{p} = (1 \oplus m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$. Indistinguishability holds by iO, because of the same reason as in $\mathsf{Hyb}_{B,1,5}$.

- $\mathsf{Hyb}_{B,3,3}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,2}, \mathsf{P2}, \mathsf{P3}_{B,2}, \mathsf{Dec}, \mathsf{SFake}_{B,2}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, ${\mu_1}^* = \mathsf{SG}(s^*, m_1^*)$, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 98 (programs of the sender) and fig. 100 (programs of the receiver).

  That is, we choose ${\mu_1}^*$ as ${\mu_1}^* = \mathsf{SG}_{k_S}(s^*, m_1^*)$ instead of computing it at random. Indistinguishability holds by the strong extracting property of the sender PRF SG (note that s^* is not used anywhere else in the distribution).

- $\mathsf{Hyb}_{B,3,4}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}_{B,1}, \mathsf{P2}, \mathsf{P3}_{B,1}, \mathsf{Dec}, \mathsf{SFake}_{B,1}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, ${\mu_1}^* = \mathsf{SG}(s^*, m_1^*)$, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 97 (programs of the sender) and fig. 100 (programs of the receiver).

  That is, in programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$ we unpuncture decryption key DK_S of the sender-fake ACE at the same set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$. Indistinguishability holds by security of constrained key of ACE, since the corresponding encryption key EK_S is already punctured at the same set.

- $\mathsf{Hyb}_{B,3,5}$. We give the adversary $(\mathsf{PP}, m_0^*, m_1^*, s', r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, where $\mathsf{PP} = \mathsf{Setup}(1^\lambda; \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}; r_{\mathsf{Setup}})$ for randomly chosen r_{Setup}; s^*, r^* are chosen at random, ${\mu_1}^* = \mathsf{SG}(s^*, m_1^*)$, ${\mu_2}^* = \mathsf{RG}(r^*, {\mu_1}^*)$, ${\mu_3}^* = \mathsf{ACE.Enc}_{\mathsf{EK}}(m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, $s' = \mathsf{ACE.Enc}_{\mathsf{EK}_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell_1^*)$, $r' = \mathsf{ACE.Enc}_{\mathsf{EK}_R}(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, L_0^*, \mathsf{prg}(\rho^*))$ for randomly chosen ρ^*. Programs can be found in fig. 96 (programs of the sender) and fig. 100 (programs of the receiver).

  That is, in program SFake we unpuncture encryption key EK_S of the sender-fake ACE at the set $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$. Indistinguishability holds by iO, since this modification doesn't change the functionality of SFake due to the fact that SFake never encrypts plaintexts with level ℓ_0^*.

  Note that $\mathsf{Hyb}_{B,3,5}$ is the same as Hyb_C, conditioned on the fact that s^*, r^* are outside of image of ACE.

Programs $\mathsf{P1}_{B,3}, \mathsf{P3}_{B,3}, \mathsf{SFake}_{B,3}$.

Program $\mathsf{P1}_{B,3}(s, m)$

Inputs: sender randomness s, message m.

Hardwired values: punctured decryption key $\mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$, key k_S of an extracting PRF SG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m = m'$ then return μ_1';
2. **Main step:**
   (a) Return $\mu_1 \leftarrow \mathsf{SG}_{k_S}(s, m)$.

Program $\mathsf{P3}_{B,3}(s, m, \mu_1, \mu_2)$

Inputs: sender randomness s, message m, the first and the second messages μ_1, μ_2 in the protocol.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{B,3}$, $\mathsf{GenZero}$, $\mathsf{Transform}$, $\mathsf{RetrieveTag}$; punctured decryption key $\mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$, punctured encryption key $\mathsf{EK}\{\overline{p}\}$ of main ACE, where $\overline{p} = (1 \oplus m_1^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$.

1. **Validity check:** if $\mathsf{P1}_{B,3}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', \mu_1', \mu_2'$ then return μ_3';
   (c) If $m, \mu_1 = m', \mu_1'$ then:
      i. If $\mu_1 \neq \mathsf{RetrieveTag}(\ell')$ then abort;
      ii. Set $L \leftarrow \mathsf{Transform}(\ell', \mu_2)$;
      iii. Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}\{\overline{p}\}}(m, \mu_1, \mu_2, L)$;
3. **Main step:**
   (a) Set $L_0 \leftarrow \mathsf{Transform}(\mathsf{GenZero}(\mu_1), \mu_2)$;
   (b) Return $\mu_3 \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}\{\overline{p}\}}(m, \mu_1, \mu_2, L_0)$.

Program $\mathsf{SFake}_{B,3}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$

Inputs: sender randomness s, real message m, fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P1}_{B,3}$, $\mathsf{GenZero}$, $\mathsf{Increment}$; punctured encryption and decryption keys $\mathsf{EK}_S\{P_{\ell_0^*}\}, \mathsf{DK}_S\{P_{\ell_0^*}\}$ of sender-fake ACE, where $P_{\ell_0^*} = \{(*, *, *, *, \ell_0^*)\}$.

1. **Validity check:** if $\mathsf{P1}_{B,3}(s, m) \neq \mu_1$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_S\{P_{\ell_0^*}\}}(s)$; if $\mathsf{out} = \mathsf{'fail'}$ goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m, \mu_1 = m', \mu_1'$ then
      i. Set $\ell_{+1} \leftarrow \mathsf{Increment}(\ell')$; if $\ell_{+1} = \mathsf{'fail'}$ then abort;
      ii. Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_{+1})$.
3. **Main step:**
   (a) Set $\ell_1 \leftarrow \mathsf{Increment}(\mathsf{GenZero}(\mu_1))$;
   (b) Return $\mathsf{ACE.Enc}_{\mathsf{EK}_S\{P_{\ell_0^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell_1)$.

Figure 123: Programs $\mathsf{P1}_{B,3}, \mathsf{P3}_{B,3}, \mathsf{SFake}_{B,3}$, used in the proof indistinguishability of explanations of the receiver for off-the-record deniability.

Programs $\mathsf{P2}_{B,1}, \mathsf{Dec}_{B,1}, \mathsf{RFake}_{B,1}$.

Program $\mathsf{P2}_{B,1}(r, \mu_1)$

Inputs: receiver randomness r, the first message μ_1 in the protocol.

Hardwired values: decryption key DK_R of receiver-fake ACE, key k_R of an extracting PRF RG.

1. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_R}(r)$; if $\mathsf{out} = '\mathsf{fail}'$ then goto main step, else parse out as $(m', \mu_1{}', \mu_2{}', \mu_3{}', L', \hat{\rho})$;
   (b) If $\mu_1 = \mu_1{}'$ then return $\mu_2{}'$;
2. **Main step:**
   (a) Return $\mu_2 \leftarrow \mathsf{RG}_{k_R}(r, \mu_1)$.

Program $\mathsf{Dec}_{B,1}(r, \mu_1, \mu_2, \mu_3)$

Inputs: receiver randomness r, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms P2, isLess, RetrieveTags; decryption key DK_R of receiver-fake ACE, punctured decryption key $\mathsf{DK}\{\overline{p}\}$ of the main ACE, where $\overline{p} = (1 \oplus m_1^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$.

1. **Validity check:** if $\mathsf{P2}(r, \mu_1) \neq \mu_2$ then abort;
2. **Trapdoor step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}_R}(r)$; if $\mathsf{out}' = '\mathsf{fail}'$ then goto main step; else parse out' as $(m', \mu_1{}', \mu_2{}', \mu_3{}', L', \hat{\rho})$;
   (b) if $\mu_1, \mu_2, \mu_3 = \mu_1{}', \mu_2{}', \mu_3{}'$ then return m';
   (c) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{\overline{p}\}}(\mu_3)$; if $\mathsf{out}'' = '\mathsf{fail}'$ then abort, else parse out'' as $(m'', \mu_1{}'', \mu_2{}'', L'')$;
   (d) If $\mu_1, \mu_2 = \mu_1{}', \mu_2{}'$ then
      i. If $(\mu_1{}', \mu_2{}') = (\mu_1{}'', \mu_2{}'') = \mathsf{RetrieveTags}(L'')$ and $\mathsf{isLess}(L', L'') = \mathsf{true}$ then return m'';
      ii. Else abort.
3. **Main step:**
   (a) $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{\overline{p}\}}(\mu_3)$; if $\mathsf{out} = '\mathsf{fail}'$ then abort, else parse out as $(m'', \mu_1{}'', \mu_2{}'', L'')$;
   (b) If $(\mu_1, \mu_2) = (\mu_1{}'', \mu_2{}'') = \mathsf{RetrieveTags}(L'')$ then return m'';
   (c) Else abort.

Program $\mathsf{RFake}_{B,1}(\hat{m}, \mu_1, \mu_2, \mu_3; \rho)$

Inputs: fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3, random coins ρ.

Hardwired values: encryption key EK_R of receiver-fake ACE, punctured decryption key $\mathsf{DK}\{\overline{p}\}$ of the main ACE, where $\overline{p} = (1 \oplus m_1^*, \mu_1{}^*, \mu_2{}^*, L_0^*)$.

1. $\mathsf{out} \leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{\overline{p}\}}(\mu_3)$; if $\mathsf{out} = '\mathsf{fail}'$ then abort, else parse out as $(m'', \mu_1{}'', \mu_2{}'', L'')$;
2. Return $r' \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_R}(\hat{m}, \mu_1, \mu_2, \mu_3, L'', \mathsf{prg}(\rho))$.

Figure 124: Programs $\mathsf{P2}_{B,1}, \mathsf{Dec}_{B,1}, \mathsf{RFake}_{B,1}$, used in the proof of lemma 55 (indistinguishability of explanations of the receiver).

Programs $\mathsf{P2}_{B,2}, \mathsf{Dec}_{B,2}, \mathsf{RFake}_{B,2}$**.**

Program $\mathsf{P2}_{B,2}(r, \mu_1)$

Inputs: receiver randomness r, the first message μ_1 in the protocol.

Hardwired values: punctured decryption key $\mathsf{DK}_R\{r^*, r'\}$ of receiver-fake ACE, punctured key $k_R\{(r^*, {\mu_1}^*), (r', {\mu_1}^*)\}$ of an extracting PRF RG, variables $r^*, r', {\mu_1}^*, {\mu_2}^*$.

1. **Trapdoor step:**
   (a) If $(r, \mu_1) = (r^*, {\mu_1}^*)$ or $(r, \mu_1) = (r', {\mu_1}^*)$ then return ${\mu_2}^*$;
   (b) If $r = r^*$ or $r = r'$ then goto main step;
   (c) out $\leftarrow$ $\mathsf{ACE.Dec}_{\mathsf{DK}_R\{r^*,r'\}}(r)$; if out $=$ $'\mathsf{fail}'$ then goto main step, else parse out as $(m', {\mu_1}', {\mu_2}', {\mu_3}', L', \hat{\rho})$;
   (d) If $\mu_1 = {\mu_1}'$ then return ${\mu_2}'$;
2. **Main step:**
   (a) Return $\mu_2 \leftarrow \mathsf{RG}_{k_R\{(r^*,{\mu_1}^*),(r',{\mu_1}^*)\}}(r, \mu_1)$.

Program $\mathsf{Dec}_{B,2}(r, \mu_1, \mu_2, \mu_3)$

Inputs: receiver randomness r, protocol transcript μ_1, μ_2, μ_3.

Hardwired values: obfuscated code of algorithms $\mathsf{P2}_{B,2}$, isLess, $\mathsf{RetrieveTags}$; punctured decryption key $\mathsf{DK}_R\{r^*, r'\}$ of receiver-fake ACE, punctured decryption key $\mathsf{DK}\{\overline{p}\}$ of the main ACE, where $\overline{p} = (1 \oplus m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$, variables $r^*, r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, m_1^*$.

1. **Validity check:** if $\mathsf{P2}_{B,2}(r, \mu_1) \neq \mu_2$ then abort;
2. **Trapdoor step:**
   (a) If $(r, \mu_1, \mu_2, \mu_3) = (r^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$ or $(r, \mu_1, \mu_2, \mu_3) = (r', {\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$ then return m_1^*;
   (b) If $(r, \mu_1, \mu_2) = (r^*, {\mu_1}^*, {\mu_2}^*)$ or $(r, \mu_1, \mu_2) = (r', {\mu_1}^*, {\mu_2}^*)$ then then goto main step;
   (c) If $r = r^*$ or $r = r'$ then goto main step;
   (d) out $\leftarrow$ $\mathsf{ACE.Dec}_{\mathsf{DK}_R\{r^*,r'\}}(r)$; if out$'$ $=$ $'\mathsf{fail}'$ then goto main step; else parse out$'$ as $(m', {\mu_1}', {\mu_2}', {\mu_3}', L', \hat{\rho})$;
   (e) if $\mu_1, \mu_2, \mu_3 = {\mu_1}', {\mu_2}', {\mu_3}'$ then return m';
   (f) out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{\overline{p}\}}(\mu_3)$; if out$''$ $= '\mathsf{fail}'$ then abort, else parse out$''$ as $(m'', {\mu_1}'', {\mu_2}'', L'')$;
   (g) If $\mu_1, \mu_2 = {\mu_1}', {\mu_2}'$ then
      i. If $({\mu_1}', {\mu_2}') = ({\mu_1}'', {\mu_2}'') = \mathsf{RetrieveTags}(L'')$ and $\mathsf{isLess}(L', L'') = \mathsf{true}$ then return m'';
      ii. Else abort.
3. **Main step:**
   (a) out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{\overline{p}\}}(\mu_3)$; if out $= '\mathsf{fail}'$ then abort, else parse out as $(m'', {\mu_1}'', {\mu_2}'', L'')$;
   (b) If $(\mu_1, \mu_2) = ({\mu_1}'', {\mu_2}'') = \mathsf{RetrieveTags}(L'')$ then return m'';
   (c) Else abort.

Program $\mathsf{RFake}_{B,2}(\hat{m}, \mu_1, \mu_2, \mu_3; \rho)$

Inputs: fake message $\hat{m}$, protocol transcript μ_1, μ_2, μ_3, random coins ρ.

Hardwired values: punctured encryption key $\mathsf{EK}_R\{S_{\hat{\rho}^*}\}$ of receiver-fake ACE, where $S_{\hat{\rho}^*} = \{*, *, *, *, *, \hat{\rho}^*\}$ for randomly chosen $\hat{\rho}^*$, punctured decryption key $\mathsf{DK}\{\overline{p}\}$ of the main ACE, where $\overline{p} = (1 \oplus m_1^*, {\mu_1}^*, {\mu_2}^*, L_0^*)$.

1. out $\leftarrow \mathsf{ACE.Dec}_{\mathsf{DK}\{\overline{p}\}}(\mu_3)$; if out $= '\mathsf{fail}'$ then abort, else parse out as $(m'', {\mu_1}'', {\mu_2}'', L'')$;
2. Return $r' \leftarrow \mathsf{ACE.Enc}_{\mathsf{EK}_R\{S_{\hat{\rho}^*}\}}(\hat{m}, \mu_1, \mu_2, \mu_3, L'', \mathsf{prg}(\rho))$.

Figure 125: Programs $\mathsf{P2}_{B,2}, \mathsf{Dec}_{B,2}, \mathsf{RFake}_{B,2}$, used in the proof of lemma 55 (indistinguishability of explanations of the receiver).

Acknowledgements

RC is a member of the Check Point Institute for Information Security. Supported by the NSF MACS project. SP's research is supported by the MIT Media Lab's Digital Currency Initiative and its funders, and earlier, was supported by the following grants: NSF MACS (CNS-1413920), DARPA IBM (W911NF-15-C-0236), Simons Investigator award agreement dated June 5th, 2012, and the Center for Science of Information (CSoI), an NSF Science & Technology Center, under grant agreement CCF-0939370.

References

[AFL16] Daniel Apon, Xiong Fan, and Feng-Hao Liu. Deniable attribute based encryption for branching programs from LWE. In *Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part II*, pages 299–329, 2016. 9, 260, 261

[BCG+18] Nir Bitansky, Ran Canetti, Sanjam Garg, Justin Holmgren, Abhishek Jain, Huijia Lin, Rafael Pass, Sidharth Telang, and Vinod Vaikuntanathan. Indistinguishability obfuscation for RAM programs and succinct randomized encodings. *SIAM J. Comput.*, 47(3):1123–1210, 2018. 49, 50

[BCP14] Elette Boyle, Kai-Min Chung, and Rafael Pass. On extractability obfuscation. In *Theory of Cryptography - 11th Theory of Cryptography Conference, TCC 2014, San Diego, CA, USA, February 24-26, 2014. Proceedings*, pages 52–73, 2014. 48, 102, 104, 266

[BGB04] Nikita Borisov, Ian Goldberg, and Eric A. Brewer. Off-the-record communication, or, why not to use PGP. In Vijay Atluri, Paul F. Syverson, and Sabrina De Capitani di Vimercati, editors, *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES 2004, Washington, DC, USA, October 28, 2004*, pages 77–84. ACM, 2004. 2

[BNNO11] Rikke Bendlin, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. Lower and upper bounds for deniable public-key encryption. In *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, pages 125–142, 2011. 1, 5, 9, 10, 11, 12, 15, 260

[BPR15] Nir Bitansky, Omer Paneth, and Alon Rosen. On the cryptographic hardness of finding a nash equilibrium. *Electronic Colloquium on Computational Complexity (ECCC)*, 22:1, 2015. 3, 7, 48, 72, 73

[BPW16] Nir Bitansky, Omer Paneth, and Daniel Wichs. Perfect structure on the edge of chaos - trapdoor permutations from indistinguishability obfuscation. In *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part I*, pages 474–502, 2016. 3, 59

[CDMW09] Seung Geol Choi, Dana Dachman-Soled, Tal Malkin, and Hoeteck Wee. Improved non-committing encryption with applications to adaptively secure protocols. In *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of*

Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings, pages 287–302, 2009. 9

[CDNO96] Ran Canetti, Cynthia Dwork, Moni Naor, and Rafail Ostrovsky. Deniable encryption. *IACR Cryptology ePrint Archive*, 1996:2, 1996. 1, 2, 9, 10, 260

[CFGN96] Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. Adaptively secure multi-party computation. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 639–648, 1996. 9

[CHJV14] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Indistinguishability obfuscation of iterated circuits and RAM programs. *IACR Cryptology ePrint Archive*, 2014:769, 2014. 3, 7, 49, 50, 52, 59, 262, 263, 264, 265

[CHK+08] Alexei Czeskis, David J. St. Hilaire, Karl Koscher, Steven D. Gribble, Tadayoshi Kohno, and Bruce Schneier. Defeating encrypted and deniable file systems: Truecrypt v5.1a and the case of the tattling OS and applications. In *3rd USENIX Workshop on Hot Topics in Security, HotSec'08, San Jose, CA, USA, July 29, 2008, Proceedings*, 2008. 261

[CIO16] Angelo De Caro, Vincenzo Iovino, and Adam O'Neill. Deniable functional encryption. In *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part I*, pages 196–222, 2016. 9, 260, 261

[CPR17] Ran Canetti, Oxana Poburinnaya, and Mariana Raykova. Optimal-rate non-committing encryption. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part III*, pages 212–241, 2017. 49

[Dac12] Dana Dachman-Soled. On the impossibility of sender-deniable public key encryption. *IACR Cryptology ePrint Archive*, 2012:727, 2012. 9, 10, 260

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Information Theory*, 22(6):644–654, 1976. 1

[DKSW09] Yevgeniy Dodis, Jonathan Katz, Adam D. Smith, and Shabsi Walfish. Composability and on-line deniability of authentication. In *Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings*, pages 146–162, 2009. 9

[GGM84] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. On the cryptographic applications of random functions. In *Advances in Cryptology, Proceedings of CRYPTO '84, Santa Barbara, California, USA, August 19-22, 1984, Proceedings*, pages 276–288, 1984. 48

[GKW17] Shafi Goldwasser, Saleet Klein, and Daniel Wichs. The edited truth. In *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I*, pages 305–340, 2017. 9, 260, 261

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984. 1

[GPS16] Sanjam Garg, Omkant Pandey, and Akshayaram Srinivasan. Revisiting the cryptographic hardness of finding a nash equilibrium. In *Advances in Cryptology - CRYPTO 2016 - 36th*

Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II, pages 579–604, 2016. 50, 264

[KLW15] Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 419–428, 2015. 3

[Nie02] Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, pages 111–126, 2002. 10

[OPW11] Adam O'Neill, Chris Peikert, and Brent Waters. Bi-deniable public-key encryption. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 525–542, 2011. 2, 9, 260, 261

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978. 1

[SW14] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 475–484, 2014. 1, 5, 6, 7, 8, 10, 11, 13, 20, 48, 59, 64, 225, 226, 232, 233, 260, 264

A On Flexible Deniability: Discussion

This weaker notion of deniability [CDNO96, OPW11, BNNO11, Dac12, AFL16, CIO16, GKW17], sometimes called *flexible deniability, multi-distributional deniability* or *dual-scheme deniability* postulates *two* schemes, S and S', where S is "deniable with respect to S'". When applied to sender-deniability, the requirement is roughly as follows: The sender can use scheme S to encrypt plaintext m with random string r to obtain ciphertext c, and then can present an appropriate fake random string $\tilde{r}$ such that c is obtained as an encryption of $m' \neq m$ with randomness $\tilde{r}$, *but under the scheme* S'. In other words, this security property assumes that the adversary doesn't know which scheme was used by the sender, S or S', and that the adversary is willing to believe the sender who claims to have used the non-deniable version. Another way to look at flexible schemes is to say that S is a "trapdoored" version of S', i.e. when parties generate a ciphertext according to deniable scheme S, they additionally generate a faking trapdoor which is required to compute fake randomness; in constrast, in S' no such trapdoor is generated. Deniability holds only as long as this trapdoor remains secret; upon coercion, the parties claim that they never generated a trapdoor to begin with.

In contrast, in the standard definition of deniable encryption [CDNO96] the adversary is fully aware of the fact that the sender is using a deniable algorithm, and thus has the right to demand to see any potential "faking trapdoors" or any information which may be required for generating fake randomness.

On a positive side, flexibly deniable encryption already guarantees *plausible deniability*, since the coercer cannot *prove* that S was used - even though it may have reasons to believe so. Thus, flexible deniability already protects parties in many scenarios where plausible deniability suffices, e.g. in court. Another benefit of flexible schemes is their efficiency: unlike fully deniable schemes (including this work and [SW14]) which, to date, are only known from obfuscatoin, known flexibly deniable schemes can be implemented in practice. Flexible deniability, due to having a weaker security guarantee, allows for fewer rounds, more efficiency, and weaker assumptions than fully deniable schemes, and requires no setup. For instance, [OPW11] build a 2-message flexibly bideniable encryption from LWE and from simulatable encryption. In fact we even have more advanced encryption schemes (like identity-based encryption [OPW11], functional encryption [CIO16], and attribute-based encryption ([AFL16])) with flexible deniability, and we have flexibly deniable encryption scheme with *succinct* keys [GKW17], where the size of a key is proportional to the number of possible fake messages (which can be smaller than the total number of possible plaintexts).

However, flexible notion of deniability has significant drawbacks. Indeed, having two different algorithms, which have two different security guarantees and which are up to the parties to choose, leaves room for suspicion, misuse, and can even cause harm to parties themselves. It also requires additional coordination between parties. But most importantly, *flexible deniability doesn't provide perhaps the most desirable benefit of deniability - preventing coercion in the first place by making it useless.* Below we explain these issues in more detail.

First, refusal to provide keys for deniable version could significantly increase the adversary's certainty that parties are lying - compared to the ideal channels case where the coercer has nothing besides parties' claims. Indeed, in the real world the opinion of the coercer will be shifted by its certainty that deniable version was used. However, this is not captured by security definition of flexible deniability, which doesn't take into account how exactly parties choose an algorithm, e.g. by assuming some distribution on the choices of S and S', or considering rational behavior. For instance, one could argue that rational players would prefer S over S' because of better security guarantees, which is further aggravated by the fact that flexible deniability could actually *harm* those who use the non-deniable version. Indeed, as [CHK$^+$08], who analyze plausible deniability of TrueCrypt hidden volume, put it, "deniability cuts both ways, and sometimes that's not a benefit".

Second, note that fully deniable encryption doesn't allow parties to prove what their plaintext was even if they want to[35]. This is crucial in preventing bribery or vote selling. In contrast, in flexibly deniable encryption parties can choose whether they want it or not by choosing deniable or non-deniable algorithm. As a result, with fully deniable encryption one could set up receipt-free voting scheme using a physical booth which, for instance, provides parties with randomness (so that they can still lie about their vote, but cannot use preset randomness to sell their vote). But if flexible scheme is used, then voters can lie about their vote but at the same time sell their true vote if they want (if deniable version is used), or can do neither (if non-deniable version is used).

Another important issue which arises in flexible setting is the need for coordination. That is, parties need a way to agree whether they run S or S', and do so by the time of encryption[36]. It is not clear how to do such coordination without another deniable channel. As a result, well-being of each party is in the other party's hands: e.g. the sender's claim will look credible only as long as the receiver also used deniable algorithm at time of encryption, also decided to fake at time of coercion, and used the same fake plaintext. This is a problem not only when the receiver turns against the sender, but also when the receiver remains honest but doesn't know what actions to take out of lack of coordination.

Finally and most importantly, as already pointed out by [OPW11], deniable encryption not only allows to

[35] As discussed before, this property only holds if parties execute the protocol correctly.

[36] However coordination is not required for correctness and semantic security, since these properties hold even if the sender and the receiver use different schemes [OPW11].

withstand coercion, but also makes in useless in the first place - just like it is useless in the ideal world, where there is no way of verifying parties' claims. However, flexible deniability doesn't give this guarantee: the coercer (who suspects that deniable version could be used) can gradually increase the pressure - be it a sum of money or "enhanced interrogation" - until the parties find it more preferable to prove what their plaintext was by disclosing keys of deniable version, S.

To summarize this discussion, we think that flexible deniability as a real-life application already suffices in many cases - e.g. when plausible deniability is sufficient, or when the coercer is not aware of the concept of deniable encryption and will be satisfied by seeing some working key. However, to obtain security guarantees of the ideal channel, one should use encryption which is (fully) deniable and off-the-record deniable.

Needless to say, we still believe that flexible deniability is a fascinating concept to explore. For instance, coming up with flexible scheme where S' is some standard encryption, e.g. RSA, would mitigate some issues mentioned above, thus making flexible deniability as good as full deniability for many practical purposes. Further, flexibly deniable encryption is an interesting primitive whose connections to non-committing encryption and full deniability are yet to be explored.

B On removing layers of obfuscation

When our construction described in section 6 is instantiated with ACE from [CHJV14], relaxed ACE described in section C, and the level system described in section 7 (which in turn uses ACE of [CHJV14]), the resulting CRS ends up containing three layers of obfuscation. Since even a single obfuscation incurs a significant blowup in the program size, ideally we would like to have only one layer of obfuscation.

In this section we explain why the whole proof of bideniability and off-the-record deniability can still go through, if we use non-obfuscated version and "unroll" all the proofs. More concretely, we do the following:

- Instead of using ACE keys and the programs of the level system, which are all obfuscated programs, we use their *non-obfuscated versions*. Still, we use one layer of obfuscation on top of programs of deniable encryption. We pad the size of the non-obfuscated programs of deniable encryption to size σ such that σ is larger than the size of any (non-obfuscated) program (including programs variants in the hybrids) of deniable encryption, ACE, relaxed ACE, or the level system.
- In the proof we replace each hybrid reducing to security of any of ACE, relaxed ACE, or the level system with a sequence of hybrids proving the corresponding property of the primitive.

Now we briefly comment on why each security reduction can still be proven. Let program C_1 of a primitive Δ_1, and program C_2 of a primitive Δ_2 be such that C_1 uses an obfuscated version of C_2, i.e. $\mathsf{iO}(C_2)$, as a black box (e.g. Δ_1 can be deniable encryption and Δ_2 can be relaxed ACE, ACE or the level system, or Δ_1 can be the level system and Δ_2 can be ACE). We denote this by $C_1[\mathsf{iO}(C_2)]$. Further, let $C_1[C_2]$ be program C_1 which uses program C_2, instead of $\mathsf{iO}(C_2)$. Note that this is syntactically well-defined since C_1 uses $\mathsf{iO}(C_2)$ as a black box and since $\mathsf{iO}(C_2)$ and C_2 have the same syntax.

Further, let all reductions in the security proof of Δ_1 use $\mathsf{iO}(C_2)$ as a black box. We claim that the "unrThen all reductions in security proofs of deniable encryption, ACE, relaxed ACE, and the level system can be classified as follows:

Reductions in the proof of security of Δ_1:

- Reductions which rely on security of Δ_2: we replace each reduction with a sequence of reductions from the proof of Δ_2, and as we argue later, they all still can be proven.
- Reductions which do not rely on security of Δ_2, but which use the fact that $\mathsf{iO}(C_2)$ has a certain functionality (e.g. an iO-based reduction, which uses the fact that the functionality of C_1 in the two consecutive hybrids doesn't change, and analyzes functionality of $\mathsf{iO}(C_2)$ as part of the argument). We claim that if such a reduction is possible with $C_1[\mathsf{iO}(C_2)]$, then it is also possible with $C_1[C_2]$. This is because iO preserves the functionality with all-but-negligible probability over the randomness of iO.
- All other reductions: these reductions merely use the fact that in the reduction it is possible to reconstruct $\mathsf{iO}(C_2)$ in polynomial time. Note that this is true for C_2 as well, thus such reductions still go through.

Reductions in the proof of security of Δ_2

- Reductions to security of obfuscation for a program C_2, relying on the fact that C_2 has the same functionality in the two consecutive hybrids: we claim that we can instead reduce to security of obfuscation for a program $\mathsf{iO}(C_1[C_2])$. Indeed, since C_1 uses $\mathsf{iO}(C_2)$ as a black box, and since iO preserves functionality except for negligible probability over the choice of randomness of iO, $C_1[C_2]$ also has the same functionality in those two hybrids. Thus, as long as we pad the program $C_1[C_2]$ sufficiently, the reduction to security of iO still holds.
- Reductions which rely on the fact that in some cases iO allows to extract a differing input of programs C_2', C_2'', given $\mathsf{iO}(C_2'), \mathsf{iO}(C_2'')$. We argue that security of there hybrids can still be reduced to security of iO and one-way functions, even though the resulting programs $C_1[C_2']$ and $C_1[C_2'']$ can be different on exponentially many inputs. Recall that those security reductions work by constructing a circuit M_2 such that M_2 is the same as C_2' or C_2'', and use it do to binary search over a differing value, which could be an input, or part of an input, or some intermediate variable in the program. But this means that the reduction in the "unrolled" proof can do the same binary search, over the same differing value, by using program $M_1 = C_1[M_2]$, which can be constructed using $\mathsf{iO}(C_1[C_2'])$, $\mathsf{iO}(C_1[C_2''])$: indeed, since M_2 is the same as either C_2' or C_2'', $C_1[M_2]$ is the same as either $C_1[C_2']$ or $C_1[C_2'']$.
- All other reductions: in such reductions we need to make sure that the reduction can reconstruct the whole distribution, which now includes an obfuscated program $\mathsf{iO}(C_1[C_2])$, together with any values the adversary is supposed to get as part of the game for primitive Δ_1. We note that this can be done: since it was possible to do in the reduction (of the proof for Δ_1) to security of Δ_2, it should be possible as well for every hybrid in security proof of Δ_2, since otherwise the reduction of the proof of Δ_1 can be used as a distinguisher for Δ_2. Indeed, since the reduction uses $\mathsf{iO}(C_2)$ as a black box, we can replace $\mathsf{iO}(C_2)$ with C_2 and the reduction still succeeds.

C Construction of relaxed ACE

In this section we describe how to modify the construction of ACE from [CHJV14] to obtain relaxed ACE (def. 9). Recall that the differences between ACE and relaxed ACE are that relaxed ACE doesn't necessarily satisfy indistinguishability of ciphertexts; that its distinguishing advantage in security of constrained decryption game is negligible for certain sets (as opposed to being proportional to size of those sets); and that it additionally satisfies symmetry.

Brief motivation and explanation of the construction. The first attempt to remove dependency on the size of the sets is perhaps to use the technique from [GPS16] - that is, instead of having a single PRF-based signature on the plaintext m, have $|m|$ signatures of each prefix of m. This allows to change the key on many inputs (with the same prefix) in a single step. However, with this approach we are not able to prove symmetry: it requires to switch $c^* = \mathsf{Enc}(m^*)$ to random and thus to puncture all keys for each PRF; however, such puncturing cannot be done without changing the functionality of the encryption program, since e.g. puncturing the PRF which is applied on the first bit already prohibits encrypting of half of all inputs.

To deal with this, we notice that in the proof of deniable encryption we use security of relaxed ACE on sets of special structure, which is either all strings ending with the same suffix of a fixed size, or all such strings except one. Thus we require relaxed ACE to be parametrized with *prefix parameter* t, which denotes the size of this prefix. [37] An encryption of m will be an ACE ciphertext where instead of a single PRF signature of m, we will have $n - t + 1$ PRF signatures of $\mathsf{suffix}_t(m), \ldots, \mathsf{suffix}_n(m)$. We say that a set S is *consistent* with some suffix suf of size t, if S consists of all strings ending with suf; we say that a plaintext m is consistent with suf, if m ends with suf. Using $n - t + 1$ signatures allows us to prove the following:

- symmetry for random c^* and $c' = \mathsf{Enc}(m^*)$, as long as encryption key is punctured at the set S, and both S and m^* are consistent with the same suffix suf of size t;
- security of constrained decryption with distinguishing advantage independent of set sizes, as long as $S_1 \setminus S_0$ is either S_{suf} (e.g. a set consistent with some suf of size t), or $S_{\mathsf{suf}} \setminus \{m\}$, where both S and m are consistent with the same suf of size t.

Security of constrained decryption follows a by-now standard proof, which punctures the key at the whole set S_{suf_i} at once (for each $i = t + 1, \ldots, n$), by adding an injective prg on top of a signature check and then switching the prg image to random (in the actual proof we instead use an injective OWF to minimize assumptions). For the case $S_1 \setminus S_0 = S_{\mathsf{suf}}$ it is enough to do one step, and for the case $S_1 \setminus S_0 = S_{\mathsf{suf}} \setminus \{m\}$ we need $n - t$ steps.[38]

Symmetry argument is essentially a Sahai-Waters [SW14] symmetry argument in the proof of deniable encryption, with a difference that they didn't use ACE as an abstraction, and we instead decided to formulate it on ACE level to shorten the main proof of deniable encryption. The proof follows essentially the same steps, except that, since we have more signatures, we also need to argue that in the proof the decryption key can be punctured at a certain set of points (this is done using an argument similar to the proof of security of constrained decryption, since encryption key is already punctured on those points). Indeed, the proof of [SW14] uses the fact that the (only) signature uniquely defines the plaintext. This is not true in our case anymore, since some signatures only define the corresponding prefix of the plaintext. This introduces "bad" plaintexts which we need to get rid of. To do this, we rely on the fact that $S \setminus \{m^*\}$ can be represented as a union of S_{suf_i}, where all suf_i are different from suffixes of m^*.

Construction of relaxed ACE. The construction of relaxed ACE is the same as the construction of ACE from [CHJV14], except that we use different programs. Namely, let $F_t, \ldots, F_n$ be injective PRFs with sparse images, mapping $t, \ldots, n$ bits, respectively, to $n_{\mathsf{out}} = O(\lambda)$ bits. Let F be a PRF mapping n_{out} bits to $O(\lambda)$ bits. Then a (possibly punctured) encryption key is obfuscated $\mathcal{G}_{\mathsf{Enc}}(m)$, a (possibly punctured) decryption key is obfuscated $\mathcal{G}_{\mathsf{Dec}}(m)$, and a ciphertext-based punctured key is obfuscated $\mathcal{G}_{\mathsf{Puncture}}(c)[c^{(0)}, c^{(1)}]$, where

[37] In the construction of deniable encryption, $t = |\ell_0|$ for the sender ACE and $t = |\mathsf{prg}(\rho)|$ for the receiver ACE.

[38] We write S_0, S_1 (sets to puncture keys at in the security game) and S_{suf} (a set denoting all strings ending with suf), somewhat abusing the notation, since the subscript means an index in the former case and a prefix in the latter. However, all our suffixes are of length at least t, so there should be no confusion.

Programs of relaxed ACE.

Program $\mathcal{G}_{\mathsf{Enc}}(m)$

Inputs: message m.

Hardwired values: keys $K_t, \ldots, K_n, K$ of PRFs $F_t, \ldots, F_n, F$; circuit C_U describing set U. Parameters t, n.

1. If $C_U(m)$ then return $\perp$;
2. For each $i = t, \ldots, n$ set $\alpha_i \leftarrow F_i(K_i; \mathsf{suffix}_i(m))$;
3. Set $\beta \leftarrow F(K; \alpha_n) \oplus m$;
4. Return $(\alpha_t, \ldots, \alpha_n, \beta)$.

Program $\mathcal{G}_{\mathsf{Dec}}(c)$

Inputs: ciphertext c.

Hardwired values: keys $K_t, \ldots, K_n, K$ of PRFs $F_t, \ldots, F_n, F$; circuit C_S. Parameters t, n.

1. Parse $c = (\alpha_t, \ldots, \alpha_n, \beta)$;
2. Set $m \leftarrow F(K; \alpha_n) \oplus \beta$
3. If $C_S(m)$ then return $\perp$;
4. For each $i = t, \ldots, n$ do: if $\alpha_i \neq F_i(K_i; \mathsf{suffix}_i(m))$ then return $\perp$;
5. Return m.

Program $\mathcal{G}_{\mathsf{Puncture}}(c)$

Inputs: ciphertext c.

Hardwired values: keys $K_t, \ldots, K_n, K$ of PRFs $F_t, \ldots, F_n, F$. In addition, strings $c^{(0)}$ and $c^{(1)}$, hardwired in lexicographic order. Parameters t, n.

1. If $c = c^{(0)}$ or $c = c^{(1)}$ then return $\perp$; *($c^{(0)}$ and $c^{(1)}$ are written in lexicographic order)*
2. Parse $c = (\alpha_t, \ldots, \alpha_n, \beta)$;
3. Set $m \leftarrow F(K; \alpha_n) \oplus \beta$
4. For each $i = t, \ldots, n$ do: if $\alpha_i \neq F_i(K_i; \mathsf{suffix}_i(m))$ then return $\perp$;
5. Return m.

Figure 126: Programs of constrained keys of relaxed ACE. By $\mathsf{suffix}_i(m)$ we denote $m_{n-i+1}, \ldots, m_n$.

one of $c^{(0)}, c^{(1)}$ is a valid ciphertext and the other is randomly chosen. Programs can be found on fig. 126.

Theorem 4. *Assuming* iO *and injective one way functions, the construction of [CHJV14] instantiated with programs on fig. 126 is a relaxed ACE for plaintext length* n *and suffix parameter* t*. Concretely, assuming* iO *is* (t_1, ε_1)*-secure and one way function is* (t_2, ε_2)*-secure, and let* (t_3, ε_3) *be such that* $\varepsilon_3 \geq \varepsilon_1^{o(1)}$*, and* $t_3 \cdot \frac{1}{\varepsilon_1}(n-t) = O(t_2)$*.*

Then the resulting ACE is $(\min(t_1, t_3), O((n-t) \cdot (\varepsilon_1 + \varepsilon_2 + \varepsilon)))$*-secure.*

Proof. **Correctness.** All necessary correctness properties follow from correctness of iO, injectivity of PRFs and can be immediately verified.

Security of constrained decryption with negligible advantage. We prove security for a harder case of $S_1 \setminus S_0 = S_{\mathsf{suf}_t} \setminus \{m^*\}$ (the case when $S_1 \setminus S_0 = S_{\mathsf{suf}_t}$ can be shown by doing a single step of this proof for the PRF F_t). Note that $S_1 \setminus S_0 = S_{\mathsf{suf}_t} \setminus \{m^*\}$ can be represented as $S_{\mathsf{suf}_n} \cup \ldots \cup S_{\mathsf{suf}_{t+1}}$, where $\mathsf{suf}_n = \overline{m_1^*}, m_2^*, \ldots, m_n^*$, $\mathsf{suf}_{n-1} = \overline{m_2^*}, m_3^*, \ldots, m_n^*$, $\mathsf{suf}_{t+1} = \overline{m_{n-t+1}^*}, m_{n-t+2}^*, \ldots, m_n^*$.

We start with a distribution corresponding to the key DK which is punctured at S_0 (which we denote by

Hyb_0) and eventially reach a distribution where the key DK is punctured at S_1 (which we denote by $\mathsf{Hyb}_{n,5}$). We show indistinguishability via a sequence of hybrids $\mathsf{Hyb}_{j,k}$ for $j = t+1, \ldots, n$, $k = 0, \ldots, 5$. Programs can be found on fig. 127:

- Hyb_0 corresponds to the game where DK is punctured at S_0, i.e. the adversary gets $(EK\{U\}, DK\{S_0\})$.
- $\mathsf{Hyb}_{j,0}$: the adversary gets $(EK\{U\}, DK^{j,0})$, where DK_i is an obfuscation of a program $\mathcal{G}^{j,0}_{\mathsf{Dec}}$ (fig. 127). Note that when $j = t+1$, $\mathsf{Hyb}_{j,0} = \mathsf{Hyb}_0$.
- $\mathsf{Hyb}_{j,1}$: the adversary gets $(EK^{j,1}, DK^{j,1})$, where $DK^{j,1}$ is an obfuscation of a program $\mathcal{G}^{j,1}_{\mathsf{Dec}}$, where $z^* = F_j(K_j; \mathsf{suf}_j))$, and $EK^{j,1}$ is an obfuscation of $\mathcal{G}^{j,1}_{\mathsf{Enc}}$. Indistinguishability from the previous hybrid follows from iO, since both pairs of programs have the same functionality. Indeed, in $\mathcal{G}^{j,0}_{\mathsf{Dec}}$ and $\mathcal{G}^{j,1}_{\mathsf{Dec}}$ we replaced the condition $\alpha_j = F_j(K_j; \mathsf{suffix}_j(m))$ with two different checks for the case $\mathsf{suffix}_j(m) \neq \mathsf{suf}_j$ and $\mathsf{suffix}_j(m) = \mathsf{suf}_j$. For the former, we didn't change the check (but punctured the key K_j at suf_j), and for the latter, we replaced the check $\alpha_j = F_j(K_j; \mathsf{suf}_j)$ with the check $g(\alpha_j) = z^*$, where $z^* = g(F_j(K_j; \mathsf{suf}_j))$. Since g is injective, this doesn't change the functionality.

  In $\mathcal{G}^{j,1}_{\mathsf{Enc}}$ we punctured the key K_j at suf_j. This is without changing the functionality, since the program outputs $\perp$ on input $m \in S_{\mathsf{suf}_j} \subset U$.
- $\mathsf{Hyb}_{j,2}$: the adversary gets $(EK^{j,1}, DK^{j,1})$, where $DK^{j,1}$ is an obfuscation of a program $\mathcal{G}^{j,1}_{\mathsf{Dec}}$, where $z^* = g(y^*)$ for random y^*, and $EK^{j,1}$ is an obfuscation of $\mathcal{G}^{j,1}_{\mathsf{Enc}}$. Indistinguishability holds by security of a punctured PRF F_j at suf_j.
- $\mathsf{Hyb}_{j,3}$: the adversary gets $(EK^{j,1}, DK^{j,3})$, where $DK^{j,3}$ is an obfuscation of a program $\mathcal{G}^{j,3}_{\mathsf{Dec}}$, where $z^* = g(y^*)$ for random y^*, and $EK^{j,1}$ is an obfuscation of $\mathcal{G}^{j,1}_{\mathsf{Enc}}$. In other words, we instruct the program to output $\perp$ instead of m when $g(\alpha_j) = z^*$.

  Similar to lemma 1 from [BCP14], we argue that if any adversary can distinguish between hybrids $\mathsf{Hyb}_{j,2}$ and $\mathsf{Hyb}_{j,3}$ and iO is secure, then we can invert the one-way function g. Note that in our case programs differ on exponentially many inputs; however, differing inputs are a subset of $\{\alpha_t, \ldots, \alpha_j = y^*, \ldots, \alpha_n, \beta\}$, where $y^* = g^{-1}(z^*)$ and other values can be arbitrary. In other words, differing inputs share the block y^*, and we can do binary search over y^* similar to how the proof of lemma 1 does a binary search over a single differing input.

  More concretely, the extractor works as follows. It creates a program M which on input $\alpha_t, \ldots, \alpha_j, \ldots, \alpha_n, \beta$ first checks if $\alpha_j < y'$ (where y' is a binary search guess for y^*, i.e. in the first iteration $y' = 2^{|\alpha_j|}/2$). If so, then M executes $\mathcal{G}^{j,1}_{\mathsf{Dec}}$, otherwise it executes $\mathcal{G}^{j,3}_{\mathsf{Dec}}$. Note that if $y^* < y'$, then M is functionally equivalent to $\mathcal{G}^{j,1}_{\mathsf{Dec}}$, and if $y^* \geq y'$, then M is functionally equivalent to $\mathcal{G}^{j,3}_{\mathsf{Dec}}$. (Indeed, if $y^* < y'$, then for all input $\alpha_j \geq y'$ the line with the check $g(\alpha_j) = z^*$ in both $\mathcal{G}^{j,1}_{\mathsf{Dec}}$, $\mathcal{G}^{j,3}_{\mathsf{Dec}}$ will never be executed, since g is injective and its only preimage $y^* < y'$. Since this is the only difference in the programs, these programs are functionally equivalent for the case $\alpha_j \geq y'$, and therefore for all inputs M is functionally equivalent to $\mathcal{G}^{j,1}_{\mathsf{Dec}}$. The case $y^* \geq y'$ can be analyzed similarly). If by assumption there is an adversary which distinguishes between $\mathsf{Hyb}_{j,2}$ and $\mathsf{Hyb}_{j,3}$ with probability at least η and iO is ν-secure, where $\nu = \eta^{o(1)}$, then the adversary can run the adversary $O(1/\eta)$ times, estimate its distinguishing probability, learn the first bit of y^*, and continue binary search similar to the proof of lemma 1.

- $\mathsf{Hyb}_{j,4}$: the adversary gets $(EK^{j,1}, DK^{j,3})$, where $DK^{j,3}$ is an obfuscation of a program $\mathcal{G}^{j,3}_{\mathsf{Dec}}$, where $z^* = g(F_j(K_j; \mathsf{suf}_j))$, and $EK^{j,1}$ is an obfuscation of $\mathcal{G}^{j,1}_{\mathsf{Enc}}$. In other words, we switch y^* back to $F_j(K_j; \mathsf{suf}_j)$ from random. Indistinguishability holds by security of a punctured PRF F_j at suf_j.

- $\mathsf{Hyb}_{j,5}$: the adversary gets $(EK\{U\}, DK^{j+1,0})$, where $DK^{j+1,0}$ is an obfuscation of a program $\mathcal{G}^{j+1,0}_{\mathsf{Dec}}$. In other words, we unpuncture the key K_j at suf_j, and, since the program now always returns $\perp$ when $\mathsf{suffix}_j(m) = \mathsf{suf}_j$, we remove the line with z^*-check and instead make the program output $\perp$ when $m \in S_{\mathsf{suf}_j}$. indistinguishability holds by iO, since this doesn't change the functionality (the reasoning why the key can be unpunctured is the same as in $\mathsf{Hyb}_{j,1}$).

  Note that $\mathsf{Hyb}_{j,5} = \mathsf{Hyb}_{j+1,0}$.

Note that in $\mathsf{Hyb}_{n,5}$ program $\mathcal{G}^{n+1,0}_{\mathsf{Dec}}$ outputs $\perp$ when $s \in S_0$ or $m \in S_{\mathsf{suf}_n} \cup \ldots \cup S_{\mathsf{suf}_{t+1}} = S_1 \setminus S_0$. In other words, it outputs $\perp$ when $m \in S_1$, and thus this program is equivalent to $DK\{S_1\}$, which concludes security proof.

Finally, note that security loss depends only logarithmically on the size of $S_1 \setminus S_0$, as required by security of constrained decryption of relaxed ACE.

Programs of relaxed ACE.

Program $\mathcal{G}^{j,1}_{\mathsf{Enc}}(m)$

Inputs: message m.

Hardwired values: keys $K_t, \ldots, K_n, K$ (where $K_j\{\mathsf{suf}_j\}$ is punctured at suf_j) of PRFs $F_t, \ldots, F_n, F$; circuit C_U describing set U. Parameters t, n.

1. If $C_U(m)$ then return $\perp$;
2. For each $i = t, \ldots, n, i \neq j$, set $\alpha_i \leftarrow F_i(K_i; \mathsf{suffix}_i(m))$; set $\alpha_j \leftarrow F_j(K_j\{\mathsf{suf}_j\}; \mathsf{suffix}_i(m))$;
3. Set $\beta \leftarrow F(K; \alpha_n) \oplus m$;
4. Return $(\alpha_t, \ldots, \alpha_n, \beta)$.

Program $\mathcal{G}^{j,0}_{\mathsf{Dec}}(c)$

Inputs: ciphertext c.

Hardwired values: keys $K_t, \ldots, K_n, K$ of PRFs $F_t, \ldots, F_n, F$; circuit C_{S_0}. Parameters t, n. Set of suffixes $\mathsf{suf}_n, \ldots, \mathsf{suf}_{t+1}$ describing $S_1 \setminus S_0$.

1. Parse $c = (\alpha_t, \ldots, \alpha_n, \beta)$;
2. Set $m \leftarrow F(K; \alpha_n) \oplus \beta$
3. If $C_{S_0}(m)$ then return $\perp$;
4. If $m \in S_{\mathsf{suf}_{j-1}} \cup S_{\mathsf{suf}_{j-2}} \cup \ldots \cup S_{\mathsf{suf}_{t+2}} \cup S_{\mathsf{suf}_{t+1}}$ then return $\perp$;
5. For each $i = t, \ldots, n$ do: if $\alpha_i \neq F_i(K_i; \mathsf{suffix}_i(m))$ then return $\perp$;
6. Return m.

Program $\mathcal{G}^{j,1}_{\mathsf{Dec}}(c)$

Inputs: ciphertext c.

Hardwired values: keys $K_t, \ldots, K_n, K$ (where $K_j\{\mathsf{suf}_j\}$ is punctured at suf_j) of PRFs $F_t, \ldots, F_n, F$; circuit C_{S_0}. Parameters t, n. Set of suffixes $\mathsf{suf}_n, \ldots, \mathsf{suf}_{t+1}$ describing $S_1 \setminus S_0$, injective owf g, value z^*.

1. Parse $c = (\alpha_t, \ldots, \alpha_n, \beta)$;
2. Set $m \leftarrow F(K; \alpha_n) \oplus \beta$
3. If $C_{S_0}(m)$ then return $\perp$;
4. If $m \in S_{\mathsf{suf}_{j-1}} \cup \ldots \cup S_{\mathsf{suf}_{t+1}}$ then return $\perp$;
5. For each $i = t, \ldots, n$, $i \neq j$ do: if $\alpha_i \neq F_i(K_i\{\mathsf{suf}_j\}; \mathsf{suffix}_i(m))$ then return $\perp$;
6. If $\mathsf{suffix}_j(m) = \mathsf{suf}_j$ then: if $g(\alpha_j) = z^*$ then return m, else return $\perp$;
7. If $\mathsf{suffix}_j(m) \neq \mathsf{suf}_j$ then: if $\alpha_j = F_j(K_j\{\mathsf{suf}_j\}; \mathsf{suffix}_j(m))$ then return m, else return $\perp$.

Program $\mathcal{G}^{j,3}_{\mathsf{Dec}}(c)$

Inputs: ciphertext c.

Hardwired values: keys $K_t, \ldots, K_n, K$ (where $K_j\{\mathsf{suf}_j\}$ is punctured at suf_j) of PRFs $F_t, \ldots, F_n, F$; circuit C_{S_0}. Parameters t, n. Set of suffixes $\mathsf{suf}_n, \ldots, \mathsf{suf}_{t+1}$ describing $S_1 \setminus S_0$, injective owf g, value z^*.

1. Parse $c = (\alpha_t, \ldots, \alpha_n, \beta)$;
2. Set $m \leftarrow F(K; \alpha_n) \oplus \beta$
3. If $C_{S_0}(m)$ then return $\perp$;
4. If $m \in S_{\mathsf{suf}_{j-1}} \cup \ldots \cup S_{\mathsf{suf}_{t+1}}$ then return $\perp$;
5. For each $i = t, \ldots, n$, $i \neq j$ do: if $\alpha_i \neq F_i(K_i\{\mathsf{suf}_j\}; \mathsf{suffix}_i(m))$ then return $\perp$;
6. If $\mathsf{suffix}_j(m) = \mathsf{suf}_j$ then: if $g(\alpha_j) = z^*$ then return $\perp$, else return $\perp$;
7. If $\mathsf{suffix}_j(m) \neq \mathsf{suf}_j$ then: if $\alpha_j = F_j(K_j\{\mathsf{suf}_j\}; \mathsf{suffix}_j(m))$ then return m, else return $\perp$.
8. Return $\perp$.

Figure 127: Programs used in the proof of security of constrained decryption of relaxed ACE.

Symmetry. Recall that from the definition of symmetry $U = S_{\mathsf{suf}_t}$ is a set of plaintexts ending with the same suffix of size t, and the challenge plaintext m^* ends with suf_t as well. Let $\mathsf{suf}^*_n, \ldots, \mathsf{suf}^*_t$ denote $n, \ldots, t$-long suffixes of m^* (note that $\mathsf{suf}_t = \mathsf{suf}^*_t$). Further, as in the proof of security of constrained decpryption, let $\mathsf{suf}_n, \ldots, \mathsf{suf}_{t+1}$ be such that $U \setminus \{m^*\} = S_{\mathsf{suf}_n} \cup \ldots \cup S_{\mathsf{suf}_{t+1}}$. (Note that for each $i = t+1, \ldots, n$ suf_i and suf^*_i only differ in the first bit).

We show symmetry of ACE in a sequence of hybrids, for $b = 0, 1$. Programs can be found on fig. 128.

- Hyb^b_0: The distribution in this hybrid is $(c^{(0)}, c^{(1)}, EK\{U\}, DK\{c^{(0)}, c^{(1)}\})$, where c_b is randomly chosen and c_{1-b} is $\mathsf{Enc}(EK, m^*)$.
- Hyb^b_1: The distribution in this hybrid is $(c^{(0)}, c^{(1)}, EK', DK')$, where EK', DK' are instead obfuscations of programs $\mathcal{G}'_{\mathsf{Enc}}$ and $\mathcal{G}'_{\mathsf{Puncture}}$, respectively. Denote $c = (\alpha_t, \ldots, \alpha_n, \beta)$, and $c^{(0)}, c^{(1)}$ accordingly (in particular, $\alpha^{(1-b)}_n = F_n(K_n; m^*)$). (fig. 128).

  We argue that indistinguishability between Hyb^b_0 and Hyb^b_1 for any b holds by iO. Indeed, since for all $i = t, \ldots, n$ $S_{\mathsf{suf}^*_i} \subset U$ and $S_{\mathsf{suf}_i} \subset U$, $\mathcal{G}'_{\mathsf{Enc}}$ outputs $\perp$ on any input $m \in S_{\mathsf{suf}^*_i}$ or $m \in S_{\mathsf{suf}_i}$, for all $i = t, \ldots, n$, anyway and thus each F_i is never computed on $\mathsf{suf}^*_i, \mathsf{suf}_i$, $i = t, \ldots, n$. Thus we can puncture each F_i at $\mathsf{suf}^*_i, \mathsf{suf}_i$, $i = t, \ldots, n$ (note that $\mathsf{suf}_t = \mathsf{suf}^*_t$ and thus F_t is only punctured once). Further, since F_n is injective, and is never run on $\mathsf{suf}^*_n = m^*$, F is never computed on $\alpha^{(1-b)}_n = F_n(K_n; m^*)$, thus we can puncture K at $\alpha^{(1-b)}_n$. Finally, since $\alpha^{(b)}_n$ is randomly chosen and F_n has sparse image, with overwhelming probability $\alpha^{(b)}_n$ is outside of the image of F_n and we can puncture key K at $\alpha^{(b)}_n$ as well.

  In $\mathcal{G}'_{\mathsf{Puncture}}$ we can puncture K at $\alpha^{(0)}_n, \alpha^{(1)}_n$ since before that there is an instruction to output $\perp$ if α_n is equal to one of these values. We argue that this instruction doesn't change the functionality: indeed, $\alpha^{(b)}_n$ is outside of the image of F_n with high probability and therefore the program would reject anyway. Next, if $\alpha = \alpha^{(1-b)}_n$, since F_n is injective, the only way to satisfy the F_n-check is to have $\beta = F(K; \alpha^{(1-b)}_n) \oplus m^* = \beta^{(1-b)}$. But then, to satisfy other PRF checks, $\alpha_t, \ldots, \alpha_{n-1}$ should be equal to $\alpha^{(1-b)}_t, \ldots, \alpha^{(1-b)}_{n-1}$, in which case $c = c^{(1-b)}$ and the program outputs $\perp$ in the very beginning.
- Hyb^b_2: The distribution in this hybrid is $(c^{(0)}, c^{(1)}, EK', DK'')$, where EK', DK'' are obfuscations of programs $\mathcal{G}'_{\mathsf{Enc}}$ and $\mathcal{G}''_{\mathsf{Puncture}}$, respectively. In other words, we instruct the program $\mathcal{G}''_{\mathsf{Puncture}}$ to output $\perp$ if $m \in S_{\mathsf{suf}_n} \cup S_{\mathsf{suf}_{n-1}} \cup \ldots \cup S_{\mathsf{suf}_{t+2}} \cup S_{\mathsf{suf}_{t+1}}$. Indistinguishability of this hybrid can be shown similarly to the proof of the security of constrained decryption. That is, for each suf_i, $i = t+1, \ldots, n$, we can make this program reject all $m \in S_{\mathsf{suf}_i}$ by puncturing the PRF F_i, changing $F_i(K_i; \mathsf{suf}_i)$ to random, replacing the PRF check with OWF check, and arguing that the program can abort (instead of outputting m) if OWF check passes, since otherwise OWF can be inverted. (Importantly, note that indeed the value $F_i(K_i; \mathsf{suf}_i)$, for $i = t+1, \ldots, n$, isn't used anywhere else in the distribution: in particular, it is not required to compute $c^{(0)}$ or $c^{(1)}$, and moreover program $\mathcal{G}'_{\mathsf{Enc}}$ only uses a punctured key $K_i\{\mathsf{suf}_i\}$).

  Indistinguishability holds by security of punctured PRFs $F_{t+1}, \ldots, F_n$, one-wayness of injective OWF, and security of iO.
- Hyb^b_3: The distribution in this hybrid is $(c^{(0)}, c^{(1)}, EK', DK''')$, where EK', DK''' are obfuscations of programs $\mathcal{G}'_{\mathsf{Enc}}$ and $\mathcal{G}'''_{\mathsf{Puncture}}$, respectively. In other words, we instruct program $\mathcal{G}'''_{\mathsf{Puncture}}$ to output $\perp$ when $m \in S_{\mathsf{suf}_t}$. We argue this doesn't change the functionality. Indeed, the condition "$m \in S_{\mathsf{suf}_n} \cup S_{\mathsf{suf}_{n-1}} \cup \ldots \cup S_{\mathsf{suf}_{t+1}}$" covers all $m \in S_{\mathsf{suf}_t}$ except m^*. Therefore requiring to output $\perp$ when

$m \in S_{\mathsf{suf}_t}$ is equivalent to additionally ask to output $\perp$ when $m = m^*$. However, when $m = m^*$, $c = c^{(1-b)}$ and therefore the program outputs $\perp$ in the very beginning.

Further, in program $\mathcal{G}'''_{\mathsf{Puncture}}$ we puncture all keys K_i, $i = t, \ldots, n$, at suf^*_i. This can be done since the program never needs to compute any of these values since when $m \in S_{\mathsf{suf}_t}$, the program outputs $\perp$.

- Hyb^b_4: The distribution in this hybrid is $(c^{(0)}, c^{(1)}, EK', DK''')$, where EK', DK''' are obfuscations of programs $\mathcal{G}'_{\mathsf{Enc}}$ and $\mathcal{G}'''_{\mathsf{Puncture}}$, respectively, and $c^{(1-b)}$ is chosen at random instead of as a result of PRFs. Security holds by security of PRFs $F, F_t, \ldots, F_n$ punctured at $\alpha^{(b)}_n, \mathsf{suf}^*_t, \ldots, \mathsf{suf}^*_n$, respectively.

Finally, note that the distributions in Hyb^0_4 and Hyb^1_4 are the same. Thus concludes the proof of the symmetry of ACE. □

D Encrypting longer plaintexts

Our main security proof holds for the case when 1-bit plaintexts are used. Here we outline the changes in the proof when the scheme is used to encrypt long plaintexts from some plaintext space $\mathcal{M}$.

The only change is that in the proof of indistinguishability of explanations of the receiver (lemma 55), instead of eliminating a single complementary ciphertext $\overline{\mu_3{}^*} = \mathsf{ACE.Enc}_{\mathsf{EK}}(1 \oplus m^*_0, \mu_1{}^*, \mu_2{}^*, L^*_0)$, we need to eliminate all complementary ciphertexts $\{\mathsf{ACE.Enc}_{\mathsf{EK}}(m, \mu_1{}^*, \mu_2{}^*, L^*_0) : m \in \mathcal{M}, m \neq m^*_0\}$. This change is required both in the proof of deniability and off-the-record deniability.

Concretely, changes are the following:

- In hybrid $\mathsf{Hyb}_{B,1,5}$ (similarly, in $\mathsf{Hyb}_{B,3,2}$) in program P3 we puncture encryption key EK of the main ACE at all points $\{(m, \mu_1{}^*, \mu_2{}^*, L^*_0) : m \in \mathcal{M}, m \neq m^*_0\}$. Indistinguishability holds by the same reasoning as in the orginal proof. The description of the program P3 on fig. 99 should be changed accordingly.
- In hybrid $\mathsf{Hyb}_{B,1,6}$ (similarly, in $\mathsf{Hyb}_{B,3,1}$) we puncture decryption key DK of the main ACE at the same set of points $\overline{p} = \{(m, \mu_1{}^*, \mu_2{}^*, L^*_0) : m \in \mathcal{M}, m \neq m^*_0\}$. Indistinguishability holds by security of constrained decryption of ACE, since the corresponding encryption key EK is already punctured at these points. The description of the programs Dec, RFake on fig. 101 should be changed accordingly. *Note however that this incurs security loss proportional to $|\mathcal{M}|$, since security loss in constrained decryption game depends on the size of the punctured set.*

Thus the proof can be adapted to the case of longer plaintexts, with additional multiplicative factor of $|\mathcal{M}|$ in security loss. However, the resulting scheme is only statically secure, i.e. both real and fake plaintexts have to be fixed before the CRS is generated. To achieve adaptive security, one can guess both plaintexts in the proof and lose another factor of $|\mathcal{M}|^2$.

Thus the scheme can be used for encrypting and denying longer messages, albeit with additional multiplicative factor of $|\mathcal{M}|^3$ in security loss.

Programs of relaxed ACE.

Program $\mathcal{G}'_{\mathsf{Enc}}(m)$

Inputs: message m.

Hardwired values: punctured keys $K_t\{\mathsf{suf}_t^*, \mathsf{suf}_t\}, K_{t+1}\{\mathsf{suf}_{t+1}^*, \mathsf{suf}_{t+1}\}, \ldots, K_n\{\mathsf{suf}_n^*, \mathsf{suf}_n\}, K\{\alpha_n^{(0)}, \alpha_n^{(1)}\}$ of PRFs $F_t, \ldots, F_n, F$; circuit C_U describing set U. Parameters t, n.

1. If $C_U(m)$ then return $\perp$;
2. For each $i = t, \ldots, n$ set $\alpha_i \leftarrow F_i(K_i\{\mathsf{suf}_i^*, \mathsf{suf}_i\}; \mathsf{suffix}_i(m))$;
3. Set $\beta \leftarrow F(K\{\alpha_n^{(0)}, \alpha_n^{(1)}\}; \alpha_n) \oplus m$;
4. Return $(\alpha_t, \ldots, \alpha_n, \beta)$.

Program $\mathcal{G}'_{\mathsf{Puncture}}(c)$

Inputs: ciphertext c.

Hardwired values: keys $K_t, \ldots, K_n, K\{\alpha_n^{(0)}, \alpha_n^{(1)}\}$ of PRFs $F_t, \ldots, F_n, F$; circuit C_U describing set U. In addition, strings $c^{(0)}$ and $c^{(1)}$, hardwired in lexicographic order. Parameters t, n.

1. If $c = c^{(0)}$ or $c = c^{(1)}$ then return $\perp$; *($c^{(0)}$ and $c^{(1)}$ are written in lexicographic order)*
2. Parse $c = (\alpha_t, \ldots, \alpha_n, \beta)$; $c^{(0)} = (\alpha_t^{(0)}, \ldots, \alpha_n^{(0)}, \beta^{(0)})$; $c^{(1)} = (\alpha_t^{(1)}, \ldots, \alpha_n^{(1)}, \beta^{(1)})$;
3. If $\alpha_n = \alpha_n^{(0)}$ or $\alpha_n = \alpha_n^{(1)}$ then return $\perp$; *($\alpha_n^{(0)}$ and $\alpha_n^{(1)}$ are written in lexicographic order)*
4. Set $m \leftarrow F(K\{\alpha_n^{(0)}, \alpha_n^{(1)}\}; \alpha_n) \oplus \beta$;
5. For each $i = t, \ldots, n$ do: if $\alpha_i \neq F_i(K_i; \mathsf{suffix}_i(m))$ then return $\perp$;
6. Return m.

Program $\mathcal{G}''_{\mathsf{Puncture}}(c)$

Inputs: ciphertext c.

Hardwired values: keys $K_t, \ldots, K_n, K\{\alpha_n^{(0)}, \alpha_n^{(1)}\}$ of PRFs $F_t, \ldots, F_n, F$; circuit C_U describing set U. In addition, strings $c^{(0)}$ and $c^{(1)}$, hardwired in lexicographic order. Parameters t, n.

1. If $c = c^{(0)}$ or $c = c^{(1)}$ then return $\perp$; *($c^{(0)}$ and $c^{(1)}$ are written in lexicographic order)*
2. Parse $c = (\alpha_t, \ldots, \alpha_n, \beta)$; $c^{(0)} = (\alpha_t^{(0)}, \ldots, \alpha_n^{(0)}, \beta^{(0)})$; $c^{(1)} = (\alpha_t^{(1)}, \ldots, \alpha_n^{(1)}, \beta^{(1)})$;
3. If $\alpha_n = \alpha_n^{(0)}$ or $\alpha_n = \alpha_n^{(1)}$ then return $\perp$; *($\alpha_n^{(0)}$ and $\alpha_n^{(1)}$ are written in lexicographic order)*
4. Set $m \leftarrow F(K\{\alpha_n^{(0)}, \alpha_n^{(1)}\}; \alpha_n) \oplus \beta$;
5. If $m \in S_{\mathsf{suf}_n} \cup S_{\mathsf{suf}_{n-1}} \cup \ldots \cup S_{\mathsf{suf}_{t+2}} \cup S_{\mathsf{suf}_{t+1}}$ then return $\perp$;
6. For each $i = t, \ldots, n$ do: if $\alpha_i \neq F_i(K_i; \mathsf{suffix}_i(m))$ then return $\perp$;
7. Return m.

Program $\mathcal{G}'''_{\mathsf{Puncture}}(c)$

Inputs: ciphertext c.

Hardwired values: punctured keys $K_t\{\mathsf{suf}_t^*\}, \ldots, K_n\{\mathsf{suf}_n^*\}, K\{\alpha_n^{(0)}, \alpha_n^{(1)}\}$ of PRFs $F_t, \ldots, F_n, F$; circuit C_U describing set U. In addition, strings $c^{(0)}$ and $c^{(1)}$, hardwired in lexicographic order. Parameters t, n.

1. If $c = c^{(0)}$ or $c = c^{(1)}$ then return $\perp$; *($c^{(0)}$ and $c^{(1)}$ are written in lexicographic order)*
2. Parse $c = (\alpha_t, \ldots, \alpha_n, \beta)$; $c^{(0)} = (\alpha_t^{(0)}, \ldots, \alpha_n^{(0)}, \beta^{(0)})$; $c^{(1)} = (\alpha_t^{(1)}, \ldots, \alpha_n^{(1)}, \beta^{(1)})$;
3. If $\alpha_n = \alpha_n^{(0)}$ or $\alpha_n = \alpha_n^{(1)}$ then return $\perp$; *($\alpha_n^{(0)}$ and $\alpha_n^{(1)}$ are written in lexicographic order)*
4. Set $m \leftarrow F(K\{\alpha_n^{(0)}, \alpha_n^{(1)}\}; \alpha_n) \oplus \beta$;
5. If $m \in S_{\mathsf{suf}_t}$ then return $\perp$;
6. For each $i = t, \ldots, n$ do: if $\alpha_i \neq F_i(K_i\{\mathsf{suf}_i^*\}; \mathsf{suffix}_i(m))$ then return $\perp$;
7. Return m.

Figure 128: Programs of constrained keys. Note that everywhere where $c^{(0)}, c^{(1)}$ or $\alpha_n^{(0)}, \alpha_n^{(1)}$ appear, they are written in lexicographic order (in particular, in the GGM-based punctured PRF, key $K\{\alpha_n^{(0)}, \alpha_n^{(1)}\}$ doesn't depend on the order of puncturing and only depends on lexicographically sorted set $\left\{\alpha_n^{(0)}, \alpha_n^{(1)}\right\}$). For convenience we denote the punctured K_t by $K_t\{\mathsf{suf}_t^*, \mathsf{suf}_t\}$ (similar to other keys), even though $\mathsf{suf}_t^* = \mathsf{suf}_t$ and the key is only punctured at one point.