A More Complete Analysis of the Signal Double Ratchet Algorithm

Alexander Bienstock[1], Jaiden Fairoze[2], Sanjam Garg[2,4], Pratyay Mukherjee[3], and Srinivasan Raghuraman[5]

[1]New York University
[2]UC Berkeley
[3]Swirlds Labs
[4]NTT Research
[5]Visa Research

Abstract

Seminal works by Cohn-Gordon, Cremers, Dowling, Garratt, and Stebila [Journal of Cryptology 2020] and Alwen, Coretti and Dodis [EUROCRYPT 2019] provided the first formal frameworks for studying the widely-used Signal Double Ratchet (DR for short) algorithm.

In this work, we develop a new Universally Composable (UC) definition $\mathcal{F}_{\mathsf{DR}}$ that we show is provably achieved by the DR protocol. Our definition captures not only the security and correctness guarantees of the DR already identified in the prior state-of-the-art analyses of Cohn-Gordon *et al.* and Alwen *et al.*, but also *more* guarantees that are absent from one or *both* of these works. In particular, we construct *six* different modified versions of the DR protocol, all of which are insecure according to our definition $\mathcal{F}_{\mathsf{DR}}$, but remain secure according to one (or both) of their definitions. For example, our definition is the first to capture CCA-style attacks possible immediately after a compromise — attacks that, as we show, the DR protocol provably resists, but were not captured by prior definitions.

We additionally show that multiple compromises of a party in a short time interval, which the DR is expected to be able to withstand, as we understand from its whitepaper, nonetheless introduce a new non-trivial (albeit minor) weakness of the DR. Since the definitions in the literature (including our $\mathcal{F}_{\mathsf{DR}}$ above) do not capture security against this more nuanced scenario, we define a new stronger definition $\mathcal{F}_{\mathsf{TR}}$ that does.

Finally, we provide a *minimalistic modification* to the DR (that we call the Triple Ratchet, or TR for short) and show that the resulting protocol securely realizes the stronger functionality $\mathcal{F}_{\mathsf{TR}}$. Remarkably, the modification incurs no additional communication cost and virtually no additional computational cost. We also show that these techniques can be used to improve communication costs in other scenarios, e.g. practical Updatable Public Key Encryption schemes and the re-randomized TreeKEM protocol of Alwen *et al.* [CRYPTO 2020] for Secure Group Messaging.

Contents

1 Introduction **4**
- 1.1 Our Contributions . . . 4
- 1.2 High-Level Summary of the Double Ratchet . . . 5
- 1.3 High-Level Summary of our New DR Definition . . . 8
- 1.4 High-Level Summary of the DR's Minor Weakness . . . 10
- 1.5 High-Level Summary of the Triple Ratchet . . . 11
- 1.6 Other Related Work . . . 12
- 1.7 Summary of the Rest of the Paper . . . 12

2 Preliminaries **13**
- 2.1 Game-Based Security and Notation . . . 13
- 2.2 Authenticated Encryption . . . 14

3 Defining Security of the Double Ratchet **16**
- 3.1 Honest Execution . . . 17
- 3.2 Execution with an Unrestricted Adversary . . . 18

4 Building Blocks **23**
- 4.1 Key Derivation Function Chains . . . 23
  - 4.1.1 Differences from ACD . . . 24
- 4.2 Forward-Secure AEAD . . . 24
  - 4.2.1 Defining FS-AEAD . . . 24
  - 4.2.2 Differences from ACD . . . 27
  - 4.2.3 Instantiating FS-AEAD . . . 27
- 4.3 Continuous Key Agreement . . . 29
  - 4.3.1 Defining CKA . . . 29
  - 4.3.2 Differences from ACD . . . 31
  - 4.3.3 Instantiating CKA . . . 33
  - 4.3.4 Instantiating CKA^+ . . . 34
  - 4.3.5 Even stronger security for CKA^+ . . . 35

5 Composition **36**
- 5.1 Constructions . . . 36
- 5.2 Vulnerability of the DR with Respect to $\mathcal{F}_{\mathsf{TR}}$ . . . 39

6 UC Security of the DR and TR **40**
- 6.1 Hybrid Algorithms H_{t^*} and the Simulator $\mathcal{S}$ . . . 41
- 6.2 Type 1 Adversaries . . . 42
- 6.3 Type 2 Adversaries . . . 44
- 6.4 Type 3 Adversaries . . . 46
- 6.5 Even Stronger Security of the TR with CKA^+ . . . 48

7 More Efficient Updatable Public-Key Encryption **49**
- 7.1 UPKE Construction from [JMM19a, ACDT20] . . . 50
- 7.2 More Efficient Construction . . . 51

A Comparison to the ACD and CCD$^+$ Secure Messaging Security Notions **57**
A.1 Definitions from ACD . . . 57
A.1.1 Secure Messaging (ACD) . . . 58
A.1.2 ACD's CKA Definition . . . 59
A.1.3 ACD's FS-AEAD Definition . . . 61
A.1.4 ACD's PRF-PRNG Defintion . . . 62
A.1.5 ACD's Composition . . . 62
A.2 Transformations to the DR and Their (In)Security . . . 62
A.2.1 T_1: Postponed FS-AEAD Key Deletion . . . 63
A.2.2 T_2: Postponed CKA Key Deletion . . . 64
A.2.3 T_3 : Eager CKA Randomness Sampling . . . 66
A.2.4 T_4: Malleable Ciphertexts . . . 68
A.2.5 T_5: CKA Bad Randomness Plaintext Trigger . . . 70
A.2.6 T_6: Removed Immediate Decryption . . . 71

B The Model in Detail **73**
B.1 UC Security: A Brief Overview . . . 73
B.1.1 The Basic Model of Computation . . . 73
B.1.2 Security of Protocols. . . . 74

1 Introduction

Background. The Signal protocol is by far the most popular end-to-end secure messaging (SM) protocol, boasting of billions of users. Based on the Off-The-Record protocol [BGB04], the core underlying technique of the Signal protocol is commonly known as the *Double Ratchet* (DR) algorithm. The DR is beautifully explained in the whitepaper [MP16a] authored by the creators of Signal, Marlinspike and Perrin. The whitepaper also outlines the desired security properties of the DR, and provides intuitions on the design rationale for achieving them. Indeed, in addition to standard security against an eavesdropper who may modify ciphertexts, the DR attempts to achieve (i) *post-compromise security* (PCS) and *forward secrecy* (FS) with respect to leakages of secret state, (ii) *resilience to bad randomness*, and (iii) *immediate decryption* (all at the same time). PCS requires the conversation to naturally and quickly recover security after a leakage on one of the (or both) parties, as long as the affected parties have good randomness (and the adversary remains passive while such recovery occurs). FS requires past messages to remain secure even after a leakage on one of the (or both) parties. Resilience to bad randomness requires that as long as both parties' secret states are secure (i.e., PCS has been achieved after any corruptions), then the conversation should remain secure, even if bad randomness is used in crafting messages. Finally, immediate decryption requires parties to — immediately upon reception of ciphertexts — obtain underlying plaintexts and place them in the correct order in the conversation, even if they arrive arbitrarily out of order and if some of them are completely lost by the network (the latter is also known as *message-loss resilience*).

However, despite the elegance and simplicity of the Double Ratchet, capturing its security turned out to be not so straightforward. The first formal analysis of the DR protocol (in fact, the whole Signal protocol) was provided by Cohn-Gordon *et al.* in the Journal of Cryptology (2020) [CCD+20] (referred to as CCD$^+$ henceforth). However, this analysis left open several questions about the cryptographic security and correctness achieved by the DR. Following in the footsteps of CCD$^+$, a more generic and comprehensive security definition of the DR was provided by Alwen *et al.* in Eurocrypt 2019 [ACD19] (referred to as ACD henceforth), with close focus on the immediate decryption property of the DR protocol. They provided a modular analysis with respect to game-based definitions proposed therein. Indeed, they introduced new abstract primitives and composed them into SM protocols (including the DR itself) that capture the above properties: Continuous Key Agreement (CKA), Forward-Secure Authenticated Encryption with Associated Data (FS-AEAD), and PRF-PRNGs. While the works of CCD$^+$ and ACD significantly improved our understanding of the DR, as we observe in this work, both definitional frameworks do not capture some of its security and functionality properties.

1.1 Our Contributions

In this work, our key aim is to develop a formal definitional framework that captures the security and correctness properties of the DR protocol as completely as possible. Moreover, we aspire for definitions that are simple to state and easy to build on (e.g., imagine executing a Private Set Intersection Protocol on top of the DR). More specifically:

- **New Definitional Framework for the DR:** We provide a new definition $\mathcal{F}_{\mathsf{DR}}$ for the DR protocol, in the Universal Composability [Can01] (UC) framework. Our definition captures all of the security and correctness guarantees of the DR provided by ACD's and CCD$^+$'s

definitions, but also *more* guarantees that are absent from one or *both* of these works. To demonstrate this, we construct *six* different (albeit somewhat contrived) modified versions of the DR protocol, all of which are insecure according to our definition, but remain secure according to ACD's and/or CCD$^+$'s definition. Some of these transformations are indeed based on analyzed (weaker) DR variants in the literature, while others are based on novel observations. For example, our definition is the first to capture CCA-style attacks that become possible on the DR immediately after a party has been compromised — attacks that, as we show, the DR provably resists, but were not captured by prior definitions. We provide an overview of our new definition's advantages in Section 1.3.

Finally, we prove that the the DR protocol, as it is described in the whitepaper [MP16a] (in its strongest form), securely realizes our ideal functionality $\mathcal{F}_{\mathsf{DR}}$. Our proof is modular and proceeds by expanding on ACD's modular definitional framework (see Section 6).

- **Non-trivial (albeit minor) weakness of the DR:** We find that multiple compromises of a party in a short time interval, which the DR should be able to withstand, as we understand from its whitepaper, nonetheless introduce a new non-trivial (albeit minor) weakness of the DR. This weakness is allowed in the definitions of both ACD and CCD$^+$, as well as $\mathcal{F}_{\mathsf{DR}}$, so we provide a new stronger definition $\mathcal{F}_{\mathsf{TR}}$ that does not allow it. We summarize this compromise scenario in Section 1.4.

- **Achieving stronger security:** Finally, we complement the above weakness by providing a minimalistic modification to the DR and prove the resulting protocol secure according to the stronger definition $\mathcal{F}_{\mathsf{TR}}$. We call this new protocol the Triple Ratchet (TR) as it adds another "mini ratchet" to the public ratchet in the DR Protocol. Remarkably, the modification incurs no additional communication cost and virtually no additional computational cost. We provide an overview of the TR in Section 1.5.

  We believe that the techniques realized here are also likely to find other applications. For instance, in Section 7, we show that our techniques can be used to improve current practical Updatable Public Key Encryption (UPKE) constructions [ACDT20,JMM19a], reducing their communication cost by an additive factor of $|G|$, where $|G|$ is the number of bits needed to represent the size of the (CDH-hard) group used in the construction, without any additional computational cost. Furthermore, the technique yields an improvement to the communication cost of the re-randomized TreeKEM (rTreeKEM) protocol of Alwen *et al.* [ACDT20] — specifically, improving the communication cost by up to roughly an additive factor of $|G| \cdot n$, where n is the number of users in the group.

1.2 High-Level Summary of the Double Ratchet

Before elaborating on our results in the subsequent sections, we first give a high-level overview of the Signal Double Ratchet. For another detailed description we refer to the Double Ratchet whitepaper [MP16a]. Readers familiar with the Double Ratchet algorithm could easily skip this section.

We note that although we here describe the double ratchet specifically in terms of its real-world implementation [MP16a], our paper still breaks it down into modular pieces which can be instantiated in several different ways, as in ACD. For the purpose of our paper, we assume that the two participants P_1 and P_2 share a common secret upon initialization. In Signal, this is achieved

via the X3DH key exchange protocol [MP16b], but we consider this out of scope for our study of the double ratchet. Using their initial shared secret, P_1 and P_2 can derive the initial *root key* σ which seeds the public ratchet. Furthermore, upon initialization P_2 also holds some secret exponent x_0 and P_1 holds the corresponding public value g^{x_0}. Once the initialization process completes, the ratcheting session begins.

At its core, the double ratchet has two key components: the outer public-key ratchet, and the inner symmetric-key ratchet (often referred to as simply the public and symmetric ratchets, respectively). ACD abstract out the symmetric ratchet as their FS-AEAD primitive, the update mechanism of the public ratchet as their PRF-PRNG primitive, and the means by which shared secrets are produced to update the public ratchet as their CKA primitive. The goal of the double ratchet is to provide distinct *message keys* to encrypt/decrypt each new message. For each message the same message key is derived by both parties using a symmetric *chain key* which itself is derived from the aforementioned root key. Naturally, this results in a key *hierarchy* with the root key at the top, chain keys at an intermediate layer, and message keys at the bottom. Observe a graphical depiction of this hierarchy in Figure 1. In the Signal double ratchet, Diffie-Hellman key exchange is used to "ratchet forward" the root key, which can then be used to establish corresponding symmetric chain keys. Message keys are then derived from the current (newest) chain key, where chain keys are updated deterministically such that multiple messages can be sent in a row before a response, and no matter which of these messages is the first to arrive, the recipient can always compute its corresponding message key *immediately*. We now introduce the concept of asynchronous epochs before describing the two ratchets and the primary properties which they achieve:

Asynchronous Sending Epochs. In the double ratchet, the parties P_1 and P_2 asynchronously alternate sending messages in *epochs* (as termed in [ACD19]): Assume that P_1 starts the conversation, sending in epoch 1 at least one message. Then once P_2 receives one of these messages, she sends messages in epoch 2. Furthermore, once P_1 receives one of these message, she starts epoch 3, and so on. We emphasize that these sending epochs are *asynchronous* – for example, even if P_2 has started sending in epoch 2, if P_1 has not yet received any such epoch 2 messages and wants to send new messages, she will still send them in epoch 1. Not until she finally receives one of P_2's epoch 2 messages will she send new messages in epoch 3.

Public Ratchet. The public ratchet forms the backbone of the double ratchet algorithm. Parties update the root key using public-key cryptography (i.e. Diffie-Hellman secrets) every time a new epoch is initiated: if P_1 wishes to start a new epoch, she must first update the root key using the Diffie-Hellman public value from P_2's latest epoch (or initialization). After deriving a new chain key from the root key, P_1 can send multiple separate messages in a row—this involves deriving a new message key for each message via the symmetric ratchet, as explained below.

We now describe the root key update process in more detail. To start a new epoch t, P_1 samples a new private exponent x_t and corresponding public value g^{x_t}. Next, she uses the public value received from P_2's latest epoch (or initialization), say $g^{x_{t-1}}$, to compute a shared secret $(g^{x_{t-1}})^{x_t} = g^{x_{t-1}x_t}$. Then, P_1 uses a two-input Key Derivation Function (KDF) to update the current root key and derive a new chain key in one go. That is, she computes $(\sigma_t, w_{t,1}) \leftarrow \mathrm{KDF}(\sigma_{t-1}, g^{x_{t-1}x_t})$. Observe that even if P_1's state was leaked before this update, as long as the parties used good randomness in sampling their Diffie-Hellman keys, the new root key and chain key will be secure. This is the key to achieving PCS. Symmetrically, even if P_1 uses bad randomness when performing this update, as

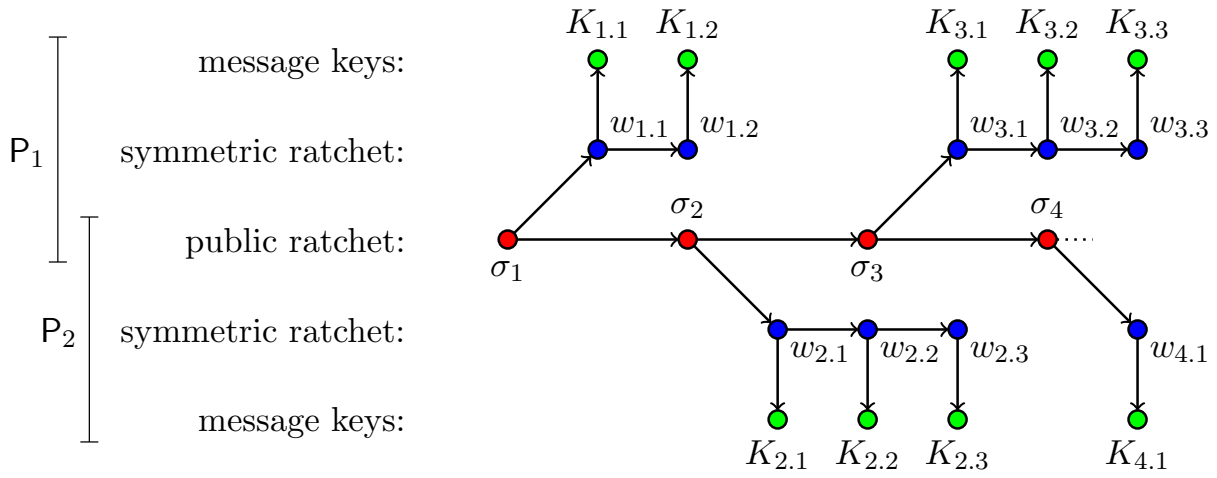

Figure 1: Sample Double Ratchet key evolution. In this depiction, P_1 sends and P_2 receives in epoch 1, followed by P_2 sending and P_1 receiving in epoch 2, and so on. As explained in the main body, initial symmetric chain keys $w_{i,1}$ for each epoch i are derived first by the sender, then also by the receiver, using the shared root keys σ_i and asynchronously exchanged shared secrets (via DDH). Then, updated symmetric chain keys $w_{i,j}$ and message keys $K_{i,j}$ are derived deterministically from $w_{i,1}$.

long as if σ_{t-1} was secure, then the new root key and chain key will be secure. Furthermore, root keys are clearly forward secret, from the security of the KDF and the fact that new Diffie-Hellman secrets are sampled independently of past ones.

P_1 includes in every message of the new epoch the fresh public share g^{x_t} to allow P_2 to compute the new shared secret $g^{x_{t-1}x_t}$ that is used to update the root key, no matter which message of the epoch she receives first. This in part is what provides for immediate decryption (and message loss resilience). When P_2 receives a message in P_1's new epoch, she recomputes the same above steps, i.e. she computes σ_t by first computing $(g^{x_t})^{x_{t-1}} = g^{x_{t-1}x_t}$ where x_{t-1} is P_2's own private share, followed by the same KDF computation. Once P_2 wishes to start her own new epoch, she generates another Diffie-Hellman pair $(x_{t+1}, g^{x_{t+1}})$ to ratchet the root key forward $(\sigma_{t+1}, w_{t+1,1}) \leftarrow \mathrm{KDF}(\sigma_t, g^{x_t x_{t+1}})$. Essentially, P_2 has refreshed her component of the Diffie-Hellman shared secret while reusing P_1's value from the previous epoch. Now, when P_1 receives a message for this epoch and again wishes to start a new one, she would similarly need to sample a new Diffie-Hellman share x_{t+2}. This process can continue asynchronously for as long as the session is active.

Symmetric Ratchet. The main purpose of the symmetric ratchet is to produce single-use symmetric keys for message encryption. When a party wishes to send (or receive) the next (ith) message in some epoch t, they derive a distinct message key $K_{t,i}$ from the symmetric chain key $w_{t,i}$ and simultaneously update the chain key. This is done by applying a KDF as follows: $(w_{t,i+1}, K_{t,i}) \leftarrow \mathrm{KDF}(w_{t,i})$ (if the KDF requires two inputs, a fixed value may be used to fill the other input). Observe that the symmetric ratchet is clearly forward secret from the security of the KDF. Note however that the symmetric ratchet does not have PCS due to its deterministic nature.

So, if P_1 just started a new epoch then she first computes initial symmetric chain key $w_{t,1}$ for the epoch as above. To derive a message key, P_1 puts this new chain key through the KDF to

compute $(w_{t,2}, K_{t,1}) \leftarrow \text{KDF}(w_{t,1})$. If P_1 wishes to send a second message, then she can derive $(w_{t,3}, K_{t,2}) \leftarrow \text{KDF}(w_{t,2})$. When P_2 receives these messages from P_1, she can repeat the key derivation in the same way as P_1 and use the subsequent message keys to decrypt the messages, no matter the order in which they arrive. The deterministic nature of the symmetric ratchet, along with including the public ratchet values in every message as above, provides immediate decryption.

1.3 High-Level Summary of our New DR Definition

In Section 3 we fully formalize our new definition for the DR in the UC framework, $\mathcal{F}_{\mathsf{DR}}$, and provide thorough discussion on it. Then in Section 6, we show that the DR UC-realizes $\mathcal{F}_{\mathsf{DR}}$. Intuitively, $\mathcal{F}_{\mathsf{DR}}$ captures all of the properties described in the last section, including all of those captured by the definitions of ACD and CCD$^+$. Here we emphasize several properties which $\mathcal{F}_{\mathsf{DR}}$ guarantees, but which ACD's and CCD$^+$'s definitions do not. We do so by providing *six* distinct transformations to the original DR protocol (denoted as $T_i(\mathsf{DR})$ for $i \in [6]$), showing their natural vulnerabilities here, and their insecurity according to $\mathcal{F}_{\mathsf{DR}}$, but security according to ACD's and CCD$^+$'s definitions. We show this formally in Appendix A.2. Although some of these transformations may be seen as artificial, they emphasize that our definition is stronger than those of ACD and CCD$^+$. Below we use the formalization of symmetric and public ratchets as done by ACD and also adapted by us – the symmetric ratchet is abstracted out as an *FS-AEAD* scheme and the public ratchet as a *CKA* scheme. We defer these definitions to Section 4.

T_1: Postponed FS-AEAD Key Deletion: This transformation slightly modifies the handling of symmetric ratchet secrets. In particular, when a party receives a new message for its counterpart's next epoch, it does not immediately delete its (no longer needed) symmetric ratchet secrets from its previous sending epoch. Instead, it waits to delete these secrets until it starts its next sending epoch (i.e., sends its next message). In that case, an injection attack can be launched as follows: only focusing on the symmetric ratchet, suppose that for a sending epoch t, P_1 derives $(w_{t,2}, K_{t,1}) \leftarrow \text{KDF}(w_{t,1})$ and sends an encrypted message using $K_{t,1}$, that is then received by P_2. Then P_2 sends a message in epoch $t+1$, which is received by P_1. Observe that unlike in (the strongest version of) the DR, $T_1(\mathsf{DR})$ keeps $w_{t,2}$ in P_1's memory even after receiving this epoch $t+1$ message from P_2. Now if P_1 is compromised then the attacker obtains $w_{t,2}$. Using this it can now launch an injection attack for P_1's sending epoch t (not just P_1's next sending epoch, $t+2$) by encrypting any arbitrary message of its choice using the next message key $(\cdot, K_{t,2}) \leftarrow \text{KDF}(w_{t,2})$ and sending that to P_2. Note that each time a sending epoch is started in the protocol, the information about how many messages were sent in the immediately past sending epoch is included. Nonetheless, that does not thwart this attack, because it is launched even before P_1 starts the next sending epoch.

Although this transformation is perhaps artificial, one can imagine scenarios in which the relative timing of messages sent by the two parties is important. Perhaps more importantly, it is clearly less secure than the standard (most secure version of) DR, but, remarkably, the version described by ACD is indeed $T_1(\mathsf{DR})$. Furthermore, as evident by ACD's security proof, their definition therefore does not require resistance against this attack; intuitively making our (and CCD$^+$'s) definition stronger than theirs in this respect.

T_2: Postponed CKA Key Deletion: A similar problem arises if the keys from the public ratchet are kept for too long. The transformed protocol works as follows: suppose that in starting a new sending epoch t, P_2 samples a secret exponent x_t and combines it with the public ratchet message

of P_1's prior sending epoch, $g^{x_{t-1}}$, to compute $I_t = g^{x_{t-1}x_t}$. Then, P_2 proceeds to send several messages using I_t (and the root key for the KDF, as described in Section 1.2) as normal. When receiving a message for the first time in sending epoch t of P_2, P_1 uses her stored secret exponent x_{t-1} and combines it with P_2's public ratchet message g^{x_t} to compute I_t. However, at this point, instead of deleting I_t (as done in the normal DR protocol), P_1 saves it in $T_2(\mathsf{DR})$. Now assume that P_1 receives all of P_2's epoch t messages. Then, when P_1 again switches to a new sending epoch she generates a new I_{t+1} (deleting the old I_t). An attack can be executed on $T_2(\mathsf{DR})$, by simply corrupting P_1 before the start of epoch $t+1$, and then using the leaked I_t to decrypt the already delivered messages sent by P_2 in epoch t – thus breaking forward security. Note: this also requires another corruption of P_2 *before* she sends messages in the attacked epoch t, to obtain the root key for the KDF. ACD's definition explicitly prevents querying the challenge oracle immediately after corruptions, and thus does not require resistance against this attack. CCD$^+$'s model does explicitly require resistance against this attack. This transformation may seem artificial, but clearly allowing the adversary to decrypt old messages should not be allowed in any formal model of the DR, and in fact is not allowed in $\mathcal{F}_{\mathsf{DR}}$.

T_3: Eager CKA Randomness Sampling: If the secret-exponent of a public-ratchet is sampled too early, then that makes the protocol vulnerable. For example, consider $T_3(\mathsf{DR})$ in which P_1 samples the exponent x_t for the next sending epoch t when still in receiving epoch $t-1$. An attacker may compromise P_1 to obtain x_t (and the root key) at this stage and use that to decrypt the messages sent in the next epoch t, thereby breaking PCS. ACD's definition does not require resistance against this attack, while $\mathcal{F}_{\mathsf{DR}}$ does, because their definition does not allow querying the challenge oracle immediately after corruptions. It is worth pointing out that the Double Ratchet whitepaper [MP16a] and CCD$^+$ present $T_3(\mathsf{DR})$ and its early sampling as their primary description of the DR (though the whitepaper later suggests deferring randomness sampling until actually sending, for better security). Therefore also CCD$^+$'s security model does not require resistance against this attack (as they prove $T_3(\mathsf{DR})$ secure in it).

T_4: Malleable Ciphertexts: If the protocol does not provide a strong non-malleability guarantee, then the DR protocol could suffer from a malleability attack according to our weaker definition $\mathcal{F}_{\mathsf{DR}}$. More specifically, if the root key is leaked, and $T_4(\mathsf{DR})$ uses a weak mechanism to update the public ratchet (note: the DR public ratchet should provide PCS here), there may exist attacks which, for example, can successfully maul DR ciphertexts encrypting m into new ones that decrypt to $m+1$. This becomes evident when we prove the DR protocol secure according to $\mathcal{F}_{\mathsf{DR}}$, which is required to protect against such an attack, as we need to rely on such a non-malleability property. Indeed, the DR seems to require modelling the public ratchet KDF as a random oracle and that the Strong Diffie-Hellman assumption (StDH) is secure (i.e., given random and independent g^a, g^b, and oracle access to $\mathbf{ddh}(g^a, \cdot, \cdot)$ which on input group elements X, Y checks if $X^a = Y$, it is hard to compute g^{ab}), in order to realize $\mathcal{F}_{\mathsf{DR}}$. To provide evidence for this requirement: the ciphertexts and key material known to the adversary in the above scenario are almost identical to that of Hashed ElGamal encryption, for which all analyses of its CCA-security of which we are aware use the same assumptions [ABR01, CS03, KM04]. However, we do not rule out a security proof from weaker assumptions.

ACD's definition however does not require resistance against such an attack since it does not allow injections after corruptions. CCD$^+$'s definition also does not require resistance against such an

attack since it only explicitly provides key-indistinguishability guarantees (which are not necessarily violated in this attack; as in standard ElGamal malleability attacks), not any authenticity, nor semantic security guarantees. As a result, both ACD's and CCD$^+$'s security proofs furthermore do not require such non-malleability guarantees from the protocol.[1]

T_5: CKA Bad Randomness Plaintext Trigger: The DR is very resilient to attacks against its source of randomness. However, in $T_5(\mathsf{DR})$, if a party samples a certain string of random bits, say the all-0 string, then it (rather artificially) sends the rest of its messages in the conversation as plaintext. In our (and ACD's) model, which require security even if the adversary can supply the parties with random bits each time they attempt to sample randomness, such a protocol is clearly insecure. However, CCD$^+$'s model only allows randomness *reveals* of *uniformly* sampled random bits. Thus sampling the all-0 string occurs with negligible probability (if we assume bit strings of poly(λ) length), so security in CCD$^+$'s model is retained. Although this attack is quite artificial, [BRV20] note that attacks on randomness sources (e.g., [HDWH12]) and/or generators (e.g., [CNE+14, YRS+09]) are not captured by randomness reveals, but are captured by randomness manipulation as in our model. Furthermore, [BRV20] show that including randomness manipulations has a concrete effect on protocol construction, particularly in Secure Messaging.

T_6: Removed Immediate Decryption: Finally, $T_6(\mathsf{DR})$ changes the DR to include the public ratchet message as part of only the first ciphertext of an epoch. It is thus pretty simple to violate the immediate decryption property required by our ideal functionality: First have P_1 send two messages m_1, m_2 in a new epoch t, generating ciphertexts c_1 and c_2. Then, attempt to deliver c_2 to P_2 (before c_1). Since c_2 does not include the public ratchet message of the epoch, P_2 will be unable to decrypt it to obtain m_2. While $\mathcal{F}_{\mathsf{DR}}$ does in fact require immediate decryption, CCD$^+$'s model does not require it (nor correctness more generally), so $T_6(\mathsf{DR})$ satisfies all formal requirements of their model. ACD's model does in fact require immediate decryption.

Although this too may be an artificial transformation, immediate decryption is an important practical property of the DR, and one of the DR's main novelties is obtaining immediate decryption at the same time as FS and PCS. Furthermore, properly modelling immediate decryption allows subsequent work to understand it, and further improve upon the DR with the requirement in mind. Indeed, many of the works which we are aware of [BSJ+17, DV17, JS18, JMM19a, PR18], besides [ACD19], which try to improve the DR do not consider immediate decryption in their security models or constructions, arguably thrusting these works outside of the practical realm.

1.4 High-Level Summary of the DR's Minor Weakness

Here we show a scenario that introduces a new non-trivial (albeit minor) weakness of the DR which demonstrates a gap between the security guarantees that the DR should achieve according to our

[1] CCD$^+$'s proof does use the gap-DH assumption (similar to the StDH assumption) and random oracles (or alternatively, the PRF-ODH assumption [BFGJ17] that is only known to reduce to StDH and random oracles) but seems to only require their full power for the initial key exchange stage, and not the DR itself. The part of their proof analyzing the DR only uses the **ddh** oracle of the StDH assumption to find g^{ab} within the adversary's random oracle queries (i.e., it does not use it for providing proper simulation, as in the initial key exchange stage), and thus we believe this part of their proof only requires the DDH assumption. This is because the adversary should not be able to distinguish g^{ab} from random g^r, and thus only queries the random oracle on g^{ab} with negligible probability, resulting in security.

understanding of its whitepaper, and those which it actually does achieve. The attack utilizes two compromises of a party in a short time interval, and stems from the fact that a party needs to hold on to the secret exponent x_t for the public ratchet that it generates in a sending epoch t until it receives a message from its counterpart's next sending epoch $t+1$. Indeed secret exponent x_t is needed until this point because the other party uses its public component to encrypt messages in epoch $t+1$. For example, consider a setting in that party P_1 is about to start a sending epoch t. At this point P_1's state has $g^{x_{t-1}}$ and P_2 has x_{t-1}. Now when the sending epoch commences, P_1 samples fresh secret (random) exponent x_t and combines that with $g^{x_{t-1}}$ to derive the CKA key $I_t = g^{x_{t-1}x_t}$, which she then combines with the root key σ_t to derive first the symmetric chain key $w_{t,1}$, followed by message key $K_{t,1}$. In this epoch P_1 sends g^{x_t} to P_2, who then derives the same key I_t by computing $(g^{x_t})^{x_{t-1}}$, and subsequently $K_{t,1}$. In the next epoch, P_2 becomes a sender. Then P_2 samples a fresh x_{t+1} to derive a new CKA key $I_{t+1} = (g^{x_t})^{x_{t+1}}$ and sends $g^{x_{t+1}}$ to P_1. Now, P_1 needs to compute I_{t+1} as $(g^{x_{t+1}})^{x_t}$. To execute this step P_1 must have stored x_t throughout its sending epoch. The attack exploits this by compromising P_1 twice in a short interval:

- first compromise P_1 before starting the sending epoch t to obtain the root key σ_t;
- then compromise P_1 at any time after she sends a few messages (at least one), but before she receives any epoch $t+1$ messages, to obtain x_t, and thus I_t;

and then combine σ_t and I_t to derive $K_{t,1}$, given which *all* messages within P_1's sending epoch t are vulnerable, including the ones that were sent between the two corruptions. Intuitively, this breaks PCS with respect to the first corruption, as well as FS with respect to the second corruption. For more details we refer to Section 5.2.

In Section 3, we provide a new ideal functionality, $\mathcal{F}_{\mathsf{TR}}$, that strengthens $\mathcal{F}_{\mathsf{DR}}$ in order to capture security against the above compromise scenario. We note that both the definitions of ACD and CCD$^+$ also did not capture this scenario.

1.5 High-Level Summary of the Triple Ratchet

Finally, we provide a minimalistic modification of the DR, which we call the Triple Ratchet protocol, or simply TR, with virtually no overhead over the DR. This protocol is secure against the compromise scenario provided in the previous section and thus realizes our stronger ideal functionality, $\mathcal{F}_{\mathsf{TR}}$. The TR protocol modifies the underlying public ratchet in a way that the sampled secret exponent is deterministically updated after starting a sending epoch; thus, adding a "mini ratchet" on top of Signal's public ratchet. In particular, using the notation from above, in the modified public ratchet, a party (say P_1) after sampling secret exponent x_t, and deriving $I_t = (g^{x_{t-1}})^{x_t}$, sends g^{x_t} as the public ratchet message, but stores $x'_t = x_t \cdot \mathsf{H}(I_1)$ instead of x_t. Once P_2 receives g^{x_t}, she also derives I_t and computes $g^{x'_t} = g^{x_t \cdot \mathsf{H}(I_t)}$ that she uses for the next public ratchet. In particular, in the next epoch when P_2 becomes the sender, she samples a fresh secret exponent x_{t+1}, and uses the key $I_{t+1} = g^{x'_t x_{t+1}}$. P_2 sends $g^{x_{t+1}}$, upon receiving which P_1 can compute I_{t+1} as $(g^{x_{t+1}})^{x'_t}$, but P_2 only stores $x'_{t+1} = x_{t+1} \cdot \mathsf{H}(I_{t+1})$, and so on. Assuming H to be a random oracle, or instead, circular-security of ElGamal encryption, we can show that given x'_t, I_t is completely hidden, rendering the attack of the previous section useless. Note that the communication cost remains the same for the modified protocol, that is one group element. The computation cost increases only slightly, specifically exactly once per epoch. We also note that, the alternate CKA

scheme based on generic KEMs proposed by [ACD19] seems to achieve this security too, albeit with doubling the communication cost.

Furthermore, as we show in Section 7, our efficient modification can also be applied to practical UPKE schemes, reducing their communication by an additive factor of $|G|$, where $|G|$ is the number of bits needed to represent the size of the (CDH-hard) group used in the schemes. Using the modified UPKE scheme, we can reduce the communication of, e.g., the rTreeKEM scheme [ACDT20] used for Secure Group Messaging by an additive factor of $|G| \cdot n$, where n is the number of users in the group.

1.6 Other Related Work

Following the first formal analysis of Signal by CCD$^+$, researchers proposed a number of protocols that provided stronger security than the DR [BSJ$^+$17, PR18, BRV20, JS18, DV19, JMM19a]. ACD however observed that in the process of strengthening security, all such protocols suffer from steep efficiency costs and loss of *immediate decryption*, rendering these protocols impractical for real-world use.

Jost, Maurer and Mularczyk [JMM19b] analyzed ratcheting with the Constructive Cryptography framework [Mau11]. They aimed to capture the security and composability of various sub-protocols, such as FS-AEAD, used in the construction of larger ratcheting protocols.

More recently, there has also been work on the X3DH key exchange protocol used in Signal, providing generalized frameworks that allow for post-quantum secure versions [BFG$^+$20, HKKP21], and analyzing its offline deniability guarantees [VGIK20, UG15, UG18, HKKP21].

1.7 Summary of the Rest of the Paper

In Section 2 we provide the preliminaries containing mostly definitions borrowed from literature. In Section 3 we provide our UC-based ideal functionalities in Figure 3. We put a lot of discussions around it for reader's convenience, and along the way explain why the transformations of Section 1.3 are insecure according to our definitions. In Section 4 we provide the building blocks required for the DR (and TR), i.e., (i) we explain the (informal) properties required from the KDF used for the public ratchet (which we model as a random oracle to handle corruptions with messages in-transit; see Section 4.1.1 for more discussion on this), (ii) we introduce FS-AEAD (formalizing the symmetric ratchet part), using essentially the same formalization as ACD (except that we require *explaining* ciphertexts to handle corruptions with messages in-transit; see Section 4.2.2 for more discussion on this), and (iii) we define the CKA primitive (capturing the public ratchet) and formally provide the details on the weaker public ratchet used in the DR, as well as the stronger (virtually as efficient) public ratchet used in the TR, along with their security. In Section 5 we detail the constructions, from the proper CKA and FS-AEAD notions, of protocols DR (Double Ratchet) and TR (Triple Ratchet), which use essentially the same presentation as ACD (fixing their error as described in T_1(DR) and modelling the KDF as a random oracle). We also formally demonstrate the weakness of the DR with respect to our stronger functionality $\mathcal{F}_{\mathsf{TR}}$. In Section 6 we provide the security analyses of the Double Ratchet DR and Triple Ratchet TR, formalized in Theorem 4, which are essentially the same as that presented by ACD (but also using standard non-malleability arguments and programming the random oracle to handle corruptions with messages in-transit; again, see Section 4.1.1). In Section 7, we show how the techniques used in the TR can also be used to reduce the communication costs of practical UPKE schemes. In Appendix A we provide the full

technical details of our transformations to the DR, their insecurity with respect to our functionality $\mathcal{F}_{\mathsf{DR}}$, and their security with respect to ACD's and/or CCD$^+$'s notion. Finally, Appendix B contains technical descriptions of the UC framework, mostly borrowed from the literature, but adapted to our setting.

2 Preliminaries

2.1 Game-Based Security and Notation

In addition to our use of the UC framework to capture the security and functionality of the DR, we also consider some game-based security definitions for the primitives that are used within the DR, i.e., games executed between a challenger and an adversary. The games have one of the following formats:

- **Unpredictability games:** First, the challenger executes the special init procedure, which sets up the game. Subsequently, the attacker is given access to a set of oracles that allow it to interact with the scheme in question. The goal of the adversary is to provoke a particular, game-specific *winning* condition. The *advantage* of an adversary $\mathcal{A}$ against construction C in an unpredictability game Γ^C is

$$\mathsf{Adv}^C_\Gamma(\mathcal{A}) \;:=\; \Pr[\mathcal{A} \text{ wins } \Gamma^C] \;.$$

- **Indistinguishability games:** In addition to setting up the game, the init procedure samples a secret bit $b \in \{0,1\}$. The goal of the adversary is to determine the value of b. Once more, upon completion of init, the attacker interacts arbitrarily with all available oracles up to the point where it outputs a guess bit b'. The adversary *wins* the game if $b = b'$. The *advantage* of an adversary $\mathcal{A}$ against construction C in an indistinguishability game Γ is

$$\mathsf{Adv}^C_\Gamma(\mathcal{A}) \;:=\; 2 \cdot |\Pr[\mathcal{A} \text{ wins } \Gamma^C] - 1/2| \;.$$

- **Recoverability games:** In such games, the attacker is once more given access to a set of oracles that allow it to interact with the scheme in question after the initial init procedure. In this case, the goal of the adversary is to recover some secret value S (usually security parameter-many bits long) that is used within the scheme. The adversary *wins* the game if they guess that the secret value is $S' = S$. The *advantage* of an adversary $\mathcal{A}$ against construction C in a recoverability game Γ is

$$\mathsf{Adv}^C_\Gamma(\mathcal{A}) \;:=\; \Pr[\mathcal{A} \text{ wins } \Gamma^C] \;.$$

With the above in mind, to describe any security (or correctness) notion, one need only specify the init oracle and the oracles available to $\mathcal{A}$. The following special keywords are used to simplify the exposition of the security games:

- **req** is followed by a condition; if the condition is not satisfied, the oracle/procedure containing the keyword is exited and all actions by it are undone.

- **win** is used to declare that the attacker has won the game; it can be used for all types of games above.

- **end** disables all oracles and returns all values following it to the attacker.

Moreover, the descriptions of some games/schemes involve *dictionaries*. For ease of notation, these dictionaries are described with the *array-notation* described next, but it is important to note that they are to be implemented by a data structure whose size grows (linearly) with the number of elements *in* the dictionary (unlike arrays):

- **Initialization:** The statement $D[\cdot] \leftarrow \perp$ initializes an *empty* dictionary D.
- **Adding elements:** The statement $D[i] \leftarrow v$ adds a value v to dictionary D with key i, overriding the value previously stored with key i if necessary.
- **Retrieval:** The expression $D[i]$ returns the value v with key i in the dictionary; if there are no values with key i, the value $\perp$ is returned.
- **Deletion:** The statement $D[i] \leftarrow \perp$ *deletes* the value v corresponding to key i.

Additionally, sometimes the random coins of certain probabilistic algorithms are made explicit. For example, $y \leftarrow A(x; r)$ means that A, on input x and with random tape r, produces output y. If r is not explicitly stated, it is assumed to be chosen uniformly at random; in this case, the notation $y \xleftarrow{\$} A(x)$ is used.

Finally, many of the protocols in this work, including the DR itself, may consist of algorithms which take in some party's state. In this case, some such algorithms, upon failing, may throw an exception (**error**), which causes the calling party's state to be rolled back to where it was before the algorithm was invoked.

2.2 Authenticated Encryption

Definition 1. *An* authenticated encryption with associated data (AEAD) scheme *is a pair of algorithms* $\mathsf{AE} = (\mathsf{Enc}, \mathsf{Dec})$ *with the following syntax:*

- ***Encryption:*** Enc *takes a key* K*, associated data* a*, and a message* m *and produces a ciphertext* $e \leftarrow \mathsf{Enc}(K, a, m)$.
- ***Decryption:*** Dec *takes a key* K*, associated data* a*, and a ciphertext* e *and produces a message* $m \leftarrow \mathsf{Dec}(K, a, e)$.

All AEAD schemes in this paper are assumed to be deterministic, i.e., all randomness stems from the key K.

Correctness. An AEAD scheme is *correct* if for all keys K and all pairs (K, a),

$$\mathsf{Dec}(K, a, \mathsf{Enc}(K, a, m)) \;=\; m\,.$$

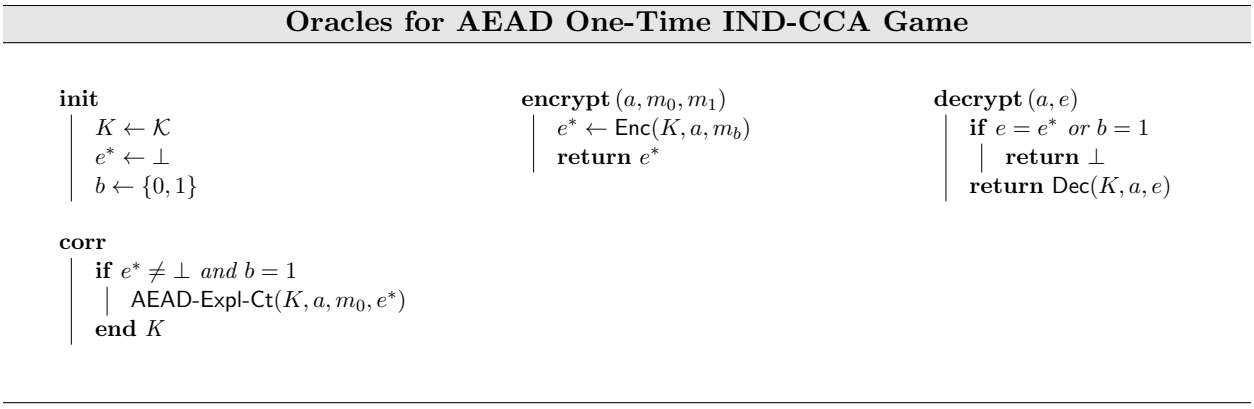

Figure 2: Oracles of the *one-time* IND-CCA security game for an AEAD scheme $(\mathsf{Enc}, \mathsf{Dec})$, where **encrypt** is a one-time oracle.

Security. In order to be used in the constructions in this paper, AEAD schemes need to satisfy one-time IND-CCA security. This is captured by the game depicted in Figure 2. It provides access to a one-time encryption oracle that on input associated data a and messages m_0, m_1, returns an encryption of message m_b, depending on a randomly chosen bit b. Moreover, the attacker may query a decryption oracle arbitrarily many times (except on the challenge ciphertext), which, however, always returns $\perp$ if $b = 1$.

An additional property that hash-based AEAD schemes satisfy and which we require (which ACD do not require since our AEAD notion is stronger than theirs) is the ability to "explain" ciphertexts. That is, the ability to program any ciphertext so that given a chosen key, message and associated data, the ciphertext decrypts with the key consistently to the message after the fact, and furthermore so that encrypting the message under the key indeed results in the same ciphertext (similar to non-committing encryption).[2] It will later be seen that such explainability is required by our ideal functionality for the DR (see Section 4.2.2).

For instance, by appropriately programming a random oracle after the fact, we can ensure that a ciphertext generated ahead of time correctly decrypts to the chosen message with the chosen key, and that the message encrypts to that ciphertext with the key. We model this "explainability" aspect via an additional algorithm called AEAD-Expl-Ct that takes an AEAD key, associated data, message, and ciphertext, and programs them to be consistent. In the security game, the adversary can query oracle **corr** at which point if $b = 1$ the challenger explains the ciphertext as an encryption of m_0 under K and returns K, or if $b = 0$, simply returns K. No matter the setting of bit b, after corruption the game ends without loss of generality as the adversary can encrypt and decrypt on its own. All AEAD schemes considered in this work are required to provide a AEAD-Expl-Ct algorithm.

The advantage of an adversary $\mathcal{A}$ attacking an AEAD scheme AE is denoted by $\mathsf{Adv}^{\mathsf{AE}}_{\text{ot-cca}}(\mathcal{A})$; the attacker is parametrized by its running time t.

Definition 2. *An AEAD scheme* AE *is* (t, ε)-one-time-CCA-secure *if for all* t*-attackers* $\mathcal{A}$*,*

$$\mathsf{Adv}^{\mathsf{AE}}_{\text{ot-cca}}(\mathcal{A}) \;\leq\; \varepsilon \;.$$

[2] Constructions used by the DR, such as AEAD based on CBC+HMAC and SIV, will satisfy this property.

3 Defining Security of the Double Ratchet

In this section, we focus on obtaining an ideal functionality $\mathcal{F}_{\mathsf{DR}}$ that captures, as completely as possible, the security provided by the Double Ratchet algorithm. We emphasize that we study the security provided by the *strongest* implementation of the DR of which we are aware. For more on this, see Appendix A. We also provide an ideal functionality $\mathcal{F}_{\mathsf{TR}}$ that captures the security of our stronger TR protocol. Both functionalities are provided in Figure 3.

$\mathcal{F}_{\mathsf{DR}}$ and $\mathcal{F}_{\mathsf{TR}}$

Notation: The ideal functionality interacts with two parties $\mathsf{P}_1, \mathsf{P}_2$, and an ideal adversary $\mathcal{S}$. The ideal functionality initializes lists of *used message-ids* $\mathsf{P}_1.\mathtt{M}$, *in-transit* messages $\mathsf{P}_1.\mathtt{T}$, *adversarially injected* message-ids $\mathsf{P}_1.\mathtt{I}$, and *vulnerable messages* $\mathsf{P}_1.\mathtt{V}$ sent by P_1 to P_2 to ϕ. Analogously, lists $\mathsf{P}_2.\mathtt{M}, \mathsf{P}_2.\mathtt{T}, \mathsf{P}_2.\mathtt{I}$ and $\mathsf{P}_2.\mathtt{V}$ are also initialized to ϕ. The ideal functionality also initializes leakage flags of both P_1 and P_2 for their corresponding (i) public ratchet secrets: $\mathsf{P}_1.\mathtt{PLEK}, \mathsf{P}_2.\mathtt{PLEK}$, (ii) current sending epoch symmetric secrets: $\mathsf{P}_1.\mathtt{CUR_SLEK}, \mathsf{P}_2.\mathtt{CUR_SLEK}$, and (iii) previous sending epoch symmetric secrets: $\mathsf{P}_1.\mathtt{PREV_SLEK}, \mathsf{P}_2.\mathtt{PREV_SLEK}$, all to 0. Further, it initializes bad-randomness flags $\mathsf{P}_1.\mathtt{BAD}, \mathsf{P}_2.\mathtt{BAD}$ and takeover possible flags $\mathsf{P}_1.\mathtt{TAKEOVER_POSS}, \mathsf{P}_2.\mathtt{TAKEOVER_POSS}$ to 0. Finally, it initializes the turn flag $\mathtt{TURN}$ as $\perp$.

- **On input** $(\mathsf{sid}, \mathtt{SETUP})$ **from** P where $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$: Send $(\mathsf{sid}, \mathtt{SETUP}, \mathsf{P})$ to $\mathcal{S}$. When $\mathcal{S}$ returns $(\mathsf{sid}, \mathtt{SETUP})$ then set $\mathtt{TURN} \leftarrow \mathsf{P}$, and send $(\mathsf{sid}, \mathtt{INITIATED})$ to both P_1 and P_2. Ignore all future messages until this step is completed for sid. Once this happens P can send the first message.
- **On input** $(\mathsf{sid}, \mathsf{mid}, \mathtt{SEND}, m)$ **from** $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$:
  1. Ignore if $\mathsf{mid} \in \mathsf{P}.\mathtt{M}$.
  2. If $\bar{\mathsf{P}}.\mathtt{CUR_SLEK} \vee (\mathsf{P}.\mathtt{V} \neq \emptyset)$ then $\mathsf{P}.\mathtt{V} \cup \{(\mathsf{sid}, \mathsf{mid}, m)\}$
  3. If $\mathtt{New}(\mathsf{P}, \mathtt{TURN}, \mathsf{P}.\mathtt{T})$[a] then set (i) $\mathsf{P}.\mathtt{PLEK} \leftarrow \mathsf{P}.\mathtt{BAD}$, (ii) $\mathsf{P}.\mathtt{CUR_SLEK} \leftarrow \bar{\mathsf{P}}.\mathtt{CUR_SLEK} \wedge (\mathsf{P}.\mathtt{PLEK} \vee \bar{\mathsf{P}}.\mathtt{PLEK})$, and (iii) $\bar{\mathsf{P}}.\mathtt{TAKEOVER_POSS} \leftarrow \mathsf{P}.\mathtt{CUR_SLEK}$.
  4. Add mid to $\mathsf{P}.\mathtt{M}$; if $\mathsf{mid} \notin \mathsf{P}.\mathtt{I}$ then add $(\mathsf{sid}, \mathsf{mid}, \mathtt{IN_TRANSIT}, m, \mathsf{P}.\mathtt{CUR_SLEK}, \mathtt{TURN})$ to $\mathsf{P}.\mathtt{T}$; and pass $(\mathsf{sid}, \mathsf{mid}, \mathtt{IN_TRANSIT}, \mathsf{P}, |m|, m')$ to $\mathcal{S}$ where $m' \leftarrow m$ if $\mathsf{P}.\mathtt{CUR_SLEK}$ and $\perp$ otherwise.
- **On input** $(\mathsf{sid}, \mathsf{mid}, \mathtt{DELIVER}, \mathsf{P}, m')$ **from** $\mathcal{S}$ where $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$:
  1. Find $(\mathsf{sid}, \mathsf{mid}, \mathtt{IN_TRANSIT}, m, \beta, \gamma) \in \mathsf{P}.\mathtt{T}$ and remove it from $\mathsf{P}.\mathtt{T}$. Skip rest of the steps if no such entry is found.
  2. If $\gamma = \mathsf{P}$ then set (i) $\mathtt{TURN} \leftarrow \bar{\mathsf{P}}$, (ii) $\mathsf{P}.\mathtt{T} \leftarrow \mathtt{Flip}(\mathsf{P}, \mathsf{P}.\mathtt{T})$,[b] (iii) $\mathsf{P}.\mathtt{PREV_SLEK} \leftarrow 0$, (iv) $\bar{\mathsf{P}}.\mathtt{PREV_SLEK} \leftarrow \bar{\mathsf{P}}.\mathtt{CUR_SLEK}$, (v) $\bar{\mathsf{P}}.\mathtt{CUR_SLEK} \leftarrow 0$, (vi) $\bar{\mathsf{P}}.\mathtt{PLEK} \leftarrow 0$, (vii) $\mathsf{P}.\mathtt{TAKEOVER_POSS} \leftarrow 0$, and (viii) $\bar{\mathsf{P}}.\mathtt{V} \leftarrow \emptyset$.
  3. If $\beta = 1$ then set $m \leftarrow m'$. Send $(\mathsf{sid}, \mathsf{mid}, \mathtt{DELIVER}, m)$ to $\bar{\mathsf{P}}$.

- **On input** $(\mathsf{sid}, \mathtt{LEAK}, \mathsf{P})$ **from** $\mathcal{S}$ where $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$:
  1. If $\neg\mathtt{New}(\mathsf{P}, \mathtt{TURN}, \mathsf{P}.\mathtt{T})$ then set $\mathsf{P}.\mathtt{CUR_SLEK} \leftarrow 1$, $\mathsf{P}.\mathtt{PLEK} \leftarrow 1$, and $\bar{\mathsf{P}}.\mathtt{TAKEOVER_POSS} \leftarrow 1$.
  2. If $\neg\mathtt{New}(\bar{\mathsf{P}}, \mathtt{TURN}, \bar{\mathsf{P}}.\mathtt{T})$ then set $\bar{\mathsf{P}}.\mathtt{CUR_SLEK} \leftarrow 1$.
  3. If $\mathtt{TURN} = \bar{\mathsf{P}}$ then set $\bar{\mathsf{P}}.\mathtt{PREV_SLEK} \leftarrow 1$.
  4. If $\mathtt{New}(\mathsf{P}, \mathtt{TURN}, \mathsf{P}.\mathtt{T}) \vee (\neg\mathtt{New}(\bar{\mathsf{P}}, \mathtt{TURN}, \bar{\mathsf{P}}.\mathtt{T}) \wedge \mathtt{TURN} = \bar{\mathsf{P}})$ then set $\mathsf{P}.\mathtt{TAKEOVER_POSS} \leftarrow 1$.
  5. Execute $\bar{\mathsf{P}}.\mathtt{T} \leftarrow \mathtt{Unsafe}(\bar{\mathsf{P}}.\mathtt{T})$[c] and $\mathsf{P}.\mathtt{T} \leftarrow \mathtt{Unsafe}'(\mathsf{P}.\mathtt{T}, \mathsf{P}.\mathtt{V})$.[d] then send $\bar{\mathsf{P}}.\mathtt{T}$ and $\mathsf{P}.\mathtt{V}$ to $\mathcal{S}$.
- **On input** $(\mathsf{sid}, \mathtt{BAD_RANDOMNESS}, \mathsf{P}, \rho)$ **from** $\mathcal{S}$ where $\rho \in \{0, 1\}$ and $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$: Set $\mathsf{P}.\mathtt{BAD} \leftarrow \rho$.
- **On input** $(\mathsf{sid}, \mathsf{mid}, \mathtt{INJECT}, \mathsf{P}, m, \delta, \gamma)$ **from** $\mathcal{S}$:

1. Skip if ($\mathsf{mid} \in \mathsf{P.I} \cup \mathsf{P.M}$) $\vee \neg(\mathsf{P.TAKEOVER_POSS} \vee \mathsf{P.PREV_SLEK} \vee \mathsf{P.CUR_SLEK})$.
2. If $\mathsf{P.TAKEOVER_POSS} \wedge \delta$ then forward all subsequent incoming messages from $\bar{\mathsf{P}}$ to $\mathcal{S}$ and from $\mathcal{S}$ for $\bar{\mathsf{P}}$ directly to $\bar{\mathsf{P}}$. Also, remove from P.T all elements of the form $(\cdot, \cdot, \mathtt{IN_TRANSIT}, \cdot, \cdot, \mathsf{P})$ and drop all subsequent incoming messages for $\bar{\mathsf{P}}$ generated by the ideal functionality (i.e., do not send them to $\bar{\mathsf{P}}$), except the ones generated according to the DELIVER command.
3. Otherwise, add mid to P.I and add to P.T (i) if $\mathtt{TURN} = \bar{\mathsf{P}} \vee \neg \mathsf{P.CUR_SLEK}$ then $(\mathsf{sid}, \mathsf{mid}, \mathtt{IN_TRANSIT}, m, 1, \perp)$, (ii) if $\mathtt{TURN} = \mathsf{P} \wedge \neg \mathsf{P.PREV_SLEK}$ then $(\mathsf{sid}, \mathsf{mid}, \mathtt{IN_TRANSIT}, m, 1, \mathsf{P})$, and (iii) else $(\mathsf{sid}, \mathsf{mid}, \mathtt{IN_TRANSIT}, m, 1, \gamma)$.

[a] $\mathtt{New}(\mathsf{P}, \mathtt{TURN}, \mathsf{P.T})$ outputs 1 if we have $\mathtt{TURN} = \mathsf{P}$ and for all $(\mathsf{sid}, \mathsf{mid}, \mathtt{IN_TRANSIT}, m, \beta, \gamma) \in \mathsf{P.T}$ we have $\gamma \neq \mathsf{P}$; otherwise output 0.
[b] $\mathtt{Flip}(\mathsf{P}, \mathsf{P.T})$ for each $(\mathsf{sid}, \mathsf{mid}, \mathtt{IN_TRANSIT}, m, \beta, \mathsf{P}) \in \mathsf{P.T}$ replaces it with $(\mathsf{sid}, \mathsf{mid}, \mathtt{IN_TRANSIT}, m, \beta, \perp)$.
[c] $\mathtt{Unsafe}(\mathsf{P.T})$ for each $(\mathsf{sid}, \mathsf{mid}, \mathtt{IN_TRANSIT}, m, \beta, \gamma) \in \mathsf{P.T}$ replaces it with $(\mathsf{sid}, \mathsf{mid}, \mathtt{IN_TRANSIT}, m, 1, \gamma)$.
[d] $\mathtt{Unsafe}'(\mathsf{P.T}, \mathsf{P.V})$ for each $(\mathsf{sid}, \mathsf{mid}, m) \in \mathsf{P.V}$, if there is a corresponding $(\mathsf{sid}, \mathsf{mid}, \mathtt{IN_TRANSIT}, m, \beta, \gamma) \in \mathsf{P.T}$, replaces it with $(\mathsf{sid}, \mathsf{mid}, \mathtt{IN_TRANSIT}, m, 1, \gamma)$.

Figure 3: The ideal functionalities $\mathcal{F}_{\mathsf{DR}}$ and $\mathcal{F}_{\mathsf{TR}}$, respectively.

3.1 Honest Execution

We start with a simplified view of the functionality where only the first three commands, namely SETUP, SEND, and DELIVER are executed. In other words, we consider a restricted view of the ideal functionality where leakage, bad randomness and injection attacks are not allowed. The adversary is still allowed to delay, reorder, and drop messages at will.

SETUP Command. This command can be initiated by either $\mathsf{P} = \mathsf{P}_1$ or $\mathsf{P} = \mathsf{P}_2$, and allows for initializing the communication channel between P and $\bar{\mathsf{P}}$. Looking ahead, in the real-world protocol, this initialization will involve sharing cryptographic secrets between the real-world P and real-world $\bar{\mathsf{P}}$, then properly initializing their states using these secrets. While the actual Signal protocol uses the X3DH key exchange [MP16b] for this, the focus of our work is to analyze the security and functionality of the double ratchet algorithm, and not X3DH. Therefore, we present a simple description for the SETUP command, that may be stronger than what X3DH achieves, but nonetheless suffices for our purposes.

We note that both P_1 and P_2 must receive $(\mathsf{sid}, \mathtt{INITIATED})$ before the communication between them can proceed. Turn status flag TURN is set to the initiator P to denote that P will be the first party to send a message.

SEND Command. This command allows $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$ to send a message m, under a unique assigned message id mid, to $\bar{\mathsf{P}}$. Naturally, the ideal functionality only allows P to send one message under each such mid, which it ensures by aborting in Step 1 if mid is already in list P.M, and subsequently adding mid to P.M in Step 4 otherwise. Now, this message might be dropped or delayed while in transit. Thus, at this point, the message is only added to the in-transit list P.T (Step 4) and the ideal-functionality waits for the instruction from the ideal-world adversary on when this message is to be delivered (if at all).

Observe that the last element of each tuple in P.T is TURN: the turn status when P attempted to send this message (i.e., when it was added to P.T). Looking ahead, this element is used in helper function $\mathtt{New}(\mathsf{P}, \mathtt{TURN}, \mathsf{P.T})$ within SEND (Step 3) to determine whether P is initiating a new

epoch when sending a message and, if so, the (in)security of the new epoch. When discussing the DELIVER command below, we will explain the role the last element of P.T plays in the logic of New(P, TURN, P.T) and further understand its role elsewhere in the functionality.

Finally, as typical with encryption, in the real-world the length of the encrypted message is often leaked by the ciphertext. Thus, the ideal functionality leaks the length of sent messages to the ideal adversary.

DELIVER Command. This command allows the ideal adversary to instruct the ideal functionality that a certain message, with unique message id mid, is no longer in-transit, and must be delivered to the recipient *immediately*, whether or not previously sent messages have already been delivered (thus transformation $T_6(\mathsf{DR})$ cannot realize the ideal functionality). The ideal functionality restricts the ideal adversary to delivering the message associated with this mid only once, which reflects the forward security of the DR – once a message is delivered, the recipient should no longer be able to decrypt it (in case she is leaked on afterwards). This is done by removing the entry for mid from P.T when it is delivered, so that subsequent deliveries cannot occur (Step 1).

As part of the delivery process (Step 2), the ideal functionality also checks if TURN was set to P when this message was sent. If so, the message was indeed the first of P's newest epoch that is delivered to $\bar{\mathsf{P}}$ (out of possibly many messages that can be the first delivered in the epoch). Thus, subsequently, it will next be $\bar{\mathsf{P}}$'s turn to start a new epoch. So, if this is the case, then TURN is flipped to $\bar{\mathsf{P}}$. Additionally, helper function Flip(P, P.T) flips the last entry of each message from P to $\bar{\mathsf{P}}$ in P.T to $\perp$. This is done so that subsequently, when P starts its next sending epoch, New(P, TURN, P.T) will return 1: TURN will flip back to P once a message of $\bar{\mathsf{P}}$'s next sending epoch is delivered to P for the first time, and there will be no element in P.T whose last entry is P. (Note: before P receives a message for $\bar{\mathsf{P}}$'s next sending epoch, P's sent messages will not start a new epoch, as captured by New(P, TURN, P.T), since TURN will be set to $\bar{\mathsf{P}}$.)

3.2 Execution with an Unrestricted Adversary

In addition to delaying, reordering, and dropping messages, we assume that the real-world adversary can: (i) provide bad randomness for both parties, (ii) leak their secret states; possibly multiple times at various points in the execution, (iii) tamper with in-transit messages between the parties, and (iv) attempt to inject messages on behalf of both parties. Here, we explain how the ideal functionality captures this behavior.

The Ideal Functionality's Flags. The ideal functionality uses several binary flags to properly capture adversarial behavior. The functionality initializes all of them to 0. Binary flag P.BAD captures bad randomness for party $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$. Naturally, P.BAD is set to 0 or 1 when the ideal-world adversary issues a $(\mathsf{sid}, \texttt{BAD_RANDOMNESS}, \mathsf{P}, \rho)$ command to the ideal functionality, depending on the the value of ρ. If P.BAD is set to 1 then P is provided with bad randomness (by the adversary, c.f. Appendix B) when she tries to sample some (thus rendering transformation $T_5(\mathsf{DR})$ insecure). Otherwise, P samples fresh randomness.

The ideal functionality further utilizes the following binary flags for each party $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$ to capture the rest of the possible adversarial behavior. We first introduce their real-world semantic meaning here before explaining (i) their evolution within the ideal functionality as a result of the ideal world adversary's behavior, then (ii) how they thus allow the ideal functionality to determine security for the session.

- $\mathsf{P}.\mathtt{PLEK}$ (Public Ratchet Secrets Leakage): If $\mathsf{P}.\mathtt{PLEK}$ is set to 1 then P's public ratchet secrets are leaked to the real-world adversary. Otherwise, they should be hidden from the real-world adversary.
- $\mathsf{P}.\mathtt{CUR_SLEK}$ (Current Sending Symmetric Ratchet Secrets Leakage): If $\mathsf{P}.\mathtt{CUR_SLEK}$ is set to 1 then the symmetric ratchet secrets of P's *current* sending epoch are leaked to the real-world adversary. Otherwise, they should be hidden from the real-world adversary
- $\mathsf{P}.\mathtt{PREV_SLEK}$ (Previous Sending Symmetric Ratchet Secrets Leakage): If $\mathsf{P}.\mathtt{PREV_SLEK}$ is set to 1 then the symmetric ratchet secrets of the *previous* sending epoch of P are leaked to the real-world adversary. Otherwise, they should be hidden from the real-world adversary.
- $\mathsf{P}.\mathtt{TAKEOVER_POSS}$ (Takeover Possible): If $\mathsf{P}.\mathtt{TAKEOVER_POSS}$ is set to 1 then the real-world adversary has the option to take over the role of P in the conversation with $\bar{\mathsf{P}}$. Otherwise, the real-world adversary should not have this option.

How the Flags are Affected by Leakages. We first describe how a leakage on one of the parties $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$ affects the above flags. For $\mathsf{P}.\mathtt{PLEK}$, when $\mathtt{New}(\mathsf{P}, \mathtt{TURN}, \mathsf{P}.\mathtt{T}) = 1$, it is P's turn to start her next sending epoch, but she has not yet started it. Thus she does not currently have any public ratchet secret state (just $\bar{\mathsf{P}}$'s public value), so there is no effect on $\mathsf{P}.\mathtt{PLEK}$ if leakage on P occurs in this case. If $\mathtt{New}(\mathsf{P}, \mathtt{TURN}, \mathsf{P}.\mathtt{T}) = 0$ when leakage on P occurs, P of course does have a public ratchet secret state, as she needs to be able to receive a message for $\bar{\mathsf{P}}$'s next sending turn; thus in command $\mathtt{LEAK}$, the ideal functionality sets $\mathsf{P}.\mathtt{PLEK}$ to 1 (Step 1). Since P never stores $\bar{\mathsf{P}}$'s public ratchet secrets, there is never any effect on $\bar{\mathsf{P}}.\mathtt{PLEK}$ when P's state is leaked.

For $\mathsf{P}.\mathtt{CUR_SLEK}$, the functionality has similar behavior. If $\mathtt{New}(\mathsf{P}, \mathtt{TURN}, \mathsf{P}.\mathtt{T}) = 1$ when leakage on P occurs, P has not yet generated the secrets for her next sending epoch, so $\mathsf{P}.\mathtt{CUR_SLEK}$ is not modified. Otherwise, P has started the epoch, and so she stores the corresponding secrets in order to send new messages for the epoch; thus in command $\mathtt{LEAK}$, we set $\mathsf{P}.\mathtt{CUR_SLEK}$ to 1 (Step 1). Additionally, if $\mathtt{New}(\bar{\mathsf{P}}, \mathtt{TURN}, \bar{\mathsf{P}}.\mathtt{T}) = 1$ then $\bar{\mathsf{P}}$ has not yet generated the secrets for her next sending epoch, so $\bar{\mathsf{P}}.\mathtt{CUR_SLEK}$ is not modified. Otherwise, $\bar{\mathsf{P}}$ has indeed started the epoch, in which case P must be able to derive the epoch's symmetric secrets (possibly using in-transit messages, which the adversary has), and thus in command $\mathtt{LEAK}$ we set $\bar{\mathsf{P}}.\mathtt{CUR_SLEK}$ to 1 (Step 2).

For $\mathsf{P}.\mathtt{PREV_SLEK}$, since in the most secure version of the DR, P only ever stores the secrets for her current sending epoch (if she has indeed started it), leakage on P has no effect on $\mathsf{P}.\mathtt{PREV_SLEK}$. However, once it is $\bar{\mathsf{P}}$'s turn to start a new sending epoch, P still stores the secrets of $\bar{\mathsf{P}}$'s previous sending epoch (in case she needs to receive messages for it; she does not yet know $\bar{\mathsf{P}}$ will never again send a message for that epoch), until she receives a message in $\bar{\mathsf{P}}$'s new epoch for the first time. Therefore, if $\mathtt{TURN} = \bar{\mathsf{P}}$ then in command $\mathtt{LEAK}$, we set $\bar{\mathsf{P}}.\mathtt{PREV_SLEK}$ to 1 (Step 3); otherwise, if $\mathtt{TURN} = \mathsf{P}$, $\bar{\mathsf{P}}.\mathtt{PREV_SLEK}$ is not modified.

Finally, for $\mathsf{P}.\mathtt{TAKEOVER_POSS}$, if it is P's turn to start a new sending epoch, then of course a leakage on P will enable the adversary to forge the first message of this new epoch and thus influence the subsequent state of $\bar{\mathsf{P}}$ upon delivery such that the adversary can take over P's role in the conversation (if it wishes). This is because the adversary will obtain the double ratchet root key, and can thus send such a message herself. Also, note that this key is derived from (i) P's previous state before she received any message for $\bar{\mathsf{P}}$'s newest epoch and (ii) any message of $\bar{\mathsf{P}}$'s newest sending epoch. Thus, additionally, if P is leaked while any message from $\bar{\mathsf{P}}$'s newest epoch

is in-transit, but before P receives any such message, then the adversary can obtain the root key as above, and so will have the ability to forge the first message of P's next sending epoch. Therefore, in command LEAK, if $\mathtt{New}(\mathsf{P}, \mathtt{TURN}, \mathsf{P}.\mathtt{T}) = 1$, or $\mathtt{New}(\bar{\mathsf{P}}, \mathtt{TURN}, \bar{\mathsf{P}}.\mathtt{T}) = 0$ and $\mathtt{TURN} = \bar{\mathsf{P}}$, then we set $\mathsf{P}.\mathtt{TAKEOVER_POSS}$ to 1 (Step 4). Otherwise, if P has already sent the first message of the epoch, and $\bar{\mathsf{P}}$ has not yet started her next sending epoch, leakage on P does not reveal the root key, so $\mathsf{P}.\mathtt{TAKEOVER_POSS}$ is not modified. In the former case, this is because P deletes the key after sending the message, and in the latter case, this is because the key does not yet exist. Furthermore, if P has indeed sent a message for her current sending epoch, then a leakage on P will provide the adversary with the new root key. The adversary will therefore be able to forge the first message for $\bar{\mathsf{P}}$'s next sending epoch. So, if $\mathtt{New}(\mathsf{P}, \mathtt{TURN}, \mathsf{P}.\mathtt{T}) = 0$ then in command LEAK, we additionally set $\bar{\mathsf{P}}.\mathtt{TAKEOVER_POSS}$ to 1 (Step 1). Otherwise, if it is P's turn to start a new epoch, and she has not yet started it, then the new root key has not yet been generated, so $\bar{\mathsf{P}}.\mathtt{TAKEOVER_POSS}$ is not modified.

How the Flags are Affected by Epoch Initialization. The effects on the ideal functionality's flags of epoch initialization via a SEND command are determined in Step 3 of the command. First, if $\mathsf{P}.\mathtt{BAD} = 1$ when starting a new epoch (i.e. P uses bad randomness to start it), then we of course set $\mathsf{P}.\mathtt{PLEK}$ to 1 (In the TR we may still here have security of P's public ratchet secret state, but we choose to capture slightly weaker security for simplicity); otherwise we set $\mathsf{P}.\mathtt{PLEK}$ to 0. Now, consider the privacy of the root key when $\bar{\mathsf{P}}.\mathtt{CUR_SLEK}$ is 1 and P is initializing a new epoch:

- If $\bar{\mathsf{P}}.\mathtt{CUR_SLEK}$ was set to 1 when $\bar{\mathsf{P}}$ initialized her newest epoch (as we explain below), then the root key must have been leaked in addition to the corresponding symmetric ratchet secrets, since they are both part of the same KDF output.
- If $\bar{\mathsf{P}}.\mathtt{CUR_SLEK}$ was set to 1 as a result of a leakage on P, then the root key must have been also leaked, since P needs it to start her new sending epoch.
- Finally, if $\bar{\mathsf{P}}.\mathtt{CUR_SLEK}$ was set to 1 as a result of a leakage on $\bar{\mathsf{P}}$, then the root key must have been also leaked, since $\bar{\mathsf{P}}$ needs it to receive a message for P's new sending epoch.

So, if $\bar{\mathsf{P}}.\mathtt{CUR_SLEK}$ is 1 when P initializes her new sending epoch, then it must be that the root key is leaked. Thus, only if P and $\bar{\mathsf{P}}$ have a secure key exchange can security for the DR be recovered, which only happens if both $\mathsf{P}.\mathtt{PLEK}$ and $\bar{\mathsf{P}}.\mathtt{PLEK}$ are 0, i.e., their public ratchet secrets are both hidden from the adversary. In this case, we set $\mathsf{P}.\mathtt{CUR_SLEK}$ to 0; otherwise, we set it to 1. If $\bar{\mathsf{P}}.\mathtt{CUR_SLEK}$ is 0 at the time of initialization, then the root key must be hidden. This is because if not, then the current symmetric ratchet secrets of $\bar{\mathsf{P}}$ would also not be hidden, since they were part of the same KDF output when $\bar{\mathsf{P}}$ started her latest sending epoch, and there were no subsequent leakages on either party. So we set $\mathsf{P}.\mathtt{CUR_SLEK}$ to 0 upon initialization, in this case.

Finally, if we do indeed set $\mathsf{P}.\mathtt{CUR_SLEK}$ to 1 at this time, as we noted above, this means that the new root key is known by the adversary, and thus the adversary could forge the first message for $\bar{\mathsf{P}}$'s next turn; otherwise the root key is hidden, and so the adversary does not have this ability. So, we set $\bar{\mathsf{P}}.\mathtt{TAKEOVER_POSS} \leftarrow \mathsf{P}.\mathtt{CUR_SLEK}$.

How the Flags are Affected by Epoch Termination. When the ideal adversary issues a DELIVER command for the first message of P's newest sending epoch, the ideal functionality needs to properly evolve the flags it uses to capture adversarial behavior (Step 2). First, when such a

delivery occurs, $\bar{\mathsf{P}}$'s latest sending epoch terminates, as her next message will be sent in a new epoch. To reflect this, upon such a delivery, the ideal functionality sets $\bar{\mathsf{P}}$.PREV_SLEK $\leftarrow \bar{\mathsf{P}}$.CUR_SLEK. Also, since $\bar{\mathsf{P}}$ deletes her public ratchet secrets upon reception of such a message, and her newest epoch has not actually started at this point, the functionality sets $\bar{\mathsf{P}}$.CUR_SLEK $\leftarrow 0$ and $\bar{\mathsf{P}}$.PLEK $\leftarrow 0$.

Furthermore, in the DR, P includes in each message of an epoch the number of messages she sent in her previous epoch (see Section 5). Thus, once $\bar{\mathsf{P}}$ receives such a message in the DR, she knows exactly how many messages P sent in her previous epoch. So, the adversary can no longer inject messages in P's previous epoch (just modify them) and there is no more adversarial action possible for that epoch, so the functionality sets P.PREV_SLEK $\leftarrow 0$. Finally, since a message for P's newest sending epoch has indeed been delivered at this point, the secrets needed to start her next sending epoch are yet to be determined. Thus, the adversary cannot yet forge a message to start her next sending epoch, so the functionality sets P.TAKEOVER_POSS $\leftarrow 0$.

Determining New Messages' Privacy and Authenticity. We know from above that if P.CUR_SLEK $= 1$, then P's current symmetric ratchet secrets are leaked to the adversary. Thus, if P issues a SEND command for message m with id mid, and P.CUR_SLEK $= 1$, then the ideal functionality leaks the corresponding message to the ideal adversary (Step 4). Additionally, the ideal functionality sets the penultimate element of mid's entry in P.T to 1. This will allow the ideal adversary to modify the message associated with mid upon delivery: the adversary will issue a DELIVER command for mid to the functionality with input modified message m', which will then be delivered $\bar{\mathsf{P}}$, instead of m (Step 3).

Otherwise, if P.CUR_SLEK $= 0$ when P issues the SEND command, then the ideal functionality only leaks the message length to the adversary and sets the penultimate element of the corresponding entry of P.T to 0, ensuring (for now) privacy and authenticity of m.

The Consequences of Leakages. When the adversary leaks on P in the real-world, the privacy of in-transit messages from $\bar{\mathsf{P}}$ to P is no longer guaranteed, since P must preserve all keys that will be necessary for authenticating and decrypting them. Therefore, when the ideal adversary issues a LEAK command on P, the ideal functionality leaks the in-transit messages from $\bar{\mathsf{P}}$ to P, P.T, to the ideal adversary, and allows the ideal adversary to modify them in the future (Step 5). The ideal functionality accomplishes the latter using helper function Unsafe($\bar{\mathsf{P}}$.T) which sets the penultimate element of each in-transit message of $\bar{\mathsf{P}}$.T to 1. As a result, the ideal adversary can modify these in-transit messages in the DELIVER command, as described above. Note that only *in-transit* messages from $\bar{\mathsf{P}}$ to P are affected (thus rendering transformation T_2(DR) insecure).

Vulnerable Messages in the DR. As explained in Section 1.4, if in the DR, the root key is leaked when it is P's turn to start a new sending epoch, but P has not yet started it, then the messages of that epoch are *vulnerable*. This means that if P is leaked on before P receives a message of $\bar{\mathsf{P}}$'s next sending epoch for the first time, the messages that P sent in her own epoch become insecure.

To capture this, the ideal functionality in the SEND command adds messages to list P.V if they are indeed vulnerable (Step 2). At the start of the epoch, this is the case if $\bar{\mathsf{P}}$.CUR_SLEK $= 1$ (as explained above); in the middle of the epoch, this is the case if P.V is non-empty. Hence, if the adversary issues a LEAK command on P, in addition to the consequences of the above paragraph, the ideal functionality *also* leaks P.V and allows for future modification of its elements that are still in-transit (Step 5). The latter is accomplished via helper function Unsafe$'$(P.T, P.V), similarly

as in $\texttt{Unsafe}(\bar{\mathsf{P}}.\texttt{T})$. Note that this scenario, and the one above, are the only ones in which secure, in-transit messages are leaked to the adversary and/or subject to modification (thus transformation $T_4(\mathsf{DR})$ is insecure). Finally, if the adversary issues a DELIVER command for the first message of $\bar{\mathsf{P}}$'s next sending epoch, the ideal functionality sets $\mathsf{P}.\texttt{V} = \emptyset$: P properly deletes the secrets which make those messages vulnerable at this time.

Injections and Takeovers. If $\mathsf{P}.\texttt{CUR_SLEK} = 1$ or $\mathsf{P}.\texttt{PREV_SLEK} = 1$, then the adversary has the secrets required to inject its own messages into P's current or previous sending epoch, respectively. Also, if $\mathsf{P}.\texttt{TAKEOVER_POSS} = 1$, then the adversary can forge the first message to be delivered in P's next sending epoch to $\bar{\mathsf{P}}$. In either case, the ideal adversary issues the INJECT command to inject message m under unique message id mid on behalf of P. Of course, the ideal functionality only allows the adversary to inject one message under each such mid, which it ensures by aborting in Step 1 if mid is already in $\mathsf{P}.\texttt{I}$, and adding it to $\mathsf{P}.\texttt{I}$ in Step 3 if not. The ideal functionality also aborts if a message with message id mid was already sent by P, i.e., it is in $\mathsf{P}.\texttt{M}$, in which case injection of mid is not allowed, only modification. If the ideal adversary injects a message with id mid that is not a takeover forgery, then before actual delivery of the injection occurs, a corresponding entry is added to $\mathsf{P}.\texttt{T}$.

Now, if $\mathsf{P}.\texttt{TAKEOVER_POSS} = 1$, and the ideal adversary inputs $\delta = 1$ to the INJECT command, indicating that it wishes to takeover for P, then the ideal functionality thereafter directly forwards messages sent from P to the ideal adversary, and vice versa (Step 2).

If the ideal adversary injects a message with id mid that is not a takeover forgery, then before actual delivery of the injection occurs, a corresponding entry is added to $\mathsf{P}.\texttt{T}$. However, the ideal functionality has to be careful to set the last element of this entry correctly:

- If $\texttt{TURN} = \bar{\mathsf{P}}$, then the first message of P's current sending epoch has already been delivered to $\bar{\mathsf{P}}$. Thus, the last element of the entry is set to $\perp$, so that if TURN is flipped to P, the entry's subsequent delivery does not prematurely flip TURN back. Moreover, if $\mathsf{P}.\texttt{CUR_SLEK} = 0$, then the adversary must be injecting into P's previous sending epoch, so for the same reason as above, we set its last entry to $\perp$.

- If $\mathsf{P}.\texttt{PREV_SLEK} = 0$ and $\texttt{TURN} = \mathsf{P}$, then the adversary must be injecting into P's current sending epoch, and moreover, it might be that the injected message could be the first of the epoch delivered to $\bar{\mathsf{P}}$. Therefore, we set TURN to P.

- If neither of the above are true, i.e., $\texttt{TURN} = \mathsf{P}$, $\mathsf{P}.\texttt{PREV_SLEK} = 1$, and $\mathsf{P}.\texttt{CUR_SLEK} = 1$, then it could be that the adversary is injecting into *either* P's previous or current sending epoch. Therefore, the ideal adversary specifies its choice of the last element with the last input γ to the INJECT command.

Actual delivery of injections is then handled in the DELIVER command, in the same simple manner as specified in the Honest Execution Section (Section 3.1). Namely, delivery of injected message with message id mid is done by removing it from $\mathsf{P}.\texttt{T}$ (if such an entry exists), and sending it to $\bar{\mathsf{P}}$. The functionality works this way in order to capture the scenario in which the real-world adversary modifies the first message of a new sending epoch for P to inform $\bar{\mathsf{P}}$ that P's last sending epoch contains more messages than it actually does. Therefore, the real-world adversary will be able to in the future inject such additional messages whenever it wants. The ideal-world adversary thus issues an INJECT command for all of these message ids at the time of the first modification,

so that later it can actually send them to $\bar{\mathsf{P}}$ using DELIVER commands (regardless of the status of the functionality's flags at that time).

If an injected message with id mid is indeed added to P.T, then the ideal functionality needs to also make sure that P can send a message with the same mid (since it does not know about the injection), but not allow the ideal adversary to deliver two messages with the same mid (since $\bar{\mathsf{P}}$ will only accept one such message in the DR). Therefore, in the SEND command, the ideal functionality checks if mid $\notin$ P.I and if so adds the corresponding message to P.T as in the honest execution. However, if mid is in P.I, the ideal functionality does not add the corresponding message to P.T, but still sends the length of the message (and the message itself if P.CUR_SLEK $= 1$) to the ideal adversary, mirroring that a ciphertext is still created in the real-world.

4 Building Blocks

In this work, we present the DR and our stronger TR protocols as modular constructions that use three components (following ACD): Key Derivation Function Chains (KDF Chains)—used to advance forward the public ratchet; continuous key-agreement (CKA)—used to generate the secrets that advance forward the public ratchet; and forward-secure authenticated encryption with associated data (FS-AEAD)—the symmetric ratchet itself. These components are presented in isolation in this section before combining them into the DR and TR schemes in Section 5. For FS-AEADand CKA we first provide security definitions for the two primitives and then constructions achieving them. Note that the DR and TR schemes we show later can use any CKA and FS-AEAD construction which satisfy their corresponding definitions. For KDF chains, we do not provide any definitions but rather achieve their security and functionality by modelling the underlying KDF as a (programmable) Random Oracle in Section 5. For a concrete instantiation of the underlying KDF, we point to that which the DR protocol uses: HKDF [KE10] with SHA-256 or SHA-512 [FIP95]. This section often follows the work of ACD verbatim.[3]

4.1 Key Derivation Function Chains

The DR protocol makes use of Key Derivation Function (KDF) Chains for the public ratchet. We take (almost) verbatim from its whitepaper by Marlinspike and Perrin [MP16a] the desired syntax and security properties of KDFs and KDF Chains: A KDF is a cryptographic function that takes as input a secret and random KDF key σ and some input data I and returns output data R (i.e., $R \leftarrow \mathrm{KDF}(\sigma, I)$). The output data R is indistinguishable from random provided the key isn't known (i.e. a KDF satisfies the requirements of a cryptographic "PRF"). If the key is not secret and random, the KDF should still provide a secure cryptographic hash of its key and input data.

To build a KDF Chain, Marlinspike and Perrin use iterated computations of a KDF, where one part of the output is used as a separate output key k and the other part is used to replace the KDF key σ, which can then be used with another input (i.e., $(\sigma', k) \leftarrow \mathrm{KDF}(\sigma, I)$). KDF chains have the following intuitive properties:

- **Resilience**: The output keys appear random to an adversary without knowledge of the KDF keys. This is true even if the adversary can control the KDF inputs.

[3] ACD use A and B to refer to the two parties of CKA and FS-AEAD. To remain consistent with the notation of our ideal functionality for Secure Messaging, we instead use P_1 and P_2.

- **FS**: Output keys from the past appear random to an adversary who learns the KDF key at some point in time.
- **PCS**: Future output keys appear random to an adversary who learns the KDF key at some point in time, provided that future inputs have added sufficient entropy.

As stated earlier, we simply model the KDF that underlies the KDF chain in the DR and TR as a (programmable) Random Oracle H. Thus, provided that either the KDF key or the input data is random and unknown to the adversary, the output will be random and unknown to the adversary, and thus both resilience as well as PCS are achieved. Likewise, FS is easily seen to be achieved.

4.1.1 Differences from ACD

Alwen *et al.* capture the above security properties of a KDF Chain in a primitive which they call *PRF-PRNGs*. They also provide a corresponding (deterministic) construction in the standard model. We refer the reader to Section 4.3 of their work for more details. However, we emphasize that a standard model PRF-PRNG construction suffices in their work only because in their SM security definition, leakages are not allowed during challenge epochs. On the other hand, our functionality requires the simulator to generate fake ciphertexts for an epoch, even if the adversary may later leak on the parties to reveal the keys for these ciphertexts. So, while these fake ciphertexts need to be at first indistinguishable from the real ciphertexts, and thus the keys used to encrypt them need to be sampled randomly, the adversary can later obtain the root KDF chain key and CKA shared secret for that epoch, from which the FS-AEAD initialization key is derived using the function underlying the KDF chain. Thus, we need to be able to properly program this function to output the correct (random) key, and so we model it as a (programmable) random oracle.

This is similar to the necessity of (programmable) random oracles for non-committing encryptions, as shown by a separation by Nielsen [Nie02] as opposed to just semantically secure encryption. Intuitively, to simulate a corruption that happens after "committing" the ciphertext one must use the random oracle programmability at its full extent – this common simulation paradigm also comes up naturally in our security analysis.

4.2 Forward-Secure AEAD

4.2.1 Defining FS-AEAD

Forward-secure authenticated encryption with associated data is a stateful primitive between a sender P_1 and a receiver P_2 and can be considered a single-epoch variant of the DR, a fact that is also evident from its security definition.

Definition 3. Forward-secure authenticated encryption with associated data (FS-AEAD) *is a tuple of algorithms* $\mathsf{FS\text{-}AEAD} = (\mathsf{FS\text{-}Init\text{-}S}, \mathsf{FS\text{-}Init\text{-}R}, \mathsf{FS\text{-}Send}, \mathsf{FS\text{-}Rcv})$*, where*

- FS-Init-S *(and similarly* FS-Init-R*) takes a key* k *and outputs a state* $v_{\mathsf{P}_1} \leftarrow \mathsf{FS\text{-}Init\text{-}S}(k)$*,*
- FS-Send *takes a state* v*, associated data* a*, and a message* m *and produces a new state and a ciphertext* $(v', e) \leftarrow \mathsf{FS\text{-}Send}(v, a, m)$*, and*
- FS-Rcv *takes a state* v*, associated data* a*, and a ciphertext* e *and produces a new state, an index, and a message* $(v', i, m) \leftarrow \mathsf{FS\text{-}Rcv}(v, a, e)$*.*

Observe that all algorithms of an FS-AEAD scheme are deterministic.

Memory management. In addition to the basic syntax above, it is useful to define the following two functions FS-Stop (called by the sender) and FS-Max (called by the receiver) for memory management:

- FS-Stop, given an FS-AEAD state v, outputs how many messages have been sent (sic) and then "erases" the FS-AEAD session corresponding to v from memory; and
- FS-Max, given a state v and an integer ℓ, remembers ℓ internally such that the session corresponding to v is erased from memory as soon as ℓ messages have been received.

These features will be useful in the full protocol (cf. Section 5) to be able to terminate individual FS-AEAD sessions when they are no longer needed. Providing a formal requirement for these additional functions is omitted. Moreover, since an attacker can infer the value of the message counter from the behavior of the protocol anyway, there is no dedicated oracle included in the security game below.

Correctness and security. Both correctness and security are built into the security game depicted in Figure 4. One can observe that it corresponds to a single epoch of the DR and it thus inherits those properties. When messages are sent, they are identified by a simple counter. Thus, for correctness, all honestly generated ciphertexts should be decrypted by the recipient to their corresponding messages with their corresponding indices.

For security, the **chall-P$_1$**(a, m_0, m_1) oracle on input messages m_0 and m_1 of the same length returns an encryption of m_b, depending on a randomly chosen bit b. FS with respect to corruptions of either party is required: If P_1 is corrupted then the adversary still should not know the underlying plaintext of previously generated ciphertexts. Moreover, if P_1 has sent i_{P_1} messages thus far, no messages with lower indices can be modified or injected. If P_2 is corrupted then the adversary still should not know the underlying plaintext of previously generated and deliverd ciphertexts. Moreover, for challenge ciphertexts that are in-transit at the time of such a corruption, we require the FS-AEAD scheme to "explain" them. That is, we require the ability to program ciphertexts so that they decrypt consistently to any chosen message after the fact, and furthermore so that encrypting the message under the corrupted state in the same manner as P_1 would have honestly done indeed results in the same ciphertext (similar to non-committing encryption).[4] We model this explicitly as an algorithm FS-Expl-In-Trans-Cts that takes the corrupted state, as well as the list of in-transit challenge messages and their corresponding associated data and ciphertexts, and programs the messages and ciphertexts to be consistent. Furthermore, we allow the initialization key k to be corrupted, in which case we require *all* challenge ciphertexts to be "explained" in the same way. We model this explicitly as an algorithm FS-Expl-Vul-Cts that takes the initial key k, as well as all sent challenge messages and their corresponding associated data and ciphertexts, and programs the messages and ciphertexts to be consistent. All FS-AEAD schemes considered in this work are required to provide both algorithms (really FS-Expl-Vul-Cts is only required for the proof of the DR, but it can easily be achieved with a random oracle, anyway).

The advantage of an attacker A against an FS-AEAD scheme FS-AEAD is denoted by the expression $\mathsf{Adv}^{\mathsf{FS\text{-}AEAD}}_{\text{fs-aead}}(\mathsf{A})$. The attacker is parameterized by its running time t and the total number of queries q it makes.

[4] Constructions used by the DR, which use an underlying AEAD based on CBC+HMAC and SIV, will satisfy this property.

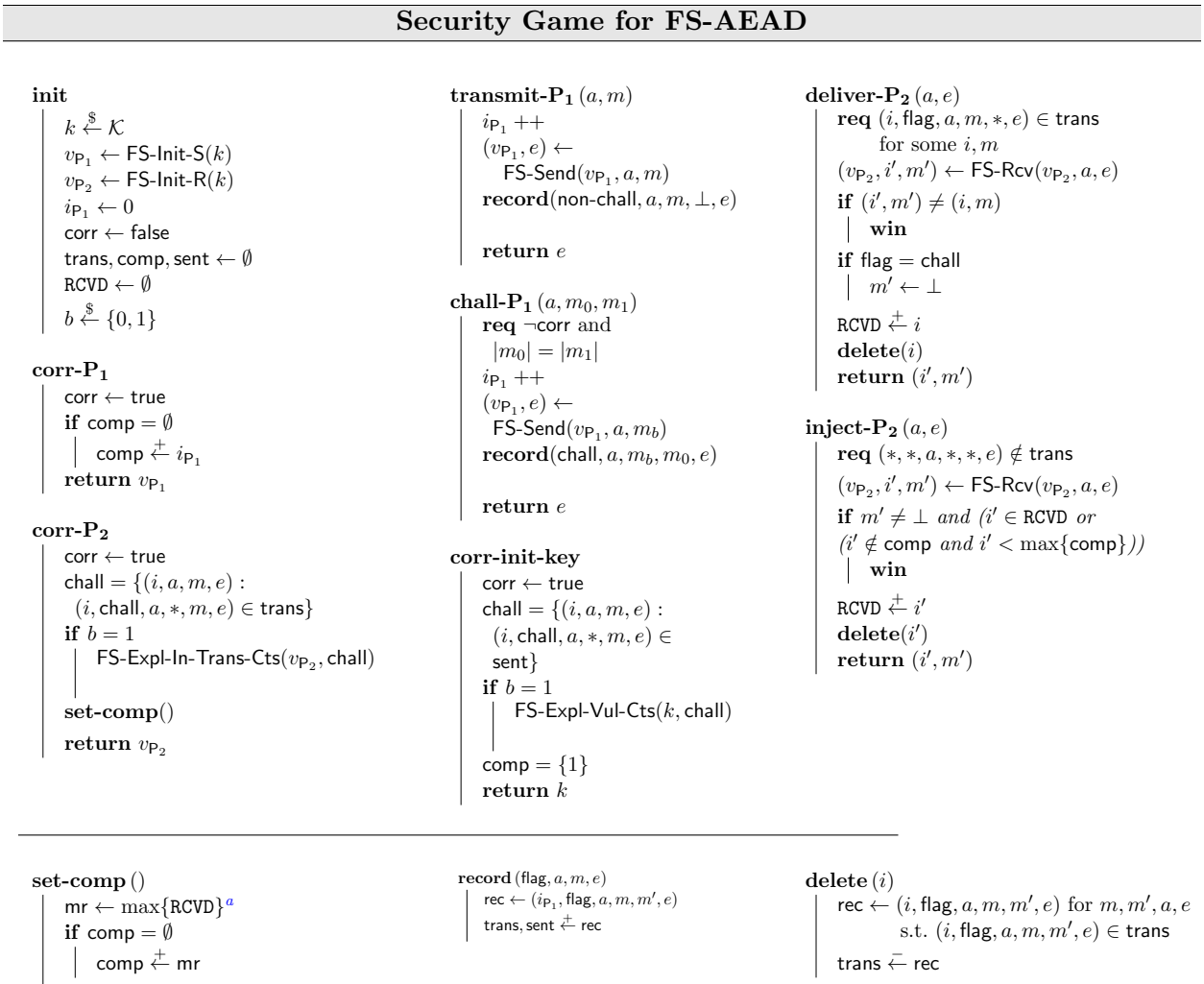

Figure 4: Oracles corresponding to party P_1 of the FS-AEAD security game for a scheme FS-AEAD $=$ (FS-Init-S, FS-Init-R, FS-Send, FS-Rcv).

Definition 4. *An FS-AEAD scheme* FS-AEAD *is* (t, q, ε)-secure *if for all* (t, q)*-attackers* $\mathcal{A}$*,*

$$\mathsf{Adv}^{\mathsf{FS\text{-}AEAD}}_{\text{fs-aead}}(\mathcal{A}) \;\leq\; \varepsilon \;.$$

Definition 5. *An FS-AEAD scheme is simply called* ε-secure *if for every* $t, q \in \text{poly}(\kappa)$ *and*

$\varepsilon \in \mathrm{negl}(\kappa)$, *where* κ *is the security parameter*

$$\mathsf{Adv}^{\mathsf{FS\text{-}AEAD}}_{\text{fs-aead}}(\mathcal{A}) \leq \varepsilon.$$

4.2.2 Differences from ACD

Explaining ciphertexts and init key corruption. Our ideal functionalities $\mathcal{F}_{\mathsf{DR}}$ and $\mathcal{F}_{\mathsf{TR}}$ of course allow the adversary to leak on either party at all times (unlike the Secure Messaging definition of ACD). Thus, although the ideal adversary may at first only get the length of a message when it is sent and still need to simulate the corresponding ciphertext, if a leak occurs on the recipient while the ciphertext is still in-transit, the ideal adversary needs to be able to *explain* the ciphertext as the real message. Thus, the underlying FS-AEAD security game must also require such explanation of in-transit challenge ciphertexts for our security proofs of the DR and TR. Furthermore, in (only) the DR, state leakages on the sender of an epoch may leak all *vulnerable* messages of the epoch to the adversary, and thus the FS-AEAD must explain *all* challenge ciphertexts sent. The above is why we need (programmable) random oracles in our instantiation below (recall Section 4.1.1 for more discussion).

Continuing the game if P_2 is corrupted. In the FS-AEAD definition of ACD, the game ends once P_2 is corrupted. However, even if such a corruption occurs, if the adversary chooses to still deliver honest ciphertexts, then we must still require correctness of the FS-AEAD.

Fixing comp. Intuitively, if an attacker corrupts P_1 then it can successfully inject messages for a later index $i > i_{\mathsf{P}_1}$. However, the ACD definition declares that the attacker wins if this happens (since for their differently defined dictionary comp in this situation, $i \notin \mathsf{comp}$ if the attacker did not make any **transmit-$\mathbf{P_1}$** or **chall-$\mathbf{P_1}$** queries, which triggers **win** in the **inject-$\mathbf{P_2}$** oracle of their game). Thus, we use our own logic for defining comp. (We also explicitly require FS-Rcv to only accept one message per index and output $\perp$ if it does receive more than one, in order to be properly used for our ideal functionalities. The security game uses set RCVD for this purpose.)

4.2.3 Instantiating FS-AEAD

An FS-AEAD scheme can be easily constructed from two components:

- an AEAD scheme $\mathsf{AE} = (\mathsf{Enc}, \mathsf{Dec})$, and
- a KDF $\mathsf{H} : \mathcal{W} \to \mathcal{W} \times \mathcal{K}$, where $\mathcal{K}$ is the key space of the AEAD scheme.

The scheme is described in Figure 5. For simplicity the states of sender P_1 and receiver P_2 are not made explicit; it consists of the variables set during initialization. The main idea of the scheme, is that P_1 and P_2 share KDF key w. KDF key w is initialized with a pre-shared key $k \in \mathcal{W}$, which is assumed to be chosen uniformly at random. Both parties keep local counters i_{P_1} and i_{P_2}, respectively.[5] P_1, when sending the i^{th} message m with associated data (AD) a, uses H to expand the current state to a new state and an AEAD key $(w, K) \leftarrow \mathsf{H}(w)$ and computes an AEAD encryption under K of m with AD $h = (i, a)$.

[5]For ease of description, the FS-AEAD state of the parties is not made explicit as a variable v.

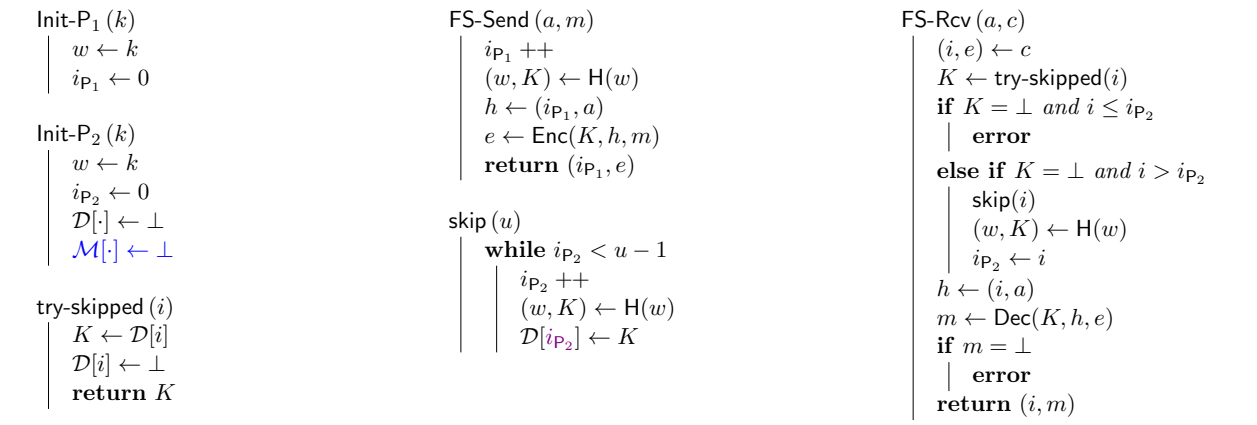

Figure 5: FS-AEAD scheme based on AEAD and a KDF H.

Since P_2 may receive ciphertexts out of order, whenever he receives a ciphertext, he first checks whether the key is already stored in a dictionary $\mathcal{D}$. If the index of the message is higher than expected (i.e., larger than $i_{\mathsf{P}_2} + 1$), P_2 skips the KDF chain ahead and stores the skipped keys in $\mathcal{D}$. In either case, once the key is obtained, it is used to decrypt. If decryption fails, FS-Rcv throws an exception (**error**), which causes the state to be rolled back to where it was before the call to FS-Rcv.

Algorithms FS-Expl-In-Trans-Cts and FS-Expl-Vul-Cts. Lastly, we explain the algorithms FS-Expl-In-Trans-Cts$(v_{\mathsf{P}_2}, \mathsf{chall})$ and FS-Expl-Vul-Cts(k, chall) provided by our FS-AEAD. In order to explain ciphertexts correctly, they rely on the explanation algorithm AEAD-Expl-Ct of the underlying AEAD scheme. Formally, for every $(i, a, m, e) \in \mathsf{chall}$, FS-Expl-In-Trans-Cts executes AEAD-Expl-Ct(K_i, a, m, e), where $K_i \leftarrow \mathcal{D}[i]$ for all $i < i_{\mathsf{P}_2}$ and for all $i > i_{\mathsf{P}_2}$, K_i is computed directly via $i - i_{\mathsf{P}_2}$ KDF invocations on current KDF key $w_{i_{\mathsf{P}_2}}$ in v_{P_2}. Algorithm FS-Expl-Vul-Cts proceeds similarly, instead computing K_i directly via i KDF invocations on initial KDF key w_0.

Theorem 1. *Assume* AE *is a* $(t', \varepsilon_{\text{aead}})$*-OT-CCA secure AEAD scheme and* H *is modelled as a (programmable) random oracle. Then, the above FS-AEAD scheme* FS-AEAD *is* (t, q, ε)*-secure for* $t \approx t'$ *and*

$$\varepsilon \;\leq\; q \cdot \varepsilon_{\text{aead}} \;.$$

Proof. The proof is a straight-forward hybrid argument: Let H_0 denote the actual FS-AEAD security game.

- In hybrid experiment H_1, the **win** condition inside the **deliver-P$_2$** oracle is removed. The perfect correctness of the proposed FS-AEAD scheme is easily seen by inspection and the correctness of the underlying AEAD scheme. Hence H_0 and H_1 are indistinguishable.
- Now, the security of H_1 follows from that of the underlying AEAD. It is obvious that injections for already received indices fail, since the FS-AEAD clearly outputs **error**. Moreover,

injections for uncompromised indices ($i \notin \mathsf{comp}$ and $i < \max\{\mathsf{comp}\}$) fail by the security of the underlying AEAD (since the attacker does not have the corresponding key). Also, we program the random oracle so that all corrupted (random) AEAD keys (and subsequent KDF keys) are simulated correctly to the attacker (for example if the attacker queries **chall-P$_1$**, followed by **corr-P$_1$** then **corr-P$_2$**). Explainability follows from that of the AEAD.

□

4.3 Continuous Key Agreement

As in the work of Alwen et al. ACD, we separate out the primitive that generates the secrets for public-ratchet updates in the DR and TR, and call it *continuous key agreement (CKA)*. In this section, we present our definitions of CKA and instantiations from the *strong-DH* (StDH) assumption [ABR01].[6] The StDH assumption is: given random and independent group elements g^a, g^b, and access to oracle $\mathbf{ddh}(g^a, \cdot, \cdot)$, which on input X, Y returns 1 if $X^a = Y$ and 0 otherwise, it is hard to compute g^{ab}.

4.3.1 Defining CKA

At a high level, CKA is a synchronous two-party protocol between P_1 and P_2. Odd rounds i consist of P_1 sending and P_2 receiving a message T_i, whereas in even rounds, P_2 is the sender and P_1 the receiver. Each round i also produces a key I_i, which is output by the sender upon sending T_i and by the receiver upon receiving T_i.

Definition 6. *A* continuous-key-agreement (CKA) scheme *is a quadruple of algorithms* $\mathsf{CKA} = (\mathsf{CKA\text{-}Init\text{-}P_1}, \mathsf{CKA\text{-}Init\text{-}P_2}, \mathsf{CKA\text{-}S}, \mathsf{CKA\text{-}R})$*, where*

- $\mathsf{CKA\text{-}Init\text{-}P_1}$ *(and similarly* $\mathsf{CKA\text{-}Init\text{-}P_2}$*) takes a key* k *and produces an initial state* $\gamma^{\mathsf{P}_1} \leftarrow \mathsf{CKA\text{-}Init\text{-}P_1}(k)$ *(and* γ^{P_2}*),*
- $\mathsf{CKA\text{-}S}$ *takes a state* γ*, and produces a new state, message, and key* $(\gamma', T, I) \xleftarrow{\$} \mathsf{CKA\text{-}S}(\gamma)$*, and*
- $\mathsf{CKA\text{-}R}$ *takes a state* γ *and message* T *and produces new state and a key* $(\gamma', I) \leftarrow \mathsf{CKA\text{-}R}(\gamma, T)$*.*

Denote by $\mathcal{K}$ *the space of initialization keys* k *and by* $\mathcal{I}$ *the space of CKA keys* I*.*

Correctness. A CKA scheme is correct if in the security game in Figure 6 (explained below), P_1 and P_2 always, i.e., with probability 1, output the same key in every round.

Security. The security property we will require a CKA scheme to satisfy is that conditioned on the transcript $T_1, T_2, \ldots$, the keys $I_1, I_2, \ldots$ are unrecoverable. An attacker against a CKA scheme is required to be passive, i.e., may not modify the messages T_i. However, it is given the power to possibly (1) control the random coins used by the sender and (2) leak the current state of either party. Given the capabilities of the adversary, it is easy to see that some keys I_i would be

[6] We do not provide CKA schemes secure according to our definitions based on LWE or generic KEMs, as in ACD. However, we note that our stronger scheme CKA^+ is intuitively at least as strong as their construction from generic KEMs, but more efficient.

Security Games for CKA

init (t^*)
- $k \xleftarrow{\$} \mathcal{K}$
- $\gamma_0^{\mathsf{P}_1} \leftarrow \mathsf{CKA\text{-}Init\text{-}P_1}(k)$
- $\gamma_0^{\mathsf{P}_2} \leftarrow \mathsf{CKA\text{-}Init\text{-}P_2}(k)$
- $t_{\mathsf{P}_1}, t_{\mathsf{P}_2} \leftarrow 0$
- $\mathsf{Recv\text{-}State}[*] \leftarrow \bot$

corr-$\mathbf{P_1}$
- **req** $\mathsf{allow\text{-}corr}_{\mathsf{P}_1}$
- **return** $\gamma_{t_{\mathsf{P}_1}}^{\mathsf{P}_1}$

send-$\mathbf{P_1}$
- t_{P_1} ++
- $(\gamma_{t_{\mathsf{P}_1}}^{\mathsf{P}_1}, T, I) \xleftarrow{\$} \mathsf{CKA\text{-}S}(\gamma_{t_{\mathsf{P}_1}}^{\mathsf{P}_1})$
- $\mathsf{Recv\text{-}State}[t_{\mathsf{P}_1}+1] \leftarrow \gamma_{t_{\mathsf{P}_1}}^{\mathsf{P}_1}$
- **return** (T, I)

send-$\mathbf{P_1}$' (r)
- t_{P_1} ++
- **req** $\mathsf{allow\text{-}bad\text{-}rand}_{\mathsf{P}_1}$
- $(\gamma_{t_{\mathsf{P}_1}}^{\mathsf{P}_1}, T, I) \leftarrow \mathsf{CKA\text{-}S}(\gamma_{t_{\mathsf{P}_1}-1}^{\mathsf{P}_1}; r)$
- $\mathsf{Recv\text{-}State}[t_{\mathsf{P}_1}+1] \leftarrow \gamma_{t_{\mathsf{P}_1}}^{\mathsf{P}_1}$
- **return** (T, I)

receive-$\mathbf{P_1}$
- t_{P_1} ++
- $(\gamma_{t_{\mathsf{P}_1}}^{\mathsf{P}_1}, *) \leftarrow \mathsf{CKA\text{-}R}(\gamma_{t_{\mathsf{P}_1}-1}^{\mathsf{P}_1}, T)$

chall-$\mathbf{P_1}$
- t_{P_1} ++
- **req** $t_{\mathsf{P}_1} = t^*$
- $(\gamma_{t_{\mathsf{P}_1}}^{\mathsf{P}_1}, T, I) \xleftarrow{\$} \mathsf{CKA\text{-}S}(\gamma_{t_{\mathsf{P}_1}-1}^{\mathsf{P}_1})$
- **return** T

test (t, T, I)
- **req** $\mathsf{Recv\text{-}State}[t] \neq \bot$
- **if** $(*, I) \leftarrow \mathsf{CKA\text{-}R}(\mathsf{Recv\text{-}State}[t], T)$
  - **return** 1
- **else**
  - **return** 0

$\mathsf{allow\text{-}corr}_{\mathsf{P}_1}, \mathsf{allow\text{-}bad\text{-}rand}_{\mathsf{P}_1} :\Longleftrightarrow$
$$\begin{cases} t_{\mathsf{P}_1} \neq t^* & t^* \text{ is odd} \\ t_{\mathsf{P}_1} \neq t^* - 1 & t^* \text{ is even} \end{cases}$$
$\mathsf{allow\text{-}corr}_{\mathsf{P}_2}, \mathsf{allow\text{-}bad\text{-}rand}_{\mathsf{P}_2} :\Longleftrightarrow$
$$\begin{cases} t_{\mathsf{P}_2} \neq t^* - 1 & t^* \text{ is odd} \\ t_{\mathsf{P}_2} \neq t^* & t^* \text{ is even} \end{cases}$$

$\mathsf{allow\text{-}corr}_{\mathsf{P}_1} :\Longleftrightarrow t_{\mathsf{P}_1} \neq t^* - 1 \ \vee \ t^*$ is odd
$\mathsf{allow\text{-}bad\text{-}rand}_{\mathsf{P}_1} :\Longleftrightarrow t_{\mathsf{P}_1} \neq t^* \ \vee \ t_{\mathsf{P}_1} \neq t^* - 1$
$\mathsf{allow\text{-}corr}_{\mathsf{P}_2} :\Longleftrightarrow t_{\mathsf{P}_2} \neq t^* - 1 \ \vee \ t^*$ is even
$\mathsf{allow\text{-}bad\text{-}rand}_{\mathsf{P}_2} :\Longleftrightarrow t_{\mathsf{P}_2} \neq t^* \ \vee \ t_{\mathsf{P}_2} \neq t^* - 1$

Figure 6: Oracles corresponding to party P_1 of the CKA security game for a scheme $\mathsf{CKA} = (\mathsf{CKA\text{-}Init\text{-}P_1}, \mathsf{CKA\text{-}Init\text{-}P_2}, \mathsf{CKA\text{-}S}, \mathsf{CKA\text{-}R})$; the oracles for P_2 are defined analogously. Conditions for the weaker security game, i.e., ε-security, are presented to the left of those for the stronger game, i.e., $(\varepsilon, +)$-security.

recoverable. The security guarantee offered by the CKA scheme would then be that even given the transcript $T_1, T_2, \ldots$, assuming certain "fine-grained" conditions around when the adversary controls the randomness used by parties and when the adversary learns the state of parties, most keys $I_1, I_2, \ldots$ are unrecoverable. It will also be the case that parties thus recover from such bad randomness and leakage issued by the adversary.

The formal security game for CKA is provided in Figure 6. It begins with a call to the **init** oracle, which initializes the states of both parties, and defines epoch counters t_{P_1} and t_{P_2}. Procedure **init** takes a value t^*, which determines in which round the challenge oracle may be called; the task of the adversary will be to recover the key I_{t^*} for that round.

Upon completion of the initialization procedure, the attacker gets to interact arbitrarily with the remaining oracles, as long as *the calls are in a "ping-pong" order*, i.e., a call to a send oracle for P_1 is followed by a receive call for P_2, then by a send oracle for P_2, etc. The attacker only gets to use the challenge oracle for epoch t^*. The attacker additionally has the capability of testing the consistency of T_t and I_t (i.e., whether the corresponding receiver in epoch t would produce key I_t on input message T_t).

The security game of Alwen et al. ACD is parametrized by Δ_{CKA}, which stands for the number of epochs that need to pass after t^* until the states do not contain secret information pertaining to the challenge. Once a party reaches epoch $t^* + \Delta_{\mathsf{CKA}}$, its state may be revealed to the attacker (via the corresponding corruption oracle). We avoid this and define two levels of security, the former weaker than the latter. At the bottom of Figure 6, the conditions $\mathsf{allow\text{-}corr}_{\mathsf{P}}$ and $\mathsf{allow\text{-}bad\text{-}rand}_{\mathsf{P}}$ for the weaker version are presented to the left of those of the stronger version. We define two levels of security in order to capture a stronger, more fine-grained security guarantee for CKA which will be useful in providing stronger security guarantees for the DR and TR as a whole when one considers the composition of all its building blocks, CKA being one of them. In the former, bad randomness is not allowed in the epochs t^* and $t^* - 1$, and corruptions are not allowed in the epoch t^* after invoking CKA-S (for the sender of epoch t^*) and before invoking CKA-R (for the receiver of epoch t^*). In the latter, bad randomness is not allowed in the epochs t^* and $t^* - 1$, and corruption of the receiver of epoch t^* is not allowed before invoking CKA-R (for epoch t^*). There is no other difference between the two notions.

The game ends (not made explicit) once both states are revealed after the challenge phase. The attacker wins the game if it eventually outputs the correct secret key I_{t^*} corresponding to the challenge message T_{t^*}. The advantage of an attacker $\mathcal{A}$ against a CKA scheme CKA is denoted by $\mathsf{Adv}^{\mathsf{CKA}}(\mathcal{A})$ and $\mathsf{Adv}^{\mathsf{CKA}^+}(\mathcal{A})$ for the weaker and stronger security guarantees, respectively. The attacker is parameterized by its running time t.

Definition 7. *A CKA scheme* CKA *is* (t, ε)-secure *if for all t-attackers $\mathcal{A}$,*

$$\mathsf{Adv}^{\mathsf{CKA}}(\mathcal{A}) \leq \varepsilon.$$

Definition 8. *A CKA scheme* CKA *is* $(t, \varepsilon, +)$-secure *if for all t-attackers $\mathcal{A}$,*

$$\mathsf{Adv}^{\mathsf{CKA}^+}(\mathcal{A}) \leq \varepsilon.$$

Definition 9. *A CKA scheme* CKA *is simply called ε-secure (resp. $(\varepsilon, +)$-secure) if for every $t \in \mathrm{poly}(\kappa)$, $\mathsf{Adv}^{\mathsf{CKA}}(\mathcal{A}) \leq \varepsilon$ (resp. $\mathsf{Adv}^{\mathsf{CKA}^+}(\mathcal{A})) \leq \varepsilon$) and $\varepsilon \in \mathrm{negl}(\kappa)$, where κ is the security parameter.*

Observe that since the TR uses a CKA with the latter, stronger security, the attack described in Section 1.4 is thwarted. This is because even if the epoch t^* sender is corrupted after invoking CKA-S, I_{t^*} should remain hidden.

Remark 1. *Many* natural *CKA schemes satisfy an additional property that given a CKA message T and key I for a given round, it is possible to deterministically compute the corresponding state of the receiving party after her execution of* CKA-R *in that round. We model this explicitly by requiring a deterministic algorithm* CKA-Der-R *that takes a message T and key I and produces the correct state $\gamma' \leftarrow \mathsf{CKA\text{-}Der\text{-}R}(T, I)$. All CKA schemes in this work are required to be* natural *(including the public-ratchet update mechanism of the* DR*).*

4.3.2 Differences from ACD

Fine-grained security guarantees. Recall the "CKA from DDH" scheme from ACD (which is the public ratchet used in the DR), $\mathsf{CKA} = (\mathsf{CKA\text{-}Init\text{-}P}_1, \mathsf{CKA\text{-}Init\text{-}P}_2, \mathsf{CKA\text{-}S}, \mathsf{CKA\text{-}R})$, that is instantiated in a cyclic group $G = \langle g \rangle$ as follows:

- The initial shared state $k = (h, x_0)$ consists of a (random) group element $h = g^{x_0}$ and its discrete logarithm x_0. The initialization for P_1 outputs $h \leftarrow \mathsf{CKA\text{-}Init\text{-}P}_1(k)$ and that for P_2 outputs $x_0 \leftarrow \mathsf{CKA\text{-}Init\text{-}P}_2(k)$.
- The send algorithm $\mathsf{CKA\text{-}S}$ takes as input the current state $\gamma = h$ and proceeds as follows: It
  1. chooses a random exponent x;
  2. computes the corresponding key $I \leftarrow h^x$;
  3. sets the CKA message to $T \leftarrow g^x$;
  4. sets the new state to $\gamma \leftarrow x$; and
  5. returns (γ, T, I).
- The receive algorithm $\mathsf{CKA\text{-}R}$ takes as input the current state $\gamma = x$ as well as a message $T = h$ and proceeds as follows: It
  1. computes the key $I = h^x$;
  2. sets the new state to $\gamma \leftarrow h$; and
  3. returns (γ, I).

Now, let x_0 be the random exponent that is part of the initial shared state, and for $i > 0$, let x_i be the random exponent picked by $\mathsf{CKA\text{-}S}$ (which was run by P_1 for odd i, and P_2 for even i) in round i. Then, we have the following:

- The key for round i is $I_i = g^{x_{i-1}x_i}$.
- The message for round i is $T_i = g^{x_i}$.
- If i is odd, and P_1 has yet to invoke $\mathsf{CKA\text{-}S}$, $\gamma^{\mathsf{P}_1} = g^{x_{i-1}}$ and $\gamma^{\mathsf{P}_2} = x_{i-1}$.
- If i is odd, and P_1 has invoked $\mathsf{CKA\text{-}S}$, but P_2 has yet to invoke $\mathsf{CKA\text{-}R}$, $\gamma^{\mathsf{P}_1} = x_i$ and $\gamma^{\mathsf{P}_2} = x_{i-1}$.
- If i is odd, and P_1 has invoked $\mathsf{CKA\text{-}S}$, and P_2 has invoked $\mathsf{CKA\text{-}R}$, $\gamma^{\mathsf{P}_1} = x_i$ and $\gamma^{\mathsf{P}_2} = g^{x_i}$.
- If i is even, and P_2 has yet to invoke $\mathsf{CKA\text{-}S}$, $\gamma^{\mathsf{P}_1} = x_{i-1}$ and $\gamma^{\mathsf{P}_2} = g^{x_{i-1}}$.
- If i is even, and P_2 has invoked $\mathsf{CKA\text{-}S}$, but P_1 has yet to invoke $\mathsf{CKA\text{-}R}$, $\gamma^{\mathsf{P}_1} = x_{i-1}$ and $\gamma^{\mathsf{P}_2} = x_i$.
- If i is even, and P_2 has invoked $\mathsf{CKA\text{-}S}$, and P_1 has invoked $\mathsf{CKA\text{-}R}$, $\gamma^{\mathsf{P}_1} = g^{x_i}$ and $\gamma^{\mathsf{P}_2} = x_i$.

Based on the above, we make the following observations:

- If i is odd and P_1 is corrupted after invoking $\mathsf{CKA\text{-}S}$, the adversary learns $\gamma^{\mathsf{P}_1} = x_i$ and since it also has access to g^{x_j} for all $j \geq 1$, the adversary learns I_i and I_{i+1}.
- If i is even and P_1 is corrupted after invoking $\mathsf{CKA\text{-}R}$, and P_2 used good randomness in picking x_i while invoking $\mathsf{CKA\text{-}S}$ in round i, the adversary learns $\gamma^{\mathsf{P}_1} = g^{x_i}$, but since it only (assuming no other corruptions) has access to g^{x_j} for all $j \geq 1$, the adversary does not learn I_i (if P_1 also used good randomness in picking x_{i-1} while invoking $\mathsf{CKA\text{-}S}$ in round $i-1$) or I_{i+1} (if P_1 also uses good randomness in picking x_{i+1} while invoking $\mathsf{CKA\text{-}S}$ in round $i+1$).

Thus, the CKA keys for two rounds are compromised only in the case where the party that has last sent a message is corrupted, and not if the party has last received a message. This allows us to consider our stronger version of the CKA security game than the one described in ACD.

Non-malleability. Consider the following scenario in the DR or TR: It is P_1's turn to start a new sending epoch, but she has not yet. Then her state is leaked, and afterwards, she sends the first message m_1 of the epoch with good randomness. Then, if P_2 started her last epoch with good randomness, and there are no other leakages, m_1 is required to remain private by $\mathcal{F}_{\mathsf{DR}}$ and $\mathcal{F}_{\mathsf{TR}}$, respectively. However, all authenticity for m_1 is lost—the adversary leaked on P_1 beforehand and thus could have generated the message herself. Therefore, we replace the indistinguishability definition of ACD with our recoverability definition and require non-malleability of CKA messages via the **test** oracle—the adversary should not be able to maul them in order to learn about the actual session messages sent in the DR or TR. See the full security proof of Theorem 4 for the DR and TR, as well as Appendix A.2.4, for more details.

4.3.3 Instantiating CKA

Note that the above scheme is *natural*, i.e., it supports a CKA-Der-R algorithm, namely, $\mathsf{CKA\text{-}Der\text{-}R}(T, I) = T$. Now, we will show that the above scheme is (t, ε)-secure. Before we do so, we introduce the Square-Diffie-Hellman (SqDH) assumption. The SqDH assumption is given random g^a, it is hard to compute g^{a^2}. We will use the SqDH assumption in the presence of a $\mathbf{ddh}(g^a, \cdot, \cdot)$ oracle as an intermediary to prove security of the above scheme from StDH. For the theorem below, let a group G be (t, ε)-SqDH-secure (resp. (t, ε)-StDH-secure) if every attacker with running time at most t has advantage at most ε at solving SqDH (resp. StDH) challenges. It has been shown [MW96, Kil01, BFGJ17, BDZ03, Gal12] that if a group G is (t, ε)-StDH-secure then it is $(t'/2, \sqrt{\varepsilon})$-SqDH-secure in the presence of a $\mathbf{ddh}(g^a, \cdot, \cdot)$ oracle, for $t' \approx t$.

Theorem 2. *Assume group G is (t, ε)-StDH-secure. Then, the above CKA scheme* CKA *is $(t'/2, \sqrt{\varepsilon})$-secure for $t \approx t'$.*

Proof. Assume w.l.o.g. that t^* is *odd*, i.e., P_1 sends the challenge; the case where t^* is even is handled analogously. Let g^a be a SqDH challenge. The reduction simulates the CKA protocol in the straight-forward way but embeds the challenge into the CKA as follows:

- in epoch $t^* - 1$, it uses $T_{t^*-1} = g^a$ and $I_{t^*-1} = g^{xa}$, where x is the exponent used to simulate $T_{t^*-2} = g^x$.
- in epoch t^*, it samples a random exponent r and uses $T_{t^*} = (g^a)^r$ and $I_{t^*} = g^{a^2 r}$ which is the key the adversary is to recover; and
- in epoch $t^* + 1$, for exponent x' (possibly generated using adversarial randomness), it uses $T_{t^*+1} = g^{x'}$ and $I_{t^*+1} = g^{arx'}$.

It is easy to verify that this correctly simulates the CKA experiment. Also note that the test oracle can be perfectly simulated with the help of a DDH oracle: if $\mathbf{test}(t^*, T, I)$ is queried, the reduction simply queries the DDH oracle on (g^a, T, I); if $\mathbf{test}(t^* + 1, T, I)$ is queried, the reduction simply queries the DDH oracle on (g^a, T^r, I); all other $\mathbf{test}()$ queries can be directly simulated. Finally, if the CKA adversary outputs correct guess $I_t^* = g^{a^2 r}$, the reduction correctly returns the SqDH challenge $(I_t^*)^{1/r} = g^{a^2}$. □

4.3.4 Instantiating CKA^+

A CKA scheme $\mathsf{CKA}^+ = (\mathsf{CKA\text{-}Init\text{-}P_1}, \mathsf{CKA\text{-}Init\text{-}P_2}, \mathsf{CKA\text{-}S}, \mathsf{CKA\text{-}R})$ can be obtained assuming random oracles or circular-secure ElGamal in a cyclic group $G = \langle g \rangle$ (with exponent space $\mathcal{X}$) using a function $\mathsf{H} : \mathcal{I} \to \mathcal{X}$ as follows:

- The initial shared state $k = (h, x_0)$ consists of a (random) group element $h = g^{x_0}$ and its discrete logarithm x_0. The initialization for P_1 outputs $h \leftarrow \mathsf{CKA\text{-}Init\text{-}P_1}(k)$ and that for P_2 outputs $x_0 \leftarrow \mathsf{CKA\text{-}Init\text{-}P_2}(k)$.
- The send algorithm $\mathsf{CKA\text{-}S}$ takes as input the current state $\gamma = h$ and proceeds as follows: It
  1. chooses a random exponent x;
  2. computes the corresponding key $I \leftarrow h^x$;
  3. sets the CKA message to $T \leftarrow g^x$;
  4. sets the new state to $\gamma \leftarrow x \cdot \mathsf{H}(I)$; and
  5. returns (γ, T, I).
- The receive algorithm $\mathsf{CKA\text{-}R}$ takes as input the current state $\gamma = x$ as well as a message $T = h$ and proceeds as follows: It
  1. computes the key $I = h^x$;
  2. sets the new state to $\gamma \leftarrow h^{\mathsf{H}(I)}$; and
  3. returns (γ, I).

Note that the above scheme is *natural*, i.e., it supports a $\mathsf{CKA\text{-}Der\text{-}R}$ algorithm, namely, $\mathsf{CKA\text{-}Der\text{-}R}(T, I) = T^{\mathsf{H}(I)}$. Now we show its security in the theorem below.

Theorem 3. *Assume group G is (t, ε)-StDH-secure. Additionally, assume the existence of a random oracle H. Then, the above CKA scheme CKA is $(t', \varepsilon, +)$-secure for $t \approx t'$. Furthermore, this CKA scheme also has dense transcripts.*

Proof. Assume w.l.o.g. that t^* is *odd*, i.e., P_1 sends the challenge; the case where t^* is even is handled analogously. Let g^a, g^b be a StDH challenge. The reduction simulates the CKA protocol in the straight-forward way but embeds the challenge into the CKA as follows:

- in epoch $t^* - 1$, it uses $T_{t^*-1} = g^a$ and $I_{t^*-1} = g^{xa\mathsf{H}(I_{t^*-2})}$, where x is the exponent used to simulate $T_{t^*-2} = g^x$.
- in epoch t^*, it uses $T_{t^*} = g^b$ and $I_{t^*} = g^{ab\mathsf{H}(I_{t^*-1})}$ which is the key the adversary is to recover, as well as sets $\gamma_{t^*}^{\mathsf{P}_1} \leftarrow y$, for random y in $\mathcal{X}$; and
- in epoch $t^* + 1$, for exponent x' (possibly generated using adversarial randomness), it uses $T_{t^*+1} = g^{x'}$ and $I_{t^*+1} = g^{yx'}$.

It is easy to verify that this correctly simulates the CKA experiment since H is a random oracle. In particular, randomly sampled y properly simulates $ar \cdot \mathsf{H}(I_{t^*})$: If the adversary does not query the random oracle on I_{t^*} then y is properly distributed. Moreover, when she makes a random oracle query for any I, the reduction can use the DDH oracle on $(g^a, g^{b\mathsf{H}(I_{t^*-1})}, I)$ so that if indeed $I = I_{t^*}$,

the reduction will know, and will then forward to its challenger $g^{ab} = I_{t^*}^{1/\mathsf{H}(I_{t^*-1})}$ before answering the CKA^+ attacker's query.

Similarly, the test oracle can be perfectly simulated with the help of the DDH oracle: if $\mathbf{test}(t^*, T, I)$ is queried, the reduction simply queries the DDH oracle on $(g^a, T^{\mathsf{H}(I_{t^*-1})}, I)$; all other $\mathbf{test}()$ queries can be directly simulated.

□

Remark 2. *The above theorem can also be proved without assuming that* H *is a random oracle, but by assuming rather that El-Gamal is circularly-secure in* G *(also in the presence of a* $\mathbf{ddh}(g^a, \cdot, \cdot)$ *oracle). Specifically, we assume that for uniformly random and independent exponents* a *and* b*, the distributions* $(g, g^a, g^b, b{\cdot}\mathsf{H}(g^{ab}))$ *and* (g, g^a, g^b, y) *are indistinguishable, where* y *is uniformly random in the exponent space* $\mathcal{X}$*. Based on this assumption (as opposed to* H *being a random oracle), we see that the embedding above still correctly simulates the CKA experiment, and essentially the same proof works for the security of the CKA scheme.*

4.3.5 Even stronger security for CKA^+

One can observe that CKA^+ is in fact even more secure than the $(t, \varepsilon, +)$-security that we proved for it. Although formalizing its full security is quite complex and does not have too much of an impact on our stronger TR protocol (see Section 6.5 for an informal description of the impact that it does have, based on the below) we here informally describe a scenario in which CKA^+ does indeed achieve stronger security.

Consider the scenario in which element x_0 of the initial shared state $k = (g^{x_0}, x_0)$ is secure, P_2 never has good randomness during the session (of course for x_0 to be secure, if P_2 generated it, then she must have had good randomness at that point, but in the real-world, this may have happened a long time ago), and P_1 has good randomness for sampling her first exponent x_1, but then never again. Moreover, assume that both P_1 and P_2 are never corrupted. Then *every* shared secret I_t will still be secure: $I_1 = g^{x_0 x_1}$ is secure by StDH since the adversary only has g^{x_1} (and maybe g^{x_0}), but not x_0 nor x_1. Shared secret $I_2 = g^{x_1 \mathsf{H}(I_1) \cdot x_2}$ is also secure by StDH if we model H as a random oracle. First, observe that in order to compute I_2, the adversary needs to query the random oracle on I_1 (for otherwise, I_2 is information-theoretically hidden). Now, since a reduction given g^{x_0} and g^{x_1} can give the adversary the same, then when the adversary queries the random oracle on I_1, the reduction can query the DDH oracle on (g^{x_0}, g^{x_1}, I_1) and submit $I_1 = g^{x_0 x_1}$ to its challenger. Further observe that in order to compute any I_t, the adversary needs to query the random oracle on I_1 (for otherwise, all of $I_2, \ldots, I_t$, where t is of course polynomial, are information-theoretically hidden). Thus, we can use the same reduction as above for every t.

Furthermore, even if P_1 is corrupted after she receives P_2's epoch $t^* - 1$ message, for some t^*, then we can still guarantee security of I_{t^*-1} from StDH, if we model H as a random oracle (of course, all preceding epochs are secure too, from the above). When leaking $\gamma_{t^*-1}^{\mathsf{P}_1}$, the reduction can simply sample random r and give g^r, in place of $g^{x_{t^*-1} \mathsf{H}(I_{t^*-1})}$. If the adversary never queries the random oracle on $\mathsf{H}(I_{t^*-1})$, then of course this is a perfect simulator. Now, as above, in order to compute I_{t^*-1}, the adversary still needs to query the random oracle on I_1, so our same reduction will still go through. Moreover, if in this situation P_1 uses good randomness for her epoch t^* message T_{t^*}, then I_{t^*} will be secure too. We just showed that for the adversary to query the random oracle on I_{t^*-1}, meaning she has already computed I_{t^*-1}, the adversary needs to still query I_1, which by our original reduction, we know it cannot. Given this information, a StDH reduction given g^r and

$g^{x_t^*}$ can simply give $\gamma_{t^*-1}^{\mathsf{P}_1} = g^r$ instead of $g^{x_{t^*-1} \cdot \mathsf{H}(I_{t^*-1})}$ for the leak on P_1 before epoch t^*, then send $T_{t^*} = g^{x_t^*}$. Then, when the adversary submits I_{t^*}, the reduction can simply forward it to its challenger.

Using similar arguments as above, if both parties use bad randomness for all subsequent epochs, and there are no more corruptions, then also all subsequent shared secrets I_t (and therefore all of them) will remain secure. Intuitively, this is because I_{t^*-1} is secure, so $\gamma_{t^*-1}^{\mathsf{P}_2}$ is as well, and P_1 used good randomness in epoch t^*, thus of course γ_{t^*} is secure too, so we can use the above argument thereafter.

5 Composition

In this section, we show how to compose the building blocks of Section 4 to construct Secure Messaging protocols DR and TR that UC-realize our two ideal functionalities of Figure 3. This section again often closely follows the work of [ACD19].

Initial Key Exchange Ideal Functionality. Before constructing the DR and TR we must introduce an ideal functionality for an initial key exchange to be used upon initialization of a session of one of our protocols. While the actual DR protocol uses the X3DH key exchange [MP16b], the focus of our work is to analyze the security and functionality of the double ratchet algorithm, and not X3DH. Therefore, we choose to present the following simple ideal functionality $\mathcal{F}_{\mathsf{KE}}^{\mathsf{CKA}}$ for key exchange that may be stronger than what X3DH offers, but nonetheless suffices for our purposes. The functionality is parameterized by a CKA protocol CKA.

On input $(\mathsf{sid}, \texttt{SETUP})$ **from** P where $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$: Send $(\mathsf{sid}, \texttt{SETUP}, \mathsf{P})$ to $\mathcal{S}$. When $\mathcal{S}$ returns $(\mathsf{sid}, \texttt{SETUP})$ then (i) sample $(\sigma_{\mathsf{root}}, k) \xleftarrow{\$} \{0,1\}^{2\lambda}$, (ii) execute $\gamma^{\mathsf{P}_1} \leftarrow \mathsf{CKA\text{-}Init\text{-}P}_1(k), \gamma^{\mathsf{P}_2} \leftarrow \mathsf{CKA\text{-}Init\text{-}P}_2(k)$, (iii) set $k^{\mathsf{P}_1} \leftarrow (\sigma_{\mathsf{root}}, \gamma^{\mathsf{P}_1}), k^{\mathsf{P}_2} \leftarrow (\sigma_{\mathsf{root}}, \gamma^{\mathsf{P}_2})$, and (iv) send $(\mathsf{sid}, \texttt{EXCHANGE}, k^{\mathsf{P}_1})$ to P and $(\mathsf{sid}, \texttt{EXCHANGE}, k^{\mathsf{P}_2})$ to $\bar{\mathsf{P}}$.

5.1 Constructions

We now provide constructions of the DR and our modified TR in the $\mathcal{F}_{\mathsf{KE}}^{\mathsf{CKA}}$-hybrid model. As in the analysis of Alwen *et al.*, our presentation of the DR differs from the actual DR protocol in a few minor aspects (see [ACD19, §5.2]), but the two are logically equivalent (and almost precisely the same otherwise). Indeed, the claim that the true DR protocol UC-realizes $\mathcal{F}_{\mathsf{DR}}$ in the $\mathcal{F}_{\mathsf{KE}}^{\mathsf{CKA}}$-hybrid model follows from Theorem 4. Importantly, we consider the version of the true DR protocol in which to increase security, parties defer new CKA secret key generation for their next sending epoch until they actually start that epoch, as opposed to when they receive the first message of the prior epoch [MP16a, §6.5]. Recall: we show in Appendix A.2.3 that while [ACD19] do the same, in their model it does not actually provide the DR any extra security.

As explained in the introduction, the main idea of the two schemes is that the two parties P_1 and P_2 keep track of the same root KDF Chain key σ_{root} and refresh it using the secrets of a CKA protocol that is run "in parallel". The corresponding outputs of the root KDF Chain are used to generate initialization keys for FS-AEAD instances. The only difference between the DR and TR is that the former uses a (t, ε)-secure CKA protocol while the latter uses a $(t, \varepsilon, +)$-secure CKA protocol.

DR and TR

$\mathsf{Init\text{-}P_1}\,(k^{\mathsf{P}_1})$
- $(\sigma_{\mathsf{root}}, \gamma) \leftarrow k^{\mathsf{P}_1}$
- $v[\cdot] \leftarrow \bot$
- $T_{\mathsf{cur}} \leftarrow \bot$
- $\ell_{\mathsf{prv}} \leftarrow 0$
- $t_{\mathsf{P}_1} \leftarrow 0$

$\mathsf{Send\text{-}P_1}\,(m)$
- **if** t_{P_1} *is even*
  - t_{P_1} ++
  - $(\gamma, T_{\mathsf{cur}}, I) \xleftarrow{\$} \mathsf{CKA\text{-}S}(\gamma)$
  - $(\sigma_{\mathsf{root}}, k) \leftarrow \mathsf{H}(\sigma_{\mathsf{root}}, I)$
  - $v[t_{\mathsf{P}_1}] \leftarrow \mathsf{FS\text{-}Init\text{-}S}(k)$
- $h \leftarrow (t_{\mathsf{P}_1}, T_{\mathsf{cur}}, \ell_{\mathsf{prv}})$
- $(v[t_{\mathsf{P}_1}], e) \leftarrow \mathsf{FS\text{-}Send}(v[t_{\mathsf{P}_1}], h, m)$
- **return** (h, e)

$\mathsf{Rcv\text{-}P_1}\,(c)$
- $(h, e) \leftarrow c$
- $(t, T, \ell) \leftarrow h$
- **req** t even and $t \leq t_{\mathsf{P}_1} + 1$
- **if** $t = t_{\mathsf{P}_1} + 1$
  - $\ell_{\mathsf{prv}} \leftarrow \mathsf{FS\text{-}Stop}(v[t_{\mathsf{P}_1}])$
  - t_{P_1} ++
  - $\mathsf{FS\text{-}Max}(v[t-2], \ell)$
  - $(\gamma, I) \leftarrow \mathsf{CKA\text{-}R}(\gamma, T)$
  - $(\sigma_{\mathsf{root}}, k) \leftarrow \mathsf{H}(\sigma_{\mathsf{root}}, I)$
  - $v[t] \leftarrow \mathsf{FS\text{-}Init\text{-}R}(k)$
- $(v[t], i, m) \leftarrow \mathsf{FS\text{-}Rcv}(v[t], h, e)$
- **if** $m = \bot$
  - **error**
- **return** (t, i, m)

Figure 7: Secure-messaging schemes DR and TR in the $\mathcal{F}_{\mathsf{KE}}^{\mathsf{CKA}}$-hybrid model based on (i) an FS-AEAD scheme FS-AEAD, (ii) (t, ε)- and $(t, \varepsilon, +)$-secure CKA schemes, respectively, and (iii) a KDF Chain using HKDF H. We assume w.l.o.g. that P_1 initializes the session. The initialization keys k^{P_1} and k^{P_2} are provided by a session of $\mathcal{F}_{\mathsf{KE}}^{\mathsf{CKA}}$, which is also initialized by P_1. The figure only shows the algorithms for P_1; P_2's algorithms are analogous, with "even" replaced by "odd".

State. In the DR and TR, party P_1 (resp. P_2) keeps an internal state s_{P_1} (resp. s_{P_2}), which is initialized by $\mathsf{Init\text{-}P_1}$ (resp. $\mathsf{Init\text{-}P_2}$) and used as well as updated by Send and Rcv. The state s_{P_1} of SM consists of the following values:

- The current key σ_{root} of the root KDF chain,
- States $v[0], v[1], v[2], \ldots$ of the various FS-AEAD instances,
- The state γ of the corresponding CKA scheme,
- The current CKA message T_{cur},
- The number of messages sent in the last completed sending epoch of P_1 ℓ_{prv}, and
- An epoch counter t_{P_1}.

We may refer to some such components of the state of P_1 throughout (e.g., in the proof of Theorem 4 in Section 6) using "dot-notation". For example, to refer to the epoch counter of party P_1, we will write "$s_{\mathsf{P}_1}.t_{\mathsf{P}_1}$". Recall (cf. Section 4.2) that once the maximum number of messages has been received for an epoch according to FS-Max, a session "erases" itself from the memory, and similarly when FS-Stop is called on a particular FS-AEAD session, it is erased. For simplicity, removing the corresponding $v[t]$ from memory is not made explicit in either case. The state s_{P_2} is defined analogously.

The algorithms. The algorithms of schemes DR and TR are depicted in Figure 7 and described in more detail below. For ease of description, the algorithms Send and Rcv are presented as Send-P_1 and Rcv-P_1, which handle the case where the algorithm is invoked by P_1; the case where the algorithm is invoked by P_2 works analogously. Moreover, to improve readability, the state s_{P_1} is not made explicit in the description: it consists of the variables set by the initialization algorithm. We also assume w.l.o.g. that P_1 initializes the session.

- **Initialization:** In the initialization procedure Init-P_1, P_1 initializes a session of $\mathcal{F}_{\mathsf{KE}}^{\mathsf{CKA}}$ to obtain initialization key $k^{\mathsf{P}_1} = (\sigma_{\mathsf{root}}, \gamma^{\mathsf{P}_1})$. It consists of the initial root KDF Chain key σ_{root} and the initial CKA state of P_1, γ^{P_1}. Furthermore, Init-P_1 also sets the initial epoch $t_{\mathsf{P}_1} \leftarrow 0$, $\ell_{\mathsf{prv}} \leftarrow 0$, and T_{cur} to a default value.

As pointed out above, in the DR and TR, P_1 and P_2 run a CKA protocol in parallel to sending their messages; the DR uses a (t, ε)-secure CKA protocol while the TR uses a $(t, \varepsilon, +)$-secure CKA protocol. To that end, P_1's first message includes the first message T_1 output by CKA-S. All subsequent messages sent by P_1 include T_1 until some message received from P_2 includes T_2. At that point P_1 would run CKA-S again and include T_3 with all her messages, and so on (cf. Section 4.3).

Upon either sending or receiving T_i for odd or even i, respectively, the CKA protocol also produces a random value I_i, which P_1 absorbs into the root KDF Chain. The resulting output k is used as key for a new FS-AEAD epoch.

- **Sending messages:** Procedure Send-P_1 allows P_1 to send a message to P_2. As a first step, Send-P_1 determines whether it is P_1's turn to send the next CKA message, which is the case if t_{P_1} is even. Whenever it is P_1's turn, Send-P_1 runs CKA-S to produce her next CKA message T and key I, which is absorbed into the root KDF Chain. The resulting value k is used as a the key for a new FS-AEAD epoch, in which P_1 acts as sender.

  Irrespective of whether it was necessary to generate a new CKA message and generate a new FS-AEAD epoch, Send-P_1 creates a header $h = (t_{\mathsf{P}_1}, T_{\mathsf{cur}}, \ell_{\mathsf{prv}})$ (see below for how ℓ_{prv} is computed in Rcv), and uses the current epoch $v[t_{\mathsf{P}_1}]$ to compute a ciphertext for (h, m) (where h is treated as associated data).

- **Receiving messages:** When a ciphertext $c = (h, e, \ell)$ with header $h = (t, T, \ell)$ is processed by Rcv-P_1, there are two possibilities:

  - $t \leq t_{\mathsf{P}_1}$ (and t even): In this case, ciphertext c pertains to an existing FS-AEAD epoch, in which case FS-Rcv is simply called on $v[t]$ to process e. If the maximum number of messages has been received for session $v[t]$, the session is removed from memory.
  - $t = t_{\mathsf{P}_1} + 1$ and t_{P_1} odd: Here, the receiver algorithm advances t_{P_1} by incrementing it and processes T with CKA-R. This produces a key I, which is absorbed into the root KDF chain to obtain a key k with which to initialize a new epoch $v[t_{\mathsf{P}_1}]$ as receiver. Then, e is processed by FS-Rcv on $v[t_{\mathsf{P}_1}]$. Note that Rcv also uses FS-Max to store ℓ as the maximum number of messages in the previous receive epoch. It also terminates its most recent sending epoch by calling FS-Stop and stores the number of messages in the old epoch in ℓ_{prv}, which will be sent along inside the header for every message of the next sending epoch.

Irrespective of whether a new CKA message was received and a new epoch created, if e is rejected by FS-Rcv, the algorithm raises an exception (**error**), which causes the entire state s_{P_1} to be rolled back to what it was before Rcv-P_1 was called.

5.2 Vulnerability of the DR with Respect to $\mathcal{F}_{\mathsf{TR}}$

Recall that the only difference between the DR and TR is that the former uses a (t, ε)-secure CKA protocol while the latter uses a $(t, \varepsilon, +)$-secure CKA protocol. As we will show in the next section, the DR UC-realizes functionality $\mathcal{F}_{\mathsf{DR}}$, whereas TR UC-realizes stronger functionality $\mathcal{F}_{\mathsf{TR}}$.

Here, we will explicitly show the vulnerability of the DR with CKA protocol CKA which prevents it from realizing the stronger $\mathcal{F}_{\mathsf{TR}}$ functionality. Briefly recall from Section 4.3.2 how CKA protocol CKA works: When P_1 sends a new CKA message, she generates random exponent x then sends message $T_1 \leftarrow g^x$ and computes shared secret $I_1 \leftarrow T_0^x$, where T_0 is the message P_2 sent in the previous epoch. Then, when P_2 sends the next message, she generates random exponent y and sends message $T_2 \leftarrow g^y$ so that the shared secret for that round is $I_2 \leftarrow T_1^y = g^{xy}$. Thus, P_1 needs to save exponent x after sending T_1 so that when she receives T_2, she can compute $I_2 = T_2^x$. Therefore, if she is corrupted after she sends T_1, but before she receives T_2, then the adversary obtains x, rendering I_1 insecure since the adversary itself can compute $I_1 \leftarrow T_0^x$.

The insecurity of I_1 in the above scenario is at the root of the vulnerability in the DR. If P_1 starts a new sending epoch, then she computes CKA message T_1 and secret I_1 as above, then FS-AEAD initialization key k for the epoch as $(\cdot, k) \leftarrow \mathsf{H}(\sigma_{\mathsf{root}}, I_1)$, using the current root KDF chain key σ_{root} that she has in her state. She may then proceed to use the FS-AEAD initialized with key k to encrypt and send several messages $m_1, \ldots, m_\ell$. As highlighted above, before P_1 receives a message for P_2's next sending epoch, she needs to keep exponent x that she used to generate CKA message T_1 in her state, so that she can compute the CKA shared secret I_2 that P_2 uses to compute the FS-AEAD initialization key for her next epoch. Therefore, consider the following two state leakages of P_1: (i) before she starts her new epoch and (ii) before she receives a new message for P_2's next epoch. The first leakage will indeed give the adversary σ_{root} and the second leakage will give the adversary x so that it can first compute I_1 as above, then FS-AEAD initialization key for that epoch as $(\cdot, k) \leftarrow \mathsf{H}(\sigma_{\mathsf{root}}, I_1)$, and finally decrypt all of the messages P_1 has sent in her epoch.

Notice that $\mathcal{F}_{\mathsf{DR}}$ will indeed provide the ideal adversary with these *vulnerable* messages upon the second leakage (the dashed part of Figure 3), while $\mathcal{F}_{\mathsf{TR}}$ will not. Thus, while the DR can UC-realize $\mathcal{F}_{\mathsf{DR}}$, it cannot UC-realize $\mathcal{F}_{\mathsf{TR}}$. TR's use of CKA^+ allows it to UC-realize stronger functionality $\mathcal{F}_{\mathsf{TR}}$, because even with both leakages on P_1 described above, the shared secret I_1 remains secure, as long as good randomness is used in the generation of T_0 and T_1 (c.f. Section 4.3.4).

We emphasize that while our improved TR protocol does UC-realize $\mathcal{F}_{\mathsf{TR}}$ (without any additional communication and only small additional computation to that of the DR), the DR itself should already, as understood by the Double Ratchet whitepaper [MP16a]: As a result of PCS, the messages in the new epoch which P_1 starts should be secure, despite *only* the leakage before she starts the epoch. Moreover, as a result of FS, the messages which P_1 already sent in her new epoch should remain secure despite *only* the second leakage of P_1. In reality, we however show that PCS and FS of the DR with respect to these two leakages no longer hold when *both* of them occur.

6 UC Security of the DR and TR

In this section, we show that the DR and TR UC-realize $\mathcal{F}_{\mathsf{DR}}$ and $\mathcal{F}_{\mathsf{TR}}$, respectively, in the $\mathcal{F}_{\mathsf{KE}}^{\mathsf{CKA}}$-hybrid model.

$\mathcal{S}_{t^*}$

Notation: The simulator algorithm interacts with the functionality $\mathcal{F}_{\mathsf{DR}}$ and the hybrid algorithm H_{t^*}. The algorithm initializes lists of *in-transit* ciphertexts $\mathsf{P}_1.\mathtt{T}$, and *vulnerable ciphertexts* $\mathsf{P}_1.\mathtt{V}$ sent by P_1 to P_2 to ϕ. Analogously, lists $\mathsf{P}_2.\mathtt{T}$ and $\mathsf{P}_2.\mathtt{V}$ are also initialized to ϕ. The algorithm also initializes leakage flags of both P_1 and P_2 for their corresponding (i) public ratchet secrets: $\mathsf{P}_1.\mathtt{PLEK}, \mathsf{P}_2.\mathtt{PLEK}$, (ii) current sending epoch symmetric secrets: $\mathsf{P}_1.\mathtt{CUR_SLEK}, \mathsf{P}_2.\mathtt{CUR_SLEK}$, and (iii) previous sending epoch symmetric secrets: $\mathsf{P}_1.\mathtt{PREV_SLEK}, \mathsf{P}_2.\mathtt{PREV_SLEK}$, all to 0. Further, it initializes bad-randomness flags $\mathsf{P}_1.\mathtt{BAD}, \mathsf{P}_2.\mathtt{BAD}$ to 0. Finally, it initializes the turn flag $\mathtt{TURN}$ as $\perp$. $\mathcal{S}_{t^*}$ also keeps track of $\mathcal{S}_{t^*}.k_t$; the FS-AEAD initialization key for each epoch t once it is generated by the corresponding sender of that epoch.

- **On input** $(\mathsf{sid}, \mathtt{SETUP}, \mathsf{P})$ **from** H_{t^*} where $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$: Sample initialization keys $k^{\mathsf{P}}, k^{\bar{\mathsf{P}}}$ according to $\mathcal{F}_{\mathsf{KE}}^{\mathsf{CKA}}$ and once $\mathcal{A}$ approves the interaction (i) run $s_{\mathsf{P}} \leftarrow \mathsf{Init\text{-}P}_1(k^{\mathsf{P}}), s_{\bar{\mathsf{P}}} \leftarrow \mathsf{Init\text{-}P}_2(k^{\bar{\mathsf{P}}})$, (ii) set $\mathtt{TURN} \leftarrow \mathsf{P}$, and (iii) return $(s_{\mathsf{P}}, s_{\bar{\mathsf{P}}})$.
- **On input** $(\mathsf{sid}, \mathsf{mid}, \mathtt{IN_TRANSIT}, \mathsf{P}, |m|, m')$ **from** H_{t^*} where $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$:
  1. Set $r \leftarrow \perp$.
  2. If $\mathtt{New}(\mathsf{P}, \mathtt{TURN}, \mathsf{P}.\mathtt{T})$[a] then:
     (a) Set (i) $\beta \leftarrow \bar{\mathsf{P}}.\mathtt{CUR_SLEK}$, (ii) $\mathsf{P}.\mathtt{PLEK} \leftarrow \mathsf{P}.\mathtt{BAD}$, and (iii) $\mathsf{P}.\mathtt{CUR_SLEK} \leftarrow \bar{\mathsf{P}}.\mathtt{CUR_SLEK} \wedge (\mathsf{P}.\mathtt{PLEK} \vee \bar{\mathsf{P}}.\mathtt{PLEK})$.
     (b) If $\mathsf{P}.\mathtt{BAD} = 0$ then sample random $r \xleftarrow{\$} \mathcal{R}$; otherwise ask $\mathcal{A}$ for random r' and set $r \leftarrow r'$.
  3. If $m' = \perp$ then set $m \leftarrow 0^{|m|}$; otherwise, set $m \leftarrow m'$.
  4. Run $(s_{\mathsf{P}}, c) \leftarrow \mathsf{Send}(s_{\mathsf{P}}, m; r)$, add $(\mathsf{sid}, \mathsf{mid}, s_{\mathsf{P}}.t, s_{\mathsf{P}}.v[s_{\mathsf{P}}.t].i, \mathtt{IN_TRANSIT}, c, \mathtt{TURN})$ to $\mathsf{P}.\mathtt{T}$, and if $\beta \vee (\mathsf{P}.\mathtt{V} \neq \emptyset)$ then add $(\mathsf{sid}, \mathsf{mid}, s_{\mathsf{P}}.v[s_{\mathsf{P}}.t].i, c)$ to $\mathsf{P}.\mathtt{V}$. Finally return (s_{P}, c).
- **On input** $(\mathsf{sid}, \mathtt{DELIVER}, \mathsf{P}, c)$ **from** H_{t^*}:
  1. Set $t_{\mathsf{prev}} \leftarrow s_{\bar{\mathsf{P}}}.t$ and run $(s_{\bar{\mathsf{P}}}, t, i, m) \leftarrow \mathsf{Rcv}(s_{\bar{\mathsf{P}}}, c)$. If $m = \perp$ then return $\perp$.
  2. If $t = t_{\mathsf{prev}} + 1$ then set (i) $\mathtt{TURN} = \bar{\mathsf{P}}$, (ii) $\mathsf{P}.\mathtt{T} \leftarrow \mathtt{Flip}(\mathsf{P}, \mathsf{P}.\mathtt{T})$,[b] (iii) $\mathsf{P}.\mathtt{PREV_SLEK} \leftarrow 0$, (iv) $\bar{\mathsf{P}}.\mathtt{PREV_SLEK} \leftarrow \bar{\mathsf{P}}.\mathtt{CUR_SLEK}$, (v) $\bar{\mathsf{P}}.\mathtt{CUR_SLEK} \leftarrow 0$, (vi) $\bar{\mathsf{P}}.\mathtt{PLEK} \leftarrow 0$, and (vii) $\bar{\mathsf{P}}.\mathtt{V} \leftarrow \emptyset$.
  3. Find $(\mathsf{sid}, \mathsf{mid}, t, i, \mathtt{IN_TRANSIT}, c, \gamma)$ in $\mathsf{P}.\mathtt{T}$, remove it from $\mathsf{P}.\mathtt{T}$, and return $(s_{\mathsf{P}}, \mathsf{sid}, \mathsf{mid}, \mathtt{DELIVER}, \mathsf{P}, m, 0, \perp, \perp)$. If no such entry is found:
     (a) If $t = t_{\mathsf{prev}} + 1$ then:
        i. If $\nexists(\mathsf{sid}, \cdot, t, \cdot, \mathtt{IN_TRANSIT}, \cdot, \mathsf{P}) \in \mathsf{P}.\mathtt{T}$ or for $(\mathsf{sid}, \cdot, t, \cdot, \mathtt{IN_TRANSIT}, c', \mathsf{P}) \in \mathsf{P}.\mathtt{T}$,$(h', e') \leftarrow c'$, $(t', T', \ell') \leftarrow h'$, $T' \neq T$, return $(s_{\bar{\mathsf{P}}}, \mathsf{sid}, (t, i), \mathtt{INJECT}, \mathsf{P}, m, 1, \perp, \perp)$.
        ii. Otherwise, return $(s_{\bar{\mathsf{P}}}, \mathsf{sid}, (t, i), \mathtt{INJECT}, \mathsf{P}, m, 0, \mathsf{P}, \ell')$, where ℓ' is as above.
     (b) Otherwise return $(s_{\bar{\mathsf{P}}}, \mathsf{sid}, (t, i), \mathtt{INJECT}, \mathsf{P}, m, 0, \perp, \perp)$.
- **On input** $(\mathsf{sid}, \mathtt{LEAK}, \mathsf{P}, \alpha)$ **from** H_{t^*} where $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$:
  1. Forward the input to $\mathcal{F}_{\mathsf{DR}}$ and receive back $\mathsf{P}.\mathtt{V}'$ and $\bar{\mathsf{P}}.\mathtt{T}'$.
  2. For receiving epochs $t \in [1, t^*]$ of P, run $\mathsf{FS\text{-}Expl\text{-}In\text{-}Trans\text{-}Cts}(s_{\mathsf{P}}.v[t], \mathsf{trans})$.[c]
  3. If $\alpha = 1$ then run $\mathsf{FS\text{-}Expl\text{-}Vul\text{-}Cts}(\mathcal{S}_{t^*}.k_{s_{\mathsf{P}}.t}, \mathsf{vul})$.[d]
  4. If $\neg\mathtt{New}(\mathsf{P}, \mathtt{TURN}, \mathsf{P}.\mathtt{T})$ then set $\mathsf{P}.\mathtt{CUR_SLEK} \leftarrow 1$ and $\mathsf{P}.\mathtt{PLEK} \leftarrow 1$. If $\neg\mathtt{New}(\bar{\mathsf{P}}, \mathtt{TURN}, \bar{\mathsf{P}}.\mathtt{T})$ then set $\bar{\mathsf{P}}.\mathtt{CUR_SLEK} \leftarrow 1$. If $\mathtt{TURN} = \bar{\mathsf{P}}$ then set $\bar{\mathsf{P}}.\mathtt{PREV_SLEK} \leftarrow 1$.

- **On input** $(\mathsf{sid}, \texttt{BAD_RANDOMNESS}, \mathsf{P}, \rho)$ **from** H_{t^*} where $\rho \in \{0,1\}$ and $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$: Set $\mathsf{P}.\texttt{BAD} \leftarrow \rho$ and forward the input to $\mathcal{F}_{\mathsf{DR}}$.

[a] $\texttt{New}(\mathsf{P}, \texttt{TURN}, \mathsf{P}.\texttt{T})$ outputs 1 if we have $\texttt{TURN} = \mathsf{P}$ and for all $(\mathsf{sid}, \mathsf{mid}, t, i, \texttt{IN_TRANSIT}, c, \gamma) \in \mathsf{P}.\texttt{T}$ we have that $\gamma \neq \mathsf{P}$; otherwise output 0.

[b] $\texttt{Flip}(\mathsf{P}, \mathsf{P}.\texttt{T})$ for each $(\mathsf{sid}, \mathsf{mid}, t', i', \texttt{IN_TRANSIT}, c, \mathsf{P}) \in \mathsf{P}.\texttt{T}$ replaces it with $(\mathsf{sid}, \mathsf{mid}, t', i', \texttt{IN_TRANSIT}, m, \bot)$.

[c] $\mathsf{trans} = \{(i, h, m, e) : (\mathsf{sid}, \mathsf{mid}, t, i, \texttt{IN_TRANSIT}, (h, e), \gamma) \in \bar{\mathsf{P}}.\texttt{T} \text{ and } (\mathsf{sid}, \mathsf{mid}, \texttt{IN_TRANSIT}, m, \beta, \gamma) \in \bar{\mathsf{P}}.\texttt{T}'\}$. If $s_{\mathsf{P}}.v[t] = \bot$ but $\mathsf{trans} \neq \emptyset$ then replace $s_{\mathsf{P}}.v[t]$ with $\mathsf{FS\text{-}Init\text{-}R}(\mathcal{S}_{t^*}.k_t)$.

[d] $\mathsf{vul} = \{(i, h, m, e) : (\mathsf{sid}, \mathsf{mid}, i, (h, e)) \in \mathsf{P}.\texttt{V} \text{ and } (\mathsf{sid}, \mathsf{mid}, m) \in \mathsf{P}.\texttt{V}'\}$.

Figure 8: Simulator algorithm $\mathcal{S}_{t^*}$.

Theorem 4. *Assume that*

- CKA *is a* $\varepsilon_{\mathsf{CKA}}$*-secure* natural *CKA scheme,*
- CKA^+ *is a* $(\varepsilon_{\mathsf{CKA}}, +)$*-secure* natural *CKA scheme,*
- $\mathsf{FS\text{-}AEAD}$ *is a* $\varepsilon_{\mathsf{FS\text{-}AEAD}}$*-secure FS-AEAD scheme, and*
- H *is modelled as a (programmable) random oracle.*

Then protocols DR *and* TR *UC-realize functionalities* $\mathcal{F}_{\mathsf{DR}}$ *and* $\mathcal{F}_{\mathsf{TR}}$ *in the* $\mathcal{F}_{\mathsf{KE}}^{\mathsf{CKA}}$*-hybrid and* $\mathcal{F}_{\mathsf{KE}}^{\mathsf{CKA}^+}$*-hybrid models, respectively, with security loss* $\varepsilon = T \cdot (\varepsilon_{\mathsf{CKA}} + \varepsilon_{\mathsf{FS\text{-}AEAD}})$*, where* T *is the number of epochs the attacker initiates.*

Recall that in Section 4.1.1 we discuss why it seems necessary to model H as a programmable random oracle. In the following, we may simply refer to the ideal functionality as $\mathcal{F}_{\mathsf{DR}}$, but such statements will hold for $\mathcal{F}_{\mathsf{TR}}$ too.

6.1 Hybrid Algorithms H_{t^*} and the Simulator $\mathcal{S}$

To prove the theorem, we proceed in a series of hybrids $H_0, H_1, \ldots, H_T$, where T is the number of epochs which the adversary initiates. Hybrid H_0 is the real world protocol DR or TR which interacts with the real world adversary $\mathcal{A}$. Hybrid H_{t^*} runs stateful simulator algorithm $\mathcal{S}_{t^*}$ for (i) setup, (ii) generation of messages from epochs 1 through t^* using input from $\mathcal{F}_{\mathsf{DR}}$, (iii) delivery of messages from epochs 1 through t^*, and (iv) for leakages of parties, the explanation of in-transit messages for epochs 1 through t^* and vulnerable messages if the party is not in an epoch after t^*. For generation of messages from epochs after t^*, hybrid H_{t^*} uses the corresponding information written to the input tapes of P_1 and P_2 in the real world and executes as in the real world. Also for delivery of messages for epochs later than t^* and leakages of states at any point, hybrid H_{t^*} behaves as in the real world. Formally, hybrid H_{t^*} (for $t^* > 0$) is defined as follows:

- **On input** $(\mathsf{sid}, \texttt{SETUP}, \mathsf{P})$ **from** $\mathcal{F}_{\mathsf{DR}}$ where $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$: Forward the input to $\mathcal{S}_{t^*}$ and once $(s_{\mathsf{P}}, s_{\bar{\mathsf{P}}})$ is received back, send $(\mathsf{sid}, \texttt{SETUP})$ back to $\mathcal{F}_{\mathsf{DR}}$.
- **On input** $(\mathsf{sid}, \mathsf{mid}, \texttt{IN_TRANSIT}, \mathsf{P}, |m|, m')$ **from** $\mathcal{F}_{\mathsf{DR}}$ where $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$: Forward the input to $\mathcal{S}_{t^*}$, receive back (s_{P}, c), and send c to $\mathcal{A}$.
- **On input** $(\mathsf{sid}, \mathsf{mid}, \texttt{SEND}, m)$ **from** $\mathcal{Z}$ **to** $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$: Run $(s_{\mathsf{P}}, c) \leftarrow \mathsf{Send}(s_{\mathsf{P}}, m)$ (asking $\mathcal{A}$ for randomness if necessary) and send c to $\mathcal{A}$.

- **On input** $(\mathsf{sid}, \texttt{DELIVER}, \mathsf{P}, c)$ **from** $\mathcal{A}$:
  1. Parse $(h, e) \leftarrow c, (t, T, \ell) \leftarrow h$.
  2. If $t \leq t^*$ then forward the input to $\mathcal{S}_{t^*}$ and:
     (a) If $\mathcal{S}_{t^*}$ returns $\perp$ then skip.
     (b) Otherwise, parse the return as $(s'_{\bar{\mathsf{P}}}, \mathsf{sid}, \mathsf{mid}, \texttt{INSTRUCTION}, \mathsf{P}, m, \delta, \gamma, \ell')$.
     (c) If $s_{\bar{\mathsf{P}}}.t \leq t^*$, then set $s_{\bar{\mathsf{P}}} \leftarrow s'_{\bar{\mathsf{P}}}$.
     (d) If $s_{\bar{\mathsf{P}}}.t > t^*$ then set $s_{\bar{\mathsf{P}}}.v[t] \leftarrow s'_{\bar{\mathsf{P}}}.v[t]$.
     (e) If $\texttt{INSTRUCTION} = \texttt{DELIVER}$ then forward $(\mathsf{sid}, \mathsf{mid}, \texttt{DELIVER}, \mathsf{P}, m)$ to $\mathcal{F}_{\mathsf{DR}}$. Otherwise, forward $(\mathsf{sid}, \mathsf{mid}, \texttt{INJECT}, \mathsf{P}, m, \delta, \gamma)$ to $\mathcal{F}_{\mathsf{DR}}$ and if $\delta = 0$ then when $\mathcal{F}_{\mathsf{DR}}$ passes activation back to H_{t^*}, send $(\mathsf{sid}, \mathsf{mid}, \texttt{DELIVER}, \mathsf{P}, m)$ to $\mathcal{F}_{\mathsf{DR}}$. Also, if $\ell' \neq \perp$ then while $\ell' < \ell$: ℓ' ++ and send $(\mathsf{sid}, (t-2, \ell'), \texttt{INJECT}, \mathsf{P}, \perp, 0, \perp)$ to $\mathcal{F}_{\mathsf{DR}}$. If $\delta = 1$ then H_{t^*} will use $s_{\bar{\mathsf{P}}}$ to communicate directly with $\mathcal{A}$ on behalf of $\bar{\mathsf{P}}$: It will generate all messages from $\bar{\mathsf{P}}$ directly using the normal Send algorithm and receive all messages for $\bar{\mathsf{P}}$ directly using the normal Rcv algorithm (starting with this one).
  3. Otherwise, run $(s_{\bar{\mathsf{P}}}, t, i, m) \leftarrow \mathsf{Rcv}(s_{\bar{\mathsf{P}}}, c)$ and write $(\mathsf{sid}, (t, i), m)$ to the output tape of $\bar{P}$.
- **On input** $(\mathsf{sid}, \texttt{LEAK}, \mathsf{P})$ **from** $\mathcal{A}$ where $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$:
  1. If $s_{\mathsf{P}}.t \leq t^*$ then set $\alpha \leftarrow 1$; otherwise, set $\alpha \leftarrow 0$.
  2. Forward $(\mathsf{sid}, \texttt{LEAK}, \mathsf{P}, \alpha)$ to $\mathcal{S}_{t^*}$ and send s_{P} to $\mathcal{A}$.
- **On input** $(\mathsf{sid}, \texttt{BAD_RANDOMNESS}, \mathsf{P}, \rho)$ **from** $\mathcal{A}$ where $\rho \in \{0, 1\}$ and $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$: If $s_{\mathsf{P}}.t \leq t^*$ then forward the input to $\mathcal{S}_{t^*}$.

Observe that H_T is the simulator $\mathcal{S}$ for the ideal world (it simply uses $\mathcal{S}_T$ for everything).

Lemma 1. *If we make the same assumptions as in Theorem 4, then for $t^* \in [T]$, hybrids H_{t^*-1} and H_{t^*} are indistinguishable with security loss $\varepsilon = \varepsilon_{\mathsf{FS\text{-}AEAD}} + \varepsilon_{\mathsf{CKA}}$.*

To show H_{t^*-1} and H_{t^*} are indistinguishable, we first make the simplifying assumptions that (i) P_1 initializes the session and (ii) P_1 sends in epoch t^*. The other cases are handled analogously. Now, for those adversaries $\mathcal{A}$ that leak on parties P_1 and P_2 and/or give them bad randomness such that after P_1 sends the first message in epoch t^*, it will be that $\mathsf{P}_1.\texttt{CUR_SLEK} = 1$: it is clear that from the correctness of CKA and $\mathsf{FS\text{-}AEAD}$ that $H_{t^*-1} \equiv H_{t^*}$, so we are done. We omit an explicit reduction to the security game of $\mathsf{FS\text{-}AEAD}$ (which captures its correctness guarantees) for brevity. Otherwise, we divide the types of adversaries such that the above does not hold into the three types of the following subsections.

6.2 Type 1 Adversaries

Type 1 adversaries $\mathcal{A}$ are those adversaries such that:

1. Before P_2 has accepted any epoch t^* message, $\mathcal{A}$ successfully forges an epoch t^* message with CKA message T in the header such that either
   (a) P_1 has not sent any epoch t^* ciphertext yet, or

(b) For the CKA message T_{t^*} that is contained in the ciphertexts P_1 has sent in epoch t^*, $T_{t^*} \neq T$;

and $\mathcal{A}$ has not queried the random oracle on $(\sigma_{\mathsf{root}}^{t^*-1}, I)$, where I is the corresponding CKA secret that P_2 would output upon receiving message with T; or

2. Before P_1 has accepted any epoch t^*+1 message, $\mathcal{A}$ successfully forges an epoch t^*+1 message with CKA message T in the header such that either

   (a) P_2 has not sent any epoch t^*+1 ciphertext yet, or

   (b) For the CKA message T_{t^*+1} that is contained in the ciphertexts P_2 has sent in epoch t^*+1, $T_{t^*+1} \neq T$;

   and $\mathcal{A}$ has not queried the random oracle on $(\sigma_{\mathsf{root}}^{t^*}, I)$, where I is the corresponding CKA secret that P_1 would output upon receiving message with T.

Lemma 2. *If we make the same assumptions as in Theorem 4, then Type* 1 *adversaries only succeed with probability* $\varepsilon_{\mathsf{FS\text{-}AEAD}}$.

Proof. We provide a reduction to the security of the underlying FS-AEAD scheme FS-AEAD to show that successful Type 1 adversaries only exist with negligible probability. Specifically, assuming that there is some successful Type 1 adversary $\mathcal{A}$, we construct reduction algorithm $\mathcal{B}_{\mathsf{FS\text{-}AEAD},1}$ that has non-negligible probability against FS-AEAD in the FS-AEAD security game, thus reaching a contradiction. $\mathcal{B}_{\mathsf{FS\text{-}AEAD},1}$ simulates hybrid H_{t^*-1} or H_{t^*} for $\mathcal{A}$, using the FS-AEAD security game oracle **init** to initialize the FS-AEAD state of P_2 or P_1 for injection in epoch t^* or t^*+1, and oracle **inject-P$_2$** for the corresponding injections.

More formally, $\mathcal{B}_{\mathsf{FS\text{-}AEAD},1}$ first samples random $b \xleftarrow{\$} \{0,1\}$ and $\mathsf{inj\text{-}guess} \xleftarrow{\$} \{\mathsf{P}_1, \mathsf{P}_2\}$ corresponding to the party to whom $\mathcal{A}$ will successfully inject a message. It then proceeds as in H_{t^*-1} with the following exceptions:

- **On input** $(\mathsf{sid}, \mathsf{mid}, \texttt{SEND}, m)$ **from** $\mathcal{Z}$ **to** $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$: If (i) $s_{\mathsf{P}}.t = t^*$ before $(s_{\mathsf{P}_1}.v[t^*], e) \leftarrow \mathsf{FS\text{-}Send}(s_{\mathsf{P}}.v[t], h, m)$ would normally be executed within Send, (ii) $\mathsf{P}_1.\texttt{CUR_SLEK} = 0$, and (iii) $b = 1$; then replace m with $m \leftarrow 0^{|m|}$. Otherwise, proceed as in H_{t^*-1}.
- **On input** $(\texttt{DELIVER}, \mathsf{P}, c)$ **from** $\mathcal{A}$:
  1. Parse $(h, e) \leftarrow c$, $(t, T, \ell) \leftarrow h$.
  2. If $t = t^* \wedge T \neq T_{t^*} \wedge \mathsf{inj\text{-}guess} = \mathsf{P}_2$ then query **inject-P$_2$**(h, e). If there are no more queries of this form then send random $b' \xleftarrow{\$} \{0,1\}$ to the challenger.
  3. If $t = t^*+1 \wedge T \neq T_{t^*+1} \wedge \mathsf{inj\text{-}guess} = \mathsf{P}_1$ then query **init** followed by **inject-P$_2$**(h, e).
  4. Otherwise proceed as in H_{t^*-1+b}.

Now, before epoch t^*, H_{t^*-1} and H_{t^*} do not diverge and thus we simulate them perfectly. In epoch t^*, before any additional leakage, if $b = 0$ we encrypt the real message m and thus simulate H_{t^*-1} perfectly; if $b = 1$ we encrypt the all 0 message and thus simulate H_{t^*} perfectly. Since $\mathcal{A}$ does not query the random oracle on $(\sigma_{\mathsf{root}}^{t^*-1}, I)$, for I corresponding to T, the corresponding FS-AEAD

intialization key k used in both H_{t^*-1} and H_{t^*} is uniformly random and unknown to $\mathcal{A}$ and thus the key sampled by the FS-AEAD challenger is distributed correctly. So, when $\mathcal{A}$ eventually submits a successful forgery, $\mathcal{B}_{\mathsf{FS\text{-}AEAD},1}$ will pass it to the challenger and the challenger will declare that $\mathcal{B}_{\mathsf{FS\text{-}AEAD},1}$ has won the game, a contradiction.

If $\mathcal{A}$ waits until epoch t^*+1 to submit a successful forgery then to deliver any well-formed epoch t^* message (either honest or an injection for which $\mathcal{A}$ has queried the random oracle on the corresponding (σ, I)), $\mathcal{B}_{\mathsf{FS\text{-}AEAD},1}$ acts as in H_{t^*-1} or H_{t^*} according to the sampled bit b. Finally, it will act as above for epoch t^*+1 attempted forgeries and pass the successful forgery to the challenger, a contradiction. □

6.3 Type 2 Adversaries

Type 2 adversaries $\mathcal{A}$ are those that are not Type 1 and query the random oracle on $(\sigma_{\mathsf{root}}^{t^*-1}, I_{t^*})$ without beforehand:

1. Any leak on P_2 after P_1 sends the first message in epoch t^*, but before P_2 receives any message that causes it to advance to epoch t^*; or
2. Any leak on P_1 after P_1 sends the first message in epoch t^*, but before P_1 receives any message that causes it to advance to epoch t^*+1.

Lemma 3. *If we make the same assumptions as in Theorem 4, then Type 2 adversaries only succeed with probability* $\varepsilon_{\mathsf{CKA}}$.

Proof. We provide a reduction to the security of the underlying CKA scheme CKA to show that successful Type 2 adversaries only exist with negligible probability. Specifically, assuming that there is some successful Type 2 adversary $\mathcal{A}$, we construct reduction algorithm $\mathcal{B}_{\mathsf{CKA}}$ that has non-negligible probability against CKA in the corresponding CKA security game, thus reaching a contradiction. $\mathcal{B}_{\mathsf{CKA}}$ simulates hybrid H_{t^*-1} or H_{t^*} for $\mathcal{A}$, using the CKA security game oracle **init** to initialize the CKA states of P_1 and P_2, oracles $\mathbf{corr\text{-}P_1}, \mathbf{corr\text{-}P_2}$ to handle leakages of CKA states, oracle $\mathbf{receive\text{-}P_1}, \mathbf{receive\text{-}P_2}$ to handle CKA message reception, and oracles $\mathbf{send\text{-}P_1}, \mathbf{send\text{-}P_2}$ and $\mathbf{send\text{-}P_1'}, \mathbf{send\text{-}P_2'}$ to handle CKA secret and message generation, except for in epoch t^*, in which $\mathbf{chall\text{-}P_1}$ is used instead.

More formally, $\mathcal{B}_{\mathsf{CKA}}$ first samples random $b \xleftarrow{\$} \{0,1\}$ and sets $\mathsf{RODict}[\cdot] \leftarrow \bot$, then proceeds as in H_{t^*-1}, with the following exceptions:

- **On input** $(\mathsf{sid}, \mathtt{SETUP}, \mathsf{P})$ **from** $\mathcal{F}_{\mathsf{DR}}$: Replace the executions of $\mathsf{CKA\text{-}Init\text{-}P_1}$ and $\mathsf{CKA\text{-}Init\text{-}P_2}$ within the executions of $\mathsf{Init\text{-}P_1}$ and $\mathsf{Init\text{-}P_2}$, respectively, in $\mathcal{S}_{t^*}$ with (i) an oracle query to $\mathbf{init}(t^*)$, (ii) set $\gamma_{\mathsf{P}} \leftarrow \mathbf{corr\text{-}P_1}$, and (iii) if $t^* \neq 1$ then set $\gamma_{\bar{\mathsf{P}}} \leftarrow \mathbf{corr\text{-}P_2}$.
- **On input** $(\mathsf{sid}, \mathsf{mid}, \mathtt{IN_TRANSIT}, \mathsf{P}, |m|, m')$ **from** $\mathcal{F}_{\mathsf{DR}}$: Replace the execution of $\mathsf{CKA\text{-}S}(\gamma^{\mathsf{P}}; r)$ within the Send execution of $\mathcal{S}_{t^*}$ with the following:
  1. If $\mathsf{P.BAD} = 0$ then an oracle call to $(T_t, I_t) \leftarrow \mathbf{send\text{-}P}$.
  2. Else if $\mathsf{P.BAD} = 1 \wedge s_{\mathsf{P}}.t = t^* - 2$ (before sending) then abort.
  3. Else if $\mathsf{P.BAD} = 1$ then an oracle call to $(T_t, I_t) \leftarrow \mathbf{send\text{-}P'}(r')$, where r' is the randomness acquired from $\mathcal{A}$ in $\mathcal{S}_{t^*}$.

4. In both non-abort cases: (i) if $s_{\mathsf{P}}.t < t^* - 1$ after sending, set $\gamma_{\mathsf{P}} \leftarrow$ **corr-P** and (ii) use the output T_t, I_t of **send-P** or **send-P'** as in H_{t^*-1}.

- **On input** $(\mathsf{sid}, \mathsf{mid}, \mathtt{SEND}, m)$ **from** $\mathcal{Z}$ **to** $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$:

1. If $s_{\mathsf{P}}.t = t^* - 1$ (before sending) then if $\mathsf{P}.\mathtt{BAD} = 1$, abort; otherwise, execute Send normally with the following exceptions:
   (a) Instead of executing $(s_{\mathsf{P}}.\gamma, T_{t^*}, I) \leftarrow \mathsf{CKA\text{-}S}(s_{\mathsf{P}}.\gamma)$, query oracle $T_{t^*} \leftarrow$ **chall-P** and set $s_{\mathsf{P}}.\gamma \leftarrow$ **corr-P**.
   Note: if we are analyzing the DR, then **corr-P** will simply exit with no return.
   (b) Instead of executing $(s_{\mathsf{P}}.\sigma_{\mathsf{root}}, k) \leftarrow \mathsf{H}(s_{\mathsf{P}}.\sigma_{\mathsf{root}}, I)$, simply sample σ and k uniformly at random and set $s_{\mathsf{P}}.\sigma_{\mathsf{root}} \leftarrow \sigma$.
   (c) If $b = 1 \wedge \mathsf{P}_1.\mathtt{CUR_SLEK} = 0$ then set $m = 0^{|m|}$.
2. If $s_{\mathsf{P}}.t = t^*$ (before sending) and $b = 1 \wedge \mathsf{P}_1.\mathtt{CUR_SLEK} = 0$ then set $m = 0^{|m|}$.
3. Proceed as in H_{t^*-1}.

- **On input** $(\mathtt{DELIVER}, \mathsf{P}, c)$ **from** $\mathcal{A}$: Parse $(h, e) \leftarrow c$, $(t, T, \ell) \leftarrow h$ then:

1. If $t < t^*$ then execute $(\gamma'_{\bar{\mathsf{P}}}, I) \leftarrow \mathsf{CKA\text{-}R}(\gamma_{\bar{\mathsf{P}}}, T)$ as in H_{t^*-1} (inside $\mathcal{S}_{t^*}$) normally. If the state is not rolled back within $\mathcal{S}_{t^*}$ then additionally make an oracle call to **receive-$\bar{\mathbf{P}}$** and set $\gamma_{\bar{\mathsf{P}}} \leftarrow \gamma'_{\bar{\mathsf{P}}}$ (otherwise, $\gamma_{\bar{\mathsf{P}}}$ stays the same as before).
2. If $t = t^* \wedge T = T_{t^*}$ then run $\mathsf{Rcv}(s_{\bar{\mathsf{P}}}, c)$ except instead of executing $(s_{\bar{\mathsf{P}}}.\gamma, I) \leftarrow \mathsf{CKA\text{-}R}(s_{\bar{\mathsf{P}}}.\gamma, T)$ and $(s_{\bar{\mathsf{P}}}.\sigma_{\mathsf{root}}, k) \leftarrow \mathsf{H}(s_{\bar{\mathsf{P}}}.\sigma_{\mathsf{root}}, I_{t^*})$:
   (a) Query **receive-$\mathbf{P_2}$** (if $\mathcal{B}_{\mathsf{CKA}}$ has not yet for T_{t^*}),
   (b) Use k sampled by $\mathcal{B}_{\mathsf{CKA}}$ above for FS-AEAD initialization, and
   (c) If $s_{\bar{\mathsf{P}}}$ is not rolled back, then set $s_{\bar{\mathsf{P}}}.\gamma \leftarrow$ **corr-$\mathbf{P_2}$** and $s_{\bar{\mathsf{P}}}.\sigma_{\mathsf{root}} \leftarrow \sigma$, where σ is that which was sampled by $\mathcal{B}_{\mathsf{CKA}}$ above.
3. If $t = t^* \wedge T \neq T_{t^*}$ then run $\mathsf{Rcv}(s_{\bar{\mathsf{P}}}, c)$ except instead of executing $(s_{\bar{\mathsf{P}}}.\gamma, I) \leftarrow \mathsf{CKA\text{-}R}(s_{\bar{\mathsf{P}}}.\gamma, T)$ and $(s_{\bar{\mathsf{P}}}.\sigma_{\mathsf{root}}, k) \leftarrow \mathsf{H}(s_{\bar{\mathsf{P}}}.\sigma_{\mathsf{root}}, I_{t^*})$; for every $(s_{\bar{\mathsf{P}}}.\sigma_{\mathsf{root}}, I)$ such that $\mathsf{RODict}[(s_{\bar{\mathsf{P}}}.\sigma_{\mathsf{root}}, I)] \neq \perp$:
   (a) Query $\mathbf{test}(t^*, T, I)$.
   (b) If the challenger returns 1 then (i) parse $(\sigma, k) \leftarrow \mathsf{RODict}[(s_{\bar{\mathsf{P}}}.\sigma_{\mathsf{root}}, I)]$ (ii) set $s_{\bar{\mathsf{P}}}.\sigma_{\mathsf{root}} \leftarrow \sigma$, (iii) set $s_{\bar{\mathsf{P}}}.\gamma \leftarrow \mathsf{CKA\text{-}Der\text{-}R}(T, I)$ and (iv) use k for the rest of $\mathsf{Rcv}(s_{\bar{\mathsf{P}}}, c)$.
   If no **test** query returns 1 then skip.
4. If $t = t^* + 1 \wedge T = T_{t^*+1}$ then run $\mathsf{Rcv}(s_{\bar{\mathsf{P}}}, c)$ except instead of executing $(s_{\bar{\mathsf{P}}}.\gamma, I) \leftarrow \mathsf{CKA\text{-}R}(s_{\bar{\mathsf{P}}}.\gamma, T)$ and $(s_{\bar{\mathsf{P}}}.\sigma_{\mathsf{root}}, k) \leftarrow \mathsf{H}(s_{\bar{\mathsf{P}}}.\sigma_{\mathsf{root}}, I_{t^*+1})$:
   (a) Query **receive-$\mathbf{P_1}$** (if $\mathcal{B}_{\mathsf{CKA}}$ has not yet for T_{t^*+1}),
   (b) Use k that $\mathcal{B}_{\mathsf{CKA}}$ naturally computes when sending the first message of epoch $t^* + 1$ for P_2, and
   (c) If $s_{\bar{\mathsf{P}}}$ is not rolled back, then set $s_{\bar{\mathsf{P}}}.\gamma \leftarrow$ **corr-$\mathbf{P_2}$** and $s_{\bar{\mathsf{P}}}.\sigma_{\mathsf{root}} \leftarrow s_{\mathsf{P}}.\sigma_{\mathsf{root}}$.
5. If $t = t^* + 1 \wedge T \neq T_{t^*+1}$ then run $\mathsf{Rcv}(s_{\bar{\mathsf{P}}}, c)$ except instead of executing $(s_{\bar{\mathsf{P}}}.\gamma, I) \leftarrow \mathsf{CKA\text{-}R}(s_{\bar{\mathsf{P}}}.\gamma, T)$ and $(s_{\bar{\mathsf{P}}}.\sigma_{\mathsf{root}}, k) \leftarrow \mathsf{H}(s_{\bar{\mathsf{P}}}.\sigma_{\mathsf{root}}, I_{t^*+1})$; for every $(s_{\bar{\mathsf{P}}}.\sigma_{\mathsf{root}}, I)$ such that $\mathsf{RODict}[(s_{\bar{\mathsf{P}}}.\sigma_{\mathsf{root}}, I)] \neq \perp$:

(a) Query $\mathbf{test}(t^*+1, T, I)$.

(b) If the challenger returns 1 then (i) parse $(\sigma, k) \leftarrow \mathsf{RODict}[(s_{\bar{\mathsf{P}}}.\sigma_{\mathsf{root}}, I)]$ (ii) set $s_{\bar{\mathsf{P}}}.\sigma_{\mathsf{root}} \leftarrow \sigma$, (iii) set $s_{\bar{\mathsf{P}}}.\gamma \leftarrow \mathsf{CKA\text{-}Der\text{-}R}(T, I)$, and (iv) use k for the rest of $\mathsf{Rcv}(s_{\bar{\mathsf{P}}}, c)$.

If no **test** query returns 1 then skip.

6. Otherwise, proceed as in H_{t^*-1+b}

- **On input** $(\texttt{QUERY}, (\sigma, I))$ **from** $\mathcal{A}$: If $s_{\mathsf{P}_1}.t \geq t^*$ and $\mathbf{test}(t^*, T_{t^*}, I) = 1$ then send I to the challenger (and the game ends). Otherwise,

1. If $\mathsf{RODict}[(\sigma, I)] = \perp$ then sample random (σ', k) and set $\mathsf{RODict}[(\sigma, I)] \leftarrow (\sigma', k)$.
2. Return $\mathsf{RODict}[(\sigma, I)]$.

First note that since we model H as a random oracle, while $\mathcal{A}$ has not queried the random oracle on $(s_{\mathsf{P}}.\sigma_{\mathsf{root}}^{t^*-1}, I_{t^*})$, the output $(s_{\mathsf{P}}.\sigma_{\mathsf{root}}^{t^*}, k)$ is always uniformly random to $\mathcal{A}$. Moreover, if $\mathcal{A}$ does make that query, $\mathcal{B}_{\mathsf{CKA}}$ forwards I_{t^*} to the challenger and the game ends. Therefore, $\mathcal{B}_{\mathsf{CKA}}$ properly simulates both H_{t^*-1} and H_{t^*} in the view of $\mathcal{A}$ when it samples uniformly random $(s_{\mathsf{P}}.\sigma_{\mathsf{root}}^{t^*}, k)$ in epoch t^*. Before epoch t^*, we send messages just as in the two hybrids and are able to corrupt the sender right afterwards (except for after sending the first message of epoch t^*-1) in the CKA game to obtain the CKA state of the sender by definition. Thus, when the other party receives a message for the first time in an epoch, we are able to use the real state to see how they would act in both hybrids. And if the receiver's state is not rolled back, this must mean that the CKA message is the same as the honestly generated one for that epoch by the sender, and so we can query the **receive-$\bar{\mathbf{P}}$** oracle to advance their state in the CKA game (since, by definition of Type 2 adversaries, P_1 must make it to t^* without a takeover of either party).

Note that Type 2 adversaries that make $\mathcal{B}_{\mathsf{CKA}}$ abort in Step 2 of an $\texttt{IN_TRANSIT}$ instruction or Step 1 of a $\texttt{SEND}$ instruction only exist with negligible probability. This is because if $\mathsf{P}_2.\texttt{BAD} = 1$ before P_2 sends the first message of epoch t^*-1 or $\mathsf{P}_1.\texttt{BAD} = 1$ before P_1 sends the first message of epoch t^* then for $\mathsf{P}_1.\texttt{CUR_SLEK}$ to be 0 after P_1 sends, it must be that $\mathsf{P}_2.\texttt{CUR_SLEK} = 0$ beforehand. Therefore, it must also be the case that after P_2 sent the first message of epoch t^*-1, $\mathsf{P}_2.\texttt{CUR_SLEK} = 0$, and no leakages on either party occurred in the interim. So, we know from the proof of indistinguishability of H_{t^*-2} and H_{t^*-1} that $\mathcal{A}$ could not have queried the random oracle on $(\sigma_{\mathsf{root}}^{t^*-2}, I_{t^*-1})$ (since if it did then $\mathsf{P}_2.\texttt{CUR_SLEK}$ would not be 0), and thus $\sigma_{\mathsf{root}}^{t^*-1}$ is uniformly random and unknown to $\mathcal{A}$. Hence, the probability of a Type 2 adversary later querying the random oracle on $(\sigma_{\mathsf{root}}^{t^*-1}, I_{t^*})$ is negligible. If $\mathcal{B}_{\mathsf{CKA}}$ does not abort then in epoch t^*: before any additional leakages, if $b = 0$, $\mathcal{B}_{\mathsf{CKA}}$ encrypts m and thus properly simulates H_{t^*-1}; otherwise, it encrypts $0^{|m|}$ and thus properly simulates H_{t^*}. Also, all message deliveries are directly simulated as in H_{t^*-1+b}.

It is clear that $\mathcal{B}_{\mathsf{CKA}}$ handles bad randomness appropriately and further for corruptions, $\mathcal{B}_{\mathsf{CKA}}$ can always provide the correct state to $\mathcal{A}$ since Type 2 adversary $\mathcal{A}$ never attempts to leak on P_2 when she is in epoch t^*-1 or P_1 when she is in epoch t^*. It is furthermore clear that once $\mathcal{B}_{\mathsf{CKA}}$ gets the random oracle query $(\sigma_{\mathsf{root}}, I_{t^*})$ from $\mathcal{A}$, **test** will return 1 and $\mathcal{B}_{\mathsf{CKA}}$ will win the CKA game, a contradiction. □

6.4 Type 3 Adversaries

Type 3 adversaries are all other adversaries that are not type 1 or 2.

Lemma 4. *If we make the same assumptions as in Theorem 4, then Type 3 adversaries only succeed with probability* $\varepsilon_{\mathsf{FS\text{-}AEAD}}$.

Proof. For Type 3 adversaries, to show $H_{t^*-1} \approx H_{t^*}$, we provide a reduction to the security of the underlying FS-AEAD scheme FS-AEAD. Specifically, assuming that there is some Type 3 adversary $\mathcal{A}$ that distinguishes between hybrids H_{t^*-1} and H_{t^*} with non-negligible probability, we construct reduction algorithm $\mathcal{B}_{\mathsf{FS\text{-}AEAD},2}$ that has non-negligible probability against FS-AEAD in the FS-AEAD security game, thus reaching a contradiction. $\mathcal{B}_{\mathsf{FS\text{-}AEAD},2}$ simulates hybrid H_{t^*-1} or H_{t^*} for $\mathcal{A}$, using the FS-AEAD security game oracle **init** to initialize the FS-AEAD states of P_1 and P_2 in epoch t^*; oracles **corr-P$_1$**, **corr-P$_2$**, and **corr-init-key** to handle leakages of FS-AEAD states of P_1, P_2, and the initialization key; and oracles **chall-P$_1$** and **transmit-P$_1$** to handle FS-AEAD message transmission.

More formally, $\mathcal{B}_{\mathsf{FS\text{-}AEAD},2}$ proceeds as in H_{t^*-1}, with the following exceptions:

- **On input** $(\mathsf{sid}, \mathsf{mid}, \texttt{SEND}, m)$ **from** $\mathcal{Z}$ **to** $\mathsf{P} \in \{\mathsf{P}_1, \mathsf{P}_2\}$:
  1. If $s_{\mathsf{P}}.t = t^* - 1$ before Send would normally be executed then replace the execution of $(s_{\mathsf{P}}.\sigma_{\mathsf{root}}, k) \leftarrow \mathsf{H}(s_{\mathsf{P}}.\sigma_{\mathsf{root}}, I_{t^*})$ and $s_{\mathsf{P}_1}.v[t^*] \leftarrow \mathsf{FS\text{-}Init\text{-}S}(k)$ with (i) oracle query **init**, (ii) sample uniformly random σ, and (iii) set $s_{\mathsf{P}}.\sigma_{\mathsf{root}} \leftarrow \sigma$.
  2. If $s_{\mathsf{P}}.t = t^*$ before $(s_{\mathsf{P}_1}.v[t^*], e) \leftarrow \mathsf{FS\text{-}Send}(s_{\mathsf{P}}.v[t], h, m)$ would normally be executed within Send, and $\mathcal{B}_{\mathsf{FS\text{-}AEAD},2}$ has not yet queried corruption oracles of the FS-AEAD game, then replace its execution with $e \leftarrow$ **chall-P$_1$**$(h, m, 0^{|m|})$ and use the output as in H_{t^*-1}.
  3. If $s_{\mathsf{P}}.t = t^*$ before $(s_{\mathsf{P}_1}.v[t^*], e) \leftarrow \mathsf{FS\text{-}Send}(s_{\mathsf{P}}.v[t], h, m)$ would normally be called within Send, and $\mathcal{B}_{\mathsf{FS\text{-}AEAD},2}$ has queried corruption oracles of the FS-AEAD game, then replace its execution with $e \leftarrow$ **transmit-P$_1$**(h, m) and use the output as in H_{t^*-1}.
  4. Otherwise, proceed as in H_{t^*-1}.
- **On input** $(\texttt{DELIVER}, \mathsf{P}, c)$ **from** $\mathcal{A}$:
  1. Parse $(h, e) \leftarrow c$, $(t, T, \ell) \leftarrow h$.
  2. If $t = t^*$ then:
     (a) If $s_{\mathsf{P}_2}.t = t^* - 1$ before Rcv would normally be called and $T \neq T_{t^*}$ then proceed as in H_{t^*-1}; otherwise replace the execution of $(s_{\mathsf{P}_2}.\sigma_{\mathsf{root}}, k) \leftarrow \mathsf{H}(s_{\mathsf{P}_2}.\sigma_{\mathsf{root}}, I_{t^*})$ and $s_{\mathsf{P}_2}.v[t^*] \leftarrow \mathsf{FS\text{-}Init\text{-}R}(k)$ with set $s_{\mathsf{P}_2}.\sigma_{\mathsf{root}} \leftarrow \sigma$.
     (b) If $\exists(\mathsf{sid}, \cdot, t^*, \cdot, \texttt{IN_TRANSIT}, c, \cdot) \in \mathsf{P}_1.\mathtt{T}$ then query oracle $(i, m) \leftarrow$ **deliver-P$_2$**(h, e) and use the output as in H_{t^*-1} (including possibly rolling back the state of P_2).
     (c) Otherwise, query oracle $(i, m) \leftarrow$ **inject-P$_2$**(h, e) and use the output as in H_{t^*-1} (including possibly rolling back the state of P_2).
     (d) If $s_{\mathsf{P}_2}.v[t^*] \neq \bot \wedge m \neq \bot$ then additionally execute $s_{\mathsf{P}_2}.v[t^*] \leftarrow$ **corr-P$_2$**.
  3. Otherwise proceed as in H_{t^*-1} (which is the same as in H_{t^*}).
- **On input** $(\mathsf{sid}, \texttt{LEAK}, \mathsf{P})$ **from** $\mathcal{A}$:
  1. If $(\mathsf{P} = \mathsf{P}_1 \wedge s_{\mathsf{P}_1}.t = t^* \wedge \mathsf{P}.\mathtt{V} \neq \emptyset) \vee (\mathsf{P} = \mathsf{P}_2 \wedge s_{\mathsf{P}_2}.t = t^* - 1)$ then query $k \leftarrow$ **corr-init-key** and program the random oracle so that $(\sigma, k) \leftarrow \mathsf{H}(\sigma_{\mathsf{root}}^{t^*-1}, I_{t^*})$.

2. If $\mathsf{P} = \mathsf{P}_1 \wedge s_{\mathsf{P}_1}.t = t^* \wedge \mathsf{P}.\mathtt{V} = \emptyset$ then query $s_{\mathsf{P}_1}.v[t^*] \leftarrow$ **corr-P$_1$**.
3. If $\mathsf{P} = \mathsf{P}_2 \wedge s_{\mathsf{P}_2}.t = t^*$ then query $s_{\mathsf{P}_2}.v[t^*] \leftarrow$ **corr-P$_2$**.
4. Otherwise proceed as in H_{t^*-1}.

- **On input bit b from $\mathcal{A}$**: Forward b to the challenger.

We now show that when the challenge bit b of the FS-AEAD security game is 0, $\mathcal{B}_{\mathsf{FS\text{-}AEAD},2}$ properly simulates H_{t^*-1} and when it is 1, $\mathcal{B}_{\mathsf{FS\text{-}AEAD},2}$ properly simulates H_{t^*}. Since $\mathcal{A}$ is a Type 3 adversary, she either does not query the random oracle on $(s_{\mathsf{P}}.\sigma_{\mathsf{root}}^{t^*-1}, I_{t^*})$, or before she makes such a query she leaks on P_2 when $s_{\mathsf{P}_2}.t = t^* - 1$ or leaks on P_1 when $s_{\mathsf{P}_1}.t = t^*$. Before such a leakage (and thus such a random oracle query), in both H_{t^*-1} and H_{t^*} the corresponding output $(s_{\mathsf{P}}.\sigma_{\mathsf{root}}^{t^*}, k)$ is uniformly random and unknown to $\mathcal{A}$. Thus implicitly using uniformly random k for FS-AEAD initialization via the challenger and using randomly sampled σ for the updated σ_{root} of epoch t^* properly simulates both hybrids. Furthermore, of course encrypting in epoch t^* before this point using the FS-AEAD **chall-P$_1$**() oracle of course properly simulates H_{t^*-1+b}, where b is the challenge bit of the FS-AEAD security game. If such a leakage does happen then we properly program the random oracle on $(\sigma_{\mathsf{root}}^{t^*-1}, I_{t^*})$ if needed, and properly explain the FS-AEAD states and ciphertexts using FS-Expl-In-Trans-Cts and FS-Expl-Vul-Cts.

It is clear that delivery of epoch t^* messages by $\mathcal{B}_{\mathsf{FS\text{-}AEAD},2}$ simulates H_{t^*-1} properly. If $\mathcal{A}$ unsuccessfully forges an epoch t^* message (before P_2 accepts a message for the epoch) then the behavior of H_{t^*-1} and H_{t^*} are identical (both reject the message). Since $\mathcal{A}$ is a Type 3 adversary, if she does successfully forge the first message that P_2 receives in epoch t^* with CKA message $T \neq T_{t^*}$, then it must be that $\mathcal{A}$ queried the random oracle on $(\sigma_{\mathsf{root}}^{t^*-1}, I)$, where I is the corresponding CKA secret associated with T. As in the proof of Lemma 3, in order for $\mathcal{A}$ to do this with non-negligible probability, then it must be that of course $\mathsf{P}_1.\mathtt{TAKEOVER_POSS} = 1$; for otherwise, after P_2 sent the first message of epoch $t^* - 1$, $\mathsf{P}_2.\mathtt{CUR_SLEK} = 0$ and so $\sigma_{\mathsf{root}}^{t^*-1}$ would be uniformly random and unknown to $\mathcal{A}$. Thus such forgeries are also successful in H_{t^*}. Furthermore, if P_2 accepts the first message of epoch t^* with proper CKA message T_{t^*}, then the reduction simulates delivery for H_{t^*} properly due to the underlying correctness and security of FS-AEAD: namely, messages are correctly decrypted with the correct index i, only one message for each index i successfully decrypts (even with injections), and injections/modifications of in-transit messages can only happen if the FS-AEAD secret state is leaked.

If the challenger declares **win** or $\mathcal{A}$ guesses the challenge bit b and $\mathcal{B}_{\mathsf{FS\text{-}AEAD},2}$ passes it along, then $\mathcal{B}_{\mathsf{FS\text{-}AEAD},2}$ wins the FS-AEAD security game, a contradiction. □

Proof of Lemma 1. Follows immediately from Lemmas 2, 3, and 4. □

Proof of Theorem 4. Follows immediately from Lemma 1. □

6.5 Even Stronger Security of the TR with CKA$^+$

We observe that the TR instantiated with our specific CKA scheme CKA$^+$ actually has even *stronger* security, but do not write $\mathcal{F}_{\mathsf{TR}}$ to reflect it, in order to tame the ideal functionality's complexity. Roughly, in $\mathcal{F}_{\mathsf{TR}}$ (Figure 3), if P_2 starts a new epoch $t^* - 1$ with bad randomness, then the functionality sets $\mathsf{P}_2.\mathtt{PLEK} \leftarrow 1$. Next, if P_1 receives a message in P_2's new epoch $t^* - 1$ and then the adversary leaks her state, the functionality of course sets $\mathsf{P}_2.\mathtt{CUR_SLEK} \leftarrow 1$. So, when P_1 starts

new epoch t^*, the functionality will set $\mathsf{P}_1.\texttt{CUR_SLEK} \leftarrow 1$, since both $\mathsf{P}_2.\texttt{CUR_SLEK}$ and $\mathsf{P}_2.\texttt{PLEK}$ are 1. Therefore, all messages of epoch t^* are deemed insecure by $\mathcal{F}_{\mathsf{TR}}$.

However, recall the stronger security of CKA^+, which we informally highlighted in Section 4.3.5. If we assume secure initialization, good randomness when P_1 sends the first message of epoch 1, no leakage except that of epoch $t^* - 1$ on P_1 above, and again good randomness when P_1 sends the first message of epoch t^*, then the CKA secret I_t for every epoch t remains secure. This is true even if P_2 always uses bad randomness, and besides from epochs 1 and t^*, P_1 also always uses bad randomness. Thus, the symmetric ratchet of every epoch except $t^* - 1$ remains secure (in-transit messages at the time of corruption are also insecure), and so notably the above messages of epoch t^* that $\mathcal{F}_{\mathsf{TR}}$ deems insecure, are indeed secure.

We remark that if P_2 is leaked before receiving a message for epoch $t^* - 1$, then all security is lost: the adversary by correctness learns I_{t^*-1} and can thus also compute the CKA state of P_2, $\gamma_{t^*-1}^{\mathsf{P}_2} = x_{t^*-1} \cdot \mathsf{H}(I_{t^*-1})$, where x_{t^*-1} is the CKA exponent that P_2 samples for epoch $t^* - 1$. So, along with $\sigma_{\mathsf{root}}^{t^*-1}$, the adversary will also have all future CKA shared secrets, and so all future messages will be insecure.

However, in the (not completely far-fetched) scenario in which initialization is secure, P_2 never has good randomness, and P_1 is leaked *after* receiving an epoch $t^* - 1$ message; if P_1 at least uses good randomness for epochs 1 and t^*, then all messages remain secure. On the other hand, in the DR, all CKA secrets I_t are of course leaked in this situation, so once P_1 is corrupted, all security is lost.

We also note that for *further* stronger security, after P sends the first message of an epoch t, instead of setting $\gamma_t^{\mathsf{P}} \leftarrow x_t \cdot \mathsf{H}(I_t)$, where x_t is the sampled exponent of epoch t, one could set $\gamma_t^{\mathsf{P}} \leftarrow x_t \cdot \mathsf{H}(\sigma_{\mathsf{root}}^{t-1}, I_t)$ (or just expand the root KDF computation $(\sigma_{\mathsf{root}}^t, k) \leftarrow \mathsf{H}(\sigma_{\mathsf{root}}^{t-1}, I_t)$ that is already in the TR). Intuitively, this should provide even stronger security, but we did not find any simple examples where security is stronger, other than the above.

7 More Efficient Updatable Public-Key Encryption

Motivated by applications in Secure (Group) Messaging, Jost *et al.* [JM20], Alwen *et al.* [ACDT20], and Dodis *et al.* [DKW21] recently utilized and built constructions of a primitive, similar to CKA, called *Updatable Public Key Encryption* (UPKE) (itself a relaxation of the stronger forward-secure public key encryption primitive [CHK04]). UPKE is an enhancement to PKE wherein beginning with an initially sampled key-pair (sk_0, pk_0), a sender can securely communicate with the receiver, such that with each ciphertext c_i, the sender updates pk_i to pk_{i+1} and further embeds within c_i information for the receiver to update the secret key sk_i to sk_{i+1} so that the next message can be encrypted under pk_{i+1} and decrypted using sk_{i+1} correctly.[7] Furthermore, if some sk_j is corrupted, the messages of all ciphertexts c_i for all $i < j$ should be hidden. This security property should be maintained even if the randomness for all such ciphertexts c_i is adversarially chosen (but that for c_j is chosen uniformly at random and hidden from the adversary). UPKE is used in, e.g., the re-randomized TreeKEM (rTreeKEM) protocol of [ACDT20], for Secure Group Messaging. We refer the reader to that paper for the full details of the application, as they are not directly relevant to our analysis of the DR.

[7] We use the notion introduced by [ACDT20], which is slightly different from those introduced by [JMM19a, DKW21], wherein *update ciphertexts* used to update the keypair are separated from the actual message ciphertexts. Our optimization of this section does not apply to this separated notion.

In this section, we optimize the (more efficient) UPKE scheme of [ACDT20], reducing the length of ciphertexts by $|G|$ bits, where $|G|$ is the number of bits needed to represent the size of the (CDH-hard) group used in the scheme. As a result, communication of the rTreeKEM protocol used in Secure Group Messaging by [ACDT20] is reduced by up to a $|G| \cdot n$ additive factor, where n is the number of users in the group. We now formally define UPKE:

Definition 10. *An* updatable public-key encryption *(UPKE) scheme is a triple of algorithms* $\mathsf{UPKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *with the following syntax:*

- *Key generation:* Gen *receives a uniformly random key* sk_0 *and outputs a fresh initial public key* $pk_0 \leftarrow \mathsf{Gen}(sk_0)$.
- *Encryption:* Enc *receives a public key* pk *and a message* m *and produces a ciphertext* c *and a new public key* pk'.
- *Decryption:* Dec *receives a secret key* sk *and a ciphertext* c *and outputs a message* m *and a new secret key* sk'.

Correctness. A UPKE scheme must satisfy the following correctness property. For any sequence of randomness and message pairs $\{r_i, m_i\}_{i=1}^{q}$,

$$\Pr[sk_0 \xleftarrow{\$} \mathcal{SK}; pk_0 \leftarrow \mathsf{Gen}(sk_0); \text{For } i \in [q], (c_i, pk_i) \leftarrow \mathsf{Enc}(pk_{i-1}, m_i; r_i); \\ (m_i', sk_i) \leftarrow \mathsf{Dec}(sk_{i-1}, c_i) : m_i = m_i'] = 1.$$

IND-CPA* security for UPKE. For any adversary $\mathcal{A}$ with running time t we consider the IND-CPA* security game:

- Sample $sk_0 \xleftarrow{\$} \mathcal{SK}, pk_0 \leftarrow \mathsf{Gen}(sk_0)$
- $\mathcal{A}$ on input pk_0 outputs $(m_0^*, m_1^*), \{r_i, m_i\}_{i=1}^{q}$
- For $i = 1, \ldots, q$, compute $(c_i, pk_i) \leftarrow \mathsf{Enc}(pk_{i-1}, m_i; r_i); (m_i, sk_i) \leftarrow \mathsf{Dec}(sk_{i-1}, c_i)$
- Compute $b \xleftarrow{\$} \{0,1\}, (c^*, pk^*) \leftarrow \mathsf{Enc}(pk_q, m_b^*), (\cdot, sk^*) \leftarrow \mathsf{Dec}(sk_q, c^*)$
- $b' \leftarrow \mathcal{A}(pk^*, sk^*, c^*)$

$\mathcal{A}$ wins the game if $b = b'$. The advantage of $\mathcal{A}$ in winning the above game is denoted by $\mathsf{Adv}_{\mathrm{cpa}^*}^{\mathsf{UPKE}}(\mathcal{A})$.

Definition 11. *An updatable public-key encryption scheme* UPKE *is* (t, ε)-CPA*-secure *if for all* t*-attackers* $\mathcal{A}$*,*

$$\mathsf{Adv}_{\mathrm{cpa}^*}^{\mathsf{UPKE}}(\mathcal{A}) \leq \varepsilon.$$

7.1 UPKE Construction from [JMM19a, ACDT20]

Here we first present the UPKE construction of [ACDT20] (itself inspired by [JMM19a]). We focus on this construction (in the Random Oracle model) since the construction of [DKW21] (in the Standard model) may not be efficient enough for practical use cases. The ROM construction is formally presented in Figure 9. A proof of the construction's IND-CPA* security is given in [ACDT20]. Note that c consists of one group element and $|m| + |G|$ bits.

UPKE of [JMM19a, ACDT20]

$\mathsf{Gen}\,(sk)$
- **return** g^{sk}

$\mathsf{Enc}\,(pk, m)$
- $(r, \delta) \xleftarrow{\$} \mathbb{Z}_p \times \mathbb{Z}_p$
- $c \leftarrow (g^r, H(pk^r) \oplus (m||\delta))$
- **return** $(c, pk \cdot g^{\delta})$

$\mathsf{Dec}\,(sk, (c_1, c_2))$
- $m||\delta \leftarrow H(c_1^{sk}) \oplus c_2$
- $sk' \leftarrow sk + \delta \mod p$
- **return** (m, sk')

Figure 9: UPKE construction of [JMM19a, ACDT20] assuming Random Oracles and the hardness of Computational Diffie-Hellman. g is the generator of a group G of prime order p and sk is in $\mathbb{Z}_p$.

More Efficient UPKE

$\mathsf{Gen}\,(sk)$
- **return** g^{sk}

$\mathsf{Enc}\,(pk, m)$
- $r \xleftarrow{\$} \mathbb{Z}_p$
- $\delta \leftarrow \mathsf{H}(pk^r)$
- $c \leftarrow (g^r, \delta \oplus m)$
- **return** $(c, pk \cdot g^{\delta})$

$\mathsf{Dec}\,(sk, (c_1, c_2))$
- $\delta \leftarrow \mathsf{H}(c_1^{sk})$
- $m \leftarrow \delta \oplus c_2$
- $sk' \leftarrow sk + \delta \mod p$
- **return** (m, sk')

Figure 10: More efficient UPKE construction assuming hash function H is modelled as a Random Oracle and the hardness of Computational Diffie-Hellman. g is the generator of a group G of prime order p and sk is in $\mathbb{Z}_p$.

7.2 More Efficient Construction

In Figure 10, we present our more efficient UPKE construction. We use a similar trick as that which was deployed in our construction of CKA^{+}. In our scheme, c consists of one group element and $|m|$ bits, eliminating an additional $|G|$ bits. As this is the same size as regular ElGamal PKE ciphertexts, our construction shows that we can lift the CPA-security of ElGamal PKE to CPA*-security without any additional communication.

Theorem 5. *Assuming the hardness of CDH over the group G, the UPKE scheme of Figure 10 is CPA*-secure if H is modelled as Random Oracles.*

Proof. Our proof follows those of [ACDT20, JMM19a] for their less efficient UPKE schemes. Given an adversary $\mathcal{A}$ that breaks the CPA* security of the UPKE scheme of Figure 10, we define an adversary $\mathcal{B}$ against the CDH problem (given challenge $(A, B) = (g^a, g^b)$) assuming H is modelled as a random oracle. $\mathcal{B}$ is defined below.

$\mathcal{B}(A, B)$:

1. Sample $b \xleftarrow{\$} \{0, 1\}, \delta \xleftarrow{\$} \mathbb{Z}_p$ and set $pk_0 \leftarrow A \cdot g^{\delta}$.
2. Execute $\mathcal{A}$ on input pk_0 and receive $(m_0^*, m_1^*), \{r_i, m_i\}_{i=1}^{q}$.

3. For $i = 1, \ldots, q$, compute $(c_i, pk_i) \leftarrow \mathsf{Enc}(pk_{i-1}, m_i; r_i)$.

4. Let $pk_q = g^{a+\sum_{i=1}^{q} \delta_i + \delta}$ and $\Delta = \sum_{i=1}^{q} \delta_i + \delta$, where each δ_i is chosen randomly as the output of H, based on the randomness r_i chosen by $\mathcal{A}$ (and ensuring consistency with the random oracle queries of $\mathcal{A}$).

5. Set $pk^* = pk_q \cdot A^{-1}$ and $sk^* = \Delta$.

6. Compute $R \xleftarrow{\$} \{0,1\}^{|m|}$, set $c^* \leftarrow (B, R)$, and output (pk^*, sk^*, c^*) to $\mathcal{A}$.

7. Let Q be the list of queries made to the random oracle by $\mathcal{A}$. Run the Diffie-Hellman self-corrector of [Sho97] with respect to Q and multiply the output by $B^{-\Delta}$ to obtain a solution.

Observe that, given the computation of steps 1-5, $\mathcal{A}$ should receive in step 6 the triple,

$$\left(pk^* = g^{\Delta}, sk^* = \Delta, c^* = \left(B, \mathsf{H}\left(g^{(a+\Delta)b}\right) \oplus m_b^*\right)\right).$$

However, in the above execution it receives,

$$\left(pk^* = g^{\Delta}, sk^* = \Delta, c^* = (B, R)\right),$$

for uniformly random R. The only way for $\mathcal{A}$ to distinguish between the above is by querying the random oracle with $g^{(a+\Delta)b}$, thus by multiplying $g^{(a+\Delta)b}B^{-\Delta}$, $\mathcal{B}$ obtains g^{ab}. □

Acknowledgements

We would like to thank Yevgeniy Dodis and Daniel Jost for helping us realize that the trick used in the CKA^+ construction can also be used to make UPKE more efficient (Section 7).

References

[ABR01] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In David Naccache, editor, *CT-RSA 2001*, volume 2020 of *LNCS*, pages 143–158, San Francisco, CA, USA, April 8–12, 2001. Springer, Heidelberg, Germany.

[ACD18] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. Cryptology ePrint Archive, Report 2018/1037, 2018. https://eprint.iacr.org/2018/1037.

[ACD19] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 129–158, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany.

[ACDT20] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In Daniele

Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 248–277, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany.

[BCH12] Nir Bitansky, Ran Canetti, and Shai Halevi. Leakage-tolerant interactive protocols. In Ronald Cramer, editor, *TCC 2012*, volume 7194 of *LNCS*, pages 266–284, Taormina, Sicily, Italy, March 19–21, 2012. Springer, Heidelberg, Germany.

[BDZ03] Feng Bao, Robert H. Deng, and Huafei Zhu. Variations of diffie-hellman problem. In *ICICS*, 2003.

[BFG+20] Jacqueline Brendel, Marc Fischlin, Felix Günther, Christian Janson, and Douglas Stebila. Towards post-quantum security for signal's x3dh handshake. In *Selected Areas in Cryptography–SAC 2020*, 2020.

[BFGJ17] Jacqueline Brendel, Marc Fischlin, Felix Günther, and Christian Janson. PRF-ODH: Relations, instantiations, and impossibility results. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 651–681, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

[BGB04] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use pgp. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 77–84, 2004.

[BRV20] Fatih Balli, Paul Rösler, and Serge Vaudenay. Determining the core primitive for optimally secure ratcheting. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 621–650, Daejeon, South Korea, December 7–11, 2020. Springer, Heidelberg, Germany.

[BSJ+17] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 619–650, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press.

[CCD+20] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. *J. Cryptol.*, 33(4):1914–1983, 2020.

[CHK04] Ran Canetti, Shai Halevi, and Jonathan Katz. Chosen-ciphertext security from identity-based encryption. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 207–222, Interlaken, Switzerland, May 2–6, 2004. Springer, Heidelberg, Germany.

[CNE+14] Stephen Checkoway, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, Hovav Shacham, and Matthew Fredrikson. On the practical exploitability of dual EC in TLS implementations. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 319–335, San Diego, CA, USA, August 20–22, 2014. USENIX Association.

[CS03] Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, November 2003.

[DKW21] Yevgeniy Dodis, Harish Karthikeyan, and Daniel Wichs. Updatable public key encryption in the standard model. 2021.

[DV17] F. Betül Durak and Serge Vaudenay. Breaking the FF3 format-preserving encryption standard over small domains. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 679–707, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

[DV19] F. Betül Durak and Serge Vaudenay. Bidirectional asynchronous ratcheted key agreement with linear complexity. In Nuttapong Attrapadung and Takeshi Yagi, editors, *IWSEC 19*, volume 11689 of *LNCS*, pages 343–362, Tokyo, Japan, August 28–30, 2019. Springer, Heidelberg, Germany.

[FIP95] PUB FIPS. 180-1. secure hash standard. *National Institute of Standards and Technology*, 17:45, 1995.

[Gal12] Steven D. Galbraith. *Mathematics of Public Key Cryptography*. Cambridge University Press, 2012.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989.

[Gol04] Oded Goldreich. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004.

[HDWH12] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In Tadayoshi Kohno, editor, *USENIX Security 2012*, pages 205–220, Bellevue, WA, USA, August 8–10, 2012. USENIX Association.

[HKKP21] Keitaro Hashimoto, Shuichi Katsumata, Kris Kwiatkowski, and Thomas Prest. An efficient and generic construction for signal's handshake (x3dh): Post-quantum, state leakage secure, and deniable. In *Public Key Cryptography (2)*, pages 410–440, 2021.

[JM20] Daniel Jost and Ueli Maurer. Overcoming impossibility results in composable security using interval-wise guarantees. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 33–62, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany.

[JMM19a] Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 159–188, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany.

[JMM19b] Daniel Jost, Ueli Maurer, and Marta Mularczyk. A unified and composable take on ratcheting. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part II*, volume 11892 of *LNCS*, pages 180–210, Nuremberg, Germany, December 1–5, 2019. Springer, Heidelberg, Germany.

[JS18] Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 33–62, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

[KE10] Hugo Krawczyk and Pasi Eronen. Hmac-based extract-and-expand key derivation function (hkdf). Technical report, RFC 5869, May, 2010.

[Kil01] Eike Kiltz. A tool box of cryptographic functions related to the Diffie-Hellman function. In C. Pandu Rangan and Cunsheng Ding, editors, *INDOCRYPT 2001*, volume 2247 of *LNCS*, pages 339–350, Chennai, India, December 16–20, 2001. Springer, Heidelberg, Germany.

[KM04] Kaoru Kurosawa and Toshihiko Matsuo. How to remove MAC from DHIES. In Huaxiong Wang, Josef Pieprzyk, and Vijay Varadharajan, editors, *ACISP 04*, volume 3108 of *LNCS*, pages 236–247, Sydney, NSW, Australia, July 13–15, 2004. Springer, Heidelberg, Germany.

[Mau11] Ueli Maurer. Constructive cryptography–a new paradigm for security definitions and proofs. In *Joint Workshop on Theory of Security and Applications*, pages 33–56. Springer, 2011.

[MP16a] M. Marlinspike and T. Perrin. The Double Ratchet Algorithm, 11 2016. https://whispersystems.org/docs/specifications/doubleratchet/doubleratchet.pdf.

[MP16b] M. Marlinspike and T. Perrin. The X3DH Key Agreement Protocol, 11 2016. https://signal.org/docs/specifications/x3dh/x3dh.pdf.

[MW96] Ueli M. Maurer and Stefan Wolf. Diffie-Hellman oracles. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 268–282, Santa Barbara, CA, USA, August 18–22, 1996. Springer, Heidelberg, Germany.

[Nie02] Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 111–126, Santa Barbara, CA, USA, August 18–22, 2002. Springer, Heidelberg, Germany.

[PR18] Bertram Poettering and Paul Rösler. Towards bidirectional ratcheted key exchange. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume

10991 of *LNCS*, pages 3–32, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

[Sho97] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266, Konstanz, Germany, May 11–15, 1997. Springer, Heidelberg, Germany.

[Sip97] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.

[UG15] Nik Unger and Ian Goldberg. Deniable key exchanges for secure messaging. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 1211–1223, Denver, CO, USA, October 12–16, 2015. ACM Press.

[UG18] Nik Unger and Ian Goldberg. Improved strongly deniable authenticated key exchanges for secure messaging. *Proc. Priv. Enhancing Technol.*, 2018(1):21–66, 2018.

[VGIK20] Nihal Vatandas, Rosario Gennaro, Bertrand Ithurburn, and Hugo Krawczyk. On the cryptographic deniability of the Signal protocol. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *ACNS 20, Part II*, volume 12147 of *LNCS*, pages 188–209, Rome, Italy, October 19–22, 2020. Springer, Heidelberg, Germany.

[YRS+09] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public: Results from the 2008 debian openssl vulnerability. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC '09, pages 15–27, New York, NY, USA, 2009. Association for Computing Machinery.

A Comparison to the ACD and CCD$^+$ Secure Messaging Security Notions

In this section we demonstrate the following *six* distinct (sometimes contrived) modifications to the DR secure messaging protocol (c.f. Figure 7) which, when instantiated with **correct** and **secure** underlying FS-AEAD and CKA schemes, are vulnerable to natural attacks – formally, we show that they are insecure with respect to our (weaker) ideal functionality $\mathcal{F}_{\mathsf{DR}}$. Despite this, we show that some of the modified protocols remain ACD-secure (transformations 1-4) and/or CCD$^+$-secure (transformations 3-6).[8]

1. **Postponed FS-AEAD Key Deletion.**
2. **Postponed CKA Key Deletion.**
3. **Eager CKA Randomness Sampling.**
4. **Malleable Ciphertexts.**
5. **CKA Bad Randomness Plaintext Trigger.**
6. **Removed Immediate Decryption.**

Transformations 1-5 satisfy the correctness while affect the security – intuitively breaking either forward security or post-compromise security. Transformation 6 preserves security, while affecting correctness – specifically immediate decryption.

Before we present the transformations and prove their (in)security, we recall the definitions of ACD [ACD19]. Although their main SM definition is game-based, the definitions of their building blocks are quite similar to ours. Since the definition of CCD$^+$ [CCD$^+$20] is quite complex and different from ours, we do not include it in our paper, and instead refer the reader to their paper. The aspects of their definition which we exploit are quite simple and do not require complete understanding of their definition. We will additionally describe the relevant specifics of their definition which our transformations exploit.

A.1 Definitions from ACD

To formally show that the transformed protocols continue to satisfy ACD's definitions we need to borrow definitions from ACD. While ACD's game-based secure message definition is completely different than our simulation based definition (Figure 3), our FS-AEAD and CKA definitions are quite similar to theirs. The differences are discussed in the respective sections (c.f. Section 4.2 and Section 4.3). To distinguish from our definitions, we refer to the a scheme which is secure according to ACD's definition as *ACD-secure*. Next we present ACD's definitions, often taken verbatim from ACD.

[8] We believe the rest to be insecure in ACD (transformations 5 and 6) and CCD$^+$ (transformations 1 and 2).

A.1.1 Secure Messaging (ACD)

ACD's game-based definition of secure messaging consists of (potentially asymmetric) initialization algorithms Init-A and Init-B, a generic sending algorithm Send, and a generic receiving algorithm Rcv. Each party maintains a state to use across invocations of the sending and receiving algorithms. Importantly, the receiving algorithm additionally outputs an epoch number and message index which is used to determine the order in which the sending party transmitted their messages. We present these algorithms formally in Definition 12.

Definition 12. *A* secure-messaging (SM) scheme *consists of four probabilistic algorithms* $\mathsf{SM} = (\mathsf{Init\text{-}A}, \mathsf{Init\text{-}B}, \mathsf{Send}, \mathsf{Rcv})$*, where*

- Init-A *(and similarly* Init-B*) takes a key* k *and outputs a state* $s_{\mathsf{A}} \leftarrow \mathsf{Init\text{-}A}(k)$*,*
- Send *takes a state* s *and a message* m *and produces a new state and a ciphertext* $(s', c) \xleftarrow{\$} \mathsf{Send}(s, m)$*, and*
- Rcv *takes a state* s *and a ciphertext* c *and produces a new state, an epoch number, an index, and a message* $(s', t, i, m) \leftarrow \mathsf{Rcv}(s, c)$*.*

Game-Based Secure Messaging. We reproduce the game-based secure messaging security notion from ACD in Figure 11. The security game consists of an initialization procedure **init**, two sending oracles **transmit-**P_1 (normal transmission) and **chall-**P_1 (challenge transmission), two receive oracles **deliver-**P_1 (honest delivery) and **inject-**P_1 (for forged ciphertexts), and a corruption oracle **corr-**P_1. These oracles (with the exception of **init**) are defined with respect to party P_1. The oracles for P_2 are defined analogously. We remark that, ACD's definition considers Alice (**A**) and Bob (**B**) instead of parties P_1 and P_2.

Due to the complexity of capturing secure messaging with game-based definitions, a number of game management functions are provided. These consist of a epoch management function **ep-mgmt**, randomness sampling function **sam-if-nec**, and record keeping functions **record** and **delete** which can be called by the attacker. Additionally, the **safe-ch**$_{\mathsf{P}}$ and **safe-inj** control the adversary's ability to make challenges and inject respectively.

The game is parametrized by an integer Δ_{SM} which relates to how fast parties recover from state compromise. The advantage of $\mathcal{A}$ against an SM scheme SM is denoted by $\mathsf{Adv}^{\mathsf{SM}}_{\mathrm{sm},\Delta_{\mathsf{SM}}}(\mathcal{A})$. The attacker is parameterized by its running time t, the total number of queries q it makes, and the maximum number of epochs q_{ep} it runs for. We define SM security in Definition 13.

Definition 13. *A secure-messaging scheme* SM *is* $(t, q, q_{\mathsf{ep}}, \Delta_{\mathsf{SM}}, \varepsilon)$-secure *if for all* (t, q, q_{ep})*-attackers* $\mathcal{A}$*,*

$$\mathsf{Adv}^{\mathsf{SM}}_{\mathrm{sm},\Delta_{\mathsf{SM}}}(\mathcal{A}) \;\leq\; \varepsilon \;.$$

Definition 14. *A secure-messaging scheme* SM *is called simply* secure *with* Δ_{SM} *if it is* $(\mathrm{poly}(\kappa), \mathrm{poly}(\kappa), \mathrm{poly}(\kappa), \Delta_{\mathsf{SM}}, \mathrm{negl}(\kappa))$*-secure.*

For more details regarding the secure messaging game-based definition, we refer the reader to Section 3 of the full version of ACD [ACD18].

Security Game for Secure Messaging

init
- $k \xleftarrow{\$} \mathcal{K}$
- $s_{\mathsf{P}_1} \leftarrow \mathsf{Init\text{-}P}_1(k)$
- $s_{\mathsf{P}_2} \leftarrow \mathsf{Init\text{-}P}_2(k)$
- $(t_{\mathsf{P}_1}, t_{\mathsf{P}_2}) \leftarrow (0,0)$
- $i_{\mathsf{P}_1}, i_{\mathsf{P}_2} \leftarrow 0$
- $t_{\mathsf{L}} \leftarrow -\infty$
- $\mathsf{trans}, \mathsf{chall}, \mathsf{comp} \leftarrow \emptyset$
- $b \xleftarrow{\$} \{0,1\}$

corr-P$_1$
- **req** $\mathsf{P}_2 \notin \mathsf{chall}$
- $\mathsf{comp} \xleftarrow{+} \mathsf{trans}(\mathsf{P}_2)$
- $t_{\mathsf{L}} \leftarrow \max(t_{\mathsf{P}_1}, t_{\mathsf{P}_2})$
- **return** s_{P_1}

transmit-P$_1$ (m, r)
- $(r, \mathsf{flag}) \leftarrow$ **sam-if-nec**(r)
- **ep-mgmt**$(\mathsf{P}_1, \mathsf{flag})$
- i_{P_1} ++
- $(s_{\mathsf{P}_1}, c) \leftarrow \mathsf{Send}(s_{\mathsf{A}}, m; r)$
- **record**$(\mathsf{P}_1, \mathsf{norm}, m, c)$
- **return** c

chall-P$_1$ (m_0, m_1, r)
- $(r, \mathsf{flag}) \leftarrow$ **sam-if-nec**(r)
- **ep-mgmt**$(\mathsf{P}_1, \mathsf{flag})$
- **req** $\mathsf{safe\text{-}ch}_{\mathsf{P}_1}$ and $|m_0| = |m_1|$
- i_{P_1} ++
- $(s_{\mathsf{P}_1}, c) \leftarrow \mathsf{Send}(m_b; r)$
- **record**$(\mathsf{P}_1, \mathsf{chall}, m_b, c)$
- **return** c

deliver-P$_2$ (c)
- **req** $(\mathsf{B}, t, i, m, c) \in \mathsf{trans}$ for some t, i, m
- $(s_{\mathsf{P}_1}, t', i', m') \leftarrow \mathsf{Rcv}(s_{\mathsf{P}_1}, c)$
- **if** $(t', i', m') \neq (t, i, m)$
  - **win**
- **if** $(t, i, m) \in \mathsf{chall}$
  - $m' \leftarrow \bot$
- $t_{\mathsf{P}_1} \leftarrow \max(t_{\mathsf{P}_1}, t)$
- **delete**(t, i)
- **return** (t', i', m')

inject-P$_2$ (c)
- **req** $(\mathsf{P}_2, c) \notin \mathsf{trans}$ and $\mathsf{safe\text{-}inj}$
- $(s_{\mathsf{P}_1}, t', i', m') \leftarrow \mathsf{Rcv}(s_{\mathsf{P}_1}, c)$
- **if** $m' \neq \bot$ *and* $(\mathsf{P}_2, t', i') \notin \mathsf{comp}$
  - **win**
- $t_{\mathsf{A}} \leftarrow \max(t_{\mathsf{P}_1}, t')$
- **delete**(t', i')
- **return** (t', i', m')

ep-mgmt $(\mathsf{P}, \mathsf{flag})$
- **if** $\mathsf{P} = \mathsf{P}_1$ *and* t_{P} *even or* $\mathsf{P} = \mathsf{P}_2$ *and* t_{P} *odd*
  - **if** $\mathsf{flag} = \mathsf{bad}$ *and* $\neg\mathsf{safe\text{-}ch}_{\mathsf{P}}$
    - $t_{\mathsf{L}} \leftarrow t_{\mathsf{P}} + 1$
  - t_{P} ++
  - $i_{\mathsf{P}} \leftarrow 0$

sam-if-nec (r)
- $\mathsf{flag} \leftarrow \mathsf{bad}$
- **if** $r = \bot$
  - $r \xleftarrow{\$} \mathcal{R}$
  - $\mathsf{flag} \leftarrow \mathsf{good}$
- **return** (r, flag)

record $(\mathsf{P}, \mathsf{flag}, m, c)$
- $\mathsf{rec} \leftarrow (\mathsf{P}, t_{\mathsf{P}}, i_{\mathsf{P}}, m, c)$
- $\mathsf{trans} \xleftarrow{+} \mathsf{rec}$
- **if** $\neg\mathsf{safe\text{-}ch}_{\mathsf{P}}$
  - $\mathsf{comp} \xleftarrow{+} \mathsf{rec}$
- **if** $\mathsf{flag} = \mathsf{chall}$
  - $\mathsf{chall} \xleftarrow{+} \mathsf{rec}$

delete (t, i)
- $\mathsf{rec} \leftarrow (\mathsf{P}, t, i, m, c)$ for some P, m, c
- $\mathsf{trans}, \mathsf{chall}, \mathsf{comp} \xleftarrow{-} \mathsf{rec}$

$\mathsf{safe\text{-}ch}_{\mathsf{P}} \;:\Longleftrightarrow\; t_{\mathsf{P}} \geq t_{\mathsf{L}} + \Delta_{\mathsf{SM}}$

$\mathsf{safe\text{-}inj} \;:\Longleftrightarrow\; \min(t_{\mathsf{A}}, t_{\mathsf{B}}) \geq t_{\mathsf{L}} + \Delta_{\mathsf{SM}}$

Figure 11: Game-based notion of secure messaging (SM) security from [ACD19]. Oracles correspond to party P_1 of the SM security game for a scheme $\mathsf{SM} = (\mathsf{Init\text{-}P}_1, \mathsf{Init\text{-}P}_2, \mathsf{Send}, \mathsf{Rcv})$. The oracles for P_2 are defined analogously. We note that the syntax above is slightly changed to parties $\mathsf{P}_1/\mathsf{P}_2$ from $\mathbf{A}/\mathbf{B}$ for consistency with our own definitions.

A.1.2 ACD's CKA Definition

The syntax and the correctness of ACD's CKA is exactly the same as ours (c.f. Definition 6). The security notion, however, diverges as we attempt to capture a more fine-grained notion. Neverthe-

ACD's Security Game for CKA

init (t^*)
- $k \xleftarrow{\$} \mathcal{K}$
- $\gamma_0^{\mathsf{P}_1} \leftarrow \mathsf{CKA\text{-}Init\text{-}P_1}(k)$
- $\gamma_0^{\mathsf{P}_2} \leftarrow \mathsf{CKA\text{-}Init\text{-}P_2}(k)$
- $t_{\mathsf{P}_1}, t_{\mathsf{P}_2} \leftarrow 0$
- $b \xleftarrow{\$} \{0,1\}$

corr-$\mathbf{P_1}$
- **req** allow-corr or $\mathsf{finished}_{\mathsf{P}_1}$
- **return** $\gamma^{\mathsf{P}_1}_{t_{\mathsf{P}_1}}$

send-$\mathbf{P_1}$
- t_{P_1} ++
- $(\gamma^{\mathsf{P}_1}_{t_{\mathsf{P}_1}}, T, I) \xleftarrow{\$} \mathsf{CKA\text{-}S}(\gamma^{\mathsf{P}_1}_{t_{\mathsf{P}_1}})$
- **return** (T, I)

send-$\mathbf{P_1}$' (r)
- t_{P_1} ++
- **req** allow-corr
- $(\gamma^{\mathsf{P}_1}_{t_{\mathsf{P}_1}}, T, I) \leftarrow \mathsf{CKA\text{-}S}(\gamma^{\mathsf{P}_1}_{t_{\mathsf{P}_1}-1}; r)$
- **return** (T, I)

receive-$\mathbf{P_1}$
- t_{P_1} ++
- $(\gamma^{\mathsf{P}_1}_{t_{\mathsf{P}_1}}, *) \leftarrow \mathsf{CKA\text{-}R}(\gamma^{\mathsf{P}_1}_{t_{\mathsf{P}_1}-1}, T)$

chall-$\mathbf{P_1}$
- t_{P_1} ++
- **req** $t_{\mathsf{P}_1} = t^*$
- $(\gamma^{\mathsf{P}_1}_{t_{\mathsf{P}_1}}, T, I) \xleftarrow{\$} \mathsf{CKA\text{-}S}(\gamma^{\mathsf{P}_1}_{t_{\mathsf{P}_1}-1})$
- **if** $b = 0$
  - **return** $(T_{t_{\mathsf{P}_1}}, I_{t_{\mathsf{P}_1}})$
- **else**
  - $I \xleftarrow{\$} \mathcal{I}$
  - **return** $(T_{t_{\mathsf{P}_1}}, I)$

$\mathsf{allow\text{-}corr}_{\mathsf{P}_1}, :\Longleftrightarrow \max(t_{\mathsf{P}_1}, t_{\mathsf{P}_2}) \leq t^* - 2$
$\mathsf{finished}_{\mathsf{P}_1}, :\Longleftrightarrow t_{\mathsf{P}} \geq t^* + \Delta_{\mathsf{CKA}}$

Figure 12: Oracles corresponding to party P_1 of the CKA security game for a scheme $\mathsf{CKA} = (\mathsf{CKA\text{-}Init\text{-}P_1}, \mathsf{CKA\text{-}Init\text{-}P_2}, \mathsf{CKA\text{-}S}, \mathsf{CKA\text{-}R})$; the oracles for P_2 are defined analogously.

less, we stick to the game-based notion analogous to ACD. We present ACD's game-based definition in Figure 12, which is adjusted to our notations of parties $\mathsf{P}_1/\mathsf{P}_2$ in contrast with ACD's $\mathbf{A}/\mathbf{B}$. The differences with our CKA definition are discussed in Section 4.3. Among them, two main differences are: (i) their definition is a indistinguishability-based definition and hence has a challenge bit in the game, whereas our definition (c.f. Definition 9) is a recoverability-based definition; (ii) their definition is parameterized by a Δ_{CKA} for the recovery period after a corruption whereas we capture recovery in a more fine-grained manner without such parameter. The parameter Δ_{CKA} stands for the number of epochs that need to pass after the challenge epoch t^* until the states do not contain secret information pertaining to the challenge. Once a party reaches epoch $t^* + \Delta_{\mathsf{CKA}}$, its state may be revealed to the attacker (via the corresponding corruption oracle). The game ends (implicitly captured above) once both states are revealed after the challenge phase. The attacker wins the game if it eventually outputs a bit $b' = b$. The advantage of an attacker $\mathcal{A}$ against the above game is denoted by $\mathsf{Adv}^{\mathsf{CKA}}_{\mathsf{ACD},\Delta_{\mathsf{CKA}}}(\mathcal{A})$. We define the ACD-security as follows:

Definition 15. *A CKA scheme* CKA *is* $(t, \Delta_{\mathsf{CKA}}, \varepsilon)$*-ACD-secure if for all attackers* $\mathcal{A}$*:*

$$\mathsf{Adv}^{\mathsf{CKA}}_{\mathsf{ACD},\Delta_{\mathsf{CKA}}}(\mathcal{A}) \leq \varepsilon$$

and whenever $t \in \mathrm{poly}(\kappa)$ *and* $\varepsilon \in \mathrm{negl}(\kappa)$ *then we say that the scheme is simply* ACD-secure *with* Δ_{CKA}*.*

For more details regarding the ACD's CKA definition we refer the reader to Section 4.1 of the full version of ACD [ACD18].

ACD's Security Game for FS-AEAD

init
- $k \xleftarrow{\$} \mathcal{K}$
- $v_{\mathsf{P}_1} \leftarrow \mathsf{FS\text{-}Init\text{-}S}(k)$
- $v_{\mathsf{P}_2} \leftarrow \mathsf{FS\text{-}Init\text{-}R}(k)$
- $i_{\mathsf{P}_1} \leftarrow 0$
- $\mathsf{corr}_{\mathsf{P}_1} \leftarrow \mathsf{false}$
- $\mathsf{trans}, \mathsf{chall}, \mathsf{comp} \leftarrow \emptyset$
- $b \xleftarrow{\$} \{0,1\}$

corr-P$_1$
- $\mathsf{corr}_{\mathsf{P}_1} \leftarrow \mathsf{true}$
- **return** v_{P_1}

corr-P$_2$
- **req** $\mathsf{chall} = \emptyset$
- **end** $(v_{\mathsf{P}_1}, v_{\mathsf{P}_2})$

transmit-P$_1$ (a, m)
- i_{P_1} ++
- $(v_{\mathsf{P}_1}, e) \leftarrow \mathsf{FS\text{-}Send}(v_{\mathsf{P}_1}, a, m)$
- **record**(good, a, m, e)
- **return** e

chall-P$_1$ (a, m_0, m_1)
- **req** $\neg\mathsf{corr}_{\mathsf{P}_1}$ and $|m_0| = |m_1|$
- i_{P_1} ++
- $(v_{\mathsf{P}_1}, e) \leftarrow \mathsf{FS\text{-}Send}(v_{\mathsf{P}_1}, a, m_b)$
- **record**$(\mathsf{chall}, a, m_b, e)$
- **return** e

deliver-P$_2$ (a, e)
- **req** $(i, a, m, e) \in \mathsf{trans}$ for some i, m
- $(v_{\mathsf{P}_2}, i', m') \leftarrow \mathsf{FS\text{-}Rcv}(v_{\mathsf{P}_2}, a, e)$
- **if** $(i', m') \neq (i, m)$
  - **win**
- **if** $(i, m) \in \mathsf{chall}$
  - $m' \leftarrow \perp$
- **delete**(i)
- **return** (i', m')

inject-P$_2$ (a, e)
- **req** $(a, e) \notin \mathsf{trans}$
- $(v_{\mathsf{P}_2}, i', m') \leftarrow \mathsf{FS\text{-}Rcv}(v_{\mathsf{P}_2}, a, e)$
- **if** $m' \neq \perp$ *and* $i' \notin \mathsf{comp}$
  - **win**
- **delete**(i')
- **return** (i', m')

---

record (flag, a, m, e)
- $\mathsf{rec} \leftarrow (i_{\mathsf{P}_1}, a, m, e)$
- $\mathsf{trans} \xleftarrow{+} \mathsf{rec}$
- **if** $\mathsf{corr}_{\mathsf{P}_1}$
  - $\mathsf{comp} \xleftarrow{+} \mathsf{rec}$
- **if** $\mathsf{flag} = \mathsf{chall}$
  - $\mathsf{chall} \xleftarrow{+} \mathsf{rec}$

delete (i)
- $\mathsf{rec} \leftarrow (i, a, m, e)$ for m, a, e s.t. $(i, a, m, e) \in \mathsf{trans}$
- $\mathsf{trans}, \mathsf{chall}, \mathsf{comp} \xleftarrow{-} \mathsf{rec}$

Figure 13: Oracles corresponding to party P_1 of the FS-AEAD security game for a scheme $\mathsf{FS\text{-}AEAD} = (\mathsf{FS\text{-}Init\text{-}S}, \mathsf{FS\text{-}Init\text{-}R}, \mathsf{FS\text{-}Send}, \mathsf{FS\text{-}Rcv})$; the oracles for P_2 are defined analogously.

A.1.3 ACD's FS-AEAD Definition

The syntax and the correctness of ACD's FS-AEAD is almost the same as ours (c.f. Definition 3) except that our FS-Stop algorithm is slightly different from theirs. However, similar to ACD we do not require explicit definitions of FS-Stop in this section. The difference comes up later while describing transformation T_1 – we defer this discussion until Appendix A.2.2. Our security notion diverges here as well although we stick to the game-based notion analogous to ACD. We present ACD's game-based definition of FS-AEAD in Figure 13, which is adjusted to our notations of parties $\mathsf{P}_1/\mathsf{P}_2$ in contrast with ACD's $\mathbf{A}/\mathbf{B}$. We discuss the differences between our definition and ACD's definition in Section 4.2. In short, the main difference is that the adversary has much stronger corruption abilities. The advantage of an attacker $\mathcal{A}$ against an FS-AEAD scheme FS-AEAD is denoted by the expression $\mathsf{Adv}^{\mathsf{FS\text{-}AEAD}}_{\mathsf{ACD}}(\mathcal{A})$. The attacker is parameterized by its running time t and the total number of queries q it makes.

Definition 16. *An FS-AEAD scheme* FS-AEAD *is* (t, q, ε)*-ACD-secure if for all* (t, q)*-attackers* $\mathcal{A}$*,*

$$\mathsf{Adv}^{\mathsf{FS\text{-}AEAD}}_{\mathsf{ACD}}(\mathcal{A}) \leq \varepsilon$$

and whenever $t, q \in \mathrm{poly}(\kappa)$ *and* $\varepsilon \in \mathrm{negl}(\kappa)$ *we simply say that* FS-AEAD *is ACD-secure.*

For more details regarding the ACD's FS-AEAD definition we refer the reader to Section 4.2 of the full version of ACD [ACD18].

A.1.4 ACD's PRF-PRNG Defintion

Instead of modelling the function which defines the root KDF chain as a random oracle (as we do), ACD propose a primitive called PRF-PRNG, which resembles both a pseudo-random function (PRF) and a pseudorandom number generator with input (PRNG). They also provide a (deterministic) construction in the standard model for PRF-PRNGs. In Section 4.1.1, we explain the insufficiency of a standard model construction of PRF-PRNGs for realizing our $\mathcal{F}_{\mathsf{DR}}$ ideal functionality. Namely, the adversary's unrestricted ability to leak parties' states requires programming a random oracle to explain these states, and further to provide non-malleability properties (as in CCA-security for hashed ElGamal [ABR01, CS03, KM04]).

It is trivial to see that a PRF-PRNG can be modelled as a random oracle in the ACD composition theorem which we include in the next section. For more details regarding ACD's PRF-PRNG definition, we refer the reader to Section 4.3 of the full version of ACD [ACD18].

A.1.5 ACD's Composition

Now, deriving from the concrete version of ACD's composition theorem we state the asymptotic version of ACD's composition theorem (adjusted for our notations/syntax and modelling of the PRF-PRNG as a random oracle).

Theorem 6. *Assume that:*

- CKA *is a ACD-secure CKA scheme with* Δ_{SM}
- FS-AEAD *is a ACD-secure FS-AEAD scheme, and*
- H *is modelled as a random oracle.*

Then the DR *protocol (c.f. Figure 7) is ACD-secure with* $\Delta_{\mathsf{SM}} = 2 + \Delta_{\mathsf{CKA}}$.

We will be using this to show security of the transformed protocols. For more details we refer to Section 5.3 of the full version of ACD [ACD18].

A.2 Transformations to the DR and Their (In)Security

Now we are ready to present the six transformations, each of which we show to be insecure/incorrect according to our weaker ideal functionality $\mathcal{F}_{\mathsf{DR}}$, but secure with respect to ACD's (Def. 13) and/or CCD$^+$'s [CCD$^+$20] definition. Note that we make the same modifications CCD$^+$ does, as necessitated by their security model, in their analysis of the DR, when analyzing the security of the transformed protocols (i.e., we remove data messages from the protocol, and send everything else in plaintext, instead of inside the associated data of the AEAD encryption of a message).

The transformed protocol $T_1(\mathsf{DR})$

$\mathsf{Init\text{-}P}_1\,(k^{\mathsf{P}_1})$
- $(\sigma_{\mathsf{root}}, \gamma) \leftarrow k^{\mathsf{P}_1}$
- $v[\cdot] \leftarrow \perp$
- $T_{\mathsf{cur}} \leftarrow \perp$
- $\ell_{\mathsf{prv}} \leftarrow 0$
- $t_{\mathsf{P}_1} \leftarrow 0$

$\mathsf{Send\text{-}P}_1\,(m)$
- **if** t_{P_1} *is even*
  - $\ell_{\mathsf{prv}} \leftarrow \mathsf{FS\text{-}Stop}(v[t_{\mathsf{P}_1} - 1])$
  - t_{P_1} ++
  - $(\gamma, T_{\mathsf{cur}}, I) \xleftarrow{\$} \mathsf{CKA\text{-}S}(\gamma)$
  - $(\sigma_{\mathsf{root}}, k) \leftarrow \mathsf{H}(\sigma_{\mathsf{root}}, I)$
  - $v[t_{\mathsf{P}_1}] \leftarrow \mathsf{FS\text{-}Init\text{-}S}(k)$
- $h \leftarrow (t_{\mathsf{P}_1}, T_{\mathsf{cur}}, \ell_{\mathsf{prv}})$
- $(v[t_{\mathsf{P}_1}], e) \leftarrow \mathsf{FS\text{-}Send}(v[t_{\mathsf{P}_1}], h, m)$
- **return** (h, e)

$\mathsf{Rcv\text{-}P}_1\,(c)$
- $(h, e) \leftarrow c$
- $(t, T, \ell) \leftarrow h$
- **req** t even and $t \leq t_{\mathsf{P}_1} + 1$
- **if** $t = t_{\mathsf{P}_1} + 1$
  - ~~$\ell_{\mathsf{prv}} \leftarrow \mathsf{FS\text{-}Stop}(v[t_{\mathsf{P}_1}])$~~
  - t_{P_1} ++
  - $\mathsf{FS\text{-}Max}(v[t-2], \ell)$
  - $(\gamma, I) \leftarrow \mathsf{CKA\text{-}R}(\gamma, T)$
  - $(\sigma_{\mathsf{root}}, k) \leftarrow \mathsf{H}(\sigma_{\mathsf{root}}, I)$
  - $v[t] \leftarrow \mathsf{FS\text{-}Init\text{-}R}(k)$
- $(v[t], i, m) \leftarrow \mathsf{FS\text{-}Rcv}(v[t], h, e)$
- **if** $m = \perp$
  - **error**
- **return** (t, i, m)

Figure 14: DR-based secure-messaging scheme as described by ACD in Page 30 of [ACD18] where the PRF-PRNG algorithms are replaced with hash functions. The differences with our version of the DR are highlighted in blue: basically the only difference is that FS-Stop is called inside the sending algorithm in ACD instead of inside the receiving algorithm in our protocol. The figure only shows the algorithms for P_1; P_2's algorithms are analogous, with "even" replaced by "odd".

A.2.1 T_1: Postponed FS-AEAD Key Deletion

In this section we present a transformation that slightly modifies the usage of FS-AEAD scheme in the DR protocol. This transformation alters the timing of the deletion of FS-AEAD states. In particular, instead of deleting the (old) FS-AEAD secret state (a.k.a. sending chain) when switching from a sending epoch to a receiving epoch, this protocol does it when the next sending epoch is started. The transformed protocol $T_1(\mathsf{DR})$ is described in Figure 14. Remarkably, ACD's presentation of the "Signal-based Secure-messaging protocol" is actually the same as this transformed protocol; their composition theorem already proved (c.f. Theorem 6) that this protocol is secure according to their definitions. However, it turns out that it is insecure according to our ideal functionality $\mathcal{F}_{\mathsf{DR}}$ (it is also insecure according to CCD$^+$'s notion). We show this by presenting an "injection attack" which is formalized below.

Lemma 5. *Suppose that the protocol $T_1(\mathsf{DR})$ is instantiated with a correct and secure CKA scheme (with any Δ_{CKA}) and a correct and secure FS-AEAD scheme. Then there exists a PPT adversary against $T_1(\mathsf{DR})$ protocol in the real world for which there is no PPT simulator that realizes the functionality $\mathcal{F}_{\mathsf{DR}}$ in the ideal world.*

Proof. We propose an explicit attack strategy for an environment and a PPT adversary in the real world. We exploit the fact that in protocol $T_1(\mathsf{DR})$ (alternatively ACD's DR-based secure-messaging protocol), parties do not update the sending chain until the next call to Send. This means that if a party is compromised within the receiving epoch t (which is in-between two sending epochs $t-1$ and $t+1$), it leaks secrets pertaining to the immediately past sending epoch $t-1$. We formalize this strategy as follows:

1. Once the setup is completed send a message m_1 from P_1 in epoch-1. Get the ciphertext c_1.
2. Deliver the ciphertext $c_1 = (h, e)$ to P_2 where $h = (t, T, \ell)$.
3. Send a message m_2 from P_2. Get the ciphertext c_2 in epoch-2
4. Deliver the ciphertext c_2 to P_1.
5. Compromise P_1. Get the secret state which includes $v[1]$.
6. Choose an arbitrary m' and then use $v[1]$ to compute $(v[1]', e') \leftarrow \mathsf{FS\text{-}Send}(v[1], h, m')$.
7. Send m' on behalf of P_1 using the injection ciphertext $c' = (h, e')$.
8. Finally, deliver c' to P_2.

Let us now explain **how the attack works and why it cannot be simulated**. Observe that P_2 accepts the *injected* message m' by executing $\mathsf{Rcv\text{-}P}_1(c')$ due to the correctness of the underlying schemes. The simulator cannot make this delivery successfully because this message m' was not in the transmission list $\mathsf{P}_1.\mathtt{T}$ and hence will be skipped by the ideal functionality in the first step of the delivery. This concludes the proof. □

Note that, we **do not need to prove** that $T_1(\mathsf{DR})$ is secure, when instantiated with secure $\mathsf{FS\text{-}AEAD}$ and CKA schemes, as in Theorem 6, because this was already proven by ACD.

A.2.2 T_2: Postponed CKA Key Deletion

We present our second transformation T_2 here, which slightly modifies the CKA scheme in the DR protocol. In particular, the modification holds on to the shared key derived by the $\mathsf{CKA\text{-}R}$ algorithm for the entire receiving epoch and is deleted only when the next time $\mathsf{CKA\text{-}S}$ is run. Given any CKA scheme for party P ($\mathsf{CKA\text{-}Init\text{-}P}, \mathsf{CKA\text{-}S}, \mathsf{CKA\text{-}R}$) we define the modified scheme ($\mathsf{CKA\text{-}Init\text{-}P}'$, $\mathsf{CKA\text{-}S}', \mathsf{CKA\text{-}R}'$) as:

- $\mathsf{CKA\text{-}Init\text{-}P}'$ is the same as $\mathsf{CKA\text{-}Init\text{-}P}$
- $\mathsf{CKA\text{-}S}'$ takes a state γ and then:
  1. parses γ as $(I_{\mathsf{prv}}, \gamma^\star)$ and set $I_{\mathsf{prv}} \leftarrow \perp$;
  2. runs $(\gamma', T, I) \xleftarrow{\$} \mathsf{CKA\text{-}S}(\gamma^\star)$;

  and then it outputs (γ', T, I)
- $\mathsf{CKA\text{-}R}'$ takes inputs (γ, T) and then:
  1. run $(\gamma^\star, I) \leftarrow \mathsf{CKA\text{-}R}(\gamma, T)$;
  2. $I_{\mathsf{prv}} \leftarrow I$
  3. $\gamma' \leftarrow (I_{\mathsf{prv}}, \gamma^\star)$

  and outputs (γ', I)

The transformed protocol $T_2(\mathsf{DR})$ can be obtained just by replacing the CKA algorithms with the modified ones and hence a detailed description is omitted. We now prove that this modified protocol is insecure in our framework (it is also insecure in CCD$^+$'s framework).

Lemma 6. *Suppose that the T_2-modified CKA scheme described above is instantiated with a secure* CKA *(Def. 9) scheme. Moreover, consider that the protocol $T_2(\mathsf{DR})$ is instantiated with a secure* FS-AEAD *scheme (Def. 5). Then there exists a PPT adversary against the $T_2(\mathsf{DR})$ protocol for which there is no PPT simulator that realizes the functionality $\mathcal{F}_{\mathsf{DR}}$ in the ideal world.*

Proof. The attacker's strategy is to compromise the sender right before a message is sent and then compromise the receiver in the next epoch. Using the information obtained from both compromises the adversary can recover all messages sent in this epoch – this information can not be simulated in the ideal world to realize the ideal functionality $\mathcal{F}_{\mathsf{DR}}$. The attacker's strategy works as follows.

1. Send a message m_1 from party P_1 to P_2 (with good randomness) and deliver it to P_2 in epoch-1.
2. Compromise P_2 and obtain σ_{root}.
3. In epoch 2, send a message m_2 from P_2 to P_1 (with good randomness) and deliver it to P_1. Get the CKA message T and ciphertext $c = (h, e)$.
4. Corrupt P_1 and obtain its state that includes $\gamma = (I_{\mathsf{prv}}, \gamma^\star)$.
5. Then recover m_2 as follows:
   - $\ldots, k \leftarrow \mathsf{H}(\sigma_{\mathsf{root}}, I_{\mathsf{prv}})$;
   - $v[2] \leftarrow \mathsf{FS\text{-}Init\text{-}S}(k)$;
   - $\ldots, m_2 \leftarrow \mathsf{FS\text{-}Rcv}(v[2], h, e)$

We argue **why this attack is possible**. First, observe that the new step added to the protocol in $\mathsf{CKA\text{-}R}'$ (highlighted in blue) stores the previous secret CKA key, I into a variable I_{prv} before this is overwritten (and thus deleted) by $(\gamma, T, I) \leftarrow \mathsf{CKA\text{-}S}(\gamma)$ in the $\mathsf{Send\text{-}P_1}$ algorithm. Furthermore, (i) the root-key σ_{root} that is obtained by the first compromise of P_2 is the same as the root key σ_{root} used by the receiver's algorithm $\mathsf{Rcv\text{-}P_1}$ and (ii) I_{prv} obtained by the compromise of P_1 is the same as the I used to encrypt m by P_2 due to the correctness of the underlying primitives. Therefore the plaintext can be derived correctly by the attacker.

Finally let us argue **how this can not be simulated**. To see this, observe that ciphertext c is delivered and there are no further corruptions besides the one on P_2 before she sends c, and the one on P_1 after she receives c. The first leak of course does not reveal m_2 to the simulator (since it has not yet been sent). More formally, from the description of $\mathcal{F}_{\mathsf{DR}}$ we observe that when the epoch changes, that is $\mathtt{New}(\mathsf{P}_2, \mathtt{TURN}, \mathsf{P}_2.\mathtt{T})$ returns 1, then the flag $\mathsf{P}_2.\mathtt{CUR_SLEK}$ is reset to 0 in Step 3 of Figure 3 – this is because P_1 and P_2 use good randomness while starting epochs 1 and 2 so that even with $\mathsf{P}_1.\mathtt{CUR_SLEK} = 1$, both $\mathsf{P}_1.\mathtt{PLEK} = \mathsf{P}_2.\mathtt{PLEK} = 0$. Also, the second leak does not reveal m_2: c is delivered so m_2 is no longer in $\mathsf{P}_2.\mathtt{T}$ and m_2 is never in $\mathsf{P}_1.\mathtt{V}$ (it is in $\mathsf{P}_2.\mathtt{V}$ which is not leaked to the simulator). Thus the simulator only knows the length of m_2 and cannot properly simulate its leakage in the real world. □

Lemma 7. *Suppose that the T_2 modified CKA scheme CKA' is instantiated with an ACD-secure CKA scheme with $\Delta_{\mathsf{CKA}} \geq 1$. Also assume that $T_2(\mathsf{DR})$ is instantiated with a secure $\mathsf{FS\text{-}AEAD}$ scheme. Then $T_2(\mathsf{DR})$ is a ACD-secure secure-messaging scheme with $\Delta_{\mathsf{SM}} = \Delta_{\mathsf{CKA}} + 2$.*

Proof. We show that if there exists a PPT adversary $\mathcal{A}$ with non-negligible advantage against the modified CKA scheme, then we can construct a PPT adversary (or reduction) $\mathcal{B}$ that breaks the underlying CKA scheme with non-negligible advantage. Both security games are played with respect to the ACD security game (c.f. Definition 15). Taking a closer look we observe that all queries from $\mathcal{A}$ can be simulated by $\mathcal{B}$ using the challenger, except a corruption query following a receive query. The state at this time additionally contains I_{prv}. Nevertheless, for any epoch $t \neq t^*$, I_{prv} is just obtained by making a send-query which returns $I = I_{\mathsf{prv}}$. But if the query sequence is $\mathsf{chall\text{-}P}_1 \longrightarrow \mathsf{receive\text{-}P}_2 \longrightarrow \mathsf{corr\text{-}P}_2$, then I_{prv} is not always obtained (because when $b = 1$ a random value is obtained instead). However, due to the restriction $\Delta_{\mathsf{CKA}} \geq 1$ such corruption query would not return anything because both $\mathsf{allow\text{-}corr}$ and $\mathsf{finished}_{\mathsf{P}_2}$ return 0 since $t_{\mathsf{P}_2} = t^*$. This concludes the proof. □

A.2.3 T_3 : Eager CKA Randomness Sampling

This transformation slightly modifies how the CKA scheme is used in the DR protocol without making any change to the CKA itself. In particular, the modification samples the randomness that is to be used in the CKA-S algorithm of the next sending epoch while it is still in the prior receiving epoch. We note that this transformation reflects the primary description of the Double Ratchet algorithm in its white paper (in the main body) [MP16a], and also CCD$^+$'s presentation. However, Perrin and Marlinspike later state that better security can be achieved if the randomness is indeed sampled within the actual sending epoch (as in our protocol). We present the modified protocol $T_3(\mathsf{DR})$ in Figure 15 and prove that this is insecure with respect to our ideal functionality $\mathcal{F}_{\mathsf{DR}}$.

We now prove the following:

Lemma 8. *Suppose that the protocol $T_3(\mathsf{DR})$ is instantiated with a secure CKA (Def. 9) scheme and a secure $\mathsf{FS\text{-}AEAD}$ scheme (Def. 5). Then there exists a PPT adversary against the $T_3(\mathsf{DR})$ protocol for which there is no PPT simulator that realizes the functionality $\mathcal{F}_{\mathsf{DR}}$ in the ideal world.*

Proof. We propose an explicit attack strategy for an environment and an adversary in the real world. The adversary never instructs the parties to use bad randomness.

1. Send a message m_1 from P_1. Get the ciphertext c_1.
2. Deliver the ciphertext c_1 to P_2.
3. Compromise P_2. Get the secret state which includes $r, \gamma^{\mathsf{P}_2}, \sigma_{\mathsf{root}}$.
4. Send a message m_2 from P_2. Get the ciphertext $c_2 = (h_2, e_2)$.
5. Use γ and r and then compute:
   (a) $(\ldots, I) \leftarrow \mathsf{CKA\text{-}S}(\gamma^{\mathsf{P}_2}; r)$;
   (b) $(\ldots, k) \leftarrow \mathsf{H}(\sigma_{\mathsf{root}}, I)$;
   (c) $v[2] \leftarrow \mathsf{FS\text{-}Init\text{-}R}(k)$;
   (d) $(\ldots, m_2) \leftarrow \mathsf{FS\text{-}Rcv}(v[2], h_2, e_2)$;

The transformed protocol $T_3(\mathsf{DR})$

$\mathsf{Init\text{-}P}_1(k^{\mathsf{P}_1})$
- $(\sigma_{\mathsf{root}}, \gamma) \leftarrow k^{\mathsf{P}_1}$
- $v[\cdot] \leftarrow \bot$
- $T_{\mathsf{cur}} \leftarrow \bot$
- $\ell_{\mathsf{prv}} \leftarrow 0$
- $t_{\mathsf{P}_1} \leftarrow 0$
- $r \xleftarrow{\$} \mathcal{R}$

$\mathsf{Send\text{-}P}_1(m)$
- **if** t_{P_1} *is even*
  - t_{P_1} ++
  - $(\gamma, T_{\mathsf{cur}}, I) \leftarrow \mathsf{CKA\text{-}S}(\gamma; r)$
  - $(\sigma_{\mathsf{root}}, k) \leftarrow \mathsf{H}(\sigma_{\mathsf{root}}, I)$
  - $v[t_{\mathsf{P}_1}] \leftarrow \mathsf{FS\text{-}Init\text{-}S}(k)$
- $h \leftarrow (t_{\mathsf{P}_1}, T_{\mathsf{cur}}, \ell_{\mathsf{prv}})$
- $(v[t_{\mathsf{P}_1}], e) \leftarrow \mathsf{FS\text{-}Send}(v[t_{\mathsf{P}_1}], h, m)$
- **return** (h, e)

$\mathsf{Rcv\text{-}P}_1(c)$
- $(h, e) \leftarrow c$
- $(t, T, \ell) \leftarrow h$
- **req** t even and $t \leq t_{\mathsf{P}_1} + 1$
- **if** $t = t_{\mathsf{P}_1} + 1$
  - $\ell_{\mathsf{prv}} \leftarrow \mathsf{FS\text{-}Stop}(v[t_{\mathsf{P}_1}])$
  - t_{P_1} ++
  - $\mathsf{FS\text{-}Max}(v[t-1], \ell)$
  - $(\gamma, I) \leftarrow \mathsf{CKA\text{-}R}(\gamma, T)$
  - $r \xleftarrow{\$} \mathcal{R}$
  - $(\sigma_{\mathsf{root}}, k) \leftarrow \mathsf{H}(\sigma_{\mathsf{root}}, I)$
  - $v[t] \leftarrow \mathsf{FS\text{-}Init\text{-}R}(k)$
- $(v[t], i, m) \leftarrow \mathsf{FS\text{-}Rcv}(v[t], h, e)$
- **if** $m = \bot$
  - **error**
- **return** (t, i, m)

Figure 15: The difference from the DR are highlighted in blue.

6. Output m_2.

It is easy to see that the attack works due to the correctness of the underlying schemes. Furthermore, note that the simulator can not obtain m_2 in the ideal world, because compromising/leaking on P_2 *before* it sends m_2 will not reveal m_2 (as in the proof of Lemma 6). □

Lemma 9. *Suppose that $T_3(\mathsf{DR})$ is instantiated with an ACD-secure CKA scheme with Δ_{CKA} and an ACD-secure $\mathsf{FS\text{-}AEAD}$ scheme. Then $T_3(\mathsf{DR})$ is a ACD-secure secure-messaging scheme with $\Delta_{\mathsf{SM}} = \Delta_{\mathsf{CKA}} + 2$.*

Proof. We construct a reduction $\mathcal{B}$ against a challenger, which is running the secure-messaging game with the original DR protocol; $\mathcal{B}$ uses an adversary $\mathcal{A}$ who is trying to gain a non-negligible advantage in a secure-messaging game running protocol $T_3(\mathsf{DR})$. Note that, the only difference in the two schemes is that in the modified scheme the randomness r is sampled one epoch earlier. Therefore, when a **corr-P** query is received from $\mathcal{A}$ followed by a **deliver**-P query, then $\mathcal{B}$ must simulate the randomness r, which it uses in the next epoch (when P becomes a sender). $\mathcal{B}$ simulates this by sampling a random r and giving that to $\mathcal{A}$. However, in order to do so, it uses the transmit oracle with randomness r: $\mathsf{transmit\text{-}P}(m, r)$. This does not work if $\mathcal{A}$ makes a **chall**-P call right after. However, since $\Delta_{\mathsf{SM}} \geq 2$, this can not happen. Hence, the simulation can be done. The other oracles are simulated straightforwardly by using the challenger as there is no other change in the protocol. This concludes the proof. □

Lemma 10. *Suppose that $T_3(\mathsf{DR})$ is instantiated with CKA from Section 4.3.3 and a secure $\mathsf{FS\text{-}AEAD}$ scheme (Def. 5). Then $T_3(\mathsf{DR})$ is a CCD$^+$-secure secure-messaging scheme.*

Proof. CCD$^+$ indeed choose to model this version of DR. Thus, security of $T_3(\mathsf{DR})$ follows from [CCD+20]. □

Remark 3. *We acknowledge that the version of the* DR *which CCD*$^+$ *analyze is simply a matter of choice. Indeed, they make the more practical choice, as this version is what is typically used in the real-world. However, we simply emphasize that from a cryptographic view-point, our model is stronger.*

A.2.4 T_4: Malleable Ciphertexts

For our proof that the DR UC-realizes $\mathcal{F}_{\mathsf{DR}}$, we crucially assume that the underlying CKA protocol provides non-malleability guarantees. More specifically, recall that in Definition 7 for CKA in this paper, we include the $\mathbf{test}(t, T, I)$ oracle, which outputs 1 if the corresponding receiver in epoch t would on input CKA message T output I; and 0 otherwise. In order to prove that the DR CKA is secure with respect to this definition, we used the StDH assumption.

In this section, we elaborate on the need for such non-malleability, by first showing that the DR building blocks can be used to build PKE, even if some of the secret information of the DR parties is not hidden. Then, we apply T_4 to the DR, which replaces the CKA secure with respect to Definition 7, with one that permits the existence of an adversary that given a ciphertext encrypting m from the above PKE scheme, can successfully maul it into a new ciphertext encrypting $m + 1$. Based on this, we show that $T_4(\mathsf{DR})$ does not UC-realize $\mathcal{F}_{\mathsf{DR}}$.

We further observe that if our derived PKE scheme is instantiated using the building blocks of the real DR protocol, then this PKE is (a variant of) hashed ElGamal. Hashed ElGamal is known to be CPA-secure based on the DDH assumption, but however, is only known to be CCA-secure under the stronger StDH assumption. Thus such malleability attacks could *in theory* exist against the DR. This is why we need to assume StDH-security for our proof that the DR UC-realizes $\mathcal{F}_{\mathsf{DR}}$, and not just DDH-security.

On the other hand, the DR does not need such non-malleability guarantees to achieve ACD-security, since ACD prove it secure based on a CKA definition without such a $\mathbf{test}()$ oracle (for the DR CKA, this is proved using only DDH). Thus, generally, if an instantiation of the DR with a CKA that prevents against malleability is transformed via T_4 into an instantiation of the DR with a CKA that is prone to malleability, then the DR is still ACD-secure but is not secure according to $\mathcal{F}_{\mathsf{DR}}$. The same can be said about CCD$^+$-security, since their security definition only requires key-indistinguishability, and no authenticity nor semantic security guarantees.

More generally, we highlight a gap in the provable security of the DR, based on whether *only* DDH is secure, or if also StDH is secure.

PKE from the DR building blocks with (partially) exposed secrets. Here, we observe that PKE can be constructed from the DR building blocks, even if the root KDF chain key σ_{root} is not hidden. Assume that there are some publicly known values σ and h.

- The generation algorithm $\mathsf{Gen}()$ first samples random CKA initialization key k, then sets $(sk, pk) \leftarrow (\mathsf{CKA\text{-}Init\text{-}P_2}(k), \mathsf{CKA\text{-}Init\text{-}P_1}(k))$.
- The encryption algorithm $\mathsf{Enc}(pk, m)$ first computes $(\cdot, T, I) \leftarrow \mathsf{CKA\text{-}S}(pk)$, then $k \leftarrow \mathsf{H}(\sigma, I)$, $v \leftarrow \mathsf{FS\text{-}Init\text{-}S}(k)$, $(\cdot, e) \leftarrow \mathsf{FS\text{-}Send}(v, (T, h), m)$, and finally $c \leftarrow (T, e)$.
- The decryption algorithm $\mathsf{Dec}(sk, c)$ parses $(c_1, c_2) \leftarrow c$ and computes $(\cdot, I) \leftarrow \mathsf{CKA\text{-}R}(sk, c_1)$, then $k \leftarrow \mathsf{H}(\sigma, I)$, $v' \leftarrow \mathsf{FS\text{-}Init\text{-}R}(k)$, and finally $(\cdot, \cdot, m) = \mathsf{FS\text{-}Rcv}(v', (c_1, h), c_2)$.

One can observe that if the above PKE scheme is instantiated using the CKA of the DR, then the PKE scheme is essentially hashed ElGamal: The secret key is exponent a, the public key is g^a, exponent b (and g^b) is sampled during encryption, then AEAD key k is derived by both the encryptor and decryptor by hashing shared DDH secret g^{ab} (and public information), and finally m is encrypted using the AEAD with key k.

Intuitively, CPA-security of this PKE scheme comes from that of the underlying CKA, since I should be hidden given just the CKA state of the CKA sender before transmitting, and the resulting CKA message that is in the ciphertext (c.f. Section 4.3). However, it is unclear if this PKE scheme provides CCA-security, particularly if the underlying CKA does not acheive the security definition with a **test**() oracle—how does the reduction answer decryption queries? In fact, there could, for example, exist some PPT adversary $\mathcal{A}$ that on input ciphertext from our above PKE scheme, $c \leftarrow \mathsf{Enc}(pk, m)$, and public information $\sigma_{\mathsf{root}}, h$, can with non-negligible probability maul this ciphertext to create c' such that $m + 1 \leftarrow \mathsf{Dec}(sk, c')$.

Malleability attack on the DR. Now assume that $T_4(\mathsf{DR})$ uses a CKA scheme that is secure with respect to the definition of ACD in Definition 15, but for which such an attacker $\mathcal{A}$ above does exist. Then, although $T_4(\mathsf{DR})$ is ACD-secure with this CKA scheme, it does not realize $\mathcal{F}_{\mathsf{DR}}$. If indeed this CKA is the standard CKA used in the DR, then we believe it is also CCD$^+$-secure (even if only DDH is secure, and StDH is not).

Lemma 11. *Assume that malleability adversary $\mathcal{A}$ above exists. Then there exists a PPT adversary against the DR for which there is no PPT simulator that realizes $\mathcal{F}_{\mathsf{DR}}$ in the ideal world.*

Proof. We propose an explicit attack strategy for an environment and an adversary in the real world.

1. After Setup (initialized by P_1), corrupt P_1. This leaks σ_{root} to the attacker.
2. Send a message m_1 from P_1. Get the ciphertext $c_1 = (h_1, e_1)$, $h_1 = (t, \ell, T)$.[9]
3. Compute $c_1' \leftarrow \mathcal{A}((T, e_1), \sigma_{\mathsf{root}}, (t, \ell))$.
4. Deliver c_1' to P_2.
5. When P_2 outputs m_1', the attacker outputs $m_1' - 1$.

Now, since $\mathcal{A}$ is successful with non-negligible probability, then with non-negligible probability, P_2 decrypts c_1' to $m_1 + 1$ and then our attacker outputs m_1.

However, in the ideal world, the simulator does not get m_1 from the functionality and therefore only with negligible probability can create ciphertext c_1 such that P_2 will decrypt it to $m_1 + 1$ and output that. More formally, from the description of $\mathcal{F}_{\mathsf{DR}}$, we observe that when P_1 starts the first epoch, although flag $\mathsf{P}_2.\mathtt{CUR_SLEK}$ is set to 1 after the first leakage, if P_1 has good randomness then both $\mathsf{P}_1.\mathtt{PLEK} = \mathsf{P}_2.\mathtt{PLEK} = 0$, so $\mathsf{P}_1.\mathtt{CUR_SLEK}$ is set to 0 and thus the simulator is only given the length of m. □

Intuitively, this attack cannot be executed according to the ACD definition, since such an injection of mauled ciphertext c_1' after corruption of P_1 is not allowed. More formally, after the

[9] We reorder for simplicity; the attack can easily be performed on the original order as presented in ACD.

The transformed protocol $T_5(\mathsf{DR})$

```
Init-P1 (k^P1)
    (σ_root, γ) ← k^P1
    v[·] ← ⊥
    T_cur ← ⊥
    ℓ_prv ← 0
    t_P1 ← 0
    plain ← 0

Send-P1 (m)
    if t_P1 is even
        t_P1 ++
        r ←$ R
        if r = 0^|r|
            plain ← 1
        (γ, T_cur, I) ← CKA-S(γ; r)
        (σ_root, k) ← H(σ_root, I)
        v[t_P1] ← FS-Init-S(k)
    if plain = 0
        h ← (t_P1, T_cur, ℓ_prv)
        (v[t_P1], e) ← FS-Send(v[t_P1], h, m)
        return (h, e)
    else
        return m

Rcv-P1 (c)
    (h, e) ← c
    (t, T, ℓ) ← h
    req t even and t ≤ t_P1 + 1
    if t = t_P1 + 1
        ℓ_prv ← FS-Stop(v[t_P1])
        t_P1 ++
        FS-Max(v[t − 1], ℓ)
        (γ, I) ← CKA-R(γ, T)
        (σ_root, k) ← H(σ_root, I)
        v[t] ← FS-Init-R(k)
    (v[t], i, m) ← FS-Rcv(v[t], h, e)
    if m = ⊥
        error
    return (t, i, m)
```

Figure 16: The differences from the DR are highlighted in blue.

corruption of P_1 in the ACD defintion, t_{L} is set to 0. Since for the DR, they prove security with respect to $\Delta_{\mathsf{SM}} = 3$, t_{P_2} will still be 0 and thus safe-inj will evaulate to false. Thus, no matter if such a malleability attack can be performed, the same level of security can be proved for the DR according to the ACD definition.

This attack does not allow the adversary to win the CCD$^+$ security game, since their game only explicitly provides key-indistinguishability guarantees (which are not necessarily violated in this attack), not any authenticity, nor semantic security guarantees. Indeed, note that regular ElGamal ciphertexts can easily be mauled without being able to distinguish the key implicitly agreed upon in the ciphertext from random: Given $c = (g^a, g^{ab}m)$, one can simply square both components to obtain $c' = (g^{2a}, g^{2ab}m^2)$ which decrypts to m^2. Thus, key-indistinguishability is still retained (from DDH), but as is well known, authenticity and semantic security guarantees (in the form of CCA security) are violated.

A.2.5 T_5: CKA Bad Randomness Plaintext Trigger

This transformation changes the DR protocol to send everything in plaintext if the all-0 random string is sampled during CKA-S. This highlights the additional power that our (and ACD's) adversary gets in choosing bad randomness for certain messages, as opposed to only randomness reveals of *uniform* randomness as in the model of CCD$^+$. Although the reader might view this transformation as highly artificial, as pointed out by [BRV20], there are real-world attacks that randomness reveals do not capture, but which allowing the adversary to choose bad randomness does capture; for example, attacks against randomness sources (e.g., [HDWH12]) and/or generators (e.g., [CNE+14, YRS+09]). Furthermore this difference has a tangible impact on protocol design, particularly in Secure Messaging, as the results of [BRV20] show.

We present the modified protocol $T_5(\mathsf{DR})$ in Figure 16 and prove that this is insecure with respect to our ideal functionality $\mathcal{F}_{\mathsf{DR}}$ (it is also insecure in ACD's notion).

Lemma 12. *Suppose that the protocol $T_5(\mathsf{DR})$ is instantiated with* CKA *from Section 4.3.3 and a secure* FS-AEAD *scheme (Def. 5). Then there exists a PPT adversary against the $T_5(\mathsf{DR})$ protocol for which there is no PPT simulator that realizes the functionality $\mathcal{F}_{\mathsf{DR}}$ in the ideal world.*

Proof. We propose a simple attack strategy for an environment and an adversary in the real world, utilizing bad randomness.

1. Send a message m from P_1 with bad randomness $0^{|r|}$. Get the ciphertext c.
2. Since the ciphertext c is just m unencrypted, output m.

It is easy to see that the attack works based on the description of $T_5(\mathsf{DR})$. Furthermore, note that the simulator can not obtain m in the ideal world, because solely instructing P_1 to use bad randomness $0^{|r|}$ when it sends m will not reveal m. This is because in Step 3 of Figure 3, only $\mathsf{P}.\texttt{PLEK} \leftarrow 1$ is set, but $\bar{\mathsf{P}}.\texttt{CUR_SLEK} = 0$, so $\mathsf{P}.\texttt{CUR_SLEK} \leftarrow 0$. Thus m is not passed to $\mathcal{S}$. □

Lemma 13. *Suppose that $T_5(\mathsf{DR})$ is instantiated with* CKA *from Section 4.3.3 and the* FS-AEAD *scheme of Figure 5. Then $T_5(\mathsf{DR})$ is a CCD$^+$-secure secure-messaging scheme.*

Proof. Since in the CCD$^+$ security model, Send is always executed with uniform randomness, the probability that the all-0 string is sampled is negligibly small. Thus the real-world game in which $T_5(\mathsf{DR})$ is used is statistically-indistinguishable from a game in which the DR is used, with CKA from Section 4.3.3, which is the version of the DR that CCD$^+$ analyzes. Since this version of the DR is secure according to CCD$^+$, we are done. □

A.2.6 T_6: Removed Immediate Decryption

This transformation changes the DR protocol to only include CKA messages in the first ciphertext of each epoch. Thus, if a party receives the ith ciphertext of an epoch before the first message, they cannot successfully decrypt it. This highlights the fact that the formal model of CCD$^+$ does not consider the notion of immediate decryption (nor correctness more generally). One might argue that this is an easy fix to make to the model of CCD$^+$. However, as [ACD19] explain, immediate decryption is an important practical requirement of the DR, and one of the main strengths of the DR is achieving PCS and FS while still maintaining immediate decryption. Furthermore, properly modelling immediate decryption allows subsequent work to understand it, and further improve upon the DR with the requirement in mind. Indeed, many of the works which we are aware of [BSJ+17, DV17, JS18, JMM19a, PR18], besides [ACD19], which try to improve the DR do not consider immediate decryption in their security models or constructions, arguably thrusting these works outside of the practical realm.

We present the modified $T_6(\mathsf{DR})$ in Figure 17 and prove that it is insecure with respect to our ideal functionality $\mathcal{F}_{\mathsf{DR}}$ (it is also insecure according to ACD's notion).

Lemma 14. *Suppose that the protocol $T_6(\mathsf{DR})$ is instantiated with* CKA *from Section 4.3.3 and the* FS-AEAD *scheme of Figure 5. Then there exists a PPT adversary against the $T_6(\mathsf{DR})$ protocol for which there is no PPT simulator that realizes the functionality $\mathcal{F}_{\mathsf{DR}}$ in the ideal world.*

Proof. We propose a simple attack strategy for an environment and an adversary in the real world, delivering messages out of order.

1. Send two messages m_1, m_2 from P_1 and obtain ciphertexts c_1, c_2.

The transformed protocol $T_6(\mathsf{DR})$

$\mathsf{Init\text{-}P}_1(k^{\mathsf{P}_1})$
- $(\sigma_{\mathsf{root}}, \gamma) \leftarrow k^{\mathsf{P}_1}$
- $v[\cdot] \leftarrow \bot$
- $T_{\mathsf{cur}} \leftarrow \bot$
- $\ell_{\mathsf{prv}} \leftarrow 0$
- $t_{\mathsf{P}_1} \leftarrow 0$

$\mathsf{Send\text{-}P}_1(m)$
- **if** t_{P_1} *is even*
  - t_{P_1} ++
  - $(\gamma, T_{\mathsf{cur}}, I) \leftarrow \mathsf{CKA\text{-}S}(\gamma)$
  - $(\sigma_{\mathsf{root}}, k) \leftarrow \mathsf{H}(\sigma_{\mathsf{root}}, I)$
  - $v[t_{\mathsf{P}_1}] \leftarrow \mathsf{FS\text{-}Init\text{-}S}(k)$
- $h \leftarrow (t_{\mathsf{P}_1}, T_{\mathsf{cur}}, \ell_{\mathsf{prv}})$
- $T_{\mathsf{cur}} \leftarrow \bot$
- $(v[t_{\mathsf{P}_1}], e) \leftarrow \mathsf{FS\text{-}Send}(v[t_{\mathsf{P}_1}], h, m)$
- **return** (h, e)

$\mathsf{Rcv\text{-}P}_1(c)$
- $(h, e) \leftarrow c$
- $(t, T, \ell) \leftarrow h$
- **req** t even and $t \leq t_{\mathsf{P}_1} + 1$
- **if** $t = t_{\mathsf{P}_1} + 1$
  - **if** $T = \bot$
    - **return** $\bot$
  - $\ell_{\mathsf{prv}} \leftarrow \mathsf{FS\text{-}Stop}(v[t_{\mathsf{P}_1}])$
  - t_{P_1} ++
  - $\mathsf{FS\text{-}Max}(v[t-1], \ell)$
  - $(\gamma, I) \leftarrow \mathsf{CKA\text{-}R}(\gamma, T)$
  - $(\sigma_{\mathsf{root}}, k) \leftarrow \mathsf{H}(\sigma_{\mathsf{root}}, I)$
  - $v[t] \leftarrow \mathsf{FS\text{-}Init\text{-}R}(k)$
- $(v[t], i, m) \leftarrow \mathsf{FS\text{-}Rcv}(v[t], h, e)$
- **if** $m = \bot$
  - **error**
- **return** (t, i, m)

Figure 17: The differences from the DR are highlighted in blue.

2. Deliver the second ciphertext c_2 to P_2.

In the real world, P_2 will output $\bot$ after receiving c_2. However, due to the definition of the ideal functionality of Figure 3, the simulator cannot force P_2 to output $\bot$. This is because no corruption has occurred and thus messages can only honestly be delivered. Thus, the simulator can only force P_2 to output m_1 or m_2, not $\bot$, and so the real world and ideal world are easily distinguished. □

Lemma 15. *Suppose that $T_6(\mathsf{DR})$ is instantiated with the* CKA *scheme from Section 4.3.3 and the* FS-AEAD *scheme of Figure 5. Then $T_6(\mathsf{DR})$ is a CCD$^+$-secure secure-messaging scheme.*

Proof. The only thing that changes in $T_6(\mathsf{DR})$ is that the CKA message T is only included in the first ciphertext of an epoch. Thus, for $i \neq 1$, the ith message key and chain key of an epoch (i.e., the i-th AEAD key K_i and seed w_i, respectively, computed by FS-AEAD) can only be computed if the first ciphertext of that epoch was delivered beforehand. In fact, every ciphertext is simply discarded by the protocol if the first ciphertext of the corresponding epoch was not yet delivered. In the language of CCD$^+$, w_i is the *input state* of the *stage* corresponding to the ith message of the epoch. This is what is revealed to the adversary upon corruption of a participant in this *stage*. Note that this input state can only be set if the first ciphertext of the epoch is delivered beforehand, since the stage is identified in part by the CKA message T of the epoch (i.e., the *ratchet public key*), which cannot be known by the receiver until such reception. However, once this first ciphertext is delivered, the protocol proceeds as usual.

Since in the CCD$^+$ security game against $T_6(\mathsf{DR})$, the adversary will receive no more information than it does against the DR, and needs to distinguish the same keys from random, security of $T_6(\mathsf{DR})$ follows from their analysis. □

B The Model in Detail

In this section we provide details of the model, a lot of which is taken from Bitanksy et al. [BCH12].

B.1 UC Security: A Brief Overview

We summarize the UC security framework of Canetti [Can01]. For brevity and simplicity, we describe a somewhat restricted variant á la [BCH12]; still, the summary is intended to provide sufficient detail for verifying the treatment in this work. The description below is taken almost verbatim from [BCH12]. Further elaboration and justification of definitional choices appears in [Can01].

B.1.1 The Basic Model of Computation

We first present the underlying model of computation, which provides the basic mechanics on top of which the notion of protocol security is defined.

Interactive Turing Machines (ITM). The basic computing element is an Interactive Turing Machine (ITM), which represents a program written for a distributed system. The UC framework uses a formalism of an ITM that augments the original formalism of [Gol04, GMR89] with some additional structure, for the purpose of capturing protocols in multi-party, multi-instance systems. Specifically, an ITM is a Turing machine (one may consider any standard definition such as [Sip97]) with the following additional constructs. It has three *special tapes* that represent three different types of information coming from external sources: (i) The *input tape* represents information coming from the "calling protocol" (for now consider a protocol to be just another ITM, we shall formalize it below); (ii) the *communication tape* represents information coming from other parties over untrusted communication links; (iii) the *subroutine output tape* represents information coming from "subroutine protocols" in a trusted way. In addition, an ITM has a special *identity tape* which cannot be written on by the ITM transition function, or program. The contents of the identity tape is interpreted as three values: The program of the ITM, represented in some canonical form; a session-identifier (sid), representing a specific protocol session; and a party identifier (pid), representing an identity of a party within that session. In the context of this work we will use the pid as an indicator of the physical device on which the ITI runs. That is, all the ITIs that represent processes that run on the same physical device (and can thus leak in a correlated way) have the same pid. Finally, to the standard ITM syntax we add the ability to perform an *external write* instruction. The semantics of this instruction are defined below.

Systems of ITMs. Running programs in a distributed system is captured as follows. An *ITM instance* (also called an ITI) $\mu \leftarrow (M, \mathsf{id})$ is an ITM M (alternatively, a program) along with a string $\mathsf{id} \leftarrow (\mathsf{sid}, \mathsf{pid})$, called the identity of μ. An ITI represents a running instance of the ITM M where the identity id is written in its identity tape. A *system of of ITMs* is a pair (M, C) where M is an ITM and $C : \{0,1\}^* \to \{0,1\}^*$ is a control function that determines the effect of the external write commands.

An *execution of a system* (M, C) of ITMs, on input x, consists of a sequence of activations of ITIs. Initially, the system consists of a single ITI with ITM M, some fixed identity (say, $\mathsf{id}_0 = (0,0)$), and x written on the input tape. This ITI, $\mu_0 \leftarrow (M, \mathsf{id}_0)$ is then activated.

In each activation of an ITI, the active ITI runs its program. The execution ends when the initial ITI μ_0 halts. The output of the execution is the output of the initial ITI.

It remains to specify the effect of the external-write operation of an arbitrary ITI $\mu^\star$, that is active at any given point. This operation specifies three things: (i) a target ITI μ (say); (ii) one of its tapes $\tau \in \{\text{input, communication, subroutine output}\}$; (iii) and the data δ to be written. When an external-write operation is carried out, the control function C is applied to the sequence of external write requests in the execution so far. Then:

1. If C returns 1 then:
   (a) If an ITI with the same identity as the target ITI does not exist in the system then a new ITI with the given specification μ.
   (b) The data δ is written to the specified tape τ of ITI μ (this is uniquely determined).

   The active ITI $\mu^\star$ becomes inactive and the target ITI μ becomes active.
2. If C returns 0 *or* the ITI $\mu^\star$ halts, then the initial ITI μ_0 is activated.
3. If C returns another value, parse that as a description of an ITM M. The effect is the same as in Case-1 except, the program of μ is replaced by M instead of the original value specified in the command.

Subroutines. An ITI μ_{sub} is a *subroutine* of another ITI μ_{main} in an execution if μ_{main} wrote to the input tape of μ_{sub} *or* μ_{sub} wrote to the subroutine output tape of μ_{main}.

Protocols and protocol instances. A protocol π is formalized as *a single ITM*, that represents the programs to be run by all the intended participants. A protocol may specify different roles, in that case the single ITM describes the programs for all the roles and the role is given to the specific party as part of an input. For example, in a secure message transmission protocol, the sender and the receiver has different roles and therefore run different programs. An *instance* (or session) of a protocol π with session identifier sid, within a system of ITMs, is the set of ITIs that run the program π and whose session identifier is sid.

Polyonomial Time ITMs. We consider ITMs that run in probabilistic polynomial time (PPT), where PPT is defined as follows: an ITM M is PPT if there exists a constant $c > 0$ such that, for any ITI μ with program M, at any point during its run, and for any contents of the random tape, the overall number of steps taken is at most n^c, where n is the overall number of bits written on the input tape of μ *minus* the overall number of bits written by μ to input tapes of other ITIs. An execution of a system of ITM is said to be run in PPT if the initial ITM is PPT and the control function is computable in probabilistic polynomial time by any TM.

B.1.2 Security of Protocols.

Recall that real world protocols that securely implement a given task are defined via comparison with an ideal process for carrying out the task. Formalizing this notion is done in several steps, as follows. First, we define the process of executing a protocol in the presence of an adversarial environment. We then define what it means for one protocol to "emulate" another protocol. Next,

we define the ideal functionality for carrying out the task. A protocol is said to securely carry out the task if it emulates the idealized protocol for that task.

The model for protocol execution. The model for executing a protocol π is parametrized by a security parameter $\kappa \in \mathbb{N}$, and three ITMs: the protocol implementation π an adversary $\mathcal{A}$, which represents the adversarial activity against a *single instance* of π, and an environment $\mathcal{Z}$, which represents the rest of the system. Specifically, to *execute the protocol* π on input x, execute the system of ITMs $(\mathcal{Z}, C_{\mathcal{A},\pi})$, $C_{\mathcal{A},\pi}$ being the control function for the protocol π in presence of an adversary $\mathcal{A}$ — it remains to describe this control function, namely the external write capabilities of each ITI.

In essence, the definition of $C_{\mathcal{A},\pi}$ captures a model where *a single instance of* π interacts with $\mathcal{Z}$ and $\mathcal{A}$, in that $\mathcal{Z}$ controls the inputs to parties and reads the outputs. All communication (via the communication tapes) *must pass through* $\mathcal{A}$. In addition, the parties of π can create subroutine ITIs, can write to the input tapes of the subroutines, and receive outputs from the subroutine on the subroutine output tapes of the calling parties. More precisely:

- *External writes by the environment:* The environment can write only to the tapes of other ITIs. The program of the first ITI invoked by the environment is set (by the control function) to be the program of the adversary $\mathcal{A}$. The programs of all the other ITIs that the environment writes to are set to be the protocol π. In addition, the SIDs of all the ITIs invoked by the environment (except $\mathcal{A}$) must be the same, which implies that all of those ITIs with the same sid belong to the *same instance of* π.

- *External writes by the adversary:* The adversary can write only to the communication tapes of ITIs. In addition, it is *not allowed* to create new ITIs; namely, if the adversary performs an external write request with non-existing target ITI, the control function returns 0.

- *External writes by other ITIs:* An ITI μ other than the environment and the adversary can write only to: (i) the subroutine output tapes of ITIs that have previously written to the input tape of μ; (ii) the input tapes of ITIs that μ has invoked; (iii) and to the input tapes of ITIs with the same session ID as μ. In addition, it can write to the communication tape of the adversary. (Writing to input tapes of ITIs is the same SID will become useful when defining ideal protocols.)

We also use the convention that creation of a new ITI must be done by writing to the input tape of that ITI; the data written in this activation must start with 1^κ , where κ is the security parameter.

Modeling party corruption specific to our setting. Since the modeling of party corruption will be central to our modeling of *leakage* and *bad randomness*, we describe it in more detail. Formally, party corruption is modeled as a special message sent by the adversary to the corrupted party (ITI). Different types of corruptions (e.g., passive, Byzantine) are modeled as parameters in the corruption message. The response of a party (ITI) to an incoming corruption message is formally treated as part of the protocol specification. This modeling has the advantage that general notions and theorems such as UC emulation, the UC theorem, and the universality of the dummy adversary apply regardless of the specific corruption model. However, some additional formalism is necessary in order to make sure that the formal corruption operation corresponds to the generally

accepted intuitive notion of party corruption. Specifically, we assume that an ITM is *corruption compliant* if its program consists of a main program σ (which can be thought of as an "operating system" of sorts), and a subroutine π which represents the actual program run by the ITM. The main program relays all inputs, incoming messages, and subroutine outputs to π, with the exception of the corruption messages sent by the adversary. The behavior of σ upon receipt of the corruption message essentially determine the corruption model.

Let us specify the behavior of σ for two salient types of corruption. In the case of passive party corruption, σ behaves as follows. When an ITI μ receives the first corruption message, the σ part of the code of μ reports that μ has been corrupted to all the ITIs that have written on μ's input tape. Upon receipt of all other corruption messages, σ returns to the adversary the entire current state of π. Note that π is never notified of the corruption message; this captures the intuitive concept that a party is generally *not aware of being passively corrupted*. Also, note that here the adversary has to explicitly ask for each new report of internal state; however this formalism is chosen for convenience only and is of no real consequence. Our **leakage** (alternatively state-compromise) is modeled as a restricted form of passive corruption, in that σ returns the secret-state only once; in particular, unlike the standard passive corruption this is just a one-time affair in which the future states of μ are not returned. Nevertheless, leakage requests can be made multiple time, each of which is addressed with a one-time state-return (plus reporting to the other ITIs that have written to μ's input tape).

In the case of Byzantine/malicious/active corruptions, it is assumed that the corruption message from the adversary includes in it a description of an ITM M . Here σ behaves the same as before, except that it immediately replaces the code M instead of π. Our **bad-randomness** (or adversarially chosen randomness) corruption can be thought of as a restricted form of malicious corruption, in that the adversary instead provides description of a specific ITM, which runs the code of π except that now the code does not read from its own randomness tape, but takes the adverserial randomness provided as part of the ITM. This is also notified to all ITIs that have written to μ's input tape. Such bad randomness is used until the adversary sends another corruption request with an ITM that runs exactly the code of π reading from its own randomness tape – again this is reported to all ITI's that have written to μ's input tape.

Let $\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}(\kappa, z)$ denote the output distribution of the environment $\mathcal{Z}$ when interacting with parties running protocol π on security parameter κ and input z. Let $\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$ denote the ensemble $\{\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}(\kappa, z)\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$.

Protocol emulation. Informally, we say that a protocol π *UC-emulates* protocol π' if for any adversary $\mathcal{A}$ there exists an adversary $\mathcal{A}'$ such that no environment $\mathcal{Z}$, on any input, can tell with non-negligible probability whether it is interacting with $\mathcal{A}$ and parties running π or it is running π'. This means that, from the view of the environment, running protocol π is "just as good" as interacting with π'.[10] This notion is formalized as follows. A distribution ensemble is called binary if it consists of distributions over $\{0, 1\}$. We have:

Definition 17. *Two binary distribution ensembles* $\{X(\kappa, a)\}_{k\in\mathbb{N},a\in\{0,1\}^*}$ *and* $\{Y(\kappa, a)\}_{k\in\mathbb{N},a\in\{0,1\}^*}$

[10]To be precise, the definition of protocol emulation only quantifies over balanced environments. An environment is balanced if at any point in time the overall length of input to the adversary is at least some polynomial in the overall length the of inputs given to the rest of the ITIs in the system. As explained in [Can01], failing to make this restriction makes the definition unreasonably strong, and also causes technical problems with the composition theorem.

are called indistinguishable (written $X \approx Y$) if for any $c, d \in \mathbb{N}$ there exists $\kappa_0 \in \mathbb{N}$ such that for all $\kappa > \kappa_0$ and for all $a \in \{0,1\}^{\kappa^d}$ we have:

$$|\Pr[X(\kappa, a)] - \Pr[Y(\kappa, a)]| < \kappa^{-c}$$

Definition 18 (Protocol emulation)**.** *Let π and π' be two protocols. We say that π* UC emulates *π' if for any adversary $\mathcal{A}$ there exists an adversary $\mathcal{A}'$ such that for any environment $\mathcal{Z}$ that outputs a value in $\{0,1\}$ we have:*

$$\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}} \approx \text{EXEC}_{\pi',\mathcal{A}',\mathcal{Z}}$$

This work makes use of the following simplified formulation of UC emulation. Let the *dummy adversary* $\mathcal{D}$ be the adversary that merely reports to the environment all the messages sent by the parties, and follows the instructions of the environment regarding which messages to deliver to parties. Then, it is enough to prove security with respect to the dummy adversary. That is:

Definition 19 (Protocol emulation with the dummy adversary)**.** *Let π and π' be two protocols. We say that π* UC emulates *π' with the dummy adversary if there exists an adversary $\mathcal{A}'$ such that for any environment $\mathcal{Z}$ that outputs a value in $\{0,1\}$ we have:*

$$\text{EXEC}_{\pi,\mathcal{D},\mathcal{Z}} \approx \text{EXEC}_{\pi',\mathcal{A}',\mathcal{Z}}$$

Claim 1 ([Can01])**.** *Protocol π* UC-emulates *protocol π' iff π* UC-emulates *π' with respect to dummy adversary.*

Ideal functionalities and ideal protocol. A key ingredient in the ideal process for a given task is the *ideal functionality* that captures the desired behavior, or in other words, the specification of that task. The ideal functionality is modeled as an ITM (representing a "trusted party") that interacts with the parties and the (ideal) adversary. For convenience, the process for realizing an ideal functionality is represented as a special type of protocol, called an *ideal protocol* denoted $I_{\mathcal{F}}$. The adversary interacting with the ideal protocols is called the simulator and denoted $\mathcal{S}$. *Executing the ideal protocol $I_{\mathcal{F}}$* on input x formally means executing the system of ITMs $(\mathcal{Z}, C_{\mathcal{S},I_{\mathcal{F}}})$, $C_{\mathcal{S},I_{\mathcal{F}}}$ being the control function for the protocol $I_{\mathcal{F}}$ in presence of the adversary $\mathcal{S}$. Informally, the execution works of the ideal protocol $I_{\mathcal{F}}$ works as follows: all parties *simply hand their inputs* to an ITI with program $\mathcal{F}$ plus a session ID that is equal to the local session ID, and party ID set to some fixed value, say $\perp$. Whenever a party in $I_{\mathcal{F}}$ receives a value from $\mathcal{F}$ on its subroutine output tape, it immediately copies this value to the subroutine output tape of the ITI that invoked it. We call the parties of the ideal protocol *dummy parties*.

Definition 20 (Realizing functionalities)**.** *Let π be a protocol, and let $\mathcal{F}$ be an ideal functionality. We say that π UC-realizes $\mathcal{F}$ if π UC-emulates $I_{\mathcal{F}}$, the ideal protocol for $\mathcal{F}$.*

Ideal functionalities and party corruption. An ideal functionality represents an ideal specification, rather than an actual program that runs on an actual, physical device. Thus, party corruption messages sent to an ideal functionality do not directly represent physical corruption. Instead, the behavior of an ideal functionality upon receipt of corruption messages from the adversary specifies the security requirements from the realizing protocols upon party corruption.

In general, ideal functionalities can modify their behavior in arbitrary ways as a function of the corruption requests received from the adversary so far. (For instance, an ideal functionality may

allow the adversary to modify sensitive information as soon as more than some number of parties have been corrupted.) Still, we define some "standard" behavior of an ideal functionality in face of corruption. Specifically, we say that an ideal functionality $\mathcal{F}$ is *standard corruption* if:

1. An instance of $\mathcal{F}$ with SID sid keeps some "ideal local state" $state_P$ for each dummy party (sid, P) that interacts with this instance of $\mathcal{F}$. (Here P is the PID of this dummy party.)
2. Upon receipt of the first "corrupt P" message from the adversary, $\mathcal{F}$ first notifies the dummy party (sid, P) that it has been corrupted. Next, in each future "corrupt p" message, $\mathcal{F}$ returns to the adversary the contents of the ideal state $state_P$.

It is stressed that a standard corruption functionality can specify additional instructions to be performed upon receipt of a corruption message; it can also alter its overall behavior as exemplified above.

Universal Composition. Let ρ^ϕ be a protocol that uses one or more instances of some protocol ϕ as a subroutine, and let π be a protocol that UC-emulates ϕ. The composed protocol ρ^π is constructed by modifying the program of ρ^ϕ so that calls to ϕ are replaced by calls to π. Similarly, subroutine outputs coming from π are treated as subroutine outputs coming from ϕ. The universal composition theorem says that protocol ρ^π behaves essentially the same as the original protocol ρ^ϕ. That is:

Theorem 7 (Universal Composition [Can01])**.** *Let π, ϕ, ρ be protocols, such that π UC-emulates ϕ. Then the protocol ρ^π UC-emulates ρ^ϕ. In particular, if ρ^ϕ UC-realizes an ideal functionality $\mathcal{F}$, then so does ρ^π.*

We note that the universal composition theorem hold only in the case where protocols π and ϕ are *modular*. Essentially, a protocol is modular if in no instance s of this protocol there is a subroutine ITI I of some ITI which is part of the instance (or a subroutine thereof), where I receives input from or sends outputs to and ITI that is not a descendant of a member of instance s. Alternatively, modular protocols are also called subroutine respecting [Can01].