Partially specified channels: The TLS 1.3 record layer without elision

Christopher Patton and Thomas Shrimpton

University of Florida

Abstract. This work advances the study of secure *stream-based* channels (Fischlin et al., CRYPTO '15) by considering the multiplexing of many data streams over a single channel. This is an essential feature of real-world protocols such as TLS. Our treatment adopts the definitional perspective of Rogaway and Stegers (CSF '09), which offers an elegant way to reason about what standardizing documents actually provide: a *partial* specification of a protocol that admits a *collection* of compliant, fully realized implementations. We formalize *partially specified channels* as the component algorithms of two parties communicating over a channel. Each algorithm has an oracle that services *specification detail* queries; intuitively, the algorithms abstract the things that are explicitly specified, while the oracle abstracts the things that are not. Our security notions, which capture a variety of privacy and integrity goals, allow the *adversary* to respond to these oracle queries; security relative to our notions implies that the channel withstands attacks in the presence of worst-case (i.e., adversarial) realizations of the specification details. Our formalization is flexible enough to provide the first provable security treatment of the TLS 1.3 record layer that does not elide optional behaviors and unspecified details.

Keywords: cryptographic standards, TLS 1.3, stream-based channels, partially specified protocols

1 Introduction

The utility of the provable security methodology is ultimately limited by how faithfully its formal models reflect reality. This is true of all theoretical pursuits, but what sets security apart are the consequences of these models being wrong: at best, proofs of security are rendered useless, even as heuristics; at worst, real systems are left vulnerable to attacks. In their 2011 article entitled "Provable security in the real world" [14], Degabriele, Paterson, and Watson highlight how this tension has played out in the deployment of cryptographic protocols:

- Academic research heavily influences the process by which cryptographic standards, like TLS [28], DTLS [27], SSH [33], and IPSec [20], are developed; but the final specification has to satisfy a number of design criteria unrelated to theory, including providing backwards compatibility and interoperability with other protocols, and accounting for competing (and often conflicting) interests of stakeholders. As a result, standards are routinely comprised of design details, parameters, and optional features that academic formalisms elide and proofs of security say nothing about.
- Standards provide guidance for implementations, but do not fully specify every aspect of the protocol. This is by design: protocols need to be flexible enough to admit many implementations with different operational constraints and security goals. But this degree of separation between theory and implementation means the way cryptographic primitives are used may not coincide with the intended model.

In short, it is the standards that are left with the unenviable task of linking the clean abstractions of theory with the complicated realities of implementation. As a result, even though we all recognize that there is a gap between theory and standard-compliant implementation, its size is hard to estimate from either side. This work considers how we might address the gap from the *middle*, by formalizing what the standard itself actually provides.

Specifically, we propose a formal framework for reasoning about secure channel protocols such as TLS, DTLS, IPSec, and SSH. We focus on the mechanisms by which data are protected, written to the channel, and processed by the receiver. Many such mechanisms have been scrutinized from a provable-security perspective [25,26], but the subject of these analyses is invariably (and often implicitly) a particular, fully realized implementation of the scheme. In contrast, the subject of our study is the *protocol specification itself:* our

goal is to formalize the *collection* of fully realized implementations that the standardizing document admits. To do so, we adopt the *partially specified protocol* framework of Rogaway and Stegers [29] (hereafter RS), an unconventional viewpoint that has the potential to transform the way researchers approach real-world cryptography. One goal of this paper is to demonstrate this potential; to this end we initiate the study of *partially specified channels* (PSCs).

We begin by devising a syntax general enough to capture all of the previously mentioned protocols (Section 2). We extend the *stream-based* channel abstraction of Fischlin et al. [17] (hereafter FGMP), so named because it treats the plaintext and ciphertext as streams of *fragments*. This accurately models the interfaces exposed by secure channel implementations. Our syntax permits *multiplexing* of many distinct plaintext streams over the same channel, an important feature of TLS and DTLS (and, arguably, an implicit feature of SSH and IPSec). Its component algorithms formalize the aspects of the channel that must be realized properly; the rest of the *specification details* are modeled via an oracle to which the algorithms have access. This abstraction provides sufficient flexibility to give a provable security treatment of *all* compliant realizations of the TLS 1.3 record layer (Section 3). The standard [28] stipulates how to encrypt, how long records may be, and rules for transforming the input data into a sequence of records. But the lengths of records, whether to pad them, how to order them, and numerous other details are left unspecified. A traditional treatment would necessarily elide these details, either by fixing or ignoring them; the result is that the analysis says nothing about the security of the vast majority of compliant implementations.

We extend the notions of privacy and integrity of FGMP to multiplexed, stream-based channels (Sections 2.2 and 2.3). Going a step further, we consider different "levels" of privacy corresponding to various security goals considered in prior works [25,17,16]. Our analysis uncovers two subtle, yet security-critical matters that would be far harder to reason about in a more traditional treatment. First, the level of privacy the record layer can provably provide depends intrinsically on the unspecified details (Theorem 2). The record layer is used to multiplex distinct plaintext streams over the same channel; thus, each record has a *content type* that associates the content to its stream. The type is encrypted along with the content, permitting implementations that, at least in principle, hide both the content and its type. This is laudable, but the specification admits implementations that leak the content type entirely. Roughly speaking, this is because the *boundaries* between records depend on the types of each record. All we can say in general is that the record layer ensures privacy of the contents of each record. (We make this point precise in Section 3.)

Second, following FGMP, our syntax regards the ciphertext as a stream, and our notion of ciphertext-stream integrity (INT-CS) implies, informally, that the receiver only consumes the stream produced by the sender. Records written to the channel are delimited by strings called *record headers*, whose values are specified by the standard. These bits are not authenticated, and the standard does not require the receiver to check that their values are correct; thus, the record layer cannot achieve our strong notion of ciphertext-stream integrity. But intuitively, the value of these bits should not impact security. Our framework provides a clean way to reconcile this intuition with our model: we show that the value of these bits are indeed irrelevant *if and only if* they are authenticated (Theorem 3).

Our analysis applies to draft 23 of the standard [28], which was current at the time of writing. We shared the above finding with the IETF working group responsible for drafting TLS 1.3, and the specification was updated so that the record header is authenticated. This change will appear in the final version of the standard, and because this is the only revision to the record layer since draft 23, our results apply to TLS 1.3.

Background. Our framework weds two existing approaches to analyzing real-world cryptography. First, we extend secure *stream-based* channels to consider *multiplexing* of plaintext streams over the same channel. This addresses a problem left open by FGMP [18] and permits, for the first time, the analysis of TLS in this setting. The second approach is the *partially specified protocol* framework of RS, which we use to reason about protocol standards without eliding the myriad optional, under-specified, or otherwise unspecified details in these standards.

Stream-based channels. The deployment of these protocol standards has out-paced the development of supporting theory. In 2000, Bellare and Namprempre [8] provided foundations for the study of probabilistic authenticated encryption (AE) schemes used in SSL/TLS, IPSec and SSH. Shortly thereafter, Rogaway [30] embellished authenticated encryption to take associated data (AEAD), moving the primitive closer to practice. Yet it was already understood that an AEAD scheme and its attendant notions of privacy and integrity

do not suffice for building secure channels. In 2002, Bellare, Kohno, and Namprempre [7] formalized *stateful* AE in order to account for replay and out-of-order delivery attacks, as well as more accurately model and analyze SSH. Their model regards ciphertexts as atomic, but ciphertexts written to the channel may be (and routinely are) *fragmented* as they traverse the network, which leaves these protocols susceptible to attacks [2]. Likewise, the APIs for real secure channels regard the input plaintext as a stream, meaning that a single logical plaintext may be presented as a sequence of fragments, too. It took another ten years for the model to be significantly extended, by Boldyreva et al. [12], to address ciphertext fragmentation and attacks that exploit it. Finally, in 2015 by FGMP formalized stream-based secure channels that address plaintext fragmentation, with updates provided in 2016 by Albrecht et al. [1]. As FGMP point out [18], these works help shed formal light on truncation [31] and cookie-cutter [11] attacks.

Although theory has advanced significantly, it still falls short of capturing an important feature that real protocols provide: a means of multiplexing a number of data streams over the same channel. The TLS 1.3 record layer, for example, handles streams for three distinct sub-protocols: *handshake*, *alert*, and *application-data*. Explicitly modeling the multiplexing of these streams is crucial to a rigorous analysis of TLS, since each of these sub-protocols has side-effects on the sender and receiver state and, hence, implications for the security provided by the channel.

Whereas FGMP regard the plaintext stream as a sequence of message fragments $M_1, M_2, \ldots$, we will consider streams of the form $(M_1, sc_1), (M_2, sc_2), \ldots$ where sc_i denotes the *stream context* of its associated message fragment. Intuitively, the stream context is metadata that allows for differentiation of fragments into logical streams, each associated to a higher-level application, protocol, etc. Following prior work, our syntax models a unidirectional channel between a sender and receiver. We decompose the sender into two randomized, stateful algorithms: the stream multiplexer (Mux), and the channel writer ($Write$). Correspondingly, we decompose the receiver into the channel reader ($Read$), and the stream demultiplexer ($Demux$). One might think it cleaner to regard the sender and receiver as atomic processes, rather than decompose them as we have done. Indeed, this abstraction is adopted in the aforementioned works. We break with this syntax in order in order to precisely capture multiplexing of streams, and to separate this functionality from the cryptographic operations that turn plaintext strings into ciphertexts. (More on this in Section 2.2.)

Partially specified protocols. The conventional approach to formalizing a cryptographic primitive yields security theorems that hold only when the constituent algorithms have been realized properly, according to some implicit reference or standard that *fully specifies what is a proper realization*. RS observed that *real* standards provide only *partial specifications* of compliant schemes: they routinely allow for optional behaviors and leave many implementation details unspecified. The RS thesis is that one does not *need* to elide these details in order to foster clean abstractions and clear mathematical reasoning. Their approach, which we adopt here, is simply to formalize what a standard *is*: a partial specification (the things that are mandated and explicitly described) plus additional specification details (everything else). RS apply this approach to authentication protocols, in particular the Needham-Schroeder-Lowe protocol. We apply it to secure channels.

Thus, we initiate the study of partially specified channels. The component algorithms Mux, $Write$, $Read$, and $Demux$ formalize the core functionalities of the sender and receiver that *must* be fully specified; the rest of the *specification details* (SD) are formalized via an oracle given to each of the algorithms. The functionality of this SD oracle is left unspecified, and in our security games, *queries made to the oracle are serviced by the adversary*. This is clearly a very strong attack model: in addition to influencing the behavior of the algorithms via their inputs, the adversary is allowed to participate in portions of their computation. The actual strength of the model depends on what quantities are exposed to the SD, and how the SD return values are used within the algorithms. At one extreme, an empty (or otherwise trivial) SD yields a traditional kind of attack model; at the other, if secret state (e.g., the key) is passed to the SD, then no security is possible. In this way, our model can provide principled guidance to the standard-writing process by surfacing choices that may be security critical.

This definitional framework admits another interpretation, one that is likely of interest in other settings: it lets us reason about security in the presence of implementation errors. One can view each algorithm as being partitioned into operations whose implementation is assumed to be correct, and those that are not. From this perspective, our attack model captures a kind of worst-case (i.e., adversarial) implementation of those operations. This is interesting because if one proves that a particular PSC construction is secure, it

makes clear which things *must* be implemented correctly and deserve the extra scrutiny of formal verification (a la [16]), and which things do not need such hard guarantees.

Related work. We have already mentioned the line of papers that our work extends [8,7,13,17,1]; this section points out important related efforts.

The miTLS project. From the standpoint of scope, the work most closely related to ours is the recent paper by Delignat-Lavaud et al. [16] (DLFK+). It provides a formal analysis of the TLS 1.3 record layer (draft 18) "as is", but their approach is fundamentally different from our own. The paper is the latest from miTLS (`mitls.org`), a project whose goal is to formally verify the security of TLS *as is*, without omitting any details. The strategy is to implement the record layer in a programming language that is amenable to formal analysis (F*), express their security goals as games in the same language, and find a formal proof that the scheme's security (in a sense they define) reduces to standard computational assumptions (also expressed in F*). This methodology amounts to a formalization of code-based game-playing techniques now common in cryptography [9]. Our work is technically different from theirs on a couple fronts. First, our analysis applies to a *set* of compliant implementations (corresponding to different realizations of the specification details), whereas their work applies only to their implementation. This flexibility extends to our security notions: we capture the goal of hiding the message length as one of many possible privacy goals, whereas this property is mandatory in their security notion. Second, our adversarial model is stronger in that it permits fragmentation of the plaintext and ciphertext streams; neither capability is considered by DLFK+. We elaborate on this and other points about their setting in Appendix B.

We do not mean to diminish the work of DLFK+ in pointing out these short comings. On the contrary, the value of their contribution (and of the miTLS project overall) is hard to overstate. They provide a reference implementation of the record layer in which we have a high degree of confidence, both in terms of security and, crucially, *correctness*. Practitioners are paying attention [28, Section 12.2], and using this reference will ultimately facilitate the development of secure production code. As such, we view our work as complimentary to DLFK+. An interesting direction would be to extend their framework to permit some degree of partial specification.

Other analyses of the record layer. In an analysis of TLS 1.2, a paper by Patterson, Ristenpart, and Shrimpton [25] put forward a notion of stateful, *length-hiding* AE that admits schemes with associated padding (to hide the plaintext length) and variable-length MACs, both features of TLS 1.2. We would call this a "traditional" analysis since their formalism necessarily elides a number of details of the protocol. Badertscher et al. [5] characterized the TLS 1.3 record layer (draft 08) as an *augmented* secure channel (ASC), which allows for sending a message with two parts: the first being private, and both parts being authenticated. It too is a more traditional analysis. Bellare and Tackmann analyze the *multi-user* security of the TLS 1.3 record layer [10]. They shed light on the following problem: if the same message is encrypted in a number of sessions, then what information does this leak about the sessions? A popular TLS endpoint, such as `google.com`, might serve billions of client a day. Many of these flows are identical (such as the initial GET); thus, an adversary who observes these identical flows can try to guess the key used for one of the clients. Its odds are improved by the sheer number of clients encrypting an identical message. This attack vector lead the designers of TLS 1.3 to "randomize" the IV used for generating the nonce; Bellare and Tackmann analyze the exact security of this approach in the multi-user setting.

2 Partially specified channels

In this section we formalize PSCs and their attendant security notions. We begin with some notation and conventions.

Notation. Let $|X|$ denote the length of a string $X \in \{0,1\}^*$ and let $|\boldsymbol{X}|$ denote the length of vector $\boldsymbol{X}$. We denote the i-th bit of string X by X_i or $X[i]$, and the i-th element of vector $\boldsymbol{X}$ by $\boldsymbol{X}_i$ or $\boldsymbol{X}[i]$. Let $\{0,1\}^{**} = (\{0,1\}^*)^*$. We define $X \,\|\, Y$ to be the concatenation of strings X and Y; let $cat\colon \{0,1\}^{**} \to \{0,1\}^*$ denote the map $\boldsymbol{X} \mapsto \boldsymbol{X}_1 \,\|\, \cdots \,\|\, \boldsymbol{X}_m$, where $|\boldsymbol{X}| = m$. Let $X[i{:}j]$ denote the substring $X_i \,\|\, \cdots \,\|\, X_j$ of X. If $i \notin [1..j]$ or $j \notin [i..|X|]$, then define $X[i{:}j] = \varepsilon$. Let $X[i{:}] = X[i{:}|X|]$ and $X[{:}j] = X[1{:}j]$. We write $X \preceq Y$ if X is a prefix of Y (i.e., $(\exists\, T \in \{0,1\}^*)\, X \,\|\, T = Y$). Let $X \,\%\, Y$ denote "remainder" of X after removing the

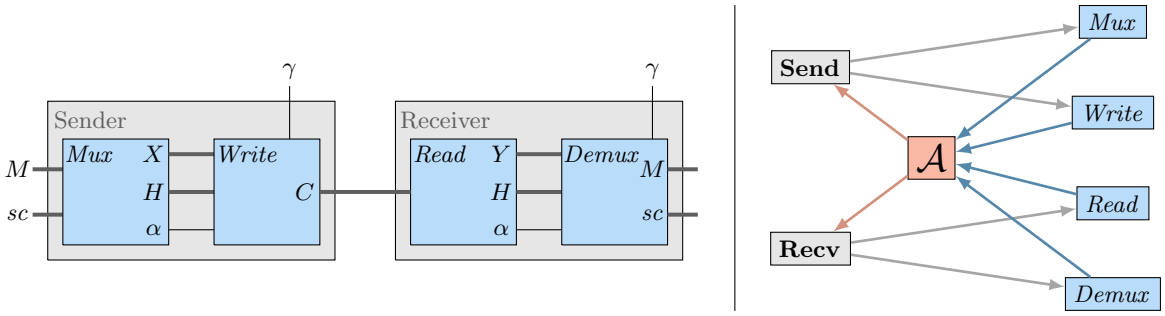

Fig. 1: left: illustration of our syntax. Right: illustration of the execution model. The game may only traverse non-cyclic paths in this call graph.

---

prefix Y, e.g., $1011 \,\%\, 10 = 11$. (If $Y \not\preceq X$, then define $X \,\%\, Y = \varepsilon$.) Let $\langle i \rangle_n$ denote the invertible encoding of integer $i \geq 0$ as an n-bit string.

Algorithms may have access to one or more oracles, written as superscripts (e.g., $\mathcal{A}^{\mathcal{O}}$). The runtime of an algorithm includes the time required to evaluate its oracle queries. If an algorithm $\mathcal{A}$ is deterministic, then we write $y \leftarrow \mathcal{A}(x)$ to denote executing $\mathcal{A}$ on input of x and assigning its output to y; if $\mathcal{A}$ is randomized or stateful, then we write $y \twoheadleftarrow \mathcal{A}(x)$. Algorithms are stateless and randomized, unless specified otherwise. An *adversary* is a stateful and randomized algorithm. If $\mathcal{X}$ is a set, then we write $x \twoheadleftarrow \mathcal{X}$ to denote sampling x randomly from $\mathcal{X}$ according to some implicitly-defined distribution. If $\mathcal{X}$ is finite, then the distribution is uniform. If $n \in \mathbb{N} \setminus \{0\}$, then let $[n] = \{x \in \mathbb{N} : 1 \leq x \leq n\}$.

Pseudocode. Our pseudocode follows the conventions of RS with a few minor differences. (Refer to [29, Section 2].) our pseudocode is statically typed. Available types are **bool** (called **boolean** in RS, an element of $\{0,1\}$), **int** (**integer** in RS, an element of $\mathbb{Z}$), **str** (**string** in RS, an element of $\{0,1\}^*$), and **struct** (**record** in RS). New types may be defined recursively from these: for example, **type struct** $\{$**str** *name*, **int** *age*$\}$ **person** declares a data structure with two fields, the first a **str** and the second an **int**. Variables may be declared with the word **declare**, e.g. **declare person** *Alice*. Variables need not be explicitly declared, in which case their type must be inferable from their initialization (i.e., the first use of the variable in an assignment statement). There are also associative arrays that map arbitrary quantities to values of a specific type. For example, **declare str** $X[\,]$ declares an associative array X. We let $X[k]$ and X_k denote the value in X associated with k. We will find it useful to explicitly define the "type" of a procedure (i.e., algorithm) by their interface. For instance, the type $\mathcal{A}(\textbf{str}\ X, \textbf{str}\ Y) \mapsto (\textbf{int}\ i, \textbf{int}\ j)$ indicates that $\mathcal{A}$ takes as input a pair of strings and outputs a pair of integers. Multiple variables of the same type may be compactly declared, e.g., as **declare str** $X, Y,$ **int** z rather than **declare str** X, **str** Y, **int** z. We also use this convention when defining procedure interfaces, e.g., $\mathcal{A}(\textbf{str}\ X, Y) \mapsto (\textbf{int}\ i, j)$.

If a variable of one type is set to a value of another type, then the variable takes the value $\diamond$, read "undefined". Uninitialized variables implicitly have the value $\diamond$. The symbol $\diamond$ is interpreted as 0 (i.e., false) in a boolean expression, as 0 in an expression involving integers, and as ε in an expression involving strings. We introduce the distinguished symbol $\perp$, read "invalid", which can be assigned to any variable regardless of type. Unlike $\diamond$, its interpretation in an expression is undefined, except that $(X = \perp)$ should evaluate to true just in case variable X was previously set to $\perp$. We remark that $\perp$ has the usual semantics in crytpographic pseudocode; the symbol $\diamond$ is useful for specifying protocols compactly.

Variables passed to procedures may be embellished with the key word **var**, which means the variable has copy-in-copy-out semantics. For example, $\mathcal{A}(\textbf{var}\ x)$ means that $\mathcal{A}$ is invoked on input x, and after $\mathcal{A}$ halts, x may have a different value. This will be useful for explicitly defining a procedure's state. A value of any type may be assigned to an anonymous variable $*$, e.g., $* \leftarrow x$. We let $\langle x_1, \ldots, x_m \rangle$ denote an invertible encoding of arbitrary values $x_1, \ldots, x_m$ as a bit string. Decoding is written as $\langle x_1, \ldots, x_m \rangle \leftarrow X$ and works like this (slightly deviating from [29, Section 2]): if there exist $x'_1, \ldots x'_{m'}$ such that $X = \langle x'_1, \ldots, x'_{m'} \rangle$, $m' = m$, and each x'_i has the same type as x_i, then set $x_i \leftarrow x'_i$ for each $i \in [m]$. Otherwise, set $x_i \leftarrow \diamond$ for each $i \in [m]$.

2.1 Syntax

Formally, a PSC is a 5-tuple of randomized algorithms $\mathcal{CH} = (\mathit{Init}, \mathit{Mux}, \mathit{Write}, \mathit{Read}, \mathit{Demux})$. All but the first are stateful and expect access to an oracle, which we generically write as $\mathcal{O}$ in the following definitions:

- $\mathit{Init}(\,) \mapsto (\mathbf{str}\ \mathit{Mu}, \mathit{Wr}, \mathit{Re}, \mathit{De})$. The *initialization algorithm* models key agreement and initialization of the sender state $(\mathit{Mu}, \mathit{Wr})$ and receiver state $(\mathit{Re}, \mathit{De})$.
- $\mathit{Mux}^{\mathcal{O}}(\mathbf{str}\ M, sc, \mathbf{var\ str}\ \mathit{Mu}) \mapsto (\mathbf{str}\ X, H, \alpha)$. The *multiplexing algorithm* takes as input a plaintext fragment M, stream context sc, and state Mu, and returns a channel fragment X, its context H, and some auxiliary output α.
- $\mathit{Write}^{\mathcal{O}}(\mathbf{str}\ X, H, \alpha, \mathbf{var\ str}\ \mathit{Wr}) \mapsto (\mathbf{str}\ C, \gamma)$. On input of a channel fragment X, context H, and auxiliary information α, and state Wr, the *channel writing algorithm* produces a ciphertext fragment C and a status message γ.
- $\mathit{Read}^{\mathcal{O}}(\mathbf{str}\ C, \mathbf{var\ str}\ \mathit{Re}) \mapsto (\mathbf{str}\ Y, H, \alpha)$. On input of a ciphertext fragment C and state Re, the *channel reading algorithm* returns a ciphertext fragment Y, its context H, and auxiliary output α.
- $\mathit{Demux}^{\mathcal{O}}(\mathbf{str}\ Y, H, \alpha, \mathbf{var\ str}\ \mathit{De}) \mapsto (\mathbf{str}\ M, sc, \gamma)$. The *demultiplexing algorithm* takes a ciphertext fragment Y with channel context H, auxiliary information α, and state De, and returns a plaintext fragment M with stream context sc, along with a status message γ.

The oracle $\mathcal{O}$ provides the *specification details* (SD) and may be invoked any number of times by the caller during its execution. The SD-oracle may have its own state and coins; to be clear, the oracle and its caller do not have joint state, and their coins are independent. We require that each of these procedures halts, regardless of coin tosses or SD responses, in a bounded number of steps that depends only on the length of its inputs.

Specification details. The SD is formalized by an oracle, so we need some way to talk about the queries that the oracles accept. Our convention will be that SD queries are of the form $\langle \mathsf{caller}, \mathsf{instruction}, x_1, \ldots, x_m \rangle$, where caller and instruction may be thought of as strings. When it is necessary to specify an SD-oracle query, we will endeavor to make them suggestive of the intended semantics under correct operation. (See Figure 7 for examples.) SD responses are also always strings, but we do not provide conventions for them.

Status messages and auxiliary outputs. All algorithms may produce some *auxiliary information* along with its outputs. This allows *Mux* and *Read* to convey state (denoted α) to *Write* and *Demux* (resp.), and allows *Write* and *Demux* to surface status information (denoted γ) to applications. (See Figure 1 for an illustration.) Among other things, this models distinguishable decryption errors [13], an attack vector that has heavily influenced the development of secure channels [32,15,24,3]. (FGMP model distinguishable errors, too.) Our consideration of information leakage via auxiliary output is inspired by a recent work by Barwell, Page, and Stam [6]. Their *subtle* AE setting models decryption leakage in a manner general enough to capture error indistinguishibility [13,17], as well as other settings for authenticated encryption [4,19].

Correctness. At this juncture, a traditional treatment would define correctness of the primitive. However, following RS, we will not explicitly define correctness of PSCs. Indeed, our aim is to define and achieve security *even for channels that are not correct:* in particular, when the SD is realized by an adversary. Quite to the contrary, one normally assumes correctness in proofs of security. We elaborate on the consequences of this choice in Appendix A.

2.2 Privacy

We recast the privacy notions of FGMP to address the multiplexing of plaintext streams and expose the specification details. Our PRIV-SR notion gives the adversary access to a pair of oracles. The **Send** oracle allows the adversary to provide the sender with arbitrary message fragments and stream contexts, where streams are distinguished by their context sc. Analogously, the **Recv** oracle allows the adversary to deliver arbitrary ciphertext fragments to the receiver. We define a PRIV-S notion from this game by removing the **Recv** oracle. In both notions, whenever a query to **Send** or **Recv** induces an SD-oracle call, that call is serviced by the adversary.

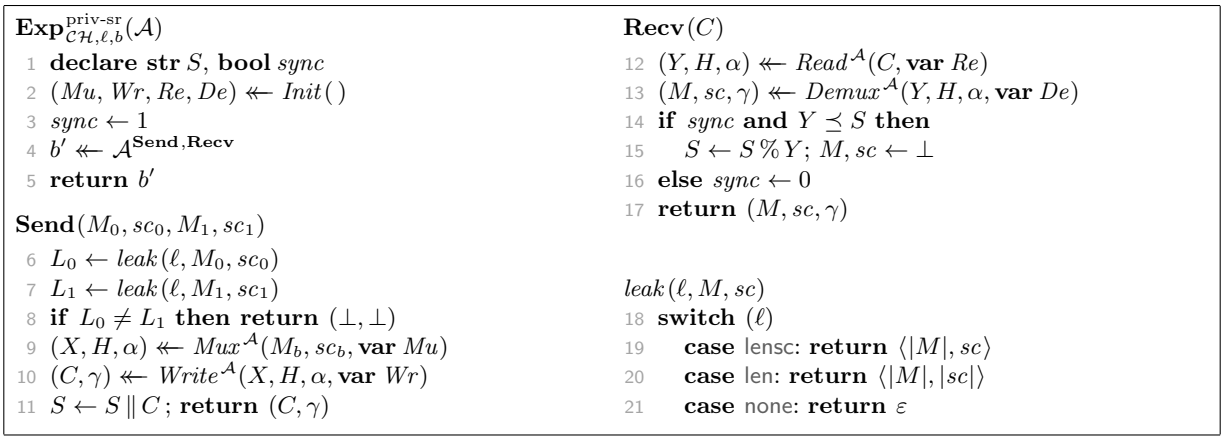

Fig. 2: The PRIV-SR notion of security for a partially-specified channel $\mathcal{CH}$. The PRIV-SR notion is parameterized by the permitted leakage $\ell \in \{\mathsf{lensc}, \mathsf{len}, \mathsf{none}\}$.

Following prior work [7,12,17] we keep track of whether the channel is *in-sync* at any given moment during the adversary's attack. Loosely, the channel is said to be in-sync if the stream of ciphertext "consumed" by the receiver "so far" is a prefix of the stream of ciphertext output by the sender. In order to avoid trivial distinguishing attacks in the PRIV-SR game, it is necessary to suppress the message fragments output by the receiver while the channel is in-sync.

Channel synchronization. We say the channel is in-sync as long as the ciphertext fragments Y output by *Read*—which models receiver-side buffering and defragmentation—remains a prefix of the ciphertext stream transmitted by the sender. In this way, the sequence of Y's output by the reader constitute the ciphertext stream "consumed" by the receiver (i.e., *Demux*) so far. This restricts the behavior of the sender-side code in a way not seen in FGMP, but the restriction appears to be minor; a natural division of labor is to have *Read* buffer the ciphertext stream and output ciphertexts that are ready to decrypt; the job of *Demux*, then, is to decrypt and process the message stream. This cleanly separates the tasks of "buffering" and "consuming" the ciphertext. The alternative would be to leave the receiver operations atomic, as FGMP have done; but this choice leads to complex security notions, as it requires handling synchronicity for a number of different cases (e.g., [18, Definition 4.1]). Decomposing the syntax in this way leads to simpler and more intuitive definitions.

The execution model. Our execution model for security games is adopted from the RS framework. It is illustrated in Figure 1. The adversary queries oracles provided by the security experiment, which in turn may invoke the adversary for fulfilling SD queries. To ensure that each oracle query completes before the next query is issued, the adversary may not issue a query while another query is pending. In effect, the adversary may not use its oracles for computing its responses to SD queries. We require that the adversary halts, regardless of coin tosses, oracle responses, or SD requests, in a bounded number of steps. By convention, the adversary's runtime includes the time needed to evaluate its queries. We silently extend this execution model and conventions to all subsequent security experiments in this paper.

The PRIV-SR and PRIV-S notions. Refer to the PRIV-SR experiment defined in Figure 2. For a given PSC $\mathcal{CH}$ and challenge bit b, the experiment compactly encapsulates three different notions of privacy, each associated to a *permitted leakage* parameter $\ell \in \{\mathsf{lensc}, \mathsf{len}, \mathsf{none}\}$. When $\ell = \mathsf{lensc}$, only message-stream privacy is captured; when $\ell = \mathsf{len}$ the notion captures privacy of both the message streams and their context; finally, $\ell = \mathsf{none}$ adds length-hiding to the list.[1]

The game begins by initializing the sender state (Mu, Wr) and receiver state (Re, De). The adversary $\mathcal{A}$ is given access to two oracles. The first, **Send**, takes as input a 4-tuple of strings (M_0, sc_0, M_1, sc_1). It first

[1] There are other parameters that may be of practical interest. For example, DLFK+ deal with whether the fragment encodes the end-of-stream [16, Definition 8].

$\mathbf{Exp}^{\text{int-cs}}_{\mathcal{CH}}(\mathcal{A})$

1 **declare str** S, **bool** $sync, win$
2 $(Mu, Wr, Re, De) \twoheadleftarrow Init(\,)$
3 $sync \leftarrow 1$; $\mathcal{A}^{\mathbf{Send},\mathbf{Recv}}$
4 **return** win

Send(M, sc)

5 $(X, H, \alpha) \twoheadleftarrow Mux^{\mathcal{A}}(M, sc, \mathbf{var}\ Mu)$
6 $(C, \gamma) \twoheadleftarrow Write^{\mathcal{A}}(X, H, \alpha, \mathbf{var}\ Wr)$
7 $S \leftarrow S \,\|\, C$
8 **return** (C, γ)

Recv(C)

9 $(Y, H, \alpha) \twoheadleftarrow Read^{\mathcal{A}}(C, \mathbf{var}\ Re)$
10 $(M, sc, \gamma) \twoheadleftarrow Demux^{\mathcal{A}}(Y, H, \alpha, \mathbf{var}\ De)$
11 **if** $sync$ **and** $Y \preceq S$ **then** $S \leftarrow S \,\%\, Y$
12 **else** $sync \leftarrow 0$
13 $\quad win \leftarrow win \vee (M \neq \bot \wedge sc \neq \bot)$
14 **return** (M, sc, γ)

$\mathbf{Exp}^{\text{int-ps}}_{\mathcal{CH}}(\mathcal{A})$

15 **declare str** $S[\,]$, **str** $R[\,]$, **bool** win
16 $(Mu, Wr, Re, De) \twoheadleftarrow Init(\,)$
17 $\mathcal{A}^{\mathbf{Send},\mathbf{Recv}}$
18 **return** win

Send(M, sc)

19 $(X, H, \alpha) \twoheadleftarrow Mux^{\mathcal{A}}(M, sc, \mathbf{var}\ Mu)$
20 $(C, \gamma) \twoheadleftarrow Write^{\mathcal{A}}(X, H, \alpha, \mathbf{var}\ Wr)$
21 $S_{sc} \leftarrow S_{sc} \,\|\, M$
22 **return** (C, γ)

Recv(C)

23 $(Y, H, \alpha) \twoheadleftarrow Read^{\mathcal{A}}(C, \mathbf{var}\ Re)$
24 $(M, sc, \gamma) \twoheadleftarrow Demux^{\mathcal{A}}(Y, H, \alpha, \mathbf{var}\ De)$
25 **if** $M \neq \bot$ **and** $sc \neq \bot$ **then**
26 $\quad R_{sc} \leftarrow R_{sc} \,\|\, M$
27 **if** $R_{sc} \not\preceq S_{sc}$ **then** $win \leftarrow 1$
28 **return** (M, sc, γ)

Fig. 3: left: game for defining ciphertext-stream integirty (INT-CS) for partially specified channel $\mathcal{CH}$. Right: game for defining plaintext-stream integrity (INT-PS) of $\mathcal{CH}$.

checks that the values of $leak(\ell, M_0, sc_0)$ and $leak(\ell, M_1, sc_1)$ are equal; if not, it returns an indication of invalidity of the query. It then executes *Mux* and *Write*, with $\mathcal{A}$ as the SD-oracle, and returns the output (C, γ) to $\mathcal{A}$. String C is appended to S, which keeps track of the sender ciphertext stream. The second oracle, called **Recv**, takes as input a ciphertext fragment C. It first invokes $(Y, H, \alpha) \twoheadleftarrow Read^{\mathcal{A}}(C, \mathbf{var}\ Re)$, then $(M, sc, \gamma) \twoheadleftarrow Demux^{\mathcal{A}}(Y, H, \alpha, \mathbf{var}\ De)$. If the channel is in-sync and Y is a prefix of the sender stream S, then the oracle "consumes" Y from the stream and suppresses the output of M and sc by setting $M, sc \leftarrow \bot$. (This is necessary because (M, sc) corresponds to an input to **Send** and might trivially leak b, depending on the permitted leakage ℓ.) Otherwise, the oracle declares the channel to be out-of-sync and outputs (M, sc, γ) without suppressing M and sc. After the adversary interacts with its oracles, it outputs a bit b', the outcome of the game. We define the advantage $\mathcal{A}$ in attacking $\mathcal{CH}$ in the PRIV-SR(ℓ) sense as

$$\mathbf{Adv}^{\text{priv-sr}}_{\mathcal{CH},\ell}(\mathcal{A}) = 2 \Pr_b\big[\,\mathbf{Exp}^{\text{priv-sr}}_{\mathcal{CH},\ell,b}(\mathcal{A}) = b\,\big] - 1\,,$$

where the probability is over the coins of the game, the adversary, and the choice of b (implicitly sampled as $b \twoheadleftarrow \{0,1\}$). In this experiment, we track the following adversarial resources: the time-complexity t, the number of **Send**-queries q_1 and the total length in bits of the inputs of each query μ_1, and the number of **Recv**-queries q_2 and their total bitlength μ_2. We define the maximum advantage of any adversary with these resources as $\mathbf{Adv}^{\text{priv-sr}}_{\mathcal{CH},\ell}(t, q_1, q_2, \mu_1, \mu_2)$.

A chosen-plaintext (fragment) attack version of PRIV-SR is obtained simply by removing the **Recv** from the experiment; we refer to this game as PRIV-S and define the PRIV-S advantage of $\mathcal{A}$ in the same way; as there is no **Recv** oracle, we drop q_2, μ_2 from the adversarial resources.

2.3 Integrity

Following FGMP, we consider integrity of both the ciphertext stream (INT-CS) and the plaintext streams (INT-PS). The first formalizes the conservative goal that the channel (i.e., the ciphertext stream) should remain in-sync, just as discussed in Section 2.2. The second formalizes a weaker property, namely that the plaintext streams carried by the channel should remain in-sync.

The INT-CS notion. Refer to the INT-CS experiment defined in Figure 3. It begins just as in the PRIV-SR game. The **Send** oracle is similar to the PRIV-SR game, except $\mathcal{A}$'s query consists of a pair (M, sc) instead of a 4-tuple. We keep track of whether the channel is in-sync in the exact same manner. If ever the

out-of-sync **Recv** oracle outputs a valid message fragment and context, then the game sets a flag $win \leftarrow 1$; the outcome of the game is the value of win after $\mathcal{A}$ halts. Define the advantage of $\mathcal{A}$ in attacking $\mathcal{CH}$ in the INT-CS sense as $\mathbf{Adv}^{\text{int-cs}}_{\mathcal{CH}}(\mathcal{A}) = \Pr\left[\mathbf{Exp}^{\text{int-cs}}_{\mathcal{CH}}(\mathcal{A}) = 1\right]$, where the probability is over the coins of the experiment and of the adversary. We define the function $\mathbf{Adv}^{\text{int-cs}}_{\mathcal{CH}}(t, q_1, q_2, \mu_1, \mu_2)$ as the maximum advantage of any adversary running in time t, making at most q_1 queries to **Send** and q_2 queries to **Recv**, and the total bit-length of its queries to **Send** (resp. **Recv**) does not exceed μ_1 (resp. μ_2) bits.

The INT-PS notion. Integrity of the plaintext streams is defined via the INT-PS game in Figure 3. This game is a bit different than the others in that we do not keep track of whether the ciphertext stream is in-sync; rather, we are concerned with the input and output plaintext streams. For each stream context $sc \in \{0,1\}^*$ queried by the adversary, we keep track of the corresponding input stream S_{sc}. (That is, $S_{sc} = cat(\boldsymbol{M})$, where $\boldsymbol{M}$ is the sequence of message fragments pertaining to sc asked of **Send**.) For each $sc \neq \perp$ output by **Recv**, we keep track of the corresponding output stream R_{sc}. (That is, $R_{sc} = cat(\boldsymbol{M})$, where $\boldsymbol{M}$ is the sequence of valid message fragments pertaining to sc output by **Recv**.) The adversary wins if at any point in the game, it holds that $R_{sc} \not\preceq S_{sc}$ for some $sc \in \{0,1\}^*$. Define the advantage of $\mathcal{A}$ in attacking $\mathcal{CH}$ in the INT-PS sense as $\mathbf{Adv}^{\text{int-ps}}_{\mathcal{CH}}(\mathcal{A}) = \Pr\left[\mathbf{Exp}^{\text{int-ps}}_{\mathcal{CH}}(\mathcal{A}) = 1\right]$, where the probability is over the coins of the experiment and of the adversary.

INT-CS $\not\Rightarrow$ INT-PS for PSCs. Traditional results for AE schemes establish an intuitive relationship between integrity of ciphertexts and plaintexts: that the former is strictly stronger than the latter. See Bellare and Namprempre [8, Theorem 3.1] in the case of stateless and randomized AE, and FGMP [18, Appendix C] for stream-based channels. It is perhaps counter-intuitive, then, that *INT-CS does not imply INT-PS* in our setting. The reason for this is that we do not require that PSCs be operationally correct in the security games; indeed, the correctness of the scheme is used in a crucial way in those proofs. However, we cannot formalize correctness for PSCs without restricting the SD oracle in some way, and doing so would reduce the generality of our results. Nevertheless, in Appendix A, we give a natural definition of correctness for *fully specified channels*—like PSCs, but with fully realized SD—that extends FGMP's correctness condition to the multiplexed setting. With this definition we show something a bit stronger than usual: that INT-CS implies INT-PS if and only if the SD is realized correctly.

2.4 Receiver-status simulatability and a generic composition

If a PSC is INT-CS secure, then an efficient attacker can do nothing but deliver the honestly produced ciphertext stream in the correct order. Thus it is intuitive that any PSC that is both PRIV-S secure and INT-CS secure will also be PRIV-SR secure, because, in effect, the **Recv** in the PRIV-SR game is useless. This is almost true; the wrinkle is that the **Recv** oracle returns a status message in addition to the message fragment and stream context. As in the FGMP setting, our syntax does not restrict the receiver (in particular, the demultiplexer) to return just one status message. Moreover, the status message may depend on the receiver state (of which a PRIV-S adversary would be ignorant), or be influenced by the adversarially controlled SD. In this section, we give a notion of security we call *receiver-status simulatablity* and show that it, PRIV-S, and INT-CS imply PRIV-SR.

The SIM-STAT notion. The notion naturally captures what the adversary learns from the receiver's state by observing the status messages it outputs. It is inspired by the ideas put forward in the subtle AE setting [6] and naturally generalizes a notion of FGMP. The SIM-STAT game (defined in Figure 4) is a simulation-based game in which the adversary is asked to distinguish the status messages output by the real receiver from those output by a simulator $\mathcal{S}$. The simulator is given the ciphertext stream S produced by the sender, as well as the input fragment C, and so it can tell if the channel is in-sync, but it is not given the receiver state. Informally, security demands that for every efficient adversary, there is an efficient simulator such that the adversary cannot distinguish real status messages from fake ones.

The game is associated to adversary $\mathcal{A}$, challenge bit b, and a *receiver-status simulator* $\mathcal{S}$. On input of C, if $b = 1$, then oracle **Recv** executes the usual receiver code and outputs γ; otherwise, the oracle executes $\mathcal{S}$ on input of (C, S), where S is the sender stream (recorded by **Send**), and with oracle access to $\mathcal{A}$ for servicing SD requests. When $\mathcal{S}$ halts and outputs a string γ, the oracle outputs γ. We define the advantage

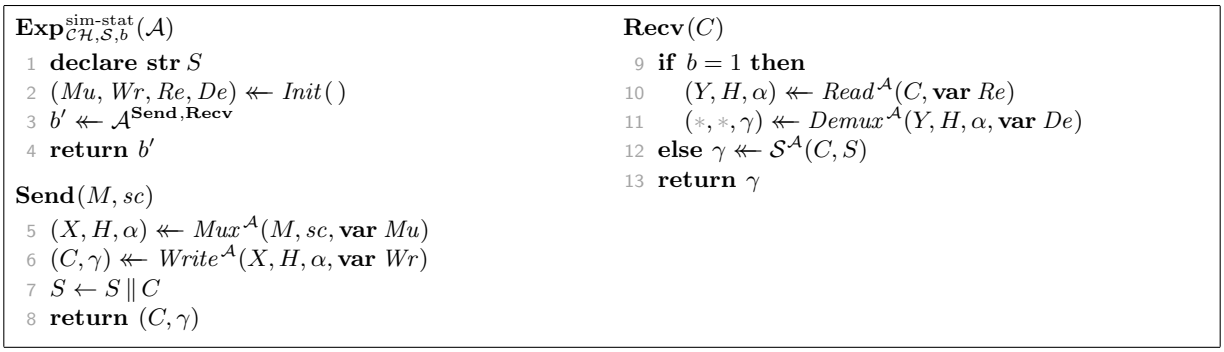

Fig. 4: the SIM-STAT game for partially-specified channel $\mathcal{CH}$.

of $\mathcal{A}$ in attacking $\mathcal{CH}$ with simulator $\mathcal{S}$ in the SIM-STAT sense as

$$\mathbf{Adv}^{\text{sim-stat}}_{\mathcal{CH},\mathcal{S}}(\mathcal{A}) = 2 \Pr_b\big[\,\mathbf{Exp}^{\text{sim-stat}}_{\mathcal{CH},\mathcal{S},b}(\mathcal{A}) = 1\,\big] - 1\,.$$

Define the maximum advantage of any t-time adversary with resources (q_1, q_2, μ_1, μ_2) in winning the game instantiated with simulator $\mathcal{S}$ as $\mathbf{Adv}^{\text{sim-stat}}_{\mathcal{CH},\mathcal{S}}(t, q_1, q_2, \mu_1, \mu_2)$. We require that $\mathcal{S}$ halts, regardless of its current state, internal coin tosses, and the result of its SD requests, in a bounded number of time steps. Its runtime also accounts for the time needed to evaluate its oracle queries; thus, its runtime depends on the time $\mathcal{A}$ takes to compute its SD responses.

PRIV-S $\wedge$ *INT-CS* $\wedge$ *SIM-STAT* $\Rightarrow$ *PRIV-SR.* We prove that for any ℓ, security in the sense of PRIV-S(ℓ), INT-CS, and SIM-STAT suffice for PRIV-SR(ℓ).

Theorem 1. *Let $\ell \in \{\mathsf{lensc}, \mathsf{len}, \mathsf{none}\}$ and let $\mathcal{CH}$ be a PSC. For every $t, s, q_1, q_2, \mu_1, \mu_2 \in \mathbb{N}$ and s-time simulator $\mathcal{S}$ it holds that*

$$\begin{aligned}\mathbf{Adv}^{\text{priv-sr}}_{\mathcal{CH},\ell}(t, \boldsymbol{r}) \le &\mathbf{Adv}^{\text{priv-s}}_{\mathcal{CH},\ell}(t + O(q_1 + sq_2), q_1, \mu_1) \\ &2\mathbf{Adv}^{\text{int-cs}}_{\mathcal{CH}}(\tilde{t}, \boldsymbol{r}) + 2\mathbf{Adv}^{\text{sim-stat}}_{\mathcal{CH},\mathcal{S}}(\tilde{t}, \boldsymbol{r})\,,\end{aligned}$$

where $\tilde{t} = t + O(q_1 + q_2)$ and $\boldsymbol{r} = (q_1, q_2, \mu_2, \mu_2)$.

This is analogous to, but much more general than [18, Theorem 4.5]. It also confirms a conjecture of FGMP; see [18, Remark 4.6]. The idea of the proof is to construct a PRIV-S adversary $\mathcal{B}$ from a given PRIV-SR adversary $\mathcal{A}$ and simulator $\mathcal{S}$ that simulates $\mathcal{A}$'s **Recv** queries using $\mathcal{S}$. What we show is that INT-CS and SIM-STAT (with respect to $\mathcal{S}$) security suffice for this reduction to work and to obtain the bound.

Proof (Theorem 1). Fix $t, s, q_1, q_2, \mu_1, \mu_2 \in \mathbb{N}$ and let $\boldsymbol{r} = (q_1, q_2, \mu_1, \mu_2)$. Let $\mathcal{A}$ be a t-time PRIV-SR adversary with resources $\boldsymbol{r}$ and let $\mathcal{S}$ be an s-time simulator. We exhibit an INT-CS adversary $\mathcal{B}$, a SIM-STAT adversary $\mathcal{C}$, and a PRIV-S adversary $\mathcal{D}$ such that

$$\mathbf{Adv}^{\text{priv-sr}}_{\mathcal{CH},\ell}(\mathcal{A}) \le 2\mathbf{Adv}^{\text{int-cs}}_{\mathcal{CH}}(\mathcal{B}) + 2\mathbf{Adv}^{\text{sim-stat}}_{\mathcal{CH},\mathcal{S}}(\mathcal{C}) + \mathbf{Adv}^{\text{priv-s}}_{\mathcal{CH},\ell}(\mathcal{D})\,,$$

where $\mathcal{B}$ and $\mathcal{C}$ run in time $t + O(q_1 + q_2)$ and each uses query resources $\boldsymbol{r}$, and $\mathcal{D}$ runs in time $t + O(q_1 + sq_2)$ and uses resources (q_1, μ_1).

The proof is by a game-playing argument; refer to games $\mathbf{G}_1, \mathbf{G}_2$, and $\mathbf{G}_3$ defined in Figure 5. Game $\mathbf{G}_1$ is the PRIV-SR notion embellished with a book-keeping flag *win*, whose value is set on line 5:17. However, the value of *win* does not affect the distribution of oracle outputs (or the game) in any way. So for any $\mathcal{A}$ and a uniform random $b \in \{0, 1\}$, the random variables $\mathbf{G}_1(\mathcal{A}) = 1$ and $\mathbf{Exp}^{\text{priv-sr}}_{\mathcal{CH},\ell,b}(\mathcal{A}) = b$ are identically and independently distributed.

Game $\mathbf{G}_2$, which includes the boxed instruction at line 5:18, is identical to game $\mathbf{G}_1$ until the flag *win* gets set. By the Fundamental Lemma of Game Playing [9], we have

$$\begin{aligned}\Pr_b\big[\,\mathbf{Exp}^{\text{priv-sr}}_{\mathcal{CH},\ell,b}(\mathcal{A}) = b\,\big] \le &\Pr\big[\,\mathbf{G}_2(\mathcal{A}) = 1\,\big] + \\ &\Pr\big[\,\mathbf{G}_2(\mathcal{A})\,\mathbf{sets}\ \mathit{win}\,\big]\,.\end{aligned} \tag{1}$$

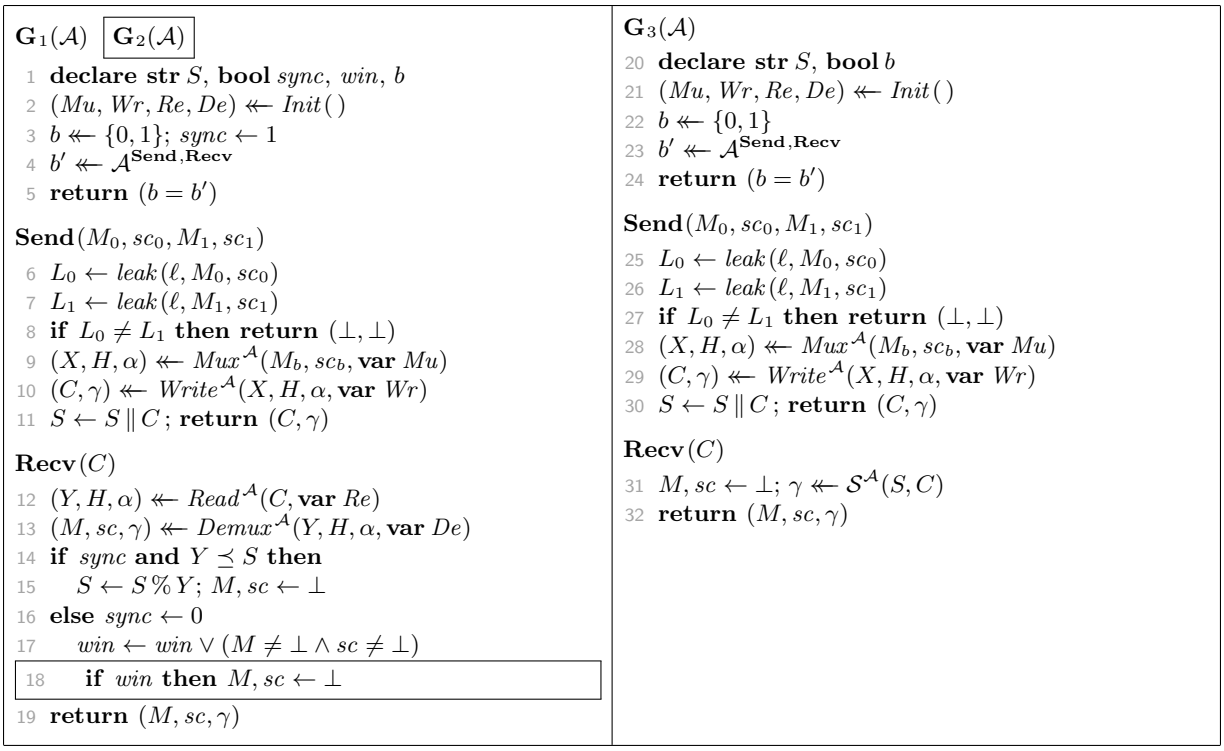

Fig. 5: games $\mathbf{G}_1$, $\mathbf{G}_2$, and $\mathbf{G}_3$ for proof of Theorem 1.

Next, we define an INT-CS adversary $\mathcal{B}^{\mathbf{Send},\mathbf{Recv}}$ as follows. It samples a bit $b \twoheadleftarrow \{0,1\}$, then executes $b' \twoheadleftarrow \mathcal{A}^{\mathbf{Send}',\mathbf{Recv}'}$. On input of (M_0, sc_0, M_1, sc_1), oracle $\mathbf{Send}'$ checks that $leak(\ell, M_0, sc_0) = leak(\ell, M_1, sc_1)$. If not, it outputs $(\bot, \bot)$; otherwise, it asks $(C, \gamma) \twoheadleftarrow \mathbf{Send}(M_b, sc_b)$ and outputs (C, γ). SD requests are simply forwarded to $\mathcal{A}$: for every SD request $req \in \{0,1\}^*$, adversary $\mathcal{B}$ computes $resp \twoheadleftarrow \mathcal{A}(req)$ and responds with $resp$. On input of C, oracle $\mathbf{Recv}'$ asks $(M, sc, \gamma) \twoheadleftarrow \mathbf{Recv}(C)$ and outputs (M, sc, γ). (Again, SD requests are forwarded to $\mathcal{A}$.) Finally, when $\mathcal{A}$ halts, adversary $\mathcal{B}$ halts. It is clear by the definition of the INT-CS game that if $\mathcal{A}$ sets win in its game, then $\mathcal{B}$ also sets win in its game. Hence, for any $\mathcal{A}$,

$$\begin{aligned} \Pr\left[\, \mathbf{G}_2(\mathcal{A}) \,\mathbf{sets}\, win \,\right] &\le \Pr\left[\, \mathbf{Exp}^{\text{int-cs}}_{\mathcal{CH}}(\mathcal{B}) \,\mathbf{sets}\, win \,\right] \\ &= \mathbf{Adv}^{\text{int-cs}}_{\mathcal{CH}}(\mathcal{B})\,. \end{aligned} \tag{2}$$

Observe that in game $\mathbf{G}_2$ the $\mathbf{Recv}$ oracle always returns $(\bot, \bot, \gamma)$, i.e., M and sc are always set to $\bot$. Thus, if the status message γ were predictable *without* knowing the reader or demultiplexer states (Re and De resp.), then game $\mathbf{G}_2$ could be simulated by a PRIV-CPA adversary, because the $\mathbf{Recv}$ oracle in $\mathbf{G}_2$ would be simulatable.

With this observation, we create game $\mathbf{G}_3$ from $\mathbf{G}_2$ by replacing the entire $\mathbf{Recv}$ code with the statement "$M, sc \leftarrow \bot; \gamma \twoheadleftarrow \mathcal{S}^{\mathcal{A}}(S, C)$", where $\mathcal{S}$ is the given simulator for the SIM-STAT security experiment. We also remove the *win* and *sync* flags, as they are no longer relevant. The definition of game $\mathbf{G}_3$ leads us to define a SIM-STAT adversary $\mathcal{C}^{\mathbf{Send},\mathbf{Recv}}$ as follows. It samples a bit $b \twoheadleftarrow \{0,1\}$ and executes $b' \twoheadleftarrow \mathcal{A}^{\mathbf{Send}',\mathbf{Recv}'}$. Queries to $\mathbf{Send}'$ are answered just as they were in the definition of adversary $\mathcal{B}$ above. On input of C, oracle $\mathbf{Recv}'$ asks $\gamma \twoheadleftarrow \mathbf{Recv}(C)$ and outputs $(\bot, \bot, \gamma)$. Finally, when adversary $\mathcal{A}$ halts and outputs b', adversary $\mathcal{C}$ halts and outputs $(b = b')$. Then for any $\mathcal{A}$, $\mathcal{S}$, and $d \in \{0,1\}$,

$$\Pr\left[\, \mathbf{G}_{3-d}(\mathcal{A}) = 1 \,\right] = \Pr\left[\, \mathbf{Exp}^{\text{sim-stat}}_{\mathcal{CH},\mathcal{S},d}(\mathcal{C}) = 1 \,\right] \tag{3}$$

and so

$$\begin{aligned} \Pr\left[\, \mathbf{G}_2(\mathcal{A}) = 1 \,\right] &= \big(\Pr\left[\, \mathbf{G}_2(\mathcal{A}) = 1 \,\right] - \Pr\left[\, \mathbf{G}_3(\mathcal{A}) = 1 \,\right]\big) \\ &\quad + \Pr\left[\, \mathbf{G}_3(\mathcal{A}) = 1 \,\right] \\ &= \mathbf{Adv}^{\text{sim-stat}}_{\mathcal{CH},\mathcal{S}}(\mathcal{C}) + \Pr\left[\, \mathbf{G}_3(\mathcal{A}) = 1 \,\right]. \end{aligned} \tag{4}$$

Lastly, we define a PRIV-S adversary $\mathcal{D}^{\mathbf{Send}}$ as follows. Initialize a variable $\mathbf{str}\, S$ and execute $b' \twoheadleftarrow \mathcal{A}^{\mathbf{Send}',\mathbf{Recv}'}$. On input of (M_0, sc_0, M_1, sc_1), $\mathbf{Send}'$ asks $(C,\gamma) \twoheadleftarrow \mathbf{Send}(M_0, sc_0, M_1, sc_1)$ (forwarding SD requests to $\mathcal{A}$), computes $S \leftarrow S \,\|\, C$, and outputs (C,γ). On input of C, oracle $\mathbf{Recv}'$ executes $\gamma \twoheadleftarrow \mathcal{S}^{\mathcal{A}}(C,S)$ and outputs $(\bot,\bot,\gamma)$. Finally, when $\mathcal{A}$ outputs b', halt and output b'. From the definition of $\mathbf{G}_3$, it is clear that for any $\mathcal{A}$,

$$\Pr\left[\,\mathbf{G}_3(\mathcal{A}) = 1\,\right] = \Pr_b\left[\,\mathbf{Exp}^{\text{priv-s}}_{\mathcal{CH},\ell,b}(\mathcal{D}) = b\,\right]. \tag{5}$$

Summarizing, we have that for every adversary $\mathcal{A}$ and simulator $\mathcal{S}$, there exist adversaries $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$ such that

$$\begin{aligned}
\mathbf{Adv}^{\text{priv-sr}}_{\mathcal{CH},\ell}(\mathcal{A}) &= 2\Big(\Pr_b\left[\,\mathbf{Exp}^{\text{priv-sr}}_{\mathcal{CH},\ell,b}(\mathcal{A}) = b\,\right]\Big) - 1 \\
&\le 2\Big(\Pr\left[\,\mathbf{G}_2(\mathcal{A}) = 1\,\right] + \mathbf{Adv}^{\text{int-cs}}_{\mathcal{CH}}(\mathcal{B})\Big) - 1 \\
&\le 2\Big(\mathbf{Adv}^{\text{sim-stat}}_{\mathcal{CH},\mathcal{S}}(\mathcal{C}) + \Pr\left[\,\mathbf{G}_3(\mathcal{A}) = 1\,\right] + \mathbf{Adv}^{\text{int-cs}}_{\mathcal{CH}}(\mathcal{B})\Big) - 1 \\
&= 2\mathbf{Adv}^{\text{sim-stat}}_{\mathcal{CH},\mathcal{S}}(\mathcal{C}) + \mathbf{Adv}^{\text{priv-s}}_{\mathcal{CH},\ell}(\mathcal{D}) + 2\mathbf{Adv}^{\text{int-cs}}_{\mathcal{CH}}(\mathcal{B}).
\end{aligned} \tag{6}$$

The claimed bound follows. To complete the proof, we note that $\mathcal{B}$ and $\mathcal{C}$ use query resources $\boldsymbol{r}$ and $\mathcal{D}$ uses query resource (q_1, μ_1). Since simulating each query requires $O(1)$ time, adversaries $\mathcal{B}$ and $\mathcal{C}$ run in time $t + O(q_1 + q_2)$, and $\mathcal{D}$ runs in time $t + O(q_1 + sq_2)$. ■

Remark. We emphasize that, although we have used SIM-STAT to prove a generic composition result, the notion is not merely a technical one. The intuition it captures is critical, since the most exploitable weaknesses in authenticated encryption schemes used in secure channels has been distinguishable error messages being surfaced in the protocol [32,15,24,3]. As a result, there has been a considerable push in the cryptographic community to make addressing these subtleties a first class consideration [13,19,6].

3 The TLS 1.3 record layer

Our study of partially specified channels streams owes much to a desire to analyze the TLS 1.3 record layer, in particular without eliding its optional features and unspecified details. So, we begin this section with an overview of some of its salient features, and a discussion of certain design choices that may have implications when the record layer is viewed through the lens of our security notions. This is followed (in Section 3.2) by a decomposition of the record layer into its component building blocks. Then we show how to securely compose these into a PSC that *nearly* formalizes the specification; we propose a small change to the standard that significantly improves flexibility of the scheme.

Note about the draft. This analysis pertains to draft 23 [28], current at the time of writing. Note that the change to the record layer we suggest here will be adopted in the final version of the protocol.[2]

3.1 Overview

TLS can be viewed as three client-server protocols executing concurrently: the *handshake* protocol handles (re-)initialization of the channel; the *record* protocol is used to exchange application data between the client and the server; and the *alert* protocol is used to close the channel. The *record layer* refers to the mechanism used to protect flows between client and server in each sub-protocol. Each of these flows is authenticated and encrypted as soon as the client and server have exchanged key material. (Usually the only unprotected messages are the initial client_hello and part of the server_hello.) Intuitively, each of these flows constitutes a logical data stream, and the record layer is a means of multiplexing these streams over a single communications channel (e.g., a TCP connection). Among the record layer's many design criteria is the need to maximize flexibility for implementations. This means, somewhat paradoxically, that the specification does

[2] Our change was implemented in draft 25; since our analysis, no other changes have been made to the record layer.

not fully specify every aspect of the construction. Rather, the record-layer specification [28, Section 5] defines some core functionalities that must be implemented and provides a set of parameters for compliant, fully realized schemes.

Content types. Each stream has an associated *content type*. Available types are handshake, application data, alert, and change ciphersuite spec (CCS); additional content types may be added subject to certain guidelines [28, Section 11]. If the client or server receives a message of unknown content type, it must send an unexpected_message alert to its peer and terminate the connection. The CCS type is only available for compatibility with systems accustomed to processing records for TLS 1.2 and earlier. Usually a CCS message must be treated as an unexpected message, but under specific conditions, it must simply be dropped.

Records. Plaintext records encode the content type, the stream fragment, the length of the fragment (which may not exceed 2^{14} bytes), and an additional field called *legacy_record_version*, whose value is fixed by the specification. (It is only present for backwards compatibility.) All flows, including unprotected ones (the initial handshake message and CCS messages) are formatted in this manner. The streams of data are transformed into a sequence of records; stream fragments of the same content type may be coalesced into a single record, but the *record boundaries* are subject to the following rules [28, Section 5.1]:

- *Handshake, no interleaving:* if two records correspond to a single handshake message, then they must be adjacent in the sequence of records.
- *Handshake, no spanning a key change:* if two records correspond to a single handshake message, then they both must precede the next key change (defined in Section 3.1). If this condition is violated, then the second record must be treated as an unexpected message.
- *Handshake and alert, no zero-length messages:* only application data records may have zero length.
- *One alert per record:* alert messages must not be fragmented across records, and a record containing an alert message must contain only that message.

Additional content types must stipulate appropriate rules for record boundaries.

Records are optionally padded and then protected using an AEAD scheme [28, Sections 5.2–5.4]. First, the record R is encoded as a string $X = R.\mathit{fragment} \,\|\, \langle R.\mathit{type}\rangle_8 \,\|\, (\langle 0\rangle_8)^p$ for some $p \in \mathbb{N}$ such that the length of the ciphertext is less than $2^{14} + 256$ bytes. The padded record X is encrypted with associated data ε (the empty string) and with a nonce N that we will define in a moment. The protected record is defined as

type struct { **int** *opaque_type*, *legacy_record_version*, *length*,
str *encrypted_record* } **TLSCiphertext**

where *opaque_type* has a fixed value (23), *legacy_record_version* has a fixed value (771, or 0x0303 in hexadecimal), and *length* is the length of *encrypted_record* in bytes. The nonce N is computed from a sequence number *seqn* and an initialization vector IV [28, Section 5.3]; both the key K and IV are derived from a shared secret [28, Sections 7.1–7.2] using an extract-and-expand key-derivation scheme [21]. The length of the IV is determined from the permitted nonce lengths of the AEAD scheme.[3] The nonce N is computed as $IV \oplus \langle \mathit{seqn}\rangle_{|IV|}$, where $0 \le \mathit{seqn} \le 2^{64} - 1$. Note that the client and server each uses a different key and IV for sending messages to the other; thus, each constitutes a unidirectional channel.

Usage limits, key changes, and protocol-level side-effects. The spec mandates that the key be changed prior to the sequence number reaching its limit of $2^{64} - 1$ in order to prevent nonce reuse. It also recommends that implementations keep track of how many bytes of plaintext have been encrypted and decrypted with a single key and to change the key before the "safety limit" of the underlying AEAD scheme has been reached.

As mentioned above, upon receipt of a message of unknown type, the receiver should send its peer an unexpected_message alert message. The alert stream is generally used to notify the recipient that the peer is tearing down its connection and will no longer write to the channel. There are *closure* alerts and *error* alerts [28, Section 6]. Both signal the tear down of the writer state, but they provide different feedback. The unexpected_message alert is an example of the latter. Error alerts are also used to indicate things like the ciphertext is inauthentic, or the record is malformed. An example of the former is close_notify, which indicates that the receiver should not expect any more data from the peer, but that no error occurred.

[3] The scheme must specify limits for valid nonce lengths per RFC 5116 [22]. The maximum must be at least 8 bytes.

The key and IV change during the normal course of the protocol. An update is always a side effect of the handshake protocol. During transmission of application data, an update is signaled by a particular handshake message described in [28, Section 4.6.3], which informs the receiver that the sender has reinitialized its state and so must do so as well. The key change re-initializes the state of the sender and receiver with a fresh key and IV (derived from the shared secret), and the sequence number is set to 0 [28, Section 5.3]. Therefore, no sender or receiver state (that is, no state that pertains to the record layer) is held over after re-initialization of the channel.

Observations about the standard. The standard defines some core functionalities, but leaves many design choices up to the implementer; our analysis aims to establish what security the record layer provides given this level of flexibility. Our approach is shaped by two questions. First, which fully specified components can be altered without impacting security? Second, which unspecified or partially specified components are security critical? We begin with a couple of observations.

Record boundaries may leak the content type. The content type of each record is encrypted along with the fragment. The intent, presumably, is to hide both the content *and* its type, but the record boundary rules stipulated by the standard make hiding the type unachievable in general. Consider the *one alert per record* rule, for example. The implementation is allowed to coalesce fragments of the same type, but a record containing an alert must contain only that alert. Thus, the *length* of each record output by the sender may (depending on the implementation) leak whether the record pertains to an alert or to application data. Of course, the standard does *permit* implementations that hide the content type of each record, but this is quite different from *mandating* this property. The take away is that *encrypting the content type does not imply indistinguishibility of the content type*, since the record boundaries depend on it.

Associated data is unauthenticated. One aspect of the scheme that is precisely defined is the format of the ciphertext transmitted on the wire. Each begins with a header composed of *opaque_type*, *legacy_record_version*, and *length*. The values of the first two fields are fixed by the spec, and the last field is crucial for correct operation, since it informs the receiver of how many bytes to read next. What should the receiver do if the header is different than specified? Changing the *length* field bits should result in the next ciphertext either being too short or too long, and so would be deemed inauthentic with overwhelming probability. If *opaque_type* or *legacy_record_version* is mangled, then it should be safe to proceed since this does not affect the inputs to decryption. However, doing so would be deemed an attack in our ciphertext-integrity setting; changing these bits means the stream is out-of-sync, but since they are not authenticated (encryption uses ε for associated data), the receiver would successfully decrypt. In fact, checking the *opaque_type* and *legacy_record_version* fields is left optional by the spec: implementations MAY check these fields are correct and abort the connection if not [28, Section 5.2]. This presents us with a dilemma: if we leave this choice up to the specification details, then there is a trivial INT-CS attack, and so in order to salvage security, we need to lift this "MAY" to a "MUST".

This dilemma points to something rather strange about the record layer's design: something that ought not be security critical—in particular, the value of the delimiter bits—*is* security critical. Indeed, this observation motivates our partially specified viewpoint. To formalize the idea that the value of the delimeter bits should not impact security, we simply let the specification details *choose* these bits itself this is safe as long as the bits are authenticated and do not depend on sensitive values. We will formalize this idea in our PSC in Section 3.3.

Remark. An alternative conclusion is that this vulnerability is only an artifact of our strong adversarial model; mangling the delimiter bits should not affect the inputs to decryption, and so does not constitute a "real attack" on privacy or integrity in an intuitive sense. To this point we offer a warning: *this intuition is correct only if down-stream handling of the plaintext is independent of the contents of these fields.* Since such behavior is beyond the scope of the TLS standard (and even our security model), these legacy fields constitute an attack surface for implementations. The risk is not inconsiderable, as it is difficult to predict how systems will evolve to make use of TLS, and of these bits in particular. Indeed, they owe their very existence to the need to maintain compatibility with older systems.

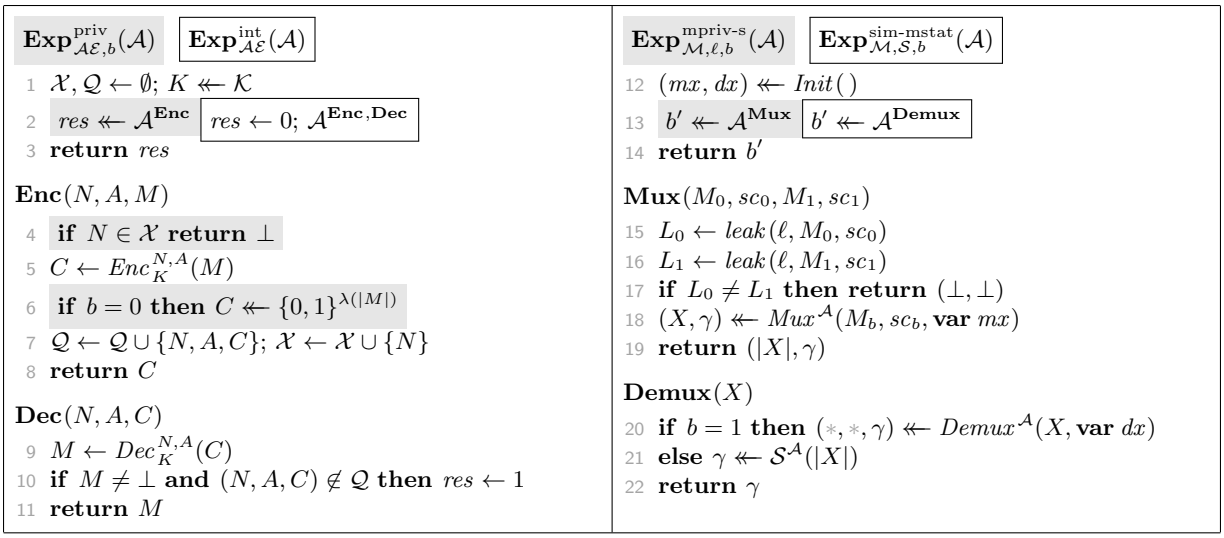

Fig. 6: left: security games PRIV and INT for AEAD scheme $\mathcal{AE} = (Enc, Dec, \lambda)$ with key space $\mathcal{K}$. Right: security games mPRIV-S and SIM-mSTAT for partially specified stream multiplexer $\mathcal{M} = (Init, Mux, Demux)$. The former has an associated permitted leakage parameter $\ell \in \{\mathsf{lensc}, \mathsf{len}, \mathsf{none}\}$; procedure *leak* is as defined in Figure 2.

3.2 The building blocks

In this section we formalize the core components of the record layer; our aim is to sweep all but these building blocks into the specification details. The first primitive, called a *stream multiplexer*, captures the non-cryptographic functionality of the underlying channel. It transforms the data streams into a sequence of channel fragments (i.e. records) such that for each stream context (i.e. content type), the output on the receiver end is a prefix of the input on the sender side. TLS offers a great deal of flexibility with respect to the stream multiplexer's operation; the flip side is that design choices here impact security of the overall construction. (Recall the discussion of record boundaries in Section 3.1.) Thus, it will be useful to consider stream multiplexers that are only partially specified. The remaining primitives are a scheme for authenticated encryption with associated data and a method of generating nonces. These are the core cryptographic functionalities and must be implemented correctly; as such, we will require these to be fully specified.

Stream multiplexers. First, a partially specified *stream-multiplexer* is a triple $\mathcal{M} = (Init, Mux, Demux)$ defined as follows.

- $Init(\,) \mapsto (\mathbf{str}\ mx, dx)$. Generates the initial state of the stream multiplexer (used by the sender) and demultiplexer (used by the receiver).
- $Mux^{\mathcal{O}}(\mathbf{str}\ M, sc, \mathbf{var\ str}\ mx) \mapsto (\mathbf{str}\ X, \gamma)$. Takes as input a plaintext fragment M, its stream context sc, and the current state mx, and returns a channel fragment X and a status message γ.
- $Demux^{\mathcal{O}}(\mathbf{str}\ X, \mathbf{var\ str}\ dx) \mapsto (\mathbf{str}\ M, sc, \gamma)$. Takes a channel fragment X and the current state dx and returns a plaintext fragment M, its stream context sc, and the status γ.

The specification details are provided by the oracle $\mathcal{O}$. Our intention is to capture only non-cryptographic functionalities with stream multiplexers. (Of course, $\mathcal{M}$ may, in principal, use some sort of cryptographic primitive, or even output encrypted records.) In order to facilitate a rigorous analysis of how design choices here impact security of the channel overall, we formulate two security properties for partially specified multiplexers. Both are defined in Figure 6.

The mPRIV-S notion. The first captures an adversary's ability to discern information about the inputs to *Mux* given (information about) its outputs. Like the PRIV-S game (Section 2.2), the mPRIV-S game

is parameterized by the permitted leakage ℓ, one of lensc, len, or none (see Figure 2), and a challenge bit b. It is given an oracle **Mux** with the same interface as **Send** in the PRIV-S game. The oracle invokes procedure *Mux* on inputs (M_b, sc_b) (and with oracle access to $\mathcal{A}$ for handling SD requests), and the adversary is asked to guess b based on the outcome of its queries. Where the games differ, however, is in the information available to the adversary. Rather than return (X, γ) directly, the oracle returns γ and only the *length of* X. This captures a much weaker property than usual indistinguishibility: rather than insisting (X, γ) not leak anything beyond $L = \mathit{leak}(\ell, M, sc)$, we insist only that $(|X|, \gamma)$ not leak anything beyond L. Define the advantage of $\mathcal{A}$ in attacking $\mathcal{M}$ in the mPRIV-S(ℓ) sense as

$$\mathbf{Adv}^{\text{mpriv-s}}_{\mathcal{M},\ell}(\mathcal{A}) = 2 \Pr_b\big[\,\mathbf{Exp}^{\text{mpriv-s}}_{\mathcal{M},\ell,b}(\mathcal{A}) = b\,\big] - 1\,.$$

Let $\mathbf{Adv}^{\text{mpriv-s}}_{\mathcal{M},\ell}(t, q, \mu)$ denote the maximum advantage of any t-time adversary making at most q queries to **Mux** with total bit-length at most μ.

The SIM-mSTAT notion. The second notion captures simulatability of the status message output by *Demux*. It is associated with a simulator $\mathcal{S}$ and a bit b. After initialization, the adversary is given access to an oracle **Demux**. On input of X, if $b = 1$, then the oracle executes procedure *Demux* on input X and returns the status message; otherwise it executes the simulator $\mathcal{S}$ on input $|X|$ and with access to $\mathcal{A}$ for servicing SD requests. Define the advantage of $\mathcal{A}$ in attacking $\mathcal{M}$ in the SIM-mSTAT sense with simulator $\mathcal{S}$ as

$$\mathbf{Adv}^{\text{sim-mstat}}_{\mathcal{M},\mathcal{S}}(\mathcal{A}) = 2 \Pr_b\big[\,\mathbf{Exp}^{\text{sim-mstat}}_{\mathcal{M},\mathcal{S},b}(\mathcal{A}) = b\,\big] - 1\,.$$

Let $\mathbf{Adv}^{\text{sim-mstat}}_{\mathcal{M},\mathcal{S}}(t, q, \mu)$ denote the maximum advantage of any t-time adversary making q queries to **Demux** with total bit-length at most μ.

AEAD schemes. We describe the syntax for authenticated encryption with associated data as prescribed by the spec [22]. An AEAD scheme is a triple $\mathcal{AE} = (\mathit{Enc}, \mathit{Dec}, \lambda)$. The last element is a function $\lambda : \mathbb{Z} \to \mathbb{Z}$ which describes the ciphertext length as a function of the plaintext length; we insist that λ is a bijection. Algorithms *Enc* and *Dec* are both deterministic and have the following interfaces:

- $\mathit{Enc}(\mathbf{str}\, K, N, A, M) \mapsto \mathbf{str}\, C$. Maps a key K, nonce N, associated data A, and plaintext M to a ciphertext C such that if $C \neq \perp$, then $|C| = \lambda(|M|) \geq |M|$.
- $\mathit{Dec}(\mathbf{str}\, K, N, A, C) \mapsto \mathbf{str}\, M$. Maps K, N, A, and C to M such that if $M \neq \perp$, then $\lambda^{-1}(|C|) = |M|$.

We may denote the execution of *Enc* on (K, N, A, M) by $\mathit{Enc}_K^{N,A}(M)$. (Similarly for *Dec*.) We respectively define the key, nonce, associated-data, and message space as the sets $\mathcal{K}, \mathcal{N}, \mathcal{A}, \mathcal{M} \subseteq \{0,1\}^*$ for which $\mathit{Enc}(K, N, A, M) \neq \perp$ if and only if $(K, N, A, M) \in \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{M}$; correctness requires that $\mathit{Dec}(K, K, N, A, \mathit{Enc}(K, N, A, M)) = M$ for every such (K, N, A, M). (This condition implies that $\mathcal{AE}$ is both *correct* and *tidy* in the sense of Namprempre, Rogaway, and Shrimpton [23].)

We will use standard notions of indistinguishibility under chosen-plaintext attack (PRIV) and integrity of ciphertexts (INT) as defined in Figure 6. As usual, the indistinguishibility game requires that the adversary not repeat a nonce. Define the PRIV advantage of adversary $\mathcal{A}$ in attacking $\mathcal{AE}$ as

$$\mathbf{Adv}^{\text{priv}}_{\mathcal{AE}}(\mathcal{A}) = 2 \Pr_b\big[\,\mathbf{Exp}^{\text{priv}}_{\mathcal{AE},b}(\mathcal{A}) = b\,\big] - 1$$

and let $\mathbf{Adv}^{\text{priv}}_{\mathcal{AE}}(t, q, \mu)$ denote the maximum advantage of any t-time adversary making at most q queries with total bit-length μ. Define the INT advantage of adversary $\mathcal{A}$ in attacking $\mathcal{AE}$ as

$$\mathbf{Adv}^{\text{int}}_{\mathcal{AE}}(\mathcal{A}) = \Pr\big[\,\mathbf{Exp}^{\text{int}}_{\mathcal{AE}}(\mathcal{A}) = 1\,\big]$$

and let $\mathbf{Adv}^{\text{int}}_{\mathcal{AE}}(t, q_1, q_2, \mu_1, \mu_2)$ be the maximum advantage of any t-time adversary making at most q_1 (resp. q_2) queries to **Enc** (resp. **Dec**) with total bit-length at most μ_1 (resp. μ_2).

```
// The sender state.
1  type struct { str ng, mx } Muxer
2  type struct { str K } Writer
// The receiver state.
3  type struct { str ng, buf } Reader
4  type struct {
5     str K, dx, bool sync
6  } Demuxer

Init( )
7  declare Muxer Mu, Writer Wr
8  declare Reader Re, Demuxer De
9  (Mu.mx, De.dx) ⇜ M.Init( )
10 Mu.ng ⇜ N.Init( ); Re.ng ← Mu.ng
11 Wr.K ⇜ K; De.K ← Wr.K
12 De.sync ← 1
13 return (Mu, Wr, Re, De)

Mux^O(M, sc, var Muxer Mu)
14 (X, α) ⇜ M.Mux^O(M, sc, var Mu.mx)
15 if X = ⋄ then return (⋄, ⋄, α)
16 N ← N.Next(var Mu.ng)
17 return (X, N, α)

Write^O(X, N, α, var Writer Wr)
18 declare str A, γ
19 ⟨A, γ⟩ ⇜ O(⟨write, create ad, |X|, α⟩)
20 if X = ⋄ then return (⋄, γ)
21 Y' ← AE.Enc(Wr.K, N, A, X)
22 if Y' = ⊥ then
23    γ ⇜ O(⟨write, invalid ptxt⟩)
24    return (⋄, γ)
25 return (A ∥ Y', γ)

Read^O(C, var Reader Re)
26 declare str α, int c, bool drop
27 Re.buf ← Re.buf ∥ C
28 ⟨c, drop, α⟩ ⇜ O(⟨read, len, Re.buf⟩)
29 Y ← Re.buf[:c]; Re.buf ← Re.buf % Y
30 if Y = ⋄ or drop then return (⋄, ⋄, α)
31 N ← N.Next(var Re.ng)
32 return (Y, N, α)

Demux^O(Y, N, α, var Demuxer De)
33 declare str X, γ, int a
34 ⟨a, γ⟩ ⇜ O(⟨demux, ad len, Y, α⟩)
35 if (Y = ⋄ and γ ≠ ⋄) or ¬De.sync
36    then return (⊥, ⊥, γ)
37 else if Y ≠ ⋄ then
38    A ← Y[:a]; Y' ← Y % A
39    X ← AE.Dec(De.K, N, A, Y')
40    if X = ⊥ then
41       De.sync ← 0
42       γ ⇜ O(⟨demux, invalid ctxt⟩)
43       return (⊥, ⊥, γ)
44 (M, sc, γ) ⇜ M.Demux^O(X, var De.dx)
45 return (M, sc, γ)
```

Fig. 7: partially specified channel $\mathbf{TLS}[\mathcal{M}, \mathcal{AE}, \mathcal{N}] = (\mathit{Init}, \mathit{Mux}, \mathit{Write}, \mathit{Read}, \mathit{Demux})$ composed of a partially specified stream multiplexer $\mathcal{M}$, an AEAD scheme $\mathcal{AE}$ with key space $\mathcal{K}$, and a nonce generator $\mathcal{N}$.

Nonce generators. Finally, a *nonce generator* is a pair of algorithms $\mathcal{N} = (\mathit{Init}, \mathit{Next})$, the first randomized and the second deterministic.

- $\mathit{Init}( ) \mapsto \mathbf{str}\ ng$. Initializes the state of the generator.
- $\mathit{Next}(\mathbf{var}\ \mathbf{str}\ ng) \mapsto \mathbf{str}\ N$. Computes the next nonce N given the current state ng and updates the state.

We associate to $\mathcal{N}$ and an integer $t \in \mathbb{N}$ a procedure Coll, which first executes $ng \twoheadleftarrow \mathit{Init}( )$, then computes $N_i \leftarrow \mathit{Next}(\mathbf{var}\ ng)$ for each $i \in [t]$. Finally, if for every $1 \le i < j \le t$ it holds that $N_i \neq N_j$, then the procedure outputs 0; otherwise it outputs 1. Define $\mathsf{coll}_{\mathcal{N}}(t) = \Pr\left[\, \mathit{Coll}_{\mathcal{N}}(t) = 1 \,\right]$.

3.3 The partially specified record layer

We are now ready to formalize TLS 1.3 record layer specification. Refer to the PSC $\mathbf{TLS}[\mathcal{M}, \mathcal{AE}, \mathcal{N}] = (\mathit{Init}, \mathit{Mux}, \mathit{Write}, \mathit{Read}, \mathit{Demux})$ defined in Figure 7. It differs from the standard (draft 23) in one small, but security-critical way: the standard mandates that the AEAD be invoked with ε as the AD, whereas in our scheme, the string A—the record header—is used as AD. To fully comply with the spec, one would replace A with ε on lines 7:21 and 7:39. However, this leads to a trivial ciphertext stream integrity attack:

suppose the sender outputs $Y = A \,\|\, Y'$. Then the adversary can deliver $A^* \,\|\, Y'$ to the receiver for some $A^* \neq A$ where $|A^*| = |A|$. If Y is consumed by the receiver, then the channel will be deemed out of sync, but the output of the receiver will be unaffected. We remind the reader that this change was adopted in the final version of the TLS 1.3 standard; what follows is a formal, partial specification of the record layer.

The procedure *Mux* invokes $\mathcal{M}$ (7:14) in order to compute the next channel fragment (i.e. record). It is designed to never operate on 0-length channel fragments (7:15); if the first input X to *Write* is undefined (i.e., $X = \varepsilon$), then it outputs a 0-length ciphertext fragment (7:20). The data on the wire is $A \,\|\, Y'$, where Y' is the ciphertext and A is a string chosen by the SD (7:19).

Defragmentation of the ciphertext is performed by *Read* and is also left largely up to the SD: first, the ciphertext fragment is appended to a buffer *buf*, then the SD is invoked to decide how much of the buffer to dequeue next. The oracle is given the contents of the buffer and outputs an integer c. It also sets a flag *drop*. If $Y = \mathit{buf}[:c] \neq \diamond \wedge \neg \mathit{drop}$ holds, then the next nonce is computed and output along with Y. Otherwise the reader outputs $Y = \diamond$ and $N = \diamond$. (Note that the *drop* flag permits the rules for handling CCS messages; such a message will never be produced by the sender, but it may be transmitted to the receiver.) Presumably, Y is equal to $A \,\|\, Y'$, where Y' is a ciphertext and A is a string chosen by the SD. On input of Y, the SD is invoked to determine the length of A (7:34). If $Y \neq \diamond$, then string Y' is decrypted (using A as associated data) and the resulting channel fragment X (i.e. record) is input to the stream demultiplexer.

If *Demux* ever encounters an invalid ciphertext, then thereafter it never outputs a valid fragment (7:34 and 7:40–42). It uses a flag *sync* to track this. If the receiver is in-sync and Y is 0-length, then *Demux* may poll the stream demultiplexer to see if a message fragment is available for outputting. (That is, line 7:43 may be invoked on $X = \varepsilon$.) Usage limits are enforced by the SD (7:19 and 7:33).

Our construction captures all protocol-level side effects in the record layer specification [28] with the exception of any sender or receiver state carried over after re-initialization of the channel. Indeed, our security model does not encompass re-initialization, since the game is defined for an already initialized channel. We made this choice because the record layer was designed so that no state is carried across key changes. (See the discussion Section 3.1.) As a result, there is no need to model key changes; security in the stronger model where the adversary controls (re-)initialization of the channel will follow from the independence of the state between changes.

Security. Let $\mathcal{CH} = \mathbf{TLS}[\mathcal{M}, \mathcal{AE}, \mathcal{N}]$ be as defined in Figure 7. Our first step is to show that PRIV of $\mathcal{AE}$ and mPRIV-S of $\mathcal{M}$ imply PRIV-S for $\mathcal{CH}$:

Theorem 2. *Let $\ell \in \{\mathsf{lensc}, \mathsf{len}, \mathsf{none}\}$. For every $t, q, \mu \in \mathbb{N}$ and $\tilde{t} = t + O(q)$ it holds that*

$$\mathbf{Adv}^{\text{priv-s}}_{\mathcal{CH},\ell}(t, q, \mu) \leq \mathbf{Adv}^{\text{mpriv-s}}_{\mathcal{M},\ell}(\tilde{t}, q, \mu) + 2\mathbf{Adv}^{\text{priv}}_{\mathcal{AE}}(\tilde{t}, q, \mu) + 2\,\mathsf{coll}_{\mathcal{N}}(q)\,.$$

Proof. Fix $t, q, \mu \in \mathbb{N}$ and let $\mathcal{A}$ be a t-time, PRIV-S adversary with query resources (q, μ). We exhibit an mPRIV-S adversary $\mathcal{B}$ and a PRIV adversary $\mathcal{C}$ such that $\mathbf{Adv}^{\text{priv-s}}_{\mathcal{CH},\ell}(\mathcal{A}) \leq \mathbf{Adv}^{\text{mpriv-s}}_{\mathcal{M},\ell}(\mathcal{B}) + 2\mathbf{Adv}^{\text{priv}}_{\mathcal{AE}}(\mathcal{C}) + 2\,\mathsf{coll}_{\mathcal{N}}(q)$, where each runs in time $t + O(q)$ and and uses the same query resources.

Let λ denote the ciphertext-length function associated with $\mathcal{AE}$. Refer to game $\mathbf{G}_1$ defined in Figure 8. One can easily check that for any $\mathcal{A}$ and a uniformly chosen b, the events $\mathbf{G}_1(\mathcal{A}) = 1$ and $\mathbf{Exp}^{\text{priv-s}}_{\mathcal{CH},\ell,b}(\mathcal{A}) = b$ are identically distributed. In game $\mathbf{G}_2$, the implementation of procedure *Write* is modified so that its output differs from game $\mathbf{G}_1$ if the flag *coll* gets set (8:15). By the Fundamental Lemma of Game Playing [9],

$$\Pr\left[\,\mathbf{G}_1(\mathcal{A}) = 1\,\right] \leq \Pr\left[\,\mathbf{G}_2(\mathcal{A}) = 1\,\right] + \Pr\left[\,\mathit{Coll}_{\mathcal{N}}(q) = 1\,\right]. \tag{7}$$

In $\mathbf{G}_2$, if a nonce N input to procedure *Write* is ever repeated, then the output Y' of the invocation of $\mathcal{AE}.\mathit{Enc}$ gets set to $\perp$ (8:15). Hence, the semantics of Y' is the same as the output of $\mathbf{Enc}(N, A, X)$ in the PRIV game for $b = 1$. In game $\mathbf{G}_3$, the invocation of $\mathcal{AE}.\mathit{Enc}$ is replaced with uniformly-chosen, $\lambda(|X|)$-bit string.

Adversary $\mathcal{C}^{\mathbf{Enc}}$ simulates $\mathcal{A}$ in game $\mathbf{G}_2$ as follows. It first initializes the multiplexer state by running $(\mathit{Mu}, *, *, *) \twoheadleftarrow \mathit{Init}(\,)$, then samples a bit b. It then executes $b' \twoheadleftarrow \mathcal{A}^{\mathbf{Send}'}$, where $\mathbf{Send}'$ is defined like $\mathbf{Send}$, except the invocation of $\mathcal{AE}.\mathit{Enc}(\mathit{Wr}.K, N, A, X)$ (8:14) is replaced with $\mathbf{Enc}(N, A, X)$. When $\mathcal{A}$ halts, adversary $\mathcal{C}$ halts and outputs $(b = b')$. By construction, we have that

$$\Pr\left[\,\mathbf{Exp}^{\text{priv}}_{\mathcal{AE},d}(\mathcal{C}) = d\,\right] = \Pr\left[\,\mathbf{G}_{3-d}(\mathcal{A}) = 1\,\right] \tag{8}$$

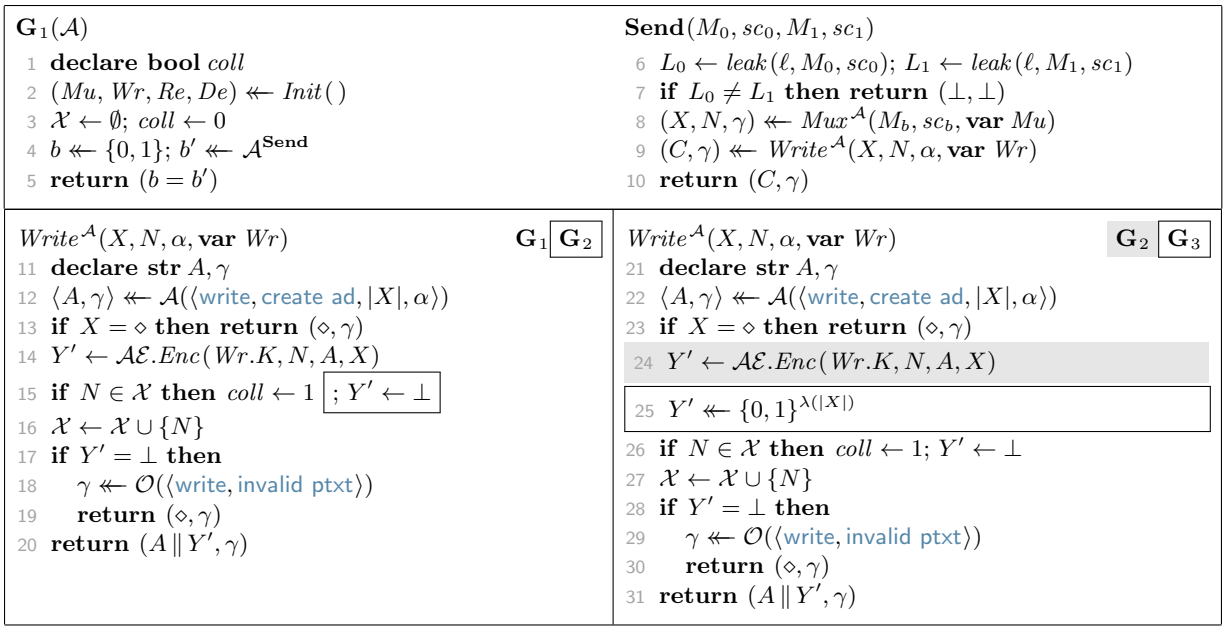

Fig. 8: Bottom: games $\mathbf{G}_1$, $\mathbf{G}_2$, and $\mathbf{G}_3$ for the proof of Theorem 2.

for each $d \in \{0,1\}$, which implies, by a standard conditioning argument, that

$$\Pr\left[\mathbf{G}_2(\mathcal{A}) = 1\right] \leq \Pr\left[\mathbf{G}_3(\mathcal{A}) = 1\right] + \mathbf{Adv}^{\mathrm{priv}}_{\mathcal{AE}}(\mathcal{C})\,. \tag{9}$$

Finally, adversary $\mathcal{B}$ is defined in Figure 9. It simulates $\mathcal{A}$ in a game $\mathbf{G}_6$, which we define in a moment. In the remainder, we will show that

$$\Pr_b\left[\mathbf{Exp}^{\text{mpriv-s}}_{\mathcal{M},\ell,b}(\mathcal{B}) = b\right] = \Pr\left[\mathbf{G}_3(\mathcal{A}) = 1\right]. \tag{10}$$

The remaining transitions do not alter the semantics of the game; they serve only to clarify the reduction. Refer to game $\mathbf{G}_4$ (Figure 9). The difference between it and $\mathbf{G}_3$ is that invocation of procedure *Mux* has been replaced with its definition. In game $\mathbf{G}_5$, returning $(\bot, \bot)$ in case $M = \bot$ or $sc = \bot$ (9:10) is deferred until after invoking $\mathcal{M}.Mux$ (9:14). In game $\mathbf{G}_6$, the string $1^{|X|}$ is passed to *Write* instead of X (9:25). But the output of *Write* does not depend on X; it only depends on $|X|$, N, α, and its current state Wr. (This is due to our revision in game $\mathbf{G}_3$.) Hence, these games are identically distributed. Now, the definition of $\mathcal{B}$'s simulated $\mathbf{Send}'$ oracle is obtained by first replacing lines 9:19–21 with an invocation of its own oracle $\mathbf{Mux}$. Then each instance of string X is replaced with integer x, the first of the outputs of $\mathbf{Mux}$. As usual, adversary $\mathcal{B}$ handles any SD requests by forwarding them to $\mathcal{A}$. It is easy to check that $\mathcal{A}$'s view is the same in the simulation as it is in game $\mathbf{G}_6$, which yields equation (14).

Putting together equations (7)–(10) and our observation about game $\mathbf{G}_1$ yields the final bound. To complete the proof, we observe that $\mathcal{B}$ and $\mathcal{C}$ each runs in time $t + O(q)$ (performing a constant amount amount of computation for each of $\mathcal{A}$'s queries) and makes at most q queries to its oracle, and the total length of the inputs does not exceed μ. ■

Next, integrity of the ciphertext stream follows easily from the ciphertext integrity of $\mathcal{AE}$:

Theorem 3. *For every $t, q_1, q_2, \mu_1, \mu_2 \in \mathbb{N}$ it holds that $\mathbf{Adv}^{\text{int-cs}}_{\mathcal{CH}}(t, \boldsymbol{r}) \leq \mathbf{Adv}^{\text{int}}_{\mathcal{AE}}(t + O(q_1 + q_2), \boldsymbol{r})$, where $\boldsymbol{r} = (q_1, q_2, \mu_1, \mu_2)$,*

Proof. Fix $t, q_1, q_2, \mu_1, \mu_2 \in \mathbb{N}$. Let $\mathcal{A}$ be a t-time INT-CS adversary using query resources (q_1, q_2, μ_1, μ_2). We exhibit an INT adversary $\mathcal{B}$ such that $\mathbf{Adv}^{\text{int-cs}}_{\mathcal{CH}}(\mathcal{A}) \leq \mathbf{Adv}^{\text{int}}_{\mathcal{AE}}(\mathcal{B})$ and $\mathcal{B}$ runs in time $t + O(q_1 + q_2)$ and uses the same query resources as $\mathcal{A}$.

Send(M_0, sc_0, M_1, sc_1) $\mathbf{G}_3$ $\boxed{\mathbf{G}_4}$

$L_0 \leftarrow leak(\ell, M_0, sc_0)$; $L_1 \leftarrow leak(\ell, M_1, sc_1)$
if $L_0 \neq L_1$ **then return** $(\bot, \bot)$
$(X, N, \alpha) \twoheadleftarrow Mux^{\mathcal{A}}(M_b, sc_b, \textbf{var } Mu)$
$(X, \alpha) \twoheadleftarrow \mathcal{M}.Mux^{\mathcal{A}}(M_b, sc_b, \textbf{var } Mu.st)$
if $X = \diamond$ **then** $N \leftarrow \diamond$
else $N \leftarrow \mathcal{N}.Next(\textbf{var } Mu.ng)$
$(C, \gamma) \twoheadleftarrow Write^{\mathcal{A}}(X, N, \alpha, \textbf{var } Wr)$
return (C, γ)

Send(M_0, sc_0, M_1, sc_1) $\mathbf{G}_4$ $\boxed{\mathbf{G}_5}$

$L_0 \leftarrow leak(\ell, M_0, sc_0)$; $L_1 \leftarrow leak(\ell, M_1, sc_1)$
if $L_0 \neq L_1$ **then return** $(\bot, \bot)$
$(X, \alpha) \twoheadleftarrow \mathcal{M}.Mux^{\mathcal{A}}(M_b, sc_b, \textbf{var } Mu.st)$
if $L_0 \neq L_1$ **then** $(X, \alpha) \leftarrow (\bot, \bot)$
else $(X, \alpha) \twoheadleftarrow \mathcal{M}.Mux^{\mathcal{A}}(M, sc, \textbf{var } mu)$
if $X = \bot$ **and** $\alpha = \bot$ **then return** $(\bot, \bot)$
if $X = \diamond$ **then** $N \leftarrow \diamond$
else $N \leftarrow \mathcal{N}.Next(\textbf{var } Mu.ng)$
$(C, \gamma) \twoheadleftarrow Write^{\mathcal{A}}(X, N, \alpha, \textbf{var } Wr)$
return (C, γ)

Send(M_0, sc_0, M_1, sc_1) $\mathbf{G}_5$ $\boxed{\mathbf{G}_6}$

$L_0 \leftarrow leak(\ell, M_0, sc_0)$; $L_1 \leftarrow leak(\ell, M_1, sc_1)$
if $L_0 \neq L_1$ **then** $(X, \alpha) \leftarrow (\bot, \bot)$
else $(X, \alpha) \twoheadleftarrow \mathcal{M}.Mux^{\mathcal{A}}(M, sc, \textbf{var } mu)$
if $X = \bot$ **and** $\alpha = \bot$ **then return** $(\bot, \bot)$
if $X = \diamond$ **then** $N \leftarrow \diamond$
else $N \leftarrow \mathcal{N}.Next(\textbf{var } Mu.ng)$
$(C, \gamma) \twoheadleftarrow Write^{\mathcal{A}}(X \boxed{1^{|X|}}, N, \alpha, \textbf{var } Wr)$
return (C, γ)

$\mathcal{B}^{\mathbf{Mux}}$:

$(*, Wr, *, *) \twoheadleftarrow Init()$
$b' \twoheadleftarrow \mathcal{A}^{\mathbf{Send}'}$; **return** b'

$\mathcal{B}$ on input $req \in \{0,1\}^*$: // Handle SD request.
$resp \twoheadleftarrow \mathcal{A}(req)$; **return** $resp$

Send$'(M_0, sc_0, M_1, sc_1)$
$(x, \alpha) \twoheadleftarrow \mathbf{Mux}(M_0, sc_0, M_1, sc_1)$
if $x = \bot$ **and** $\alpha = \bot$ **then return** $(\bot, \bot)$
if $x = 0$ **then** $N \leftarrow \diamond$
else $N \leftarrow \mathcal{N}.Next(\textbf{var } Mu.ng)$
$(C, \gamma) \twoheadleftarrow Write^{\mathcal{A}}(1^x, N, \alpha, \textbf{var } Wr)$
return (C, γ)

$\mathbf{G}_1(\mathcal{A})$ $\boxed{\mathbf{G}_2(\mathcal{A})}$

declare str $S, S^*, Y^*, T[\,]$, **bool** $sync[\,], win[\,]$
$(Mu, Wr, Re, De) \twoheadleftarrow Init()$
$sync_1, sync_2 \leftarrow 1$
$\mathcal{A}^{\mathbf{Send}, \mathbf{Recv}}$
return win_1

Send(M, sc)
$(X, N, \alpha) \twoheadleftarrow Mux^{\mathcal{A}}(M, sc, \textbf{var } Mu)$
$(C, \gamma) \twoheadleftarrow Write^{\mathcal{A}}(X, N, \alpha, \textbf{var } Wr)$
$S \leftarrow S \,\|\, C$; $S^* \leftarrow S^* \,\|\, C$
return (C, γ)

Recv(C)
$(Y, N, \alpha) \twoheadleftarrow Read^{\mathcal{A}}(C, \textbf{var } Re)$
if $sync_1$ **and** $Y \preceq S$ **then**
  $Y^* \leftarrow Y^* \,\|\, Y$; $S \leftarrow S \,\%\, Y$
else $sync_1 \leftarrow 0$
$(M, sc, \gamma) \twoheadleftarrow Demux^{\mathcal{A}}(Y, N, \alpha, \textbf{var } De)$
if $\neg sync_1$ **then** $win_1 \leftarrow win_1 \vee (M \neq \bot \wedge sc \neq \bot)$
return (M, sc, γ)

$Write^{\mathcal{A}}(X, N, \alpha, \textbf{var } Wr)$
declare str A, γ
$\langle A, \gamma \rangle \twoheadleftarrow \mathcal{A}(\langle \text{write}, \text{create ad}, |X|, \alpha \rangle)$
if $X = \diamond$ **then return** $(\diamond, \gamma)$
$Y' \leftarrow \mathcal{AE}.Enc(Wr.K, N, A, X)$
$T[N, A, Y'] \leftarrow X$
if $Y' = \bot$ **then**
  $\gamma \twoheadleftarrow \mathcal{O}(\langle \text{write}, \text{invalid ptxt} \rangle)$
  return $(\diamond, \gamma)$
return $(A \,\|\, Y', \gamma)$

$Demux^{\mathcal{A}}(Y, N, \alpha, \textbf{var } De)$
declare str X, γ, **int** a
$\langle a, \gamma \rangle \twoheadleftarrow \mathcal{A}(\langle \text{demux}, \text{ad len}, Y, \alpha \rangle)$
if $(Y = \diamond$ **and** $\gamma \neq \diamond)$ **or** $\neg sync_2$ **then**
  return $(\bot, \bot, \gamma)$
else if $Y \neq \diamond$ **then**
  $A \leftarrow Y[:a]$; $Y' \leftarrow Y \,\%\, A$
  if $T[N, A, Y'] \neq \diamond$ **then** $X \leftarrow T[N, A, Y']$
  else
    $X \leftarrow \mathcal{AE}.Dec(De.K, N, A, Y')$
    if $X \neq \bot$ **then** $win_2 \leftarrow 1$ $\boxed{; X \leftarrow \bot}$
  if $X = \bot$ **then**
    $sync_2 \leftarrow 0$; $\gamma \twoheadleftarrow \mathcal{A}(\langle \text{demux}, \text{invalid ctxt} \rangle)$
    return $(\bot, \bot, \gamma)$
$(M, sc, \gamma) \twoheadleftarrow \mathcal{M}.Demux^{\mathcal{A}}(X, \textbf{var } De.dx)$
return (M, sc, γ)

Fig. 9: top: games $\mathbf{G}_4$, $\mathbf{G}_5$, and $\mathbf{G}_6$ and adversary $\mathcal{B}$ for proof of Theorem 2. Note that procedure $Write()$ in the definition of $\mathcal{B}$ is as defined in games $\mathbf{G}_3$–$\mathbf{G}_6$. Bottom: games for proof of Theorem 3.

Refer to games $\mathbf{G}_1$ and $\mathbf{G}_2$ defined in Figure 9. The first is a modified version of the INT-CS game with adversary $\mathcal{A}$ and the partially-specified channel $\mathcal{CH}$. The changes preserve the semantics of the game and are only meant to clarify our argument. First, we have renamed some variables: in particular, *sync* has been renamed to $sync_1$, *win* to win_1, and *De.sync* to $sync_2$. Second, we rearranged the logic used to check if the channel is in sync so that $sync_1$ gets set before invoking procedure *Demux* (9:13). Third, the game declares an array $\mathbf{str}\, T[\,]$ used by procedure *Write* to associate a given plaintext X to its ciphertext Y', as well as the nonce N and associated data A used to encrypt it (9:21). Before invoking $\mathcal{AE}.Dec$, the *Demux* procedure checks if the inputs are stored in T (9:31). Fourth, we added a flag win_2 (9:34). Finally, we declare two strings: S^*, used to track the whole sender-ciphertext stream (and not just the undelivered part); and Y^*, used to track the ciphertext stream consumed by the **Recv** oracle, i.e. $Y^* = cat(\boldsymbol{Y})$ where $\boldsymbol{Y}_i$ is the fragment output by *Read* in the i-th **Recv** query. One can easily check that the random variables $\mathbf{G}_1(\mathcal{A}) = 1$ and $\mathbf{Exp}^{\text{int-cs}}_{\mathcal{CH}}(\mathcal{A}) = 1$ are identically distributed.

Game $\mathbf{G}_2$ is identical to $\mathbf{G}_1$ until the flag win_2 gets set, at which point the revised game sets X to $\perp$. This ensures that the next branch (9:35) is taken if $T[N, A, Y'] \neq \diamond$ (9:32). There exists an INT adversary $\mathcal{B}$ such that

$$\Pr\left[\,\mathbf{G}_1(\mathcal{A}) = 1\,\right] \leq \Pr\left[\,\mathbf{G}_2(\mathcal{A}) = 1\,\right] + \mathbf{Adv}^{\text{int}}_{\mathcal{AE}}(\mathcal{B})\,. \tag{11}$$

Adversary $\mathcal{B}^{\mathbf{Enc},\mathbf{Dec}}$ simulates $\mathcal{A}$ in game $\mathbf{G}_1$. Its definition is precisely the pseudocode in Figure 9, except line (9:20) is replaced by "$Y' \leftarrow \mathbf{Enc}(N, A, X)$", and lines (9:33–34) are replaced with "$X \leftarrow \mathbf{Dec}(N, A, Y')$". (Note that $\mathcal{B}$ runs in time $t + O(q_1 + q_2)$, makes as many queries to its oracles as $\mathcal{A}$ does, and the queries have the same bit length.) By definition of the INT game, we have that $\Pr\left[\,\mathbf{Exp}^{\text{int}}_{\mathcal{AE}}(\mathcal{B})\,\mathbf{sets}\ win\,\right] = \Pr\left[\,\mathbf{G}_1(\mathcal{A})\,\mathbf{sets}\ win_2\,\right]$. Applying the Fundamental Lemma of Game Playing [9] yields equation (11).

Consider the probability that $\mathbf{G}_2(\mathcal{A})$ sets $win_1 \leftarrow 1$. We begin with a few definitions. Let $\mathsf{Win}(f)$ denote the event that when $\mathcal{A}^{\mathbf{Send},\mathbf{Recv}}$ halts, the variable win_1 has the value $f \in \{0, 1\}$. (Note that $\mathsf{Win}(1)$ and $\mathbf{G}_2(\mathcal{A}) = 1$ are the same event.) We write $\mathsf{Unsync}_1(i)$ to denote the event that $sync_1 \leftarrow 0$ is set during $\mathcal{A}$'s i-th query to **Recv**, and this is the first such query. (Note that if $\mathsf{Unsync}_1(i)$ holds, then $sync_1 = 0$ for every subsequent query.) We define $\mathsf{Unsync}_2(i)$ in kind.

A couple of observations. First, if $\mathsf{Win}(1)$ holds, then $\mathsf{Unsync}_1(i)$ holds for some $1 \leq i \leq r$. (Flag win_1 can only be set on line 9:15; reaching this point implies that $sync_1 = 0$.) Second, if $\mathsf{Unsync}_2(j)$ holds for some $1 \leq j \leq r$, then for every $j \leq q \leq r$, the output of the q-th **Recv** query is (M, sc, γ), where $M = \perp$ and $sc = \perp$. (This is made clear by lines 9:27 and 9:36.) Hence, no query following (and including) the q-th sets $win_1 \leftarrow 1$.

We now show that if $\mathsf{Unsync}_1(i)$ holds, then so does $\mathsf{Unsync}_2(i)$. Suppose that the i-th query to **Recv** is the first to set $sync_1 \leftarrow 0$, and let C denote the input to the oracle. The i-th query setting $sync_1 \leftarrow 0$ implies that $Read^{\mathcal{A}}(C, \mathbf{var}\ Re)$ output a triple of strings (Y, N, α) such that $Y \not\preceq S$ (9:10–11). We examine the possible values of $sync_2$ after the next execution of *Demux* (9:14). If $sync_2 = 0$ prior to execution of line 9:14, then we are done; so suppose that $sync_2 = 1$. $Y \not\preceq S$ implies that $Y \neq \varepsilon$, so the branch at line 9:30 is taken. If $T[N, Y[:a], Y[a+1:]] = \diamond$ holds for every $a \in \mathbb{N}$, then the branch on line 9:33 will get taken and $sync_2$ will get set to 0. Suppose to the contrary that $T[N, Y[:a], Y[a+1:]] \neq \diamond$ for some $a \in \mathbb{N}$. By definition of *Write*, this means that Y is a substring of S^*, i.e. there exists some string P such that $P \,\|\, Y \preceq S^*$. Because the i-th is the first query to set $sync_1 \leftarrow 0$, it must be the case that $P = Y^*$. But $S = S^* \,\%\, Y^*$ (9:12), so $Y \preceq S$, a contradiction. Therefore, $\mathsf{Unsync}_1(i)$ implies $\mathsf{Unsync}_2(i)$.

Suppose that $\mathsf{Win}(1)$ holds. This implies $\mathsf{Unsync}_1(i)$ holds for some $1 \leq i \leq r$, which in turn implies $\mathsf{Unsync}_2(i)$ holds (as we just saw). But this means that no query following (and including) the i-th sets $win_1 \leftarrow 1$, so $\mathsf{Win}(1)$ cannot hold. We conclude that $\Pr\left[\,\mathbf{G}_2(\mathcal{A}) = 1\,\right] = 0$. ■

In Appendix A, we show how to achieve plaintext-stream integrity (INT-PS) for this scheme. Loosely, what we show is that if we restrict the adversary such that its SD-query responses ensure correct operation of the channel, then security in the INT-CS sense implies INT-PS since; thus, in this setting, security for $\mathcal{CH}$ follows from the INT security of $\mathcal{AE}$ via Theorem 3. Next, a similar argument allows us to reduce the SIM-STAT security of $\mathcal{CH}$ to the SIM-mSTAT security of $\mathcal{M}$:

Theorem 4. *For every $t, s, q_1, q_2, \mu_1, \mu_2 \in \mathbb{N}$ and every s-time simulator $\mathcal{T}$, there exists an $(t + O(s + \mu_2))$-time simulator $\mathcal{S}$ such that that $\mathbf{Adv}^{\text{sim-stat}}_{\mathcal{CH},\mathcal{S}}(t, \boldsymbol{r}) \leq \mathbf{Adv}^{\text{sim-mstat}}_{\mathcal{M},\mathcal{T}}(\tilde{t}, q_2, \mu_2) + \mathbf{Adv}^{\text{int}}_{\mathcal{AE}}(\tilde{t}, \boldsymbol{r})$, where $\tilde{t} = t + O(q_1 + q_2)$, and $\boldsymbol{r} = (q_1, q_2, \mu_1, \mu_2)$.*

Initialization of $\mathcal{S}$:

1 **declare str** buf, **bool** $sync[\,]$
2 $sync_1 \leftarrow 1$

$\mathcal{S}^{\mathcal{A}}$ **on input** (C, S):

3 $(Y, N, \alpha) \twoheadleftarrow Read^{\mathcal{A}}(C)$
4 **if** $sync_1$ **and** $Y \preceq S$ **then** $S \leftarrow S \,\%\, Y$
5 **else** $sync_1 \leftarrow 0$
6 $(M, sc, \gamma) \twoheadleftarrow Demux^{\mathcal{A}}(Y, N, \alpha)$
7 **return** γ

$Read^{\mathcal{A}}(C)$

8 **declare str** α, **int** c, **bool** $drop$
9 $buf \leftarrow buf \,\|\, C$
10 $\langle c, drop, \alpha\rangle \twoheadleftarrow \mathcal{A}(\langle \mathsf{read}, \mathsf{len}, buf\rangle)$
11 $Y \leftarrow buf[:c]$; $buf \leftarrow buf \,\%\, Y$
12 **if** $Y = \diamond$ **or** $drop$ **then return** $(\diamond, \bot, \alpha)$
13 **return** $(Y, \bot, \alpha)$

$Demux^{\mathcal{A}}(Y, N, \alpha)$

14 **declare str** γ, **int** x, a
15 $\langle a, \gamma\rangle \twoheadleftarrow \mathcal{A}(\langle \mathsf{demux}, \mathsf{ad\ len}, Y, \alpha\rangle)$
16 **if** $(Y = \diamond$ **and** $\gamma \neq \diamond)$ **or** $\neg sync_1$ **then**
17 **return** $(\bot, \bot, \gamma)$
18 **else if** $Y \neq \diamond$ **then**
19 $x \leftarrow \lambda^{-1}(|Y| - a)$
20 **if** $x \leq 0$ **or** $\neg sync_1$ **then**
21 $sync_1 \leftarrow 0$
22 $\gamma \twoheadleftarrow \mathcal{A}(\langle \mathsf{demux}, \mathsf{invalid\ ctxt}\rangle)$
23 **return** $(\bot, \bot, \gamma)$
24 $M, sc \leftarrow \bot$; $\gamma \twoheadleftarrow \mathcal{T}^{\mathcal{A}}(x)$
25 **return** (M, sc, γ)

Fig. 10: SIM-STAT simulator $\mathcal{S}$ for proof of Theorem 4.

The proof begins with the same argument used in Theorem 3, which lets us transition into a setting in which **Recv** queries are evaluated without invoking $\mathcal{AE}.Dec$. This allows us to construct a SIM-mSTAT adversary $\mathcal{B}$ and a SIM-STAT simulator $\mathcal{S}$, such that for every SIM-mSTAT simulator $\mathcal{T}$, we simulate SIM-STAT adversary $\mathcal{A}$ in its game with $\mathcal{S}$.

Proof (Theorem 4). Fix $t, s, q_1, q_2, \mu_1, \mu_2 \in \mathbb{N}$. Let $\mathcal{A}$ be a t-time SIM-STAT adversary with query resources $\boldsymbol{r} = (q_1, q_2, \mu_1, \mu_2)$ and let $\mathcal{T}$ be an s-time SIM-mSTAT simulator. To prove the claim, we exhibit a SIM-mSTAT adversary $\mathcal{B}$, SIM-STAT simulator $\mathcal{S}$, and INT adversary $\mathcal{C}$ such that

$$\mathbf{Adv}^{\text{sim-stat}}_{\mathcal{CH},\mathcal{S}}(\mathcal{A}) \leq \mathbf{Adv}^{\text{sim-mstat}}_{\mathcal{M},\mathcal{T}}(\mathcal{B}) + \mathbf{Adv}^{\text{int}}_{\mathcal{AE}}(\mathcal{C})\,,$$

where both $\mathcal{B}$ and $\mathcal{C}$ have runtime $t + O(q_1 + q_2)$, $\mathcal{B}$ uses query resources (q_2, μ_2), and $\mathcal{C}$ uses query resources $\boldsymbol{r}$.

Let λ denote the ciphertext-length function associated with $\mathcal{AE}$. (Recall that λ is a bijection by definition.) let $\mathcal{S}$ be the simulator in Figure 10. Just as in the proof of Theorem 3, we begin with a game $\mathbf{G}^b_1$ (Figure 11) instrumented to clarify the reduction. By the same argument yielding equation (11), there exists an INT adversary $\mathcal{C}$ such that

$$\Pr\big[\,\mathbf{Exp}^{\text{sim-stat}}_{\mathcal{CH},\mathcal{S},1}(\mathcal{A}) = 1\,\big] \leq \Pr\big[\,\mathbf{G}^1_1(\mathcal{A},\mathcal{S}) = 1\,\big] + \mathbf{Adv}^{\text{int}}_{\mathcal{AE}}(\mathcal{C})\,. \tag{12}$$

Moreover, adversary $\mathcal{C}$ runs in time $t + O(q_1 + q_2)$ and makes as many queries to its oracles as $\mathcal{A}$ does.

Now consider game $\mathbf{G}^b_2$ (Figure 11). This changes the condition on line (11:31) so that X gets set on the next line if $T[N, A, Y']$ is defined *and* $x = \lambda^{-1}(|Y| - a) > 0$. But the former condition implies the latter (by definition of λ and *Write*), so this change has no affect on the outcome of the game. Next, game $\mathbf{G}^b_3$ replaces the invocation of $\mathcal{M}.Demux$ (11:51) on input of X with execution of the simulator $\mathcal{T}$ on input of $x = |X|$. (The simplification on lines 11:44–46 are the result of no longer needing the variable X and do not impact the outcome.) There exists an adversary $\mathcal{B}$ such that

$$\Pr\big[\,\mathbf{Exp}^{\text{sim-mstat}}_{\mathcal{M},\mathcal{T},d}(\mathcal{B}) = 1\,\big] = \Pr\big[\,\mathbf{G}^1_{3-d}(\mathcal{A},\mathcal{S}) = 1\,\big] \tag{13}$$

for every $d \in \{0, 1\}$. Adversary $\mathcal{B}^{\mathbf{Demux}}$ simulates adversary $\mathcal{A}$ in game $\mathbf{G}^1_2$. It is defined by the pseudocode used to define the game, except line 11:50 is replaced with "$M, sc \leftarrow \bot$; $\gamma \twoheadleftarrow \mathbf{Demux}(X)$". SD requests are forwarded to $\mathcal{A}$. When $\mathcal{A}$ halts and outputs b', adversary $\mathcal{B}$ halts and outputs b'. Then $\mathcal{B}$ runs in time $t + O(q_1 + q_2)$ and uses query resources (q_2, μ_2). Now consider the revisions in game $\mathbf{G}^b_4$. The first change is to replace $sync_2$ with $sync_1$ on line 11:55 and 11:60. The second is to change the condition on line 11:59 so that the branch is taken if $x < 0$ *or* $\neg sync_1$ (rather than $T[N, A, Y']$ being undefined). Recall that in

the proof of Theorem 4,we showed that when a **Recv**-query sets $sync_1 \leftarrow 0$ for the first time, the same query also sets $sync_2 \leftarrow 0$. This implies that $\Pr\left[\,\mathbf{G}_3^1(\mathcal{A},\mathcal{S})=1\,\right] \le \Pr\left[\,\mathbf{G}_4^1(\mathcal{A},\mathcal{S})=1\,\right]$. Summarizing, and observing that the events $\mathbf{Exp}^{\text{sim-stat}}_{\mathcal{CH},\mathcal{S},0}(\mathcal{A})=1$ and $\mathbf{G}_4^0(\mathcal{A},\mathcal{S})=1$ are identically distributed,

$$\begin{aligned}\Pr\left[\,\mathbf{Exp}^{\text{sim-stat}}_{\mathcal{CH},\mathcal{S},1}(\mathcal{A})=1\,\right] &\le \Pr\left[\,\mathbf{G}_4^1(\mathcal{A},\mathcal{S})=1\,\right] \\ &\quad + \mathbf{Adv}^{\text{sim-mstat}}_{\mathcal{M},\mathcal{T}}(\mathcal{B}) + \mathbf{Adv}^{\text{int}}_{\mathcal{AE}}(\mathcal{C}) \\ \mathbf{Adv}^{\text{sim-stat}}_{\mathcal{CH},\mathcal{S}}(\mathcal{A}) &\le \Pr\left[\,\mathbf{G}_4^1(\mathcal{A},\mathcal{S})=1\,\right] \\ &\quad - \Pr\left[\,\mathbf{Exp}^{\text{sim-stat}}_{\mathcal{CH},\mathcal{S},0}(\mathcal{A})=1\,\right] \\ &\quad + \mathbf{Adv}^{\text{sim-mstat}}_{\mathcal{M},\mathcal{T}}(\mathcal{B}) + \mathbf{Adv}^{\text{int}}_{\mathcal{AE}}(\mathcal{C})\end{aligned} \tag{14}$$

and so

$$\begin{aligned}\mathbf{Adv}^{\text{sim-stat}}_{\mathcal{CH},\mathcal{S}}(\mathcal{A}) &\le \Pr_b\left[\,\mathbf{G}_4^b(\mathcal{A},\mathcal{S})=b\,\right] - 1/2 \\ &\quad + \mathbf{Adv}^{\text{sim-mstat}}_{\mathcal{M},\mathcal{T}}(\mathcal{B}) + \mathbf{Adv}^{\text{int}}_{\mathcal{AE}}(\mathcal{C})\,,\end{aligned} \tag{15}$$

where equation 15 follows from conditioning on the value of b. But the definition of $\mathcal{S}$, executed in case $b=0$, is equivalent to the game code executed when $b=1$ in $\mathbf{G}_4^b$. Hence, adversary $\mathcal{A}$'s view in the game is independent of the challenge bit, so the first term in the last line is $1/2$.

Finally, note that the runtime of the simulator $\mathcal{S}$ is linear in the total bit length of $\mathcal{A}$'s **Recv** queries and incurs a cost of $O(s)$ (the runtime of $\mathcal{T}$) for each query. Hence, each execution of $\mathcal{S}$ runs in $t + O(s + \mu_2)$ time. (The t factor comes from using $\mathcal{A}$ to handle SD requests.) ■

Finally, putting together Theorems 1, 2, 3, and 4 yields our result for the PRIV-SR security of $\mathcal{CH}$:

Corollary 1. *For every $t, s, q_1, q_2, \mu_1, \mu_2 \in \mathbb{N}$ and s-time simulator $\mathcal{S}$, it holds that*

$$\begin{aligned}\mathbf{Adv}^{\text{priv-sr}}_{\mathcal{CH},\ell}(t,\boldsymbol{r}) \le \mathbf{Adv}^{\text{mpriv-s}}_{\mathcal{M},\ell}(\hat{t},q_1,\mu_1) + 2\mathbf{Adv}^{\text{sim-mstat}}_{\mathcal{M},\mathcal{S}}(\tilde{t},q_2,\mu_2) + \\ 4\mathbf{Adv}^{\text{int}}_{\mathcal{AE}}(\tilde{t},\boldsymbol{r}) + 2\mathbf{Adv}^{\text{priv}}_{\mathcal{AE}}(\hat{t},q_1,\mu_1) + 2\,\mathsf{coll}_{\mathcal{N}}(q_1)\,,\end{aligned}$$

where $\tilde{t} = t + O(q_1+q_2)$, $\hat{t} = O(q_1 + q_2(t+s+\mu_2))$, $\boldsymbol{r} = (q_1,q_2,\mu_1,\mu_2)$, and $\ell \in \{\mathsf{lensc},\mathsf{len},\mathsf{none}\}$.

3.4 Discussion

The preceding analysis offers good news about TLS 1.3. We regarded the record layer as a multiplexed, stream-based channel, a setting which accurately models secure channels as they are used in practice. We formalized it as a partially specified channel, allowing us to encapsulate in one scheme (see Figure 7) the myriad implementations that its standardizing document admits. We confirm its privacy and integrity in our strong adversarial model, but with two important caveats: first, whether the record layer hides the length, content, or type of input streams depends crucially on details left unspecified by the standard. Nevertheless, our results—specifically, Theorems 2 and 4—provide guidance on how to develop implementations that achieve a target security goal. Concretely, the goal is a propety of the stream multiplexer used to construct the channel. The second caveat is that draft 23 of the record layer does not achieve security in the sense of ciphertext-stream integrity; we suggested a simple change to the standard so that it provably does (Theorem 3), which was adopted in the latest draft.

Our partial specification of the record layer is simple and flexible; our hope is that this paradigm will help shape the standard-writing process. Thinking formally about what the protocol *must* get right and what it *may* get wrong provides principled guidance in its development. This paper leaves open a number of directions for future work. Our notions of security apply to settings in which an out-of-order packet is regarded as an attack (e.g., TLS and SSH); our framework can be applied to other notions of security appropriate for settings in which packet loss is expected (e.g., DTLS and IPSec). Beyond channels, we hope to see the Rogaway-Stegers framework applied more broadly. e.g., to the TLS handshake.

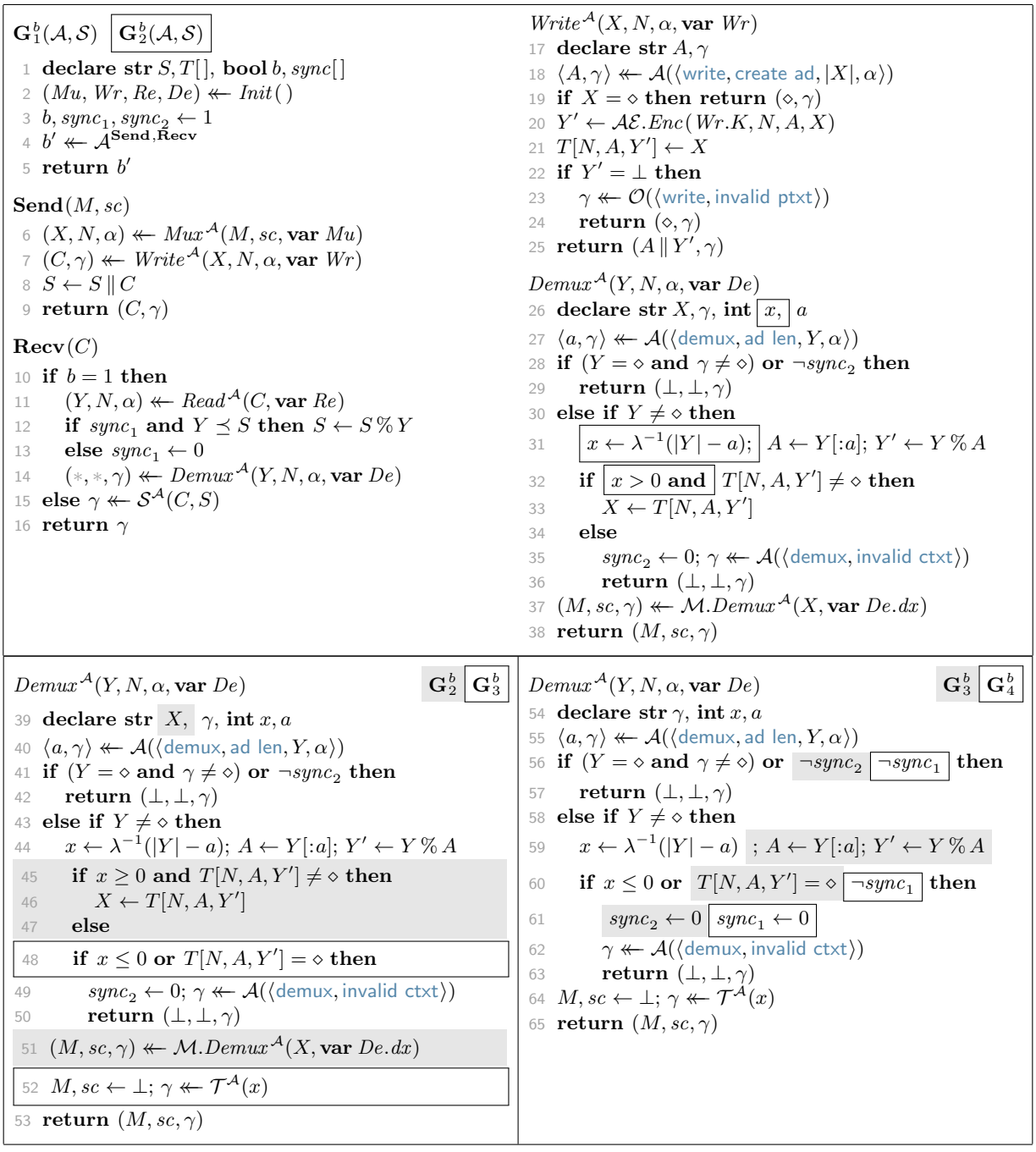

Fig. 11: Games for proof of Theorem 5.

References

1. Albrecht, M.R., Degabriele, J.P., Hansen, T.B., Paterson, K.G.: A surfeit of SSH cipher suites. In: Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security, ACM (2016) 1480–1491
2. Albrecht, M.R., Paterson, K.G., Watson, G.J.: Plaintext recovery attacks against SSH. In: Proceedings of the 30th IEEE Symposium on Security and Privacy, IEEE (2009) 16–26
3. AlFardan, N.J., Paterson, K.G.: Lucky thirteen: Breaking the TLS and DTLS record protocols. In: 2013 IEEE Symposium on Security and Privacy, IEEE (2013) 526–540
4. Andreeva, E., Bogdanov, A., Luykx, A., Mennink, B., Mouha, N., Yasuda, K.: How to securely release unverified plaintext in authenticated encryption. In: Advances in Cryptology – ASIACRYPT 2014, Springer Berlin Heidelberg (2014) 105–125
5. Badertscher, C., Matt, C., Maurer, U., Rogaway, P., Tackmann, B.: Augmented secure channels and the goal of the TLS 1.3 record layer. In: Provable Security, Springer International Publishing (2015) 85–104
6. Barwell, G., Page, D., Stam, M.: Rogue decryption failures: Reconciling ae robustness notions. In: Proceedings of the 15th IMA International Conference on Cryptography and Coding, Springer International Publishing (2015) 94–111
7. Bellare, M., Kohno, T., Namprempre, C.: Breaking and provably repairing the SSH authenticated encryption scheme: A case study of the Encode-then-Encrypt-and-MAC paradigm. ACM Trans. Inf. Syst. Secur. **7**(2) (2004) 206–241
8. Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. Cryptology ePrint Archive, Report 2000/025 (2000) `https://eprint.iacr.org/2000/025`.
9. Bellare, M., Rogaway, P.: The security of triple encryption and a framework for code-based game-playing proofs. In: Proceedings of the 24th Annual International Conference on The Theory and Applications of Cryptographic Techniques, Springer-Verlag (2006) 409–426
10. Bellare, M., Tackmann, B.: The multi-user security of authenticated encryption: AES-GCM in TLS 1.3. In: Advances in Cryptology – CRYPTO 2016, Springer Berlin Heidelberg (2016) 247–276
11. Bhargavan, K., Lavaud, A.D., Fournet, C., Pironti, A., Strub, P.Y.: Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In: Proceedings of the 35th IEEE Symposium on Security and Privacy, IEEE (2014) 98–113
12. Boldyreva, A., Degabriele, J.P., Paterson, K.G., Stam, M.: Security of symmetric encryption in the presence of ciphertext fragmentation. In: Advances in Cryptology – EUROCRYPT 2012, Springer Berlin Heidelberg (2012) 682–699
13. Boldyreva, A., Degabriele, J.P., Paterson, K.G., Stam, M.: On symmetric encryption with distinguishable decryption failures. In: Fast Software Encryption, Springer Berlin Heidelberg (2014) 367–390
14. Degabriele, J.P., Paterson, K., Watson, G.: Provable security in the real world. IEEE Security & Privacy **9**(3) (2011) 33–41
15. Degabriele, J.P., Paterson, K.G.: On the (in)Security of IPsec in MAC-then-encrypt Configurations. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, ACM (2010) 493–504
16. Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Protzenko, J., Rastogi, A., Swamy, N., Zanella-Beguelin, S., Bhargavan, K., Pan, J., Zinzindohoue, J.K.: Implementing and proving the TLS 1.3 record layer. In: Proceedings of the 38th IEEE Symposium on Security and Privacy (SP), IEEE (2017) 463–482
17. Fischlin, M., Günther, F., Marson, G.A., Paterson, K.G.: Data is a stream: Security of stream-based channels. In: Advances in Cryptology – CRYPTO 2015, Springer Berlin Heidelberg (2015) 545–564
18. Fischlin, M., Günther, F., Marson, G.A., Paterson, K.G.: Data is a stream: Security of stream-based channels. Cryptology ePrint Archive, Report 2017/1191 (2017) `https://eprint.iacr.org/2017/1191`.
19. Hoang, V.T., Krovetz, T., Rogaway, P.: Robust authenticated-encryption AEZ and the problem that it solves. In: Advances in Cryptology – EUROCRYPT 2015, Springer Berlin Heidelberg (2015) 15–44
20. Kent, S., Seo, K.: Security architecture for the internet protocol. RFC 4301, RFC Editor (December 2005) `http://www.rfc-editor.org/rfc/rfc4301.txt`.
21. Krawczyk, H.: Cryptographic extraction and key derivation: The HKDF scheme. In: Advances in Cryptology – CRYPTO 2010, Springer Berlin Heidelberg (2010) 631–648
22. McGrew, D.: An interface and algorithms for authenticated encryption. RFC 5116, RFC Editor (January 2008) `http://www.rfc-editor.org/rfc/rfc5116.txt`.
23. Namprempre, C., Rogaway, P., Shrimpton, T.: Reconsidering generic composition. In: Advances in Cryptology – EUROCRYPT 2014, Springer Berlin Heidelberg (2014) 257–274
24. Paterson, K.G., AlFardan, N.J.: Plaintext-recovery attacks against datagram TLS. In: 19th Annual Network and Distributed System Security Symposium, NDSS. (2012)
25. Paterson, K.G., Ristenpart, T., Shrimpton, T.: Tag size *does* matter: Attacks and proofs for the tls record protocol. In: Advances in Cryptology – ASIACRYPT 2011, Springer Berlin Heidelberg (2011) 372–389

26. Paterson, K.G., Watson, G.J.: Plaintext-dependent decryption: A formal security treatment of SSH-CTR. Cryptology ePrint Archive, Report 2010/095 (2010) https://eprint.iacr.org/2010/095.
27. Rescorla, E., Tschofenig, H., Modadugu, N.: The Datagram Transport Layer Security (DTLS) Protocol Version 1.3. Internet-Draft draft-ietf-tls-dtls13-22, IETF Secretariat (2017) https://tools.ietf.org/html/draft-ietf-tls-dtls13-22.
28. Rescorla, E.: The Transport Layer Security (TLS) Protocol version 1.3. Internet-Draft draft-ietf-tls-tls13-23, IETF Secretariat (2018) https://tools.ietf.org/html/draft-ietf-tls-tls13-23.
29. Rogaway, P., Stegers, T.: Authentication without elision. In: 2009 22nd IEEE Computer Security Foundations Symposium, IEEE (2009) 26–39
30. Rogaway, P.: Authenticated-encryption with associated-data. In: Proceedings of the 9th ACM Conference on Computer and Communications Security, ACM (2002) 98–107
31. Smyth, B., Pironti, A.: Truncating TLS connections to violate beliefs in web applications. In: Presented as part of the 7th USENIX Workshop on Offensive Technologies, USENIX (2013)
32. Vaudenay, S.: Security flaws induced by CBC padding — Applications to SSL, IPSEC, WTLS... In: Advances in Cryptology — EUROCRYPT 2002, Springer Berlin Heidelberg (2002) 534–545
33. Ylonen, T., Lonvick, C.: The secure shell (SSH) protocol architecture. RFC 4251, RFC Editor (January 2006) http://www.rfc-editor.org/rfc/rfc4251.txt.

A Fully specified channels

Conspicuously absent from our treatment in Section 2 is a correctness condition for PSCs. Indeed, it is undesirable to require correctness in our setting; we want our results to hold up even when the SD is realized by the adversary. But this choice is not without consequences, since we cannot *assume* correctness in proofs of security, as is so often done in cryptography [8,18]. In particular, contrary to prior settings, it is not the case that ciphertext-stream integrity implies plaintext-stream integrity for PSCs. We show this with a counter example, then show how to restrict the SD in order to recover the classic result.

INT-CS $\not\Rightarrow$ INT-PS for PSCs. Let $\mathcal{CH} = (\mathit{Init}, \mathit{Mux}, \mathit{Write}, \mathit{Read}, \mathit{Demux})$ be a PSC. We define from this a new PSC $\mathcal{CH}' = (\mathit{Init}, \mathit{Mux}, \mathit{Write}, \mathit{Read}', \mathit{Demux}')$, where Read' and Demux' are given in Figure 12. Whatever were the SD associated to PSC $\mathcal{CH}$, we add to this a new reader specification detail that on input $\langle \mathsf{read}, \mathsf{output\ 1} \rangle$ returns a bit c. Under any correct realization of $\mathcal{CH}'$ this bit must be 1. The Read' algorithm takes input runs $(Y, H, \alpha) \twoheadleftarrow \mathit{Read}(C, \mathbf{var}\ \mathit{Re})$, and outputs $(Y, H \,\|\, c, \alpha)$. Likewise, we add two new demultiplexer SD hooks: first, one that on input $\langle \mathsf{demux}, \mathsf{frag}, M \rangle$, where M is a string or $\perp$, and returns a string F; second, one that on input $\langle \mathsf{demux}, \mathsf{ctx}, \mathit{sc} \rangle$, where sc is a string or $\perp$, and returns a string φ. The Demux' algorithm, on input $(X, H', \alpha, \mathbf{var}\ \mathit{De})$ parses H' into H and the extra bit, and executes $\mathit{Demux}^{\mathcal{O}}(Y, H, \alpha, \mathbf{var}\ \mathit{De})$. If the extra bit is 1, then Demux' returns whatever Demux did. Otherwise, the output fragment M is replaced with F and the stream context sc gets replaced with φ.

It is easy to show that for every $\mathcal{A}$, there exists $\mathcal{A}'$ such that $\mathbf{Adv}^{\text{int-cs}}_{\mathcal{CH}}(\mathcal{A}) = \mathbf{Adv}^{\text{int-cs}}_{\mathcal{CH}'}(\mathcal{A}')$. Consider the following INT-PS attack: choose any string C and ask it of **Recv**. On SD request $\langle \mathsf{read}, \mathsf{output\ 1} \rangle$, the attacker responds with $\langle 0 \rangle$. On SD request $\langle \mathsf{demux}, \mathsf{frag}, M \rangle$ for some $M \in \{0,1\}^* \cup \{\perp\}$, if $M \neq \perp$, then choose any string $F \not\preceq M$ and output it; otherwise, choose any string $F \neq \varepsilon$ and output it. On SD request $\langle \mathsf{demux}, \mathsf{ctx}, \mathit{sc} \rangle$ for some $\mathit{sc} \in \{0,1\}^* \cup \{\perp\}$, if $\mathit{sc} \neq \perp$, then output $\varphi = \mathit{sc}$; otherwise, choose any string φ and output it. Then $S_\varphi = \diamond$ by definition, and the adversary ensures that $R_\varphi \neq \diamond$. Clearly $R_\varphi \not\preceq S_\varphi$, and the adversary wins with probability 1. ◊

$\mathit{Read}'^{\mathcal{O}}(C, \mathbf{var}\ \mathit{Re})$
1 **declare bool** b
2 $(Y, H, \alpha) \twoheadleftarrow \mathit{Read}^{\mathcal{O}}(C, \mathbf{var}\ \mathit{Re})$
3 $\langle c \rangle \twoheadleftarrow \mathcal{O}(\langle \mathsf{read}, \mathsf{output\ 1} \rangle)$
4 **return** $(Y, H \,\|\, c, \alpha)$

$\mathit{Demux}'^{\mathcal{O}}(X, H', \alpha, \mathbf{var}\ \mathit{De})$
5 $\ell \leftarrow |H'|$; $H \leftarrow H'[1{:}\ell - 1]$; $c \leftarrow H'_\ell$
6 $(M, \mathit{sc}, \gamma) \twoheadleftarrow \mathit{Demux}^{\mathcal{O}}(Y, H, \alpha, \mathbf{var}\ \mathit{De})$
7 **if** $c = 1$ **then return** (M, sc, γ)
8 $F \twoheadleftarrow \mathcal{O}(\langle \mathsf{demux}, \mathsf{frag}, M \rangle)$
9 $\varphi \twoheadleftarrow \mathcal{O}(\langle \mathsf{demux}, \mathsf{ctx}, \mathit{sc} \rangle)$
10 **return** (F, φ, γ)

Fig. 12: Procedures Read' and Demux'.

$\mathit{Gets}(\boldsymbol{M}, \boldsymbol{s}, sc)$
1 $M' \leftarrow \varepsilon$
2 **for** $i \leftarrow 1$ **to** $|\boldsymbol{s}|$ **do**
3 **if** $\boldsymbol{s}_i = sc$ **then** $M' \leftarrow M' \,\|\, \boldsymbol{M}_i$
4 **return** M'

$\mathit{Corr}(\boldsymbol{C}', \boldsymbol{M}, \boldsymbol{s})$
5 $(\mathit{Mu}, \mathit{Wr}, \mathit{Re}, \mathit{De}) \twoheadleftarrow \mathit{Init}$
6 $\boldsymbol{C} \twoheadleftarrow \mathit{Sends}(\boldsymbol{M}, \boldsymbol{s}, \mathit{Mu}, \mathit{Wr})$
7 $(\boldsymbol{Y}', \boldsymbol{M}', \boldsymbol{s}') \twoheadleftarrow \mathit{Recvs}(\boldsymbol{C}', \mathit{Re}, \mathit{De})$
8 **return** $(\boldsymbol{C}, \boldsymbol{Y}', \boldsymbol{M}', \boldsymbol{s}')$

$\mathit{Sends}(\boldsymbol{M}, \boldsymbol{s}, \mathit{Mu}, \mathit{Wr})$ // $\boldsymbol{M}, \boldsymbol{s} \in \{0,1\}^{**}$
9 **for** $i \leftarrow 1$ **to** $|\boldsymbol{s}|$ **do**
10 $(X, H, \alpha) \twoheadleftarrow \mathit{Mux}^{\mathcal{S}}(\boldsymbol{M}_i, \boldsymbol{s}_i, \textbf{var } \mathit{Mu})$
11 $(\boldsymbol{C}_i, *) \twoheadleftarrow \mathit{Write}^{\mathcal{S}}(X, H, \alpha, \textbf{var } \mathit{Wr})$
12 **return** $\boldsymbol{C}$

$\mathit{Recvs}(\boldsymbol{C}', \mathit{Re}, \mathit{De})$
13 **for** $i \leftarrow 1$ **to** $|\boldsymbol{C}'|$ **do**
14 $(\boldsymbol{Y}'_i, H, \alpha) \twoheadleftarrow \mathit{Read}^{\mathcal{R}}(\boldsymbol{C}'_i, \textbf{var } \mathit{Re})$
15 $(\boldsymbol{M}'_i, \boldsymbol{s}'_i, *) \twoheadleftarrow \mathit{Demux}^{\mathcal{R}}(\boldsymbol{Y}'_i, H, \alpha, \textbf{var } \mathit{De})$
16 **return** $(\boldsymbol{Y}', \boldsymbol{M}', \boldsymbol{s}')$

$\mathbf{G}(\mathcal{A})$
1 **declare str** $R[\,], S[\,], T$
2 **declare bool** $\mathit{sync}, \mathit{win}[\,]$
3 $(\mathit{Mu}, \mathit{Wr}, \mathit{Re}, \mathit{De}) \twoheadleftarrow \mathit{Init}(\,)$
4 $\mathit{sync} \leftarrow 1$; $\mathcal{A}^{\mathbf{Send}, \mathbf{Recv}}$

$\mathbf{Send}(M, sc)$
5 $(X, H, \alpha) \twoheadleftarrow \mathit{Mux}^{\mathcal{A}}(M, sc, \textbf{var } \mathit{Mu})$
6 $(C, \gamma) \twoheadleftarrow \mathit{Write}^{\mathcal{A}}(X, H, \alpha, \textbf{var } \mathit{Wr})$
7 $T \leftarrow T \,\|\, C$; $S_{sc} \leftarrow S_{sc} \,\|\, M$
8 **return** (C, γ)

$\mathbf{Recv}(C)$
9 $(Y, H, \alpha) \twoheadleftarrow \mathit{Read}^{\mathcal{A}}(C, \textbf{var } \mathit{Re})$
10 $(M, sc, \gamma) \twoheadleftarrow \mathit{Demux}^{\mathcal{A}}(Y, H, \alpha, \textbf{var } \mathit{De})$
11 **if** sync **and** $Y \preceq T$ **then** $T \leftarrow T \,\%\, Y$
12 **else** $\mathit{sync} \leftarrow 0$
13 **if** $M \neq \bot \wedge sc \neq \bot$ **then**
14 $R_{sc} \leftarrow R_{sc} \,\|\, M$
15 **if** $R_{sc} \not\preceq S_{sc}$ **then** $\mathit{win}_2 \leftarrow 1$
16 **if** $\neg \mathit{sync}$ **then** $\mathit{win}_1 \leftarrow 1$
17 **return** (M, sc, γ)

Fig. 13: top: procedures for defining correctness of FSC $(\mathcal{CH}, \mathcal{S}, \mathcal{R})$. Bottom: a game for proving Theorem 5.

The attack just described exploits the fact that the adversary controls the SD. Note, too, that the adversarial handling of the SD does not result in a correct realization of $\mathcal{CH}'$. This raises the question of whether or not there is a separation when the PSC is correctly realized.

A partially specified channel is transformed into a *fully specified channel* (FSC) by instantiating the SD oracle for each algorithm. Given a PSC $\mathcal{CH}$, we define an FSC as a triple $(\mathcal{CH}, \mathcal{S}, \mathcal{R})$, where $\mathcal{S}$ and $\mathcal{R}$ are randomized and stateful algorithms that instantiate the SD oracle for the sender $(\mathit{Mux}, \mathit{Write})$ and receiver $(\mathit{Read}, \mathit{Demux})$, respectively. We may define correctness FSCs as follows:

Definition 1. Refer to procedures *Gets* and *Corr* defined in Figure 13. We say that FSC $(\mathcal{CH}, \mathcal{S}, \mathcal{R})$ is *correct* if for every $\boldsymbol{C}', \boldsymbol{M}, \boldsymbol{s} \in \{0,1\}^{**}$ such that $|\boldsymbol{M}| = |\boldsymbol{s}|$ and $sc \in \{0,1\}^{*}$, it holds that

$$\Pr\big[(\boldsymbol{C}, \boldsymbol{Y}', \boldsymbol{M}', \boldsymbol{s}') \twoheadleftarrow \mathit{Corr}(\boldsymbol{C}', \boldsymbol{M}, \boldsymbol{s}) : \\ \mathit{cat}(\boldsymbol{Y}') \preceq \mathit{cat}(\boldsymbol{C}) \Rightarrow \mathit{Gets}(\boldsymbol{M}', \boldsymbol{s}', sc) \preceq \mathit{Gets}(\boldsymbol{M}, \boldsymbol{s}, sc)\big] = 1\,.$$

We say that PSC $\mathcal{CH}$ *has a correct realization* if there exists a pair $(\mathcal{S}, \mathcal{R})$ such that the FSC $(\mathcal{CH}, \mathcal{S}, \mathcal{R})$ is correct. ◊

The definition says that each plaintext stream output by the receiver must be a prefix of the corresponding stream input to the sender, as long as the ciphertext stream *consumed* by the receiver is a prefix of the ciphertext stream *produced* by the sender. This naturally generalizes the correctness condition of FGMP for single stream-based channels [18, Definition 3.1].

INT-CS ⇒ INT-PS for correct FSCs. We show that if the SD are handled by $\mathcal{A}$ in a manner that yields a *correct* FSC, then the traditional relationship holds.

Theorem 5. *For every adversary $\mathcal{A}$, if $(\mathcal{CH}, \mathcal{A}, \mathcal{A})$ is a correct FSC, then* $\mathbf{Adv}^{\text{int-ps}}_{\mathcal{CH}}(\mathcal{A}) \leq \mathbf{Adv}^{\text{int-cs}}_{\mathcal{CH}}(\mathcal{A})$.

Proof. Consider the game $\mathbf{G}$ defined in Figure 13. It combines the game logic of INT-CS and INT-PS so that flag win_1 has the semantics as the *win* flag of the INT-CS game and win_2 has the semantics of the *win* flag in the INT-PS game. Then by definition, $\mathbf{Adv}^{\text{int-cs}}_{\mathcal{CH}}(\mathcal{A}) = \Pr\big[\,\mathbf{G}(\mathcal{A})\,\textbf{sets sets}\,\mathit{win}_1\,\big]$ and

$\mathbf{Exp}^{\text{stae}}_{\Pi,b}(\mathcal{A})$

1 **declare str** $E[\,]$, **state** $D[\,]$
2 $\pi \xleftarrow{} Gen(\,)$ // Sets $\pi.seqn = 0$.
3 $b' \xleftarrow{} \mathcal{A}^{\mathbf{Enc},\mathbf{Dec},\mathbf{GenD}}$
4 **return** b'

Enc(**str** H, M)

5 **if** $b = 1$ **then** $C \leftarrow Enc(\mathbf{var}\ \pi, H, M)$
6 **else**
7 $\quad c \leftarrow cipherlen(|M|)$; $C \xleftarrow{} \{0,1\}^c$
8 $\quad \pi.seqn \leftarrow \pi.seqn + 1$
9 $E[\pi.\, seqn - 1, H, C] \leftarrow M$
10 **return** C

GenD(**int** d)

11 **if** $D[d] \neq \perp$ **then return** $\perp$
12 $D[d] \xleftarrow{} GenD(\pi)$ // Sets $D[d].seqn = 0$.

Dec(**int** d, **str** H, C)

13 **if** $D[d] \neq \perp$ **then return** $\perp$
14 **if** $b = 1$ **then** $M \leftarrow Dec(\mathbf{var}\ D[d], H, C)$
15 **else**
16 $\quad M \leftarrow E[D[d].seqn, H, C\,]$
17 $\quad$ **if** $M \neq \perp$ **then**
18 $\quad\quad D[d].seqn \leftarrow D[d].seqn + 1$
19 **return** M

$\mathbf{Exp}^{\text{lhse}}_{\Pi,b}(\mathcal{A})$

20 **declare str** $E[\,]$, **state** $D[\,]$
21 $\pi \xleftarrow{} Gen(\,)$ // Sets $\pi.seqn = 0$.
22 $b' \xleftarrow{} \mathcal{A}^{\mathbf{Enc},\mathbf{Dec},\mathbf{GenD}}$
23 return b'

Enc(**int** ℓ, **str** F) // What if $\ell < |F|$?

24 $R_0 \leftarrow 0^\ell$; $R_1 \leftarrow finalize(R_0)$
25 **if** $b = 1$ **then** $V \leftarrow Enc(\mathbf{var}\ \pi, \ell, F)$
26 **else** $V \leftarrow Enc(\mathbf{var}\ \pi, \ell, R_{final(F)})$
27 $E[\pi.\, seqn - 1, V] \leftarrow F$
28 **return** V

GenD(**int** d)

29 **if** $D[d] \neq \perp$ **then return** $\perp$
30 $D[d] \xleftarrow{} GenD(\pi)$ // Sets $D[d].seqn = 0$.

Dec(**int** d, **str** V)

31 **if** $D[d] \neq \perp$ **then return** $\perp$
32 **if** $b = 1$ **then** $F \leftarrow Dec(\mathbf{var}\ D[d], V)$
33 **else**
34 $\quad$ **if** $closed(D[d])$ **then return** $\perp$
35 $\quad F \leftarrow E[D[d].seqn, V]$
36 $\quad$ **if** $F \neq \perp$ **then** $D[d].seqn \leftarrow D[d].seqn + 1$
37 $\quad$ **if** $F \neq \perp$ **and** $final(F)$ **then** $close(\mathbf{var}\ D[d])$
38 **return** F

Fig. 14: **Left:** StAE security of stateful AE scheme $\Pi = (Gen, GenD, Enc, Dec, cipherlen)$. Type **state** is implicitly defined as a **struct** with **int** *seqn* being one of its attributes; otherwise **state** is defined by the scheme. **Right:** Lhse security of content-hiding, stateful AE scheme $\Pi = (Gen, GenD, Enc, Dec, final, finalize, closed, close)$.

$\mathbf{Adv}^{\text{int-ps}}_{\mathcal{CH}}(\mathcal{A}) = \Pr\left[\,\mathbf{G}(\mathcal{A})\ \mathbf{sets\ sets}\ win_2\,\right]$. To prove the claim, it suffices to show that if $\mathbf{G}(\mathcal{A})$ sets win_2, then it also sets win_1. A sufficient condition is that if at any point in the game, if $R_{sc} \not\preceq S_{sc}$ for some sc, then $\neg sync$ holds. Suppose that $sync$ holds. Then satisfying Definition 1 implies that $R_{sc} \preceq S_{sc}$ for every sc. ■

The results above imply the following:

Corollary 2. *For every adversary* $\mathcal{A}$, $\mathbf{Adv}^{\text{int-ps}}_{\mathcal{CH}}(\mathcal{A}) \leq \mathbf{Adv}^{\text{int-cs}}_{\mathcal{CH}}(\mathcal{A})$ *if and only if* $(\mathcal{CH}, \mathcal{A}, \mathcal{A})$ *is correct.*

B The notions of Delignat-Lavaud et al.

In this appendix we discuss the security notions of DLFK+ [16] for stateful authenticated encryption. These notions were devised in order to formalize the security properties of their F* implementation of the TLS 1.3 record layer (draft 18).

Change of notation. For the purpose of presenting their notions, it will be convenient to change a convention used in the rest of this paper. In this appendix, uninitialized variables (or elements of an associative array) now implicitly have the value $\perp$ instead of $\diamond$.

The StAE notion. Figure 14 defines the DLFK+'s security notion for stateful, authenticated encryption. It is largely an extension of AEAD security to stateful schemes that use a *sequence number* to generate nonces, but it incorporates an additional feature designed to cope with a real-world problem. We elaborate below.

Multiple receivers. The *Gen* procedure generates a secret (modeling the outcome of the handshake) and outputs the state of the encrypting party. The decrypting party's state is initialized via the *GenD* procedure,

which takes as input the encrypting party's state. The security game models a setting in which there is one encryptor and any number of decryptors. This is intended to cope with the fact that "[i]n practice, it is difficult to prevent multiple honest servers from decrypting and processing the same 0-RTT encryption stream" [16, Section 6]. In order to reduce latency of the connection, TLS 1.3 allows clients to send early application data using a shared secret derived from a prior session before the new key exchange is finished. This so-called *0-RTT* data is piggy-backed on the client's first handshake flow, thus reducing the time to wait before application data can start flowing the other direction. In fact, web servers usually balance their load across multiple front-end servers. In order to support 0-RTT data, it is necessary that each of these servers share the state needed to resume the old session. As a result, it is possible for an adversary to replay 0-RTT data to more than one front-end server. This motivates DLFK+'s consideration of multiple receivers.

The possibility of multiple decrypting parties appears to be a non-trivial extension of the usual security model, and something that our setting does not capture. Follow-on work should verify that our construction remains secure when the model is augmented in this fashion. Intuitively, this gives the adversary additional power in the sense that there are now multiple ciphertext streams that may go out of sync; we conjecture this would degrade the privacy and integrity bounds by no more than a factor of the number of channels.

StAE does not support ciphertext fragmentation. Consider the following attack. First, choose some $H, M \in \{0,1\}^*$ such that $\mathit{cipherlen}(|M|) > 1$ and ask $C \leftarrow \mathbf{Enc}(H, M)$. Next, ask $M_1 \leftarrow \mathbf{Dec}(1, H, C[1])$ followed by $M_2 \leftarrow \mathbf{Dec}(1, H, C[2{:}])$. If $b = 0$, then $M_1 = M_2 = \bot$. Suppose that $b = 1$. If the scheme does support fragmentation, then the correctness condition on the scheme [12, Definition 3.2] would imply that $M_2 \neq \bot$. Therefore, for a scheme to be deemed secure in the StAE sense, it *must not* support fragmentation.

The Lhse notion. DLFK+ define a stronger notion that captures three additional goals. First and foremost, this notion incorporates *length hiding*, which obscures the length of the message fragments. Second, the *content type* is encoded by the fragment itself, and hence is kept private. Third, the syntax is extended so that the sender may signal the *end-of-stream* to the receiver; security demands that, upon receipt of this signal, the peer close the channel.

A content-hiding, stateful AE scheme is composed of eight algorithms. The first four — *Gen*, *GenD*, *Enc*, and *Dec* — are much the same as before, except that (1) encryption takes as input an $\mathbf{int}\,\ell$ that specifies the length of the padded fragment, and (2) the associated data is dropped from encryption and decryption. The remaining algorithms are used to signal closure of the channel. Algorithm $\mathit{final}(\mathbf{str}) \mapsto \mathbf{bool}$ tests if a fragment encodes the end-of-stream, i.e., is the *final* fragment, and algorithm $\mathit{finalize}(\mathbf{str}) \mapsto \mathbf{str}$ encodes its input as the final fragment. Algorithm $\mathit{closed}(\mathbf{state}) \mapsto \mathbf{bool}$ tests if the peer's state indicates that the channel has been closed, and $\mathit{close}(\mathbf{var\ state})$ closes the channel.

We highlight the important differences between Lhse and StAE security. (Refer to Figure 14.) Line 14:24 defines two strings. The first, R_0, is the all-zero string of the specified length (ℓ), and R_1 is the finalized version of that string. If $b = 1$ (the "real" world), then the **Enc** encrypts the input F; if $b = 0$ (the "simulated" world), it encrypts one of R_0 and R_1, depending on whether the F is a final fragment, i.e., if $\mathit{final}(F) = 1$ holds. The simulated decryption oracle is defined so that if it receives a ciphertext output by the encryption oracle corresponding to a final fragment, then it closes the stream (14:37). This mandates that *Dec* call *close* on the state upon receipt of a final fragment.

None of the three additional properties captured by Lhse are mandated by the TLS 1.3 specification [28]. Since draft 09, the content type has been moved from the associated data to the scope of the plaintext being encrypted. This might signal that the authors of the spec intend that the content type not be discernible from the ciphertext stream, but since the record boundaries depend on the content type, this is not true of every implementation (see the discussion in Section 3.1). (Though it is certainly true of DLFK+'s.) Next, length hiding MAY be used to mitigate traffic analysis attacks, but this too is not mandatory. Finally, the document does not mandate the end-of-stream semantics as defined in the Lhse game; certainly the application might make good use of such a functionality (as suggested by DLFK+, see [16, Section 7]), but specification is silent on the subject. Nevertheless, the end-of-stream semantics could be captured as a *permitted leakage parameter* in our PRIV-SR notion (see Section 2.2).

Record layer security. Finally, DLFK+ define a game for modeling the security provided by the overall record-layer *protocol*. In addition to the content-hiding properties of Lhse, they allow the adversary to re-

initialize the channel at will, modeling the key changes that occur during the normal execution of the protocol. For their implementation of the record layer, they are able to show that Lhse of the underlying stateful AE scheme implies record layer security, losing only a hybrid term in the reduction [16, Theorem 4]. Roughly speaking, they show that permitting q_i key changes is equivalent to executing the Lhse game q_i times. This follows easily from the observation that no state is carried over after re-initializing the channel. Thus, the ability to re-initialize the channel does not really give the adversary more power, at least with respect to the record layer.