Partially Specified Channels

The TLS 1.3 Record Layer without Elision

Christopher Patton
University of Florida
cjpatton@ufl.edu

Thomas Shrimpton
University of Florida
teshrim@ufl.edu

ABSTRACT

We advance the study of secure *stream-based* channels (Fischlin et al., CRYPTO '15) by considering the multiplexing of many data streams over a single channel, 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 provides *specification details*; the algorithms abstract the things that must be explicitly specified, while the oracle abstracts the things that need not be. Our security notions, which capture a variety of privacy and integrity goals, allow the *adversary* to respond to these oracle queries; security relative to these notions implies that the channel withstands attacks in the presence of worst-case (i.e., adversarial) realizations of the specification details. We apply this framework to a formal treatment of the TLS 1.3 record and, in doing so, show that its security hinges crucially upon details left unspecified by the standard.

CCS CONCEPTS

• **Security and privacy → Cryptography**;

KEYWORDS

provable security, cryptographic standards, TLS 1.3, stream-based channels, partially specified protocols

ACM Reference Format:
Christopher Patton and Thomas Shrimpton. 2018. Partially Specified Channels: The TLS 1.3 Record Layer without Elision. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3243734.3243789

1 INTRODUCTION

As protocols such as TLS [29], SSH [36], IPSec [22], and QUIC [20] have evolved, so have the formal tools used to analyze them. Often it is the protocol standards themselves, rather than fully realized implementations, that inspire and guide mathematical abstractions of these protocols, but their complexity makes the task of developing these abstractions quite challenging and prone to missing subtle attacks. Much of this complexity stems from the fact that protocols are only *partially* specified. The TLS 1.3 standard [30], whose record layer mechanism is the subject of this paper, contains numerous "SHOULDs", "SHOULD NOTs" and "MAYs." Each of these provides a guideline, but not a rule (those are "MUSTs" and "MUST NOTs"), for compliant realizations of the standard. In addition, and like other protocol standards, TLS 1.3 leaves many implementation details unspecified. Thus, the standard actually describes a *collection* of implementations that share a core set of behaviors.

Standards are not more explicit and prescriptive for good reason. To be broadly adopted, they need to be flexible in the face of a variety of deployment concerns, such as backwards compatibility, interoperability with other protocols, and limitations of existing infrastructure. They also need to balance performance with security and account for competing (and often conflicting) interests of stakeholders. But this need for flexibility presents an important challenge to provable security: namely, deciding which of the standard's guidelines and unspecified implementation details are relevant to security, and so should be captured in the model.

The implications of these modeling choices are often clear only after an attack is found, leading to what Degabrielle et al. [15] call the *model-attack-remodel cycle*. A prominent example is the case of padding-oracle attacks. The MAC-then-encode-then-encrypt construction, used to provide authenticated encryption in many early secure channel protocols, is provably secure [27], but only in a model in which decryption does not surface distinguishable errors. Yet compliant implementations of these protocols did make visible the cause of decryption failures (in particular, whether the encoding was invalid or the MAC was incorrect), leading to plaintext-recovery attacks [16, 26, 35]. The research community reacted by incorporating distinguishable errors into updated models [14, 18], but left more subtle attack vectors unaddressed [3], leading in turn to more sophisticated models [6, 21]. This reactive evolution of the adversarial model is to be expected. But since standards only partially specify the protocol, it is hard to anticipate where vulnerabilities might arise in implementations.

This work explores a definitional viewpoint that may help us to be more proactive, by making explicit in the security model which parts of the protocol are fully specified, and which are not. Concretely, our goal is to establish the security of the TLS 1.3 record layer [29], which (partially) specifies how plaintext and ciphertext data are formatted, encrypted, and transmitted from sender to receiver. To this end, we formalize a new primitive that we call a *partially specified channel*.

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243789

Modeling the TLS 1.3 record layer. The starting point of our model is the *stream-based channel* abstraction, introduced by Fischlin et al. [18] (hereafter FGMP). The FGMP syntax for stream-based channels accurately captures the interfaces exposed by real secure-channel implementations in that it treats the sender- and receiver-side inputs and outputs as streams of *fragments*, as opposed to atomic messages. (It also admits distinguishable error messages.) We augment their syntax in order to account for *multiplexing* of many data streams over the same channel, as this is an essential feature of many secure channel protocols, including TLS 1.3. And although this protocol is our focus, we expect our syntax should be applicable to the authenticated encryption mechanism in other protocols, such as SSH, IPSec, QUIC, and DTLS [31].

We extend the FGMP notions of privacy and integrity to this setting. There are two main flavors of privacy: the first, PRIV-S, is analogous to indistinguishibility under chosen-plaintext attack, since the adversary only controls the sender's inputs; in the second, PRIV-SR, we also allow the adversary to mount chosen-ciphertext fragment attacks. With each of these, we consider different "degrees" of privacy corresponding to various security goals considered in prior works [17, 18, 27]. For integrity, we formalize two notions: integrity of ciphertext streams (INT-CS) and plaintext streams (INT-PS). Following FGMP, we show how to achieve PRIV-SR security from a scheme that is both PRIV-S and INT-CS secure; just as with FGMP, we will need an additional property called status simulatability (SIM-STAT). Our notions are applicable to settings in which reliable transport (e.g., via TCP) is expected, and failure of the underlying transport mechanism to deliver stream fragments in order is deemed in attack (as in TLS and SSH).

A number of implementation details that are not specified by TLS 1.3 are relevant in the adversarial model of FGMP. For example, there are explicit rules that govern the manner in which plaintext fragments are buffered and coalesced into atomic plaintext records, but the specification leaves many design choices up to the implementation. In order to establish the security of the record layer in this setting, we first need to determine how to reason about these missing pieces. To do so, we apply the *partially specified protocol* approach of Rogaway and Stegers [33] (RS) to the study of secure channels. Loosely speaking, a partially specified channel (PSC) consists of named algorithms for the sender and receiver operations that each take a *specification details* (SD) oracle. The algorithms form the cryptographic core of the secure channel, and hence the part that must be realized precisely; everything that is not explicitly part of the cryptographic core is handled by the oracle. Crucially, in our security notions, it is the adversary itself who will service calls to the SD-oracle. Thus, a proof of security for a particular PSC implies that all details swept into the SD-oracle are irrelevant with respect to these definitions; they can be implemented to behave in an adversarial manner, without concern.

Our results. We found this definitional viewpoint to be a useful tool for determining which pieces of the record layer specification are security critical and which are not. In particular, our formal treatment of the record layer uncovers two subtle and security-critical matters. First, the degree of privacy the record layer can provably provide depends intrinsically on the unspecified details (Theorem 5.1). 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 content 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 leakage occurs because the *boundaries* between records depend on the content types of each record. In general, we can conclude only that the record layer ensures privacy of *the contents* of each of the data streams. (We make this point precise in Section 5.)

Second, following FGMP, our notion of ciphertext-stream integrity implies 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 5.2).

Our analysis applies to draft 23 [29], which was current at the time of writing. We shared our findings with the IETF working group responsible for standardizing TLS 1.3 and the specification was updated so that the record header is authenticated. This change appears in the final version of the standard [30].

Roadmap of the paper. The next section motivates our analytical framework, putting it in context with prior work on secure channels and partially specified protocols. Section 3 outlines additional related work on TLS. In Section 4 we formulate our syntax and adversarial model, and define our notions of privacy (Section 4.2) and integrity (Section 4.3). Section 5 presents our formal treatment of the record layer and discusses some limitations of our model with respect to TLS. We conclude in Section 6 with directions for future work. This paper is an extended abstract; all proofs of security can be found in the full version of this paper [28].

2 PSCs IN RELATION TO PRIOR WORK

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 [19] 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 the standard itself.

Stream-based secure channels. We summarize important landmarks in the development of the theory of secure channels. 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 [32] 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 (BKN) [7] formalized *stateful* AE in order to account for replay and out-of-order delivery attacks, as well as to 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. [13], 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 [19], these works help shed formal light on truncation [34] and cookie-cutter [12] attacks. (However, as we discuss in Section 5.3.2, their work is somewhat limited with regard to these.)

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 necessary for 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 4.2.)

Partially specified protocols. In their treatment of the SSH protocol, BKN introduce a paradigm they call *Encode-then-Encrypt-and-MAC*, which cleanly abstracts many of the details of the SSH specification. In particular, they treat the details of encoding as a generic transform and give a sufficient condition on this transform for the security of the overall protocol. Of course, this idea—and more generally, the *Encode-then-Encipher* paradigm [9]—is applicable to the problem of analyzing TLS 1.3. But our consideration of stream-based channels makes our adversary considerably stronger than that considered by BKN. It stands to reason, then, that there are details of the protocol and implementation that are relevant to the stronger model, but not the weaker one. (In particular, we must at least account for processing of plaintext- and ciphertext-stream fragments.) How shall we go about uncovering what these security-critical matters are?

There are many ways to appraoch this problem. The approach of RS, 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. The component algorithms of a PSC, *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 are relevant to security.

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 [17]), and which things do not need such hard guarantees.

3 RELATED WORK

The miTLS project. From the standpoint of scope, the work most closely related to ours is the recent paper by Delignat-Lavaud et al. [17] (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 [10]. 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.

Our notions are also more flexible: 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 the full version of this paper [28].

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 [29, 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 Paterson, Ristenpart, and Shrimpton [27] 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. 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. Bellare and Tackmann analyze the *multi-user* security of the TLS 1.3 record layer [11]. 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 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.

4 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 prefix Y, e.g., $1011 \,\%\, 10 = 11$. (If $Y \not\preceq X$, then define $X \,\%\, Y = \varepsilon$.) Let $\langle i \rangle_n$ denote an 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 $y \twoheadleftarrow \mathcal{A}(x)$. If $\mathcal{X}$ is a set, then we write $x \twoheadleftarrow \mathcal{X}$ to denote sampling x randomly from $\mathcal{X}$ according to some distribution; if $\mathcal{X}$ is finite and the distribution is unspecified, then it 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 [33, 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 its 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 cryptographic pseudocode; the symbol $\diamond$ is useful for specifying protocols compactly.

A value of any type may be assigned to an anonymous variable $*$, e.g., $* \leftarrow x$, but the value of $*$ is undefined in an expression. We let $\langle x_1, \ldots, x_m \rangle$ denote an invertible encoding of arbitrary values $x_1, \ldots, x_m$ as a string. Decoding is written as $\langle x_1, \ldots, x_m \rangle \leftarrow X$ and works like this (slightly deviating from [33, 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 $i \in [m]$.

Finally, it is customary in cryptographic pseudocode to pass all variables by *value*; for technical reasons, which will become apparent later on, we also permit variables to be passed by *reference*. Specifically, variables passed to procedures may be embellished with the keyword **var**. If the variable appears on the left hand side of an assignment statement, then this immediately changes the

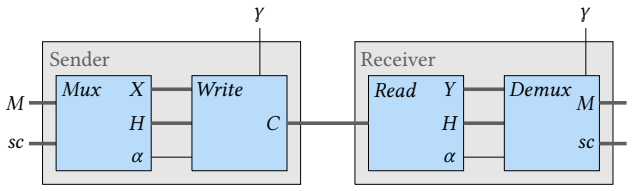


Figure 1: Illustration of our syntax.

value of the variable; when used in an expression, the variable is treated as its value. A procedure's interface makes explicit which inputs must be passed by reference. For example, in a procedure $\mathcal{A}(\textbf{int } x, \textbf{var int } y) \mapsto \textbf{int } z$, variable x is passed by value, while y is passed by reference.[1]

4.1 Syntax

Our syntax is illustrated in Figure 1. 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 expect access to an oracle, which we generically write as $\mathcal{O}$ in the following definitions:

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

The oracle $\mathcal{O}$ provides the specification details 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-oracle responses, in a bounded number of steps that depends only on the length of its inputs.

Our convention will be that SD-oracle queries are always strings of the form $\langle \text{caller}, \text{instruction}, x_1, \ldots, x_m \rangle$, 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 4 for examples.) SD-oracle responses are also always strings, but we do not define conventions for them.

[1]The keyword **var** as used by RS serves a similar purpose, but is semantically different. In their setting, a variable embellished with **var** has copy-in-copy-out semantics, which means its value is only changed when the procedure goes out of scope.

4.1.1 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 [14], an attack vector that has heavily influenced the development of secure channels [3, 16, 26, 35]. (FGMP model distinguishable errors, too.) Our consideration of information leakage via auxiliary output is inspired by a paper by Barwell, Page, and Stam [6]. Their *subtle* AE setting models decryption leakage in a manner general enough to capture error indistinguishability [14, 18], as well as other settings for authenticated encryption [4, 21].

4.1.2 Correctness. Conventionally, one would define a correctness condition as part of the syntax for this new primitive. Following RS, however, we will not explicitly define correctness of PSCs, as our aim will be to achieve security *even for channels that are not correct:* in particular, when the SD is realized by an adversary. We elaborate on the consequences of this choice in the full version of this paper [28], but note that this means we will not be able to assume correctness in our security proofs.

4.2 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.

Following prior work [7, 13, 18] 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.

4.2.1 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., by *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

$\mathbf{Exp}^{\text{priv-sr}}_{\mathcal{CH},\ell,b}(\mathcal{A})$
1 **declare str** S, Env, **bool** $sync$
2 $(Mu, Wr, Re, De) \twoheadleftarrow Init()$
3 $sync \leftarrow 1$
4 $b' \twoheadleftarrow \mathcal{A}_1^{\mathbf{Send},\mathbf{Recv}}(\mathbf{var}\ Env)$
5 **return** b'

$\mathbf{Send}(M_0, sc_0, M_1, sc_1)$
6 $L_0 \leftarrow leak(\ell, M_0, sc_0)$
7 $L_1 \leftarrow leak(\ell, M_1, sc_1)$
8 **if** $L_0 \neq L_1$ **then return** $(\perp, \perp)$
9 $(X, H, \alpha) \twoheadleftarrow Mux^{\mathbf{SD}}(M_b, sc_b, \mathbf{var}\ Mu)$
10 $(C, \gamma) \twoheadleftarrow Write^{\mathbf{SD}}(X, H, \alpha, \mathbf{var}\ Wr)$
11 $S \leftarrow S \,\|\, C$
12 **return** (C, γ)

$\mathbf{Recv}(C)$
13 $(Y, H, \alpha) \twoheadleftarrow Read^{\mathbf{SD}}(C, \mathbf{var}\ Re)$
14 $(M, sc, \gamma) \twoheadleftarrow Demux^{\mathbf{SD}}(Y, H, \alpha, \mathbf{var}\ De)$
15 **if** $sync$ **and** $Y \preceq S$ **then**
16 $\quad S \leftarrow S \,\%\, Y$; $M, sc \leftarrow \perp$
17 **else** $sync \leftarrow 0$
18 **return** (M, sc, γ)

$\mathbf{SD}(I)$
19 $O \twoheadleftarrow \mathcal{A}_2(I, \mathbf{var}\ Env)$; **return** O

$leak(\ell, M, sc)$
20 **switch** (ℓ)
21 $\quad$ **case** lensc: **return** $\langle |M|, sc \rangle$
22 $\quad$ **case** len: **return** $\langle |M|, |sc| \rangle$
23 $\quad$ **case** none: **return** ε

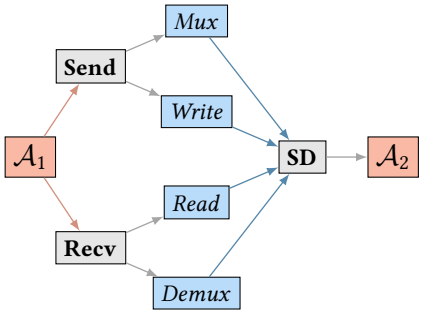


Figure 2: left: game for defining PRIV-SR and PRIV-S security of partially specified channel $\mathcal{CH}$, where $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$. Right: the call graph of the game (who may call whom).

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., [19, Definition 4.1]).

4.2.2 The adversary. Our execution model for security games is adopted from the RS framework, but we will be a bit more precise in our formulation. 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 formalize this idea as follows. An adversary is a pair of stateful, randomized algorithms with interfaces $\mathcal{A}_1(\mathbf{var\ str}\ Env) \mapsto \mathbf{bool}$ and $\mathcal{A}_2(\mathbf{str}\ I, \mathbf{var\ str}\ Env) \mapsto \mathbf{str}\ O$. Most games in this paper begin by declaring a variable Env of type **str**, which is used to share state between $\mathcal{A}_1$ and $\mathcal{A}_2$. These games also define an oracle **SD** that, on input of a string I, executes $O \twoheadleftarrow \mathcal{A}_2(I, \mathbf{var}\ Env)$ and returns O. When $\mathcal{A}_1$ makes a query to **Send** or **Recv** and a PSC algorithm is invoked, the PSC algorithm is given oracle access to **SD** for making SD queries. Algorithm $\mathcal{A}_2$ may change the value of Env as a side effect, allowing it to convey information to $\mathcal{A}_1$; algorithm $\mathcal{A}_1$ may also convey information to $\mathcal{A}_2$ by modifying the value of Env.

In the remainder, we will often denote the pair $(\mathcal{A}_1, \mathcal{A}_2)$ by $\mathcal{A}$ for convenience. We require that each of these algorithms halt, regardless of coin tosses or oracle responses, in a bounded number of steps that depends only on the length of their inputs. By convention, the adversary's runtime includes the time needed to evaluate its queries. An adversary is called *t-time* if both $\mathcal{A}_1$ and $\mathcal{A}_2$ halt in at most t time steps. We silently extend this execution model and these conventions to all subsequent security experiments in this paper.

4.2.3 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 $\ell \in \{\text{lensc}, \text{len}, \text{none}\}$. When $\ell = \text{lensc}$, only message-stream privacy is captured; when $\ell = \text{len}$ the notion captures privacy of both the message streams and their context; finally, $\ell = \text{none}$ adds length-hiding to the list.[2]

Let $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$. The game begins by initializing the adversary state Env, sender state (Mu, Wr), and receiver state (Re, De). Algorithm $\mathcal{A}_1$ is then executed with 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 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 **SD** as the SD oracle, and returns the output (C, γ) to $\mathcal{A}_1$. (Recall that $\mathcal{A}_2$ may update Env as a side effect of the **SD** queries made by Mux and $Write$.) String C is appended to S, which keeps track of the sender ciphertext stream. The second oracle, **Recv**, takes as input a ciphertext fragment C and invokes $(Y, H, \alpha) \twoheadleftarrow Read^{\mathbf{SD}}(C, \mathbf{var}\ Re)$, then $(M, sc, \gamma) \twoheadleftarrow Demux^{\mathbf{SD}}(Y, H, \alpha, \mathbf{var}\ De)$. 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 $M, sc \leftarrow \perp$. (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

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

where the probability is over the coins of the game, $\mathcal{A}_1$, $\mathcal{A}_2$, 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 of the adversary (that is, the maximum runtime of either $\mathcal{A}_1$ or $\mathcal{A}_2$), 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

[2] There are other parameters that may be of practical interest. For example, DLFK+ deal with whether the fragment encodes the end-of-stream [17, Definition 8].

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.

4.3 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 4.2.1. The second formalizes a weaker property, namely that the plaintext streams carried by the channel should remain in-sync.

4.3.1 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 $\mathcal{A}_1$'s queries consist of pairs (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 $\mathcal{A}_1$ 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.

4.3.2 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 ⇏ 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 [19, 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. 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 the full version of this paper [28], we give a natural definition of correctness for *fully specified channels*—like PSCs, but with a fully realized SD-oracle—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-oracle is realized correctly.

4.4 Receiver-status simulatability and a generic composition

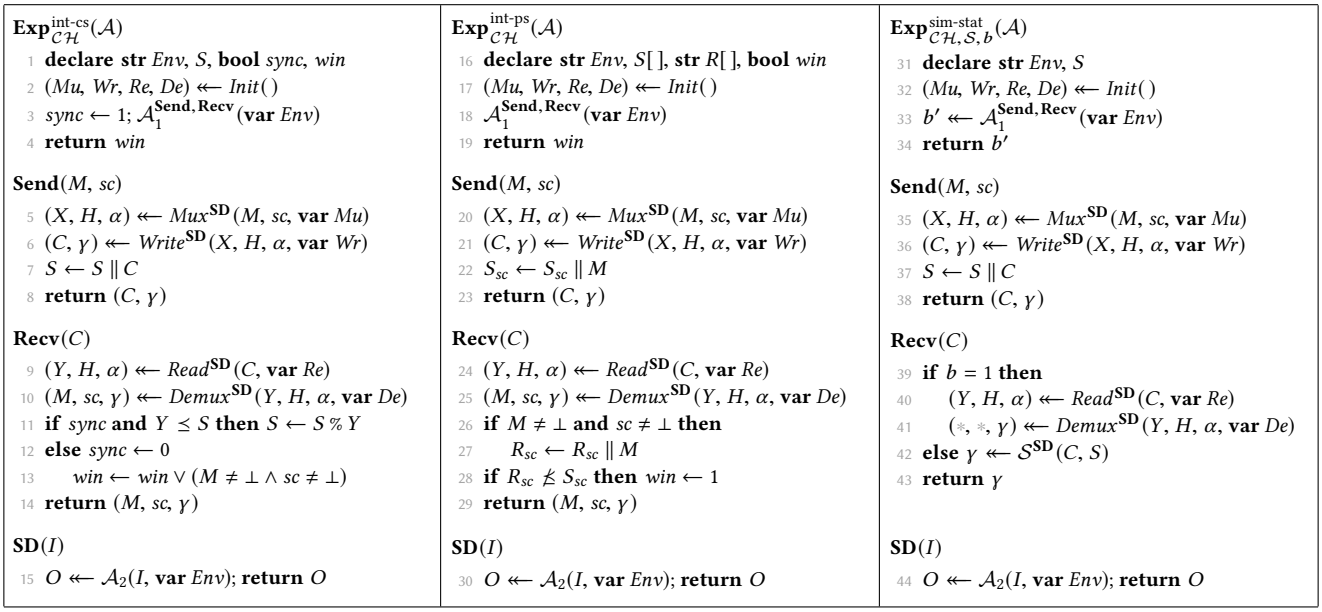
If a PSC is INT-CS secure, then an efficient attacker can do nothing but deliver the honestly produced ciphertext stream in the correct order. Thus it is intuitive that any PSC that is both PRIV-S secure and INT-CS secure will also be PRIV-SR secure, because, in effect, the **Recv** in the PRIV-SR game is useless. This is almost true; the wrinkle is that the **Recv** oracle returns status information 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* (SIM-STAT) and show that it, PRIV-S, and INT-CS imply PRIV-SR.

4.4.1 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 3) is a simulation-based game in which the adversary is asked to distinguish the status information 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 with non-negligible probability.

The game is associated to adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, 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 **SD** for servicing SD requests. When $\mathcal{S}$ halts and outputs a string γ, the oracle outputs γ. We define the advantage of $\mathcal{A}$ in attacking $\mathcal{CH}$ with simulator $\mathcal{S}$ in the SIM-STAT sense as

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

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.



$\mathbf{Exp}^{\text{int-cs}}_{\mathcal{CH}}(\mathcal{A})$
1 **declare str** *Env*, *S*, **bool** *sync*, *win*
2 $(Mu, Wr, Re, De) \twoheadleftarrow Init()$
3 $sync \leftarrow 1$; $\mathcal{A}_1^{\mathbf{Send},\mathbf{Recv}}(\mathbf{var}\ Env)$
4 **return** *win*

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

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

SD(I)
15 $O \twoheadleftarrow \mathcal{A}_2(I, \mathbf{var}\ Env)$; **return** O

$\mathbf{Exp}^{\text{int-ps}}_{\mathcal{CH}}(\mathcal{A})$
16 **declare str** *Env*, $S[\,]$, **str** $R[\,]$, **bool** *win*
17 $(Mu, Wr, Re, De) \twoheadleftarrow Init()$
18 $\mathcal{A}_1^{\mathbf{Send},\mathbf{Recv}}(\mathbf{var}\ Env)$
19 **return** *win*

Send(M, sc)
20 $(X, H, \alpha) \twoheadleftarrow Mux^{\mathbf{SD}}(M, sc, \mathbf{var}\ Mu)$
21 $(C, \gamma) \twoheadleftarrow Write^{\mathbf{SD}}(X, H, \alpha, \mathbf{var}\ Wr)$
22 $S_{sc} \leftarrow S_{sc} \parallel M$
23 **return** (C, γ)

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

SD(I)
30 $O \twoheadleftarrow \mathcal{A}_2(I, \mathbf{var}\ Env)$; **return** O

$\mathbf{Exp}^{\text{sim-stat}}_{\mathcal{CH},\mathcal{S},b}(\mathcal{A})$
31 **declare str** *Env*, *S*
32 $(Mu, Wr, Re, De) \twoheadleftarrow Init()$
33 $b' \twoheadleftarrow \mathcal{A}_1^{\mathbf{Send},\mathbf{Recv}}(\mathbf{var}\ Env)$
34 **return** b'

Send(M, sc)
35 $(X, H, \alpha) \twoheadleftarrow Mux^{\mathbf{SD}}(M, sc, \mathbf{var}\ Mu)$
36 $(C, \gamma) \twoheadleftarrow Write^{\mathbf{SD}}(X, H, \alpha, \mathbf{var}\ Wr)$
37 $S \leftarrow S \parallel C$
38 **return** (C, γ)

Recv(C)
39 **if** $b = 1$ **then**
40 $(Y, H, \alpha) \twoheadleftarrow Read^{\mathbf{SD}}(C, \mathbf{var}\ Re)$
41 $(*, *, \gamma) \twoheadleftarrow Demux^{\mathbf{SD}}(Y, H, \alpha, \mathbf{var}\ De)$
42 **else** $\gamma \twoheadleftarrow \mathcal{S}^{\mathbf{SD}}(C, S)$
43 **return** γ

SD(I)
44 $O \twoheadleftarrow \mathcal{A}_2(I, \mathbf{var}\ Env)$; **return** O

Figure 3: games for defining INT-CS (left), INT-PS (middle), and SIM-STAT (right) security for partially specified channel $\mathcal{CH}$. Let $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$.

4.4.2 PRIV-S ∧ INT-CS ∧ SIM-STAT ⇒ 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 4.1. *Let $\ell \in \{\text{lensc}, \text{len}, \text{none}\}$ 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*

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

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 [19, Theorem 4.5]. It also confirms a conjecture of FGMP; see [19, 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. The proof is given in the full version [28].

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 important: distinguishable error messages have been exploited repeatedly [3, 16, 26, 35] to attack AE-powered secure-channel protocols. As a result, there has been a considerable push in the cryptographic community to make addressing this subtlety a first class consideration [6, 14, 21].

5 THE TLS 1.3 RECORD LAYER

Our study of partially specified channels 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 5.2) 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 [29], 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 [30].

5.1 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 not fully specify every aspect of the construction. Rather, the record-layer specification [29, 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 [29, 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 [29, 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 5.1). 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 [29, 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* [29, Section 5.3]; both the key K and *IV* are derived from a shared secret [29, Sections 7.1–7.2] using an extract-and-expand key-derivation scheme [23]. The length of the *IV* is determined from the permitted nonce lengths of the AEAD scheme.[3] The nonce N is computed as $\mathit{IV} \oplus \langle \mathit{seqn}\rangle_{|\mathit{IV}|}$, 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 [29, 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.

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 [29, 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 [29, 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.

5.1.1 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 its indistinguishability*, since the record boundaries depend on it.

[3] The scheme must specify limits for valid nonce lengths per RFC 5116 [24]. The maximum must be at least 8 bytes.

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 [29, 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 5.3.

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.

5.2 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 side 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 5.1.1.) 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.

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

- $\mathit{Init}() \mapsto (\mathbf{str}\ mx, dx)$. Generates the initial state of the stream multiplexer (used by the sender) and demultiplexer (used by the receiver).
- $\mathit{Mux}^{\mathcal{O}}(\mathbf{str}\ M, sc, \mathbf{var\ str}\ mx) \mapsto (\mathbf{str}\ X, \gamma)$. 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.
- $\mathit{Demux}^{\mathcal{O}}(\mathbf{str}\ X, \mathbf{var\ str}\ dx) \mapsto (\mathbf{str}\ M, sc, \gamma)$. 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 4.

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 4.2), 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. We again formalize the adversary as a pair of algorithms $(\mathcal{A}_1, \mathcal{A}_2)$. The first, $\mathcal{A}_1$, 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 **SD** for handling SD requests, which in turn invokes $\mathcal{A}_2$), 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 $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ in attacking $\mathcal{M}$ in the mPRIV-S(ℓ) sense as

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

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

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

Init()
declare Muxer Mu, Writer Wr
declare Reader Re, Demuxer De
(Mu.mx, De.dx) ⇐ M.Init()
Mu.ng ⇐ N.Init(); Re.ng ← Mu.ng
Wr.K ⇐ K; De.K ← Wr.K
De.sync ← 1
return (Mu, Wr, Re, De)

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

Write^O(X, N, α, var Writer Wr)
declare str A, γ
⟨A, γ⟩ ⇐ O(⟨write, create ad, |X|, α⟩)
if X = ⋄ then return (⋄, γ)
Y′ ← AE.Enc(Wr.K, N, A, X)
if Y′ = ⊥ then
    γ ⇐ O(⟨write, invalid ptxt⟩)
    return (⋄, γ)
return (A ‖ Y′, γ)

Read^O(C, var Reader Re)
declare str α, int c, bool drop
Re.buf ← Re.buf ‖ C
⟨c, drop, α⟩ ⇐ O(⟨read, len, Re.buf⟩)
Y ← Re.buf[:c]; Re.buf ← Re.buf % Y
if Y = ⋄ or drop then return (⋄, ⋄, α)
N ← N.Next(var Re.ng)
return (Y, N, α)

Demux^O(Y, N, α, var Demuxer De)
declare str X, γ, int a
⟨a, γ⟩ ⇐ O(⟨demux, ad len, Y, α⟩)
if (Y = ⋄ and γ ≠ ⋄) or ¬De.sync
    then return (⊥, ⊥, γ)
else if Y ≠ ⋄ then
    A ← Y[:a]; Y′ ← Y % A
    X ← AE.Dec(De.K, N, A, Y′)
    if X = ⊥ then
        De.sync ← 0
        γ ⇐ O(⟨demux, invalid ctxt⟩)
        return (⊥, ⊥, γ)
(M, sc, γ) ⇐ M.Demux^O(X, var De.dx)
return (M, sc, γ)
```

Figure 4: top-left: PSC TLS$[\mathcal{M}, \mathcal{AE}, \mathcal{N}]$ = $(Init, Mux, Write, Read, Demux)$ composed of partially specified stream multiplexer $\mathcal{M}$, AEAD scheme $\mathcal{AE}$ with key space $\mathcal{K}$, and nonce generator $\mathcal{N}$. Top-right: the mPRIV-S and SIM-mSTAT games associated with $\mathcal{M}$ and adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$. Bottom-right: the PRIV and INT games associated with $\mathcal{AE}$ and AEAD adversary $\mathcal{A}$.

```
Exp^{mpriv-s}_{M,ℓ,b}(A)   Exp^{sim-mstat}_{M,S,b}(A)
Env ← ε; (mx, dx) ⇐ Init()
b′ ⇐ A_1^{Mux}(var Env)
b′ ⇐ A_1^{Demux}(var Env)
return b′

Mux(M_0, sc_0, M_1, sc_1)
L_0 ← leak(ℓ, M_0, sc_0)
L_1 ← leak(ℓ, M_1, sc_1)
if L_0 ≠ L_1 then return (⊥, ⊥)
(X, γ) ⇐ Mux^{SD}(M_b, sc_b, var mx)
return (|X|, γ)

Demux(X)
if b = 1 then (*, *, γ) ⇐ Demux^{SD}(X, var dx)
else γ ⇐ S^A(|X|)
return γ

SD(I)
O ⇐ A_2(I, var Env); return O

Exp^{priv}_{AE,b}(A)   Exp^{int}_{AE}(A)
X, Q ← ∅; K ⇐ K
res ⇐ A^{Enc}   res ← 0; A^{Enc,Dec}
return res

Enc(N, A, M)
if N ∈ X return ⊥
C ← Enc_K^{N,A}(M)
if b = 0 then C ⇐ {0,1}^{λ(|M|)}
Q ← Q ∪ {N, A, C}; X ← X ∪ {N}
return C

Dec(N, A, C)
M ← Dec_K^{N,A}(C)
if M ≠ ⊥ and (N, A, C) ∉ Q then res ← 1
return M
```

message; otherwise it executes the simulator $\mathcal{S}$ on input $|X|$ and with access to **SD** for servicing SD requests. Define the advantage of $\mathcal{A}$ in attacking $\mathcal{M}$ in the SIM-mSTAT sense with simulator $\mathcal{S}$ as

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

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.

5.2.2 AEAD schemes. We describe the syntax for authenticated encryption with associated data as prescribed by the spec [24]. An AEAD scheme is a triple $\mathcal{AE} = (Enc, Dec, \lambda)$. 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:

- $Enc(\mathbf{str}\ K, N, A, M) \mapsto \mathbf{str}\ C$. Maps a key K, nonce N, associated data A, and plaintext M to a ciphertext C such that if $C \neq \perp$, then $|C| = \lambda(|M|) \geq |M|$.
- $Dec(\mathbf{str}\ K, N, A, C) \mapsto \mathbf{str}\ M$. 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 $Enc_K^{N,A}(M)$. (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 $Enc(K, N, A, M) \neq \perp$ if and only if $(K, N, A, M) \in \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{M}$; correctness requires that $Dec(K, K, N, A, Enc(K, N, A, M)) = M$ 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 [25].)

We will use standard notions of indistinguishability under chosen-plaintext attack (PRIV) and integrity of ciphertexts (INT) as defined in Figure 4. As usual, the indistinguishability game requires that the adversary not repeat a nonce. The adversary for the PRIV and

INT games is simply a randomized algorithm $\mathcal{A}() \mapsto \textbf{bool}$. that expects access to one or more oracles. To distinguish it from other adversaries in this paper, we will refer to it as an AEAD adversary. Define the PRIV advantage of adversary $\mathcal{A}$ in attacking $\mathcal{AE}$ as

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

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

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

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).

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

- $\mathit{Init}() \mapsto \textbf{str}\ \mathit{ng}$. Initializes the state of the generator.
- $\mathit{Next}(\textbf{var str}\ \mathit{ng}) \mapsto \textbf{str}\ N$. 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 $\mathit{ng} \twoheadleftarrow \mathit{Init}()$, then computes $N_i \leftarrow \mathit{Next}(\textbf{var}\ \mathit{ng})$ 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 $\text{coll}_{\mathcal{N}}(t) = \Pr\left[\, \mathit{Coll}_{\mathcal{N}}(t) = 1 \,\right]$.

5.3 The partially specified record layer

We are now ready to formalize the record layer specification. Refer to the PSC $\mathbf{TLS}[\mathcal{M},\mathcal{AE},\mathcal{N}]$ defined in Figure 4. It differs from the standard (draft 23) in one small, but security-critical way: the standard mandates that the AEAD scheme 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 4:21 and 4:39. 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 note that this attack is not an artificat of our security model. The strength of our model—and hence the possibility of this attack—is inherited from the stream-based channel setting; if one were to directly extend FGMP's syntax and security notions so that they encompass multiplexing, then the record layer woul have the same problem.

The procedure *Mux* invokes $\mathcal{M}$ (4:14) in order to compute the next channel fragment (i.e. record). It is designed to never operate on 0-length records (4:15); if the first input X to *Write* is undefined (i.e., $X = \varepsilon$), then it outputs a 0-length ciphertext fragment (4:20). The data on the wire is $A \,\|\, Y'$, where Y' is the ciphertext and A is a string chosen by the SD (4:19).

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 (4:34). 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 (4:34 and 4:40–42). 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 4:43 may be invoked on $X = \varepsilon$.) Usage limits are enforced by the SD (4:19 and 4:33).

Our construction captures all protocol-level side effects in the record layer specification [29] 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 5.1.)

5.3.1 Security. Let $\mathcal{CH} = \mathbf{TLS}[\mathcal{M},\mathcal{AE},\mathcal{N}]$ be as defined in Figure 4. Our first step is to show that PRIV of $\mathcal{AE}$ and mPRIV-S of $\mathcal{M}$ imply PRIV-S for $\mathcal{CH}$:

Theorem 5.1. *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*

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

The proof first appeals to the PRIV security of $\mathcal{AE}$ to transition to a "random" setting in which **Send** queries are evaluated without using $\mathcal{AE}.\mathit{Enc}$ and instead just generate a random string of the appropriate length. To get there, we first need to upper bound the probability that a nonce is repeated. To complete the proof, we construct an mPRIV-S adversary that simulates a PRIV-S in this random setting. This is made possible by virtue of the length of ciphertexts only depending on the length of the plaintext, i.e., the information provided to the adversary. The full proof, and the proof for all subsequent theorems, can be found in the full version of this paper [28].

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

Theorem 5.2. *For every $t, q_1, q_2, \mu_1, \mu_2 \in \mathbb{N}$ it holds that*

$$\mathbf{Adv}^{\text{int-cs}}_{\mathcal{CH}}(t,\boldsymbol{r}) \le \mathbf{Adv}^{\text{int}}_{\mathcal{AE}}(t + O(q_1+q_2),\boldsymbol{r})\,.$$

where $\boldsymbol{r} = (q_1, q_2, \mu_1, \mu_2)$,

To prove this, it suffices to show that if the game's *sync* flag gets set to 0, then with overwhelming probability, the flag *De.sync* in the PSC gets set to 0 as well. Next, a similar argument allows us to reduce the SIM-STAT security of $\mathcal{CH}$ to the SIM-mSTAT security of $\mathcal{M}$:

Theorem 5.3. *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*

$$\mathbf{Adv}^{\text{sim-stat}}_{\mathcal{CH},\mathcal{S}}(t, \boldsymbol{r}) \le \mathbf{Adv}^{\text{sim-mstat}}_{\mathcal{M},\mathcal{T}}(\tilde{t}, q_2, \mu_2) + \mathbf{Adv}^{\text{int}}_{\mathcal{AE}}(\tilde{t}, \boldsymbol{r}),$$

where $\tilde{t} = t + O(q_1 + q_2)$, and $\boldsymbol{r} = (q_1, q_2, \mu_1, \mu_2)$.

The proof begins with the same argument used in Theorem 5.2, 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 $\mathcal{S}$. Finally, putting together Theorems 4.1, 5.1, 5.2, and 5.3 yields our result for the PRIV-SR security of $\mathcal{CH}$:

Corollary 5.4. *For every $t, s, q_1, q_2, \mu_1, \mu_2 \in \mathbb{N}$ and s-time simulator $\mathcal{S}$, it holds that*

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

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 $\ell \in \{\text{lensc}, \text{len}, \text{none}\}$.

5.3.2 Limitations of our analysis. The stream multiplexer, $\mathcal{M}$, is responsible for record fragmentation, encoding (including the content type), and padding. It is also responsible for the length and order of records. As discussed in Section 5.1, all of these details matter for security; as we have just seen, the mPRIV-S and SIM-mSTAT notions make clear what properties $\mathcal{M}$ must possess in order for the record layer to be secure in the PRIV-SR sense.

We emphasize, however, that PRIV-SR security says nothing about whether a particular implementation of the record layer is *operationally correct*. (For example, whether $\mathcal{CH}$ properly handles streams depends on how $\mathcal{M}$ encodes the content type.) All it says is that whether the record layer is correct is irrelevant for PRIV-SR security. But in the absence of a proof of correctness, attacks in the INT-PS sense are possible, including important real-world attacks such as truncation attacks [34]. In the full version [28], we describe a sufficient condition for $\mathcal{CH}$ under which it achieves INT-PS security. 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. (This reflects a result of FGMP.) Thus, security for $\mathcal{CH}$ follows from the INT security of $\mathcal{AE}$ via Theorem 5.2. An interesting question is whether correctness of $\mathcal{M}$, along with INT security of $\mathcal{AE}$, suffices for INT-PS of $\mathcal{CH}$. We leave this for future work.

The subject of this paper is the mechanism by which data streams are protected in TLS 1.3. Our model permits the study of the security of data transmitted between key changes. (See the discussion in Section 5.1.) This is valid, since under appropriate assumptions about the underlying key-derivation function used in TLS, the record-layer state is effectively independent between key changes. However, one limitation of our model is that we cannot say anything about the security of the *concatenation* of data sent *across* key changes. In particular, consider the concatenation of the application-data stream sent in the early-data phase and in the post-handshake phase. Early data is replayable, since the adversary can send this data to any number of valid recipients in possession of a pre-shared key shared with the client. Our model cannot account for such replay attacks. This also limits our ability to study truncation attacks [34], since these may involve data sent across key changes. Finally, we note that since we have analyzed TLS 1.3 in isolation, our results say nothing about the record layer specifications in TLS 1.2, 1.1, 1.0, SSL 3, and so on.

6 CONCLUSION

Despite these limitations, 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 4) 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 5.1 and 5.3—provide guidance on how to develop implementations that achieve a target security goal. Concretely, this goal is a property 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 5.2), which was adopted in the final version.

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. Although the partially specified protocol framework is not the only way to reason about how unspecified or underspecified matters affect security, we found it to be a useful tool for discovering what these security-critical matters are in the first place. 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.

7 ACKNOWLEDGMENTS

We thank Mihir Bellare and the CCS program committee for their useful feedback. This work was supported by NSF grants CNS-1564444 and CNS-1816375.

REFERENCES

[1] Martin R. Albrecht, Jean Paul Degabriele, Torben Brandt Hansen, and Kenneth G. Paterson. 2016. A Surfeit of SSH Cipher Suites. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1480–1491.

[2] Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. 2009. Plaintext Recovery Attacks Against SSH. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*. IEEE, 16–26.

[3] N. J. AlFardan and K. G. Paterson. 2013. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 526–540.

[4] Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Nicky Mouha, and Kan Yasuda. 2014. How to Securely Release Unverified Plaintext in Authenticated Encryption. In *Advances in Cryptology – ASIACRYPT 2014*. Springer Berlin Heidelberg, 105–125.

[5] Christian Badertscher, Christian Matt, Ueli Maurer, Phillip Rogaway, and Björn Tackmann. 2015. Augmented Secure Channels and the Goal of the TLS 1.3 Record Layer. In *Provable Security*. Springer International Publishing, 85–104.

[6] Guy Barwell, Daniel Page, and Martijn Stam. 2015. Rogue Decryption Failures: Reconciling AE Robustness Notions. In *Proceedings of the 15th IMA International Conference on Cryptography and Coding*. Springer International Publishing, 94–111.

[7] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. 2004. 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] Mihir Bellare and Chanathip Namprempre. 2000. Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm. Cryptology ePrint Archive, Report 2000/025. https://eprint.iacr.org/2000/025.

[9] Mihir Bellare and Phillip Rogaway. 2000. Encode-Then-Encipher Encryption: How to Exploit Nonces or Redundancy in Plaintexts for Efficient Cryptography. In *Advances in Cryptology — ASIACRYPT 2000*. Springer Berlin Heidelberg, 317–330.

[10] Mihir Bellare and Phillip Rogaway. 2006. 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, 409–426.

[11] Mihir Bellare and Björn Tackmann. 2016. The Multi-user Security of Authenticated Encryption: AES-GCM in TLS 1.3. In *Advances in Cryptology – CRYPTO 2016*. Springer Berlin Heidelberg, 247–276.

[12] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti, and P. Y. Strub. 2014. Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*. IEEE, 98–113.

[13] Alexandra Boldyreva, Jean Paul Degabriele, Kenneth G. Paterson, and Martijn Stam. 2012. Security of Symmetric Encryption in the Presence of Ciphertext Fragmentation. In *Advances in Cryptology – EUROCRYPT 2012*. Springer Berlin Heidelberg, 682–699.

[14] Alexandra Boldyreva, Jean Paul Degabriele, Kenneth G. Paterson, and Martijn Stam. 2014. On Symmetric Encryption with Distinguishable Decryption Failures. In *Fast Software Encryption*. Springer Berlin Heidelberg, 367–390.

[15] J. P. Degabriele, K. Paterson, and G. Watson. 2011. Provable Security in the Real World. *IEEE Security & Privacy* 9, 3 (2011), 33–41.

[16] Jean Paul Degabriele and Kenneth G. Paterson. 2010. On the (in)Security of IPsec in MAC-then-encrypt Configurations. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*. ACM, 493–504.

[17] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Beguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue. 2017. Implementing and Proving the TLS 1.3 Record Layer. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP)*. IEEE, 463–482.

[18] Marc Fischlin, Felix Günther, Giorgia Azzurra Marson, and Kenneth G. Paterson. 2015. Data Is a Stream: Security of Stream-Based Channels. In *Advances in Cryptology – CRYPTO 2015*. Springer Berlin Heidelberg, 545–564.

[19] Marc Fischlin, Felix Günther, Giorgia Azzurra Marson, and Kenneth G. Paterson. 2017. Data Is a Stream: Security of Stream-Based Channels. Cryptology ePrint Archive, Report 2017/1191. https://eprint.iacr.org/2017/1191.

[20] Google. [n. d.]. QUIC, a multiplexed stream transport over UDP. https://www.chromium.org/quic, accessed 13 Feb 2018.

[21] Viet Tung Hoang, Ted Krovetz, and Phillip Rogaway. 2015. Robust Authenticated-Encryption AEZ and the Problem That It Solves. In *Advances in Cryptology – EUROCRYPT 2015*. Springer Berlin Heidelberg, 15–44.

[22] S. Kent and K. Seo. 2005. *Security Architecture for the Internet Protocol*. RFC 4301. RFC Editor. http://www.rfc-editor.org/rfc/rfc4301.txt http://www.rfc-editor.org/rfc/rfc4301.txt.

[23] Hugo Krawczyk. 2010. Cryptographic Extraction and Key Derivation: The HKDF Scheme. In *Advances in Cryptology – CRYPTO 2010*. Springer Berlin Heidelberg, 631–648.

[24] D. McGrew. 2008. *An Interface and Algorithms for Authenticated Encryption*. RFC 5116. RFC Editor. http://www.rfc-editor.org/rfc/rfc5116.txt http://www.rfc-editor.org/rfc/rfc5116.txt.

[25] Chanathip Namprempre, Phillip Rogaway, and Thomas Shrimpton. 2014. Reconsidering Generic Composition. In *Advances in Cryptology – EUROCRYPT 2014*. Springer Berlin Heidelberg, 257–274.

[26] Kenneth G. Paterson and Nadhem J. AlFardan. 2012. Plaintext-Recovery Attacks Against Datagram TLS. In *19th Annual Network and Distributed System Security Symposium, NDSS*.

[27] Kenneth G. Paterson, Thomas Ristenpart, and Thomas Shrimpton. 2011. Tag Size *does* Matter: Attacks and Proofs for the TLS Record Protocol. In *Advances in Cryptology – ASIACRYPT 2011*. Springer Berlin Heidelberg, 372–389.

[28] Christopher Patton and Thomas Shrimpton. 2018. Partially specified channels: The TLS 1.3 record layer without elision. Cryptology ePrint Archive, Report 2018/634. https://eprint.iacr.org/2018/634.

[29] Eric Rescorla. 2018. *The Transport Layer Security (TLS) Protocol Version 1.3*. Internet-Draft draft-ietf-tls-tls13-23. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-tls-tls13-23.txt https://tools.ietf.org/html/draft-ietf-tls-tls13-23.

[30] E. Rescorla. 2018. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. RFC Editor.

[31] E. Rescorla, H. Tschofenig, and N. Modadugu. 2017. *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3*. Internet-Draft draft-ietf-tls-dtls13-22. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-tls-dtls13-22.txt https://tools.ietf.org/html/draft-ietf-tls-dtls13-22.

[32] Phillip Rogaway. 2002. Authenticated-encryption with Associated-data. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*. ACM, 98–107.

[33] P. Rogaway and T. Stegers. 2009. Authentication without Elision. In *2009 22nd IEEE Computer Security Foundations Symposium*. IEEE, 26–39.

[34] Ben Smyth and Alfredo Pironti. 2013. Truncating TLS Connections to Violate Beliefs in Web Applications. In *Presented as part of the 7th USENIX Workshop on Offensive Technologies*. USENIX.

[35] Serge Vaudenay. 2002. Security Flaws Induced by CBC Padding — Applications to SSL, IPSEC, WTLS.... In *Advances in Cryptology — EUROCRYPT 2002*. Springer Berlin Heidelberg, 534–545.

[36] T. Ylonen and C. Lonvick. 2006. *The Secure Shell (SSH) Protocol Architecture*. RFC 4251. RFC Editor. http://www.rfc-editor.org/rfc/rfc4251.txt http://www.rfc-editor.org/rfc/rfc4251.txt.