2017 IEEE Symposium on Security and Privacy

Implementing and Proving the TLS 1.3 Record Layer

Antoine Delignat-Lavaud, Cédric Fournet,
Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi,
Nikhil Swamy, Santiago Zanella-Béguelin
Microsoft Research
{antdl,fournet,markulf,protz,aseemr,
nswamy,santiago}@microsoft.com

Karthikeyan Bhargavan, Jianyang Pan,
Jean Karim Zinzindohoué
INRIA Paris-Rocquencourt
karthikeyan.bhargavan@inria.fr,
panyang314@gmail.com,
jean-karim.zinzindohoue@inria.fr

***Abstract*—The record layer is the main bridge between TLS applications and internal sub-protocols. Its core functionality is an elaborate form of authenticated encryption: streams of messages for each sub-protocol (handshake, alert, and application data) are fragmented, multiplexed, and encrypted with optional padding to hide their lengths. Conversely, the sub-protocols may provide fresh keys or signal stream termination to the record layer.**

Compared to prior versions, TLS 1.3 discards obsolete schemes in favor of a common construction for Authenticated Encryption with Associated Data (AEAD), instantiated with algorithms such as AES-GCM and ChaCha20-Poly1305. It differs from TLS 1.2 in its use of padding, associated data and nonces. It also encrypts the content-type used to multiplex between sub-protocols. New protocol features such as early application data (0-RTT and 0.5-RTT) and late handshake messages require additional keys and a more general model of stateful encryption.

We build and verify a reference implementation of the TLS record layer and its cryptographic algorithms in F*, a dependently typed language where security and functional guarantees can be specified as pre- and post-conditions. We reduce the high-level security of the record layer to cryptographic assumptions on its ciphers. Each step in the reduction is verified by typing an F* module; for each step that involves a cryptographic assumption, this module precisely captures the corresponding game.

We first verify the functional correctness and injectivity properties of our implementations of one-time MAC algorithms (Poly1305 and GHASH) and provide a generic proof of their security given these two properties. We show the security of a generic AEAD construction built from any secure one-time MAC and PRF. We extend AEAD, first to stream encryption, then to length-hiding, multiplexed encryption. Finally, we build a security model of the record layer against an adversary that controls the TLS sub-protocols. We compute concrete security bounds for the AES_128_GCM, AES_256_GCM, and CHACHA20_POLY1305 ciphersuites, and derive recommended limits on sent data before re-keying.

We plug our implementation of the record layer into the miTLS library, confirm that they interoperate with Chrome and Firefox, and report initial performance results. Combining our functional correctness, security, and experimental results, we conclude that the new TLS record layer (as described in RFCs and cryptographic standards) is provably secure, and we provide its first verified implementation.

I. Introduction

Transport Layer Security (TLS) is the main protocol for secure communications over the Internet. With the fast growth of TLS traffic (now most of the Web [48]), numerous concerns have been raised about its security, privacy, and performance. These concerns are justified by a history of attacks against deployed versions of TLS, often originating in the record layer.

History and Attacks Wagner and Schneier [49] report many weaknesses in SSL 2.0. The MAC construction offers very weak security regardless of the encryption strength. The padding length is unauthenticated, allowing an attacker to truncate fragments. Stream closure is also unauthenticated; although an end-of-stream alert was added in SSL 3.0, truncation attacks persist in newer TLS versions [12, 44].

The original MAC-pad-encrypt mode is not generically secure [31] and is brittle in practice, despite encouraging formal results for specific algorithms [2, 11, 39]. Many padding oracle attacks have surfaced over the years, ranging from attacks exploiting straightforward issues (such as implementations sending padding error alerts after decryption) to more advanced attacks using side channels [1, 37]. Although well understood, padding oracle attacks remain difficult to prevent in TLS implementations [45]. The CBC mode of operation is also not secure against chosen-plaintext attacks when the IV is predictable (as in TLS 1.0), which is exploited in the BEAST attack [20]. Random explicit IVs [15] and CBC mode for 64-bit block ciphers [10] are also vulnerable to birthday attacks. Finally, fragment compression can be exploited in adaptive chosen-plaintext attacks to recover secrets [40].

Even with provably-secure algorithms, functional correctness and memory safety are essential, inasmuch as implementation bugs can easily nullify security guarantees. For instance, the OpenSSL implementation of ChaCha20-Poly1305 has been found to contain arithmetic flaws [14] and more recently, a high severity buffer overflow vulnerability [47].

Changes in TLS 1.3 The IETF aims to robustly fix the

DOI 10.1109/SP.2017.58

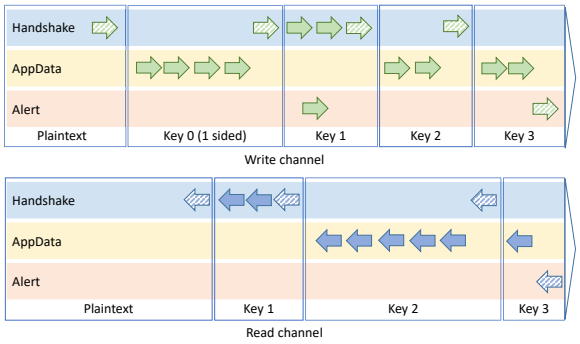

Figure 1. Multiplexing of sub-protocol streams by the record layer, depicting a TLS 1.3 draft-19 0-RTT handshake with re-keying.

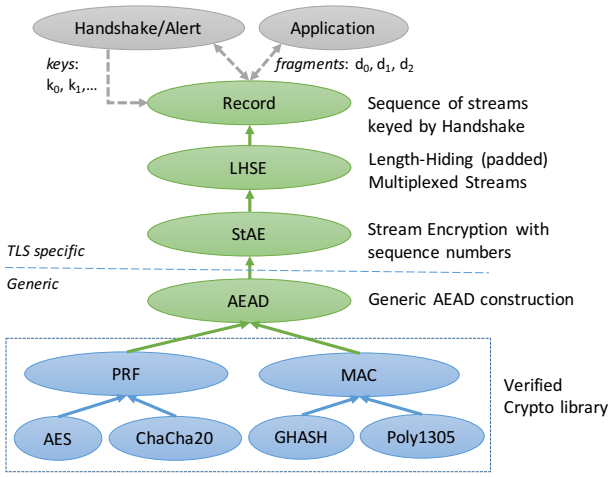

Figure 2. Modular structure of our proof. Green arrows denote security reductions proved by typing.

weaknesses of the record layer by adopting a single AEAD mode for all ciphersuites, thus deprecating all legacy modes (MAC-only, MAC-pad-encrypt, RFC 7366 [26] encrypt-then-MAC, compress-then-encrypt). The new AEAD mode is designed to be provably-secure and modular, supporting algorithms such as AES-GCM, AES-CCM, and ChaCha20-Poly1305 within the same framework. The usage of AEAD has also been improved: authentication no longer relies on associated data, whereas implicit nonces derived from initialization vectors (IV) and sequence numbers yield better security and performance.

What is the Record Layer? TLS involves establishing and using many encryption keys. In the key exchange literature, a common viewpoint is to treat each key generated in the handshake as belonging to a specific, independent application. Under this model, the handshake encryption key is used only by the handshake to encrypt its own messages, and must be separate from the application data key used only to encrypt application data fragments. This model does not fit the actual use of keys in TLS: it fails to capture TLS 1.2 renegotiation (where handshake messages are interleaved with the application data stream), TLS 1.3 post-handshake authentication and re-keying, or even alerts in any TLS version. In our modularization of TLS, following Bhargavan et al. [11], we consider that each sub-protocol of TLS—handshake, change cipher spec (CCS), alert and application data (AppData)—defines its own data stream. The role of the record is to multiplex all of these streams into one, corresponding to network messages after fragmentation, formatting, padding, and optional record-layer encryption. Under this model, the record layer is the exclusive user for all non-exported keys generated by the handshake, and there is no need to assign keys to any given sub-protocol stream.

Figure 1 illustrates the stream multiplexing for a TLS 1.3 connection with 0-RTT data and one re-keying from the point of view of the client. Separate channels are used for writing and reading. Within each channel, a band in the figure represents a stream, and arrows represent message fragments (incoming for left arrows, outgoing for right arrows) over time (flowing from left to right). Dashed arrows represent fragments used to signal key changes to the record layer. In TLS 1.2, CCS messages signal key changes; in TLS 1.3 this function is taken over by handshake messages.

Related Work Since the first draft of TLS 1.3 in April 2014, the handshake and key schedule have undergone significant analysis efforts [17, 18, 23, 28, 29, 32] as their design evolved over 19 iterations (at the time of writing). In contrast, few authors have analyzed changes to the record layer: Fischlin et al. [22] and Badertscher et al. [3] analyze an early draft that did not feature many of the current changes (for instance, it still relied on associated data to authenticate record meta-data), and Bellare and Tackmann [7] specifically focus on the way nonces are derived from IVs. This focus on the handshake protocol may be explained by the difficulty of analyzing the record independently of the handshake, and more generally, of defining the precise scope of its functionality. Authenticated and Confidential Channel Establishment (ACCE) [27, 33] is a game-based model that combines the handshake and the record protocols. While ACCE models capture complex features of TLS 1.2 such as renegotiation [25], its focus is primarily on the handshake, and it is unclear how to capture features such as post-handshake authentication [19]. Other limits of ACCE models are discussed in [3].

Our contributions We provide a reference implementation of the TLS record layer and its underlying cryptographic algorithms using the F* programming language. We define security as an indistinguishability game and show a reduction with concrete bounds (Table I) for any distinguisher to standard, low-level cryptographic assumptions. Our proof structure is depicted in Figure 2; from the bottom up:

1) We build a generic library for one-time message authentication codes (MAC) based on the Wegman-Carter-Shoup construction. We implement the GHASH and Poly1305 algorithms, and we prove the functional correctness (with regards to mathematical specifications), memory safety, and encoding injectivity of the resulting low-level code (§III). Similarly, we build a library for pseudo-random functions (PRF), and provide functionally-correct, memory-safe implementations for AES and ChaCha20 (§IV). We express the security guarantees of these libraries using cryptographic idealizations backed by game-based assumptions.

2) We describe a generic AEAD construction that captures both RFC 5288 [41] for AES-GCM (as described in NIST SP800-38D [21]) and RFC 7539 [38] for ChaCha20-Poly1305 through an interface compatible with RFC 5116 [36]. We show that this construction satisfies a standard notion of AEAD security (§V) that combines indistinguishability under chosen-plaintext attacks (IND-CPA) with ciphertext integrity (INT-CTXT). Our proof applies to our F⋆ implementation and, based on the idealizations of our two libraries, is verified by typing—as are the following three steps:

3) From AEAD, we build and verify stream encryption, which uses AEAD nonces and record sequence numbers according to the TLS version-specific format (§VI).

4) From stream encryption, we build a length-hiding encryption interface by adding padding, the TLS-specific content type multiplexing, and version-specific associated data (§VII).

5) From length-hiding stream encryption with multiplexing, we implement the TLS record layer by adding interfaces to the handshake and alert sub-protocols that extend streams to sequences of streams by installing and enabling keys (§VIII). This captures novel protocol features of TLS 1.3 such as early application data (0-RTT and 0.5 RTT), late handshake messages, and re-keying. Based on our security bound, we propose a re-keying strategy that compensates for potential weaknesses in AES-GCM.

6) We evaluate our implementation of the TLS record layer (§IX) by linking our AES-GCM and ChaCha20-Poly1305 ciphersuites to the handshake implementation of miTLS [11]. We confirm network interoperability with other TLS libraries both for TLS 1.2 and TLS 1.3 draft-14 and draft-18. Our code and formal development maximize reuse between TLS 1.2 and 1.3.

Additional Materials An extended version of this paper with additional details and proofs of the results in §III, §IV, and §V is available from a companion website https://project-everest.github.io/record/. This website also provides links to online repositories that include all verified code we report on in the paper, as well as instructions on how to verify, extract and run the code.

II. Compositional Verification by Typing

To implement and verify the record layer, we adopt a compositional approach to functional correctness and cryptographic security based on F⋆ [46], a dependently-typed programming language. This section explains our approach on two examples: arithmetic in a prime field for the Poly1305 algorithm, and basic authenticated encryption. We refer the reader to Fournet et al. [24] for a general presentation of our approach and Barthe et al. [4] for a probabilistic semantics of F⋆ and additional cryptographic examples.

We use F⋆ not only to implement cryptographic constructions, but also as the formal syntax for their game-based security definitions. This is akin to the approach taken by Bhargavan et al. [13] in their proof of a TLS 1.2 handshake implementation using F7, an ancestor of F⋆. In contrast to F7, F⋆ supports an imperative programming style that is closer to pseudo-code used by cryptographers in code-based games [5]. For most of the paper, we use such pseudo-code instead of more precise and verbose F⋆ code. We do not assume familiarity with F⋆ and elide details of its syntax, such as type annotations, that are not relevant for the developments in this paper.

Functional Correctness of Poly1305 As a first example in F⋆ syntax, we specify arithmetic in the field $GF(2^{130}-5)$ for the Poly1305 MAC algorithm as follows:

```
let p = 2^130 − 5 (* the prime order of the field *)
type elem = n:nat {n < p} (* the type of field elements *)
let x +@ y : Tot elem = (x + y) % p (* field addition *)
let x *@ y : Tot elem = (x * y) % p (* field multiplication *)
```

This code uses F⋆ mathematical (unbounded) natural numbers to define the prime order p of the field and the type of field elements. (The formula `{n < p}` line 2 states that this type is inhabited by natural numbers n smaller than p.) The code also defines two infix operators +@ and *@ for addition and multiplication in the field, relying on primitive, unbounded integer arithmetic. Their result is annotated with types `Tot elem`, to indicate that these operations are pure total functions that return field elements. Hence, the F⋆ typechecker automatically checks that their result is in the field; it would report an error if e.g. we omitted the reduction modulo p. These operations are convenient to specify polynomial computations (see §III-B) but highly inefficient.

Instead, typical 32-bit implementations of Poly1305 represent field elements as mutable arrays of 5 unsigned 32-bit integers, each holding 26 bits. This low-level representation evenly spreads out the bits across the integers, so that carry-overs during arithmetic operations can be delayed. It also enables an efficient modulo operation for p. We show below an excerpt of the interface of our lower level verified implementation, relying on the definitions above to specify its correctness.

```
abstract type repr = buffer UInt32.t 5 (* 5-limb representation *)
val select: memory → r:repr → Tot elem (* current value held in r *)
val multiply: e0:repr → e1:repr → ST unit
  (requires live e0 ∧ live e1 ∧ disjoint e0 e1)
  (modifies e0)
  (ensures select e0′ = select e0 *@ select e1)
```

The type repr defines the representation of field elements as F⋆ buffers (i.e., mutable arrays) of 5 32-bit integers. It is marked as abstract, to hide this representation from the rest of the code. Functions are declared with a series of argument types (separated by →) ending with a return type and an effect (e.g. Tot or ST). Functions may have logical pre- and post-conditions that refer to their arguments, their result, and their effects on the memory. If they access buffers, they typically have a pre-condition requiring their caller to prove that the buffers are 'live' in the current memory (that is, they have been allocated and haven't been de-allocated yet) and they also explicitly state which buffers they modify.

The total function select is used only in specifications; it reads the value of an element from the program memory. We use it, for example, in the stateful specification of multiply. In the types above, we keep the memory argument implicit, writing select e0 and select e0′ for the values of e0 in initial and final memories, respectively. (In real F⋆ code, pre- and post-conditions take these memories as explicit arguments.)

The multiply function is marked as ST, to indicate a stateful computation that may use temporary stack-based allocations. It requires that its arguments e0 and e1 be live and disjoint; it computes the product of its two arguments and overwrites e0 with the result. Its post-condition specifies the result in terms of the abstract field multiplication of the arguments.

Implementing and proving that multiply meets its mathematical specification involves hundreds of lines of source code, relying on a custom Bignum library with lemmas on integer representations and field arithmetic (see §IX). Such code is easy to get wrong, but once F⋆ typechecks it, we are guaranteed that our low-level code is safe (e.g. it never accesses buffers out of bound, or de-allocated buffers) and functionally correct (since their results are fully specified). All F⋆ types and specifications are then erased, so that the compiled code only performs efficient low-level operations.

Authenticated Encryption: Real Interface Let us consider a simplified version of the authenticated encryption (AE) functionality at the core of the TLS record layer. In F⋆, we may write an AE module with the following interface:

```
val ℓ_p: nat
val ℓ_c: nat
type lbytes (ℓ:nat) = b:bytes{length b = ℓ}
type bbytes (ℓ:nat) = b:bytes{length b ≤ ℓ}
type plain = lbytes ℓ_p
type cipher = lbytes ℓ_c
abstract type key
val keygen: unit → ST key
val decrypt: key → cipher → Tot (option plain)
val encrypt: k:key → p:plain → ST (c:cipher{decrypt k c = Some p})
```

Plaintexts and ciphertexts are represented here as immutable bytestrings of fixed lengths ℓ_p and ℓ_c. We frequently rely on type abbreviations to statically enforce length checks for fixed-length bytestrings using lbytes ℓ, and for bounded-length bytestrings using bbytes ℓ. (Our presentation uses immutable bytestrings for simplicity, whereas our record-layer implementation also uses mutable buffers of bytes.)

Next, our interface defines an abstract type key; values of this type can only be generated via keygen and accessed via encrypt and decrypt. The internal representation of keys is hidden from all other modules to protect their integrity and secrecy. The function keygen needs to generate randomness by calling an effectful external function; so we give this function the ST effect to indicate that the computation is impure and stateful (even though it does not explicitly modify the memory). In particular, two calls to keygen may yield different results. The function encrypt would typically generate a nonce for use in the underlying AE construction, and hence is also marked as stateful. In contrast, decrypt is deterministic, so is marked with the Tot effect. Its result is an optional plain value: either Some p if decryption succeeds, or None otherwise. In pseudo-code we write $\perp$ for brevity.

Our interface does not express any security guarantees yet, but it does require a functional correctness guarantee, namely that decryption undoes encryption.

Authenticated Encryption: Security Given an AE scheme, one usually measures its concrete security as the advantage of an adversary $\mathcal{A}$ that attempts to guess the value of b in the following game:

$$\frac{\textbf{Game } \mathsf{Ae}(\mathcal{A}, \mathsf{AE})}{b \xleftarrow{\$} \{0,1\};\ L \leftarrow \varnothing;\ k \xleftarrow{\$} \mathsf{AE.keygen}()\\ b' \leftarrow \mathcal{A}^{\mathsf{Encrypt},\mathsf{Decrypt}}();\ \textbf{return } (b \stackrel{?}{=} b')}$$

$$\frac{\textbf{Oracle } \mathsf{Encrypt}(p)}{\textbf{if } b \textbf{ then } c \xleftarrow{\$} \mathsf{byte}^{\ell_c};\ L[c] \leftarrow p\\ \textbf{else } c \leftarrow \mathsf{AE.encrypt}\ k\ p\\ \textbf{return } c} \qquad \frac{\textbf{Oracle } \mathsf{Decrypt}(c)}{\textbf{if } b \textbf{ then } p \leftarrow L[c]\\ \textbf{else } p \leftarrow \mathsf{AE.decrypt}\ k\ c\\ \textbf{return } p}$$

The adversary $\mathcal{A}$ is a program that can call the two oracle functions to encrypt and decrypt using a secret key k. In the real case ($b = 0$) they just call the real AE implementation. In the ideal case ($b = 1$), Encrypt returns a randomly sampled ciphertext and stores the associated plaintext in a log L, while Decrypt performs decryption by looking up the plaintext in the log, returning $\perp$ when there is no plaintext associated with the ciphertext. Ideal AE is perfectly secure, inasmuch as the ciphertext does not depend on the plaintext. Thus, we define AE security by saying that the attacker cannot easily distinguish between the real and ideal cases.

For this game, we define $\mathcal{A}$'s advantage probabilistically as $|2\Pr[\mathsf{Ae}(\mathcal{A}, \mathsf{AE})] - 1|$, e.g. an adversary flipping a coin to guess b will succeed with probability $\frac{1}{2}$ and has 0 advantage.

In this paper, we adopt a more flexible notation for

indistinguishability games: we keep the sampling of b and the call to the adversary implicit, and instead indicate the oracles available to this adversary. Hence, we write the game above (with the same oracles) equivalently as

$$\frac{\textbf{Game } \mathsf{Ae}^b(\mathsf{AE})}{L \leftarrow \varnothing\ ;\ k \xleftarrow{\$} \mathsf{AE.keygen}();\ \textbf{return } \{\mathsf{Encrypt}, \mathsf{Decrypt}\}}$$

This notation facilitates the re-use of oracles for building other games, much like F⋆ modules. In general, we write G^b to refer to an indistinguishability game G where the adversary $\mathcal{A}$ tries to guess the value of the random bit b by calling the oracles returned by G. For all such games, we equivalently define the advantage as $\left|\Pr[\mathcal{A}^{\mathsf{G}^1} = 1] - \Pr[\mathcal{A}^{\mathsf{G}^0} = 1]\right|$.

Embedding games in F⋆ modules Although we wrote the game Ae^b above in pseudo-code, each game in this paper reflects a verified F⋆ module, written e.g. AE^b, that uses a boolean flag b to select between real and ideal implementations of the underlying cryptographic module AE. For example, AE^b may define the key type and encrypt function as

```
abstract type key = {key: AE.key; log: encryption_log}
let encrypt (k:key) (p:plain) =
  if b then
    let c = random_bytes ℓc in
    k.log ← k.log ++ (c,p);
    c
  else AE.encrypt k.key p
```

where the (private) key representation now includes both the real key and the ideal encryption log. The encrypt function uses k.log to access the current log, and ++ to append a new entry, much as the Encrypt oracle.

Idealization Interfaces The idealized module AE^b can be shown to implement the following typed interface that reflects the security guarantee of the Ae^b game:

```
abstract type key
val log: memory → key → Spec (seq (cipher × plain))
val keygen: unit → ST k:key
  (ensures b ⇒  log k′ = ∅)
val encrypt: k:key → p:plain → ST (c:cipher)
  (ensures b ⇒  log k′ = log k ++ (c,p))
val decrypt: k:key → c:cipher  → ST (o:option plain)
  (ensures b ⇒  o = lookup c (log k) ∧  log k' = log k)
```

The interface declares keys as abstract, hiding both the real key value and the ideal log, and relies on the log to specify the effects of encryption and decryption. To this end, it provides a log function that reads the current content of the log—a sequence of ciphertexts and plaintexts. This function is marked as Spec, indicating that it may be used *only in specification* and will be discarded by the compiler after typechecking.

Each of the 3 ensures clauses above uses this proof-only function to specify the state of the log before (log k) and after the call (log k′). Hence, the interface states that, in the ideal case, the function keygen creates a key with an empty log; encrypt k p returns a ciphertext c and extends the log for k with an entry mapping c to p; and decrypt k c returns exactly the result of looking up for c in the current log. This post-condition formally guarantees that decrypt succeeds if and only if it is passed a ciphertext that was generated by encrypt; in other words it guarantees both functional correctness and authentication (a notion similar to INT-CTXT).

AE^b is also parametrized by a module Plain^b that defines abstract plaintexts, with an interface that allows access to their concrete byte representation only when $b = 0$ (for real encryption). By typing AE^b, we verify that, when $b = 1$, our idealized functionality is independent (information-theoretically) from the values of the plaintexts it processes.

From the viewpoint of the application, the plaintext abstraction guarantees that AE^1 preserves the confidentiality and integrity of encrypted data (as in classic information flow type systems). An application can rely on this fact to prove application-level guarantees. For instance, an application may prove, as an invariant, that only well-formed messages are encrypted under a given key, and thus that parsing and processing a decrypted message always succeeds.

Probabilistic Semantics We model randomness (e.g. random_bytes) using primitive sampling functions. Two Boolean terminating F⋆ programs A^0 and A^1 are equivalent, written $A^0 \approx A^1$, when they return true with the same probability. They are ϵ-equivalent, noted $A^0 \approx_\epsilon A^1$, when $|\Pr[A^1 \Downarrow \mathsf{true}] - \Pr[A^0 \Downarrow \mathsf{true}]| \leq \epsilon$ where $\Pr[A \Downarrow v]$ denotes the probability that program A evaluates to value v according to the probabilistic semantics of F⋆. These definitions extend to program evaluation contexts, written $A^b[\_]$, in which case ϵ depends on the program plugged into the context, which intuitively stands for the adversary. Equipped with these definitions, we can develop code-based game-playing proofs following the well-established approach of Bellare and Rogaway [5] directly applied to F⋆ programs rather than pseudo-code. For example, we can reformulate AE security as $\mathsf{AE}^1[\mathcal{A}] \approx_\epsilon \mathsf{AE}^0[\mathcal{A}]$, where $\mathcal{A}$ now ranges over well-typed Boolean programs parameterized by the two functions encrypt and decrypt defined by AE^b. Our definition of ϵ-equivalence between real and ideal implementations of AE^b matches the definition of $\mathcal{A}$'s advantage in the Ae^b game.

Concrete security definitions and reductions As illustrated for AE below, our security definitions will consist of a game and a notation for the adversary advantage, parameterized by a measure of oracle use (e.g. how many times an adversary calls an oracle is called). We intend to provide concrete bounds on those advantages, as a function of their parameters. To this end, our reduction theorems will relate the advantage for a given construction to the advantages of its building blocks.

Definition 1 (AE-security): Given AE, let $\epsilon_{\mathsf{Ae}}(\mathcal{A}[q_e, q_d])$ be the advantage of an adversary $\mathcal{A}$ that makes q_e queries to Encrypt and q_d queries to Decrypt in the $\mathsf{Ae}^b(\mathsf{AE})$ game.

Equipped with this definition and our idealized interface for AE, we can prove the security of programs using ideal AE ($b = 1$), say with advantage ϵ, and then bound the advantage of the same programs using real AE ($b = 0$) to $\epsilon + \epsilon_{\mathsf{Ae}}(\mathcal{A}[q_e, q_d])$.

We can either assume that this definition holds for our real AE module with an ϵ_{Ae} that is small for realistic adversaries (possibly relying on functional correctness and some prior proof of security), or we can prove that our AES-GCM module (say) achieves some bound on ϵ_{Ae} by reduction to a simpler assumptions on the AES cipher module. In later sections, we will show how we can precisely compute the adversary $\mathcal{A}$'s advantage in the game above from a related adversary $\mathcal{B}$'s advantage in winning the PRF game on the underlying cipher (e.g. AES). The proof relies on standard cryptographic game transformations that are applied, on paper, at the level of F⋆ code, combined with functional correctness proofs about the real and ideal code, verified automatically by F⋆.

Games vs Idealized Modules We conclude this presentation of our approach by discussing differences between the games on paper and the modules of our implementation.

Standard-compliant modules include many details elided in informal games; they also use lower level representations to yield more efficient code, and require additional type annotations to keep track of memory management.

These modules are part of a general-purpose verified cryptographic libraries, providing real functionality (when idealizations flags are off) so they always support multiple instances of their functionality. Here for instance, AE^b has a function to generate keys, passed as parameters to the encrypt function, whereas the game oracle uses a single, implicit key. (This difference can usually be handled by a standard hybrid-argument reduction.)

Modules rely on the F⋆ type system to enforce the rules of the games. Hence, dynamic checks in games (say, to test whether a nonce has already been used) are often replaced with static pre-conditions on typed adversaries. Similarly, types enforce many important but trivial conditions, such as the length of oracle arguments, and are often kept implicit in the paper.

III. One-Time MACs

Anticipating on §V, the AEAD construction uses fresh key materials for each message, so we consider authentication when keys are used to compute at most one MAC.

We treat two main constructions, GHASH and Poly1305, using the same definitions, code, and proofs, inasmuch as possible. We initially suppose that the whole key is freshly generated for each MAC (as in ChaCha20-Poly1305) before presenting the general case where a part of the key is shared between multiple MACs (as in AES-GCM).

A. One-time MAC functionality and security

We outline below our interface for message authentication code (MAC), omitting its functional specification (see §IX).

```
val ℓ_{k0}: nat                    (* static key length, may be 0 *)
val ℓ_k: n:nat {ℓ_{k0} ≤ ℓ_k}      (* total key length *)
val ℓ_t: nat                       (* tag length *)
val ℓ_m: nat                       (* maximal message length *)
type key0 = lbytes ℓ_{k0}          (* static key shared between MACs *)
type key = lbytes ℓ_k              (* one−time key (including static key) *)
type tag = lbytes ℓ_t              (* authentication tag *)
type message = b:bbytes ℓ_b {wellformed b}
val keygen0: unit → ST key0
val keygen: key0 → ST key
val verify: key → message → tag → Tot bool
val mac: k:key → m:message → Tot (t:tag{verify k m t})
```

This interface defines concrete byte formats for keys, tags, and messages. Authenticated messages are strings of at most ℓ_m bytes that comply with an implementation-specific well-formedness condition. (We need such a condition for GHASH.) We let m range over well-formed messages.

Key-generation functions are marked as stateful (ST) to reflect their use of random sampling. Static keys of type key_0 may be used to generate multiple one-time keys of type key. (For example, keygen may concatenate the static key with $\ell_k - \ell_{k_0}$ random bytes.) To begin with, we assume $\ell_{k_0} = 0$ so that k_0 is the empty string ε.

The two main functions produce and verify MACs. Their correctness is captured in the verify post-condition of mac: verification succeeds at least on the tags correctly produced using mac with matching key and message.

One-Time Security MAC security is usually defined using computational unforgeability, as in the following game:

Game UF-1CMA($\mathcal{A}$, MAC)

$k \xleftarrow{\$} \mathsf{MAC.keygen}(\varepsilon)$; $log \leftarrow \bot$
$(m^\star, t^\star) \leftarrow \mathcal{A}^{\mathsf{Mac}}$
return $\mathsf{MAC.verify}(k, m^\star, t^\star)$
  $\wedge\ log \neq (m^\star, t^\star)$

Oracle Mac(m)

if $log \neq \bot$ **return** $\bot$
$t \leftarrow \mathsf{MAC.mac}(k, m)$
$log \leftarrow (m, t)$
return t

The oracle permits the adversary a single chosen-message query (recorded in *log*) before trying to produce a forgery. The advantage of $\mathcal{A}$ playing the UF-1CMA game is defined as $\epsilon_{\mathsf{UF\text{-}1CMA}}(\mathcal{A}[\ell_m]) \triangleq \Pr[\mathsf{UF\text{-}1CMA}(\mathcal{A}, \mathsf{MAC}) = 1]$.

We seek a stronger property for AEAD—the whole ciphertext must be indistinguishable from random bytes—and we need a decisional game for type-based composition, so we introduce a variant of unforgeability that captures indistinguishability from a random tag (when r is set).

Definition 2 (IND-UF-1CMA): Let $\epsilon_{\mathsf{Mac1}}(\mathcal{A}[\ell_m, q_v])$ be the advantage of an adversary $\mathcal{A}$ that makes q_v Verify queries on messages of length at most ℓ_m in the following game:

Game $\mathsf{Mac1}^b(\mathsf{MAC})$

$k \xleftarrow{\$} \mathsf{MAC.keygen}(\varepsilon)$; $log \leftarrow \bot$
return {Mac, Verify}

Oracle $\mathsf{Verify}(m^\star, t^\star)$

if b **return** $log = (m^\star, t^\star)$
return $\mathsf{MAC.verify}(k, m^\star, t^\star)$

Oracle $\mathsf{Mac}(m)$

if $log \neq \bot$ **return** $\bot$
$t \leftarrow \mathsf{MAC.mac}(k, m)$
if $b \wedge r$
  $t \xleftarrow{\$} \mathsf{byte}^{\mathsf{MAC}.\ell_t}$
$log \leftarrow (m, t)$
return t

In this game, the MAC oracle is called at most once, on some chosen message m; it returns a tag t and logs (m, t). Conversely, Verify is called q_v times before and after calling MAC. When b is set, the game idealizes MAC in two ways: verification is replaced by a comparison with the log; and (when r is also set) the tag is replaced with random bytes.

We show (in the full paper) that our definition implies UF-1CMA when $q_v \geq 1$ and that random tags are neither necessary nor sufficient for unforgeability. We are not aware of much prior work on Mac1 with r set; a pairwise independent hash function would satisfy our IND-UF-1CMA definition but may require longer keys [42].

Multi-Instance Security with a Shared Key In the AEAD construction, we instantiate a one-time MAC for every encryption and decryption. AES-GCM uses a static MAC key derived from the AEAD key and shared between all MAC instances. This state sharing is not captured by the games above. To this end, we extend the $\mathsf{Mac1}^b$ game into a multi-instance version $\mathsf{MMac1}^b$ with a setup that invokes the keygen_0 function to generate any key materials reused across instances.

In the multi-instance case it is convenient to support two kinds of instances: honest instances are created with Keygen and idealized as in $\mathsf{Mac1}^b$; dishonest instances are created with Coerce and use the real implementation regardless of b. (Formally Coerce does not make the model stronger, as an adversary can run all algorithms on his own. However, the finer model is useful in hybrid arguments and for composition with a PRF in §IV.)

Definition 3 (m-IND-UF-1CMA):
Let $\epsilon_{\mathsf{MMac1}}(\mathcal{A}[\ell_m, q_v, q_i])$ be the advantage of an adversary $\mathcal{A}$ that creates q_i instances and makes q_v Verify queries overall on messages of length at most ℓ_m in the game:

Game $\mathsf{MMac1}^b(\mathsf{MAC})$

$log \leftarrow \varnothing$; $k \leftarrow \varnothing$; $H \leftarrow \varnothing$
$k_0 \xleftarrow{\$} \mathsf{MAC.keygen}_0()$
return {MAC, Verify, Coerce, Keygen}

Oracle $\mathsf{Mac}(n, m)$

if $k[n] = \bot$ **return** $\bot$
if $log[n] \neq \bot$ **return** $\bot$
$t \leftarrow \mathsf{MAC.mac}(k[n], m)$
if $b \wedge n \in H$
  $t \xleftarrow{\$} \mathsf{byte}^{\mathsf{MAC}.\ell_t}$
$log[n] \leftarrow (m, t)$
return t

Oracle $\mathsf{Keygen}(n)$

if $k[n] \neq \bot$ **return** $\bot$
$k[n] \leftarrow \mathsf{MAC.keygen}(k_0)$
$H \leftarrow H \cup n$

Oracle $\mathsf{Coerce}(n, k)$

if $k[n] \neq \bot$ **return** $\bot$
$k[n] \leftarrow k$

Oracle $\mathsf{Verify}(n, m, t)$

if $k[n] = \bot$ **return** $\bot$
$v \leftarrow \mathsf{MAC.verify}(k[n], m, t)$
if $b \wedge n \in H$
  $v \leftarrow log[n] = (m, t)$
return v

We confirm that Mac1 is a special case of MMac1 security and that, even with a static key, it suffices to consider a single verification query. (The proofs are in the full paper.)

*Lemma 1 (*MMac1 *reduces to* Mac1*):* Given $\mathcal{A}$ against $\mathsf{MMac1}^b$, when $\ell_{k_0} = 0$, we construct $\mathcal{B}$ against $\mathsf{Mac1}^b$ (linearly in q_i) such that: $\epsilon_{\mathsf{MMac1}}(\mathcal{A}[\ell_m, q_v, q_i]) \leq q_i\ \epsilon_{\mathsf{Mac1}}(\mathcal{B}[\ell_m, q_v])$.

Lemma 2: Given $\mathcal{A}$ against $\mathsf{MMac1}^b$ we construct $\mathcal{B}$ such that: $\epsilon_{\mathsf{MMac1}}(\mathcal{A}[\ell_m, q_v, q_i]) \leq q_v\ \epsilon_{\mathsf{MMac1}}(\mathcal{B}[\ell_m, 1, q_i])$.

Verified Implementation m-IND-UF-1CMA security reflects the type-based security specification of our idealized module $\mathsf{MMac1}^b$, which has an interface of the form

```
val log: memory → key → Spec (option (message × tag))
val mac: k:key → m:message → ST (t:tag)
  (requires log k = None)
  (ensures log k′ = Some(m,t))
val verify: k:key → m:message → t:tag → ST (v:bool)
  (ensures b ⇒ v = (log k′ = Some(m,t)))
```

The types of mac and verify express the gist of our security property: the specification function log gives access to the current content of the log associated with a one-time key; mac requires that the log be empty (None in F*) thereby enforcing our one-time MAC discipline; verify ensures that, when b is set, verification succeeds if and only if mac logged exactly the same message and tag. Their implementation is automatically verified by typing $\mathsf{MMac1}^b$. However, recall that typing says nothing about the security loss incurred by switching b—this is the subject of the next subsection.

Our verified implementation of $\mathsf{MMac1}^b$ supports the two constructions described next, including code and functional correctness proofs for their algorithms. It also provides a more efficient interface for computing MACs incrementally. Instead of actually concatenating all authenticated materials in a message, the user creates a stateful hash, then repeatedly appends 16-byte words to the hash, and finally calls mac or verify on this hash, with a type that binds the message to the final hash contents in their security specifications. Our code further relies on indexed abstract types to separate keys and hashes for different instances of the functionality, and to support static key compromise.

B. Wegman-Carter-Shoup (WCS) Constructions

Next, we set up notations so that our presentation applies to multiple constructions, including GHASH and Poly1305; we factor out the encodings to have a core security assumption on sequences of field elements; we verify their injectivity; we finally prove concrete bounds in general, and in particular for GHASH and Poly1305.

From bytes to polynomials and back In addition to fixed lengths for keys and tags, the construction is parameterized by

- a field $\mathbb{F}$;

- an encoding function $\overline{\cdot}$ from messages to polynomials in $\mathbb{F}$, represented as sequences of coefficients $\overline{m} \in \mathbb{F}^*$.
- a truncation function from $e \in \mathbb{F}$ to $\mathsf{tag}(e) \in \mathsf{byte}^{\ell_t}$;

The key consists of two parts: an element $r \in \mathbb{F}$ and a one-time pad $s \in \mathsf{byte}^{\ell_t}$. We assume that r and s are sampled uniformly at random, from some $R \subseteq \mathbb{F}$ and from byte^{ℓ_t}, respectively. We write $r \| s \leftarrow k$ for the parsing of key materials into r and s, including the encoding of r into R.

Generic Construction Given a message m encoded into the sequence of d coefficients $\overline{m}_0, \ldots, \overline{m}_{d-1}$ of a polynomial $\overline{m}(x) = \sum_{i=1..d} \overline{m}_{d-i} x^i$ in $\mathbb{F}$, the tag is computed as:

$$\begin{array}{rl} \mathsf{hash}_r(m) \leftarrow \mathsf{tag}(\overline{m}(r)) & \text{in } \mathbb{F} \text{ before truncation} \\ \mathsf{mac}(r\|s, m) \leftarrow \mathsf{hash}_r(m) \boxplus s & \text{in } \mathsf{byte}^{\ell_t} \end{array}$$

where the blinding operation $\boxplus$ is related to addition in $\mathbb{F}$ (see specific details below). We refer to $\mathsf{hash}_r(m)$, the part of the construction before blinding, as the hash.

Next, we describe the two instantiations employed in TLS.

GHASH [21] uses the Galois field $GF(2^{128})$, defined as the extension $GF(2)[x]/x^{128}+x^7+x^2+x+1$, that is, the field of polynomials with Boolean coefficients modulo the irreducible polynomial $x^{128}+x^7+x^2+x+1$. Such polynomials are represented as 128-bit vectors. Conveniently, polynomial addition, the blinding operation $\boxplus$, and its inverse $\boxminus$ simply correspond to 128-bit XOR. Polynomial multiplication is also efficiently supported on modern processors. The message encoding $\overline{\cdot}$ splits the input message into 16-byte words, seen as integers in $0..2^{128}-1$; and the tag truncation is the identity. For AES-GCM, GHASH has a keygen0 function that samples a single $r \xleftarrow{\$} GF(2^{128})$ shared across all MAC instances.

Poly1305 [8] uses the field $GF(p)$ for $p = 2^{130} - 5$, that is, the prime field of integer addition and multiplication modulo p, whose elements can all be represented as 130-bits integers. Its message encoding $\overline{\cdot}$ similarly splits the input message into 16-byte words, seen as integers in $0..2^{128}-1$, then adds 2^ℓ to each of these integers, where ℓ is the word length in bits. (Hence, the encoding precisely keeps track of the length of the last word; this feature is unused for AEAD, which applies its own padding to ensure $\ell = 128$.) The truncation function is $\mathsf{tag}(e) = e \mod 2^{128}$. The blinding operation $\boxplus$ and its inverse $\boxminus$ are addition and subtraction modulo 2^{128}. For ChaCha20-Poly1305, both r and s are single-use ($\ell_{k_0} = 0$) but our proof also applies to the original Poly1305-AES construction [8] where r is shared.

Injectivity Properties We intend to authenticate messages, not just polynomial coefficients. To this end, we instantiate our wellformed predicate on messages and show (in F*) that

$\forall$ ($m0$: bytes) ($m1$: bytes).
(wellformed $m0$ $\wedge$ wellformed $m1$ $\wedge$
Poly.equals $\overline{m0}$ $\overline{m1}$) $\Rightarrow$ $m0 = m1$

where Poly.equals specifies the equality of two formal polynomials by comparing their sequences of coefficients, extending the shorter sequence with zero coefficients if necessary. This enables the (conditional) composition of MACs with suitable well-formedness predicates for AEAD in TLS. This is required for GHASH as it is otherwise subject to 0-message truncations.

We verify that the property above suffices to prove that both encodings are secure, and also that it holds in particular once we define wellformed as the range of formatted messages for AEAD (which are 16-byte aligned and embed their own lengths; see §V). We also confirm by typing that, with Poly1305, there is no need to restrict messages: its encoding is injective for all bytestrings [8, Theorem 3.2].

Security We give a theorem similar to those in prior work [8, 30, 43] but parameterized by the underlying field $\mathbb{F}$, encoding $\overline{\cdot}$, truncation tag, and blinding operation $\boxplus$. The theorem covers all uses of AES-GCM and ChaCha20-Poly1305 in TLS.

Consider the MMac1 definition, covering both shared and fresh values for r. Let q_v be the number of oracle calls to Verify (for which $log[n] \neq (m^\star, t^\star)$) and d a bound on the size (expressed in number of field elements) of the messages in calls to Mac and Verify.

Theorem 1: The Wegman-Carter-Shoup construction for messages in $\mathbb{F}^{d-1}$ is m-IND-UF-1CMA secure with concrete bound $\epsilon_{\mathsf{MMac1}}(\mathcal{A}[\ell_m, q_v, q_i]) = \frac{d.\tau.q_v}{|R|}$ with $d = \ell_m/16$, and $\tau = 1$ for GHASH and $\tau = 8$ for Poly1305.

The proof (in the full paper) uses Lemma 2 and establishes a bound $\frac{d\cdot\tau}{|R|}$ for an adversary that makes a single Verify query. This bound follows from an $\frac{d\cdot\tau}{|R|}$-almost-$\boxminus$-universal property, which has been separately proved for GHASH [35] and Poly1305 [8]; the full paperalso includes its proof for all instantiations of hash_r for TLS.

Concrete bounds for GHASH: The range size for r is 2^{128} and there is no tag truncation, hence by Lemma 2 we get a straight $\epsilon = \frac{d\cdot q_v}{2^{128}}$, so for TLS the main risk is a failure of our PRF assumption on AES, discussed in §VII.

Concrete bound for Poly1305: The effective range R of r is reduced, first by uniformly sampling in $0..2^{128}-1$, then by clamping 22 bits, to uniformly sampling one element out of $|\ R\ | = 2^{106}$ potential values. We lose another 3 bits of security from the truncation of $\mathbb{F}$ to byte^{ℓ_t} and by applying Lemma 2 we arrive at $\epsilon = \frac{d\cdot q_v}{2^{103}}$.

IV. Pseudo-Random Functions for AEAD

We now consider the use of symmetric ciphers in counter mode, both for keying one-time MACs and for generating one-time pads for encryption. We model ciphers as PRFs. For TLS, we will use AES or ChaCha20, and discuss PRF/PRP issues in §VII. A pseudo-random function family

PRF implements the following interface:

```
type key
val keygen: unit → ST key
val ℓ_d : nat (* fixed domain length *)
val ℓ_b : nat (* fixed block length *)
type domain = lbytes ℓ_d
type block = lbytes ℓ_b
val eval: key → domain → Tot block (* functional specification *)
```

This interface specifies an abstract type for keys and a key-generation algorithm. (Type abstraction ensures that these keys are used only for PRF computations.) It also specifies concrete, fixed-length bytestrings for the domain and range of the PRF, and a function to compute the PRF. We refer to the PRF outputs as blocks. As usual, we define security as indistinguishability from a uniformly random function with lazy sampling.

Definition 4 (PRF security): Let $\epsilon_{\mathsf{Prf}}(\mathcal{A}[q_b])$ be the advantage of an adversary $\mathcal{A}$ that makes q_b Eval queries in the game:

Game $\mathsf{Prf}^b(\mathsf{PRF})$	**Oracle** $\mathsf{Eval}(m)$
$T \leftarrow \varnothing$	**if** $T[m] = \bot$
$k \xleftarrow{\$} \mathsf{PRF.keygen}()$	**if** b **then** $T[m] \xleftarrow{\$} \mathsf{byte}^{\ell_b}$
return {Eval}	**else** $T[m] \leftarrow \mathsf{PRF.eval}(k, m)$
	return $T[m]$

The AEAD constructions we consider use PRFs both to generate keys for the one-time MAC used to authenticate the ciphertext and to generate a one-time pad for encryption and decryption. Accordingly, we partition the domain and use a specialized security definition, with a separate eval function for each usage of the PRF. (This will enable us to give more precise types to each of these functions.)

We assume the PRF domain consists of concatenations of a fixed-sized counter j and a nonce n, written $j\|n$. This notation hides minor differences between AEAD algorithm specifications, e.g. AES-GCM uses $n\|j$ instead $j\|n$. Our implementation handles these details, and verifies that $j\|n$ is injective for all admissible values of j and n.

For key generation, AES-GCM uses the PRF to derive both a static MAC key k_0 generated from the PRF (with nonce and counter 0) and a 1-time MAC key for each nonce (with counter 1) whereas Poly1305 uses a pure 1-time MAC key for each nonce (with counter 0). To handle both cases uniformly, we introduce a parameter $j_0 \in \{0, 1\}$ to shift the counter before concatenation with the nonce. In the following, we assume a *compatible* MAC, meaning that either $j_0 = 0 \wedge \ell_{k_0} = 0 \wedge \ell_k \leq \ell_b$ or $j_0 = 1 \wedge \ell_{k_0} \leq \ell_b \wedge \ell_k - \ell_{k_0} \leq \ell_b$.

For pad generation, counter mode encrypts plaintext blocks as $p \oplus \mathsf{eval}(j\|n)$ and decrypts by applying the same pad to the ciphertext. In the PrfCtr game below, we separate encryption and decryption, and we fuse the block generation and the XOR, so that we can give separate types to plaintexts and ciphertexts. (We truncate the block in case it is smaller than the input, as required for the last block in counter-mode.)

*Definition 5 (*PrfCtr *security):* Given PRF and MAC, let $\epsilon_{\mathsf{PrfCtr}}(\mathcal{A}[q_b, q_g])$ be the advantage of an adversary $\mathcal{A}$ that makes q_b queries to either EvalEnx or EvalDex and q_g queries to EvalKey in the following game:

Game $\mathsf{PrfCtr}^b(\mathsf{PRF}, \mathsf{MAC})$	**Oracle** $\mathsf{EvalKey}(j\|n)$
$T \leftarrow \varnothing;\ R \leftarrow \varnothing$	**if** $j \neq j_0$ **return** $\bot$
$k \xleftarrow{\$} \mathsf{PRF.keygen}()$	**if** $T[j\|n] = \bot$
$k_0 \xleftarrow{\$} \mathsf{MAC.keygen0}()$	**if** b
if $j_0 \wedge \neg b$	$k_m \xleftarrow{\$} \mathsf{MAC.keygen}(k_0)$
$o \leftarrow \mathsf{PRF.eval}(k, 0^{\ell_b})$	**else**
$k_0 \leftarrow \mathsf{truncate}(o, \mathsf{MAC}.\ell_{k_0})$	$o \leftarrow \mathsf{PRF.eval}(k, j\|n)$
return {EvalKey, EvalEnx, EvalDex}	$k_m \leftarrow \mathsf{truncate}(k_0\|o, \ell_k)$
	$T[j\|n] \leftarrow k_m$
	return $T[j\|n]$

Oracle $\mathsf{EvalEnx}(j\|n, p)$	**Oracle** $\mathsf{EvalDex}(j\|n, c)$
if $j \leq j_0$ **return** $\bot$	**if** $j \leq j_0$ **return** $\bot$
$o \xleftarrow{\$} \mathsf{Eval}(j\|n)$	$o \xleftarrow{\$} \mathsf{Eval}(j\|n)$
$c \leftarrow p \oplus \mathsf{truncate}(o, \|p\|)$	$p \leftarrow c \oplus \mathsf{truncate}(o, \|c\|)$
return c	**return** p

Lemma 3 (PrfCtr^b *reduces to* Prf^b*):* Given PRF, MAC, and $\mathcal{A}$ against $\mathsf{PrfCtr}^b(\mathsf{PRF}, \mathsf{MAC})$, we construct $\mathcal{B}$ against $\mathsf{Prf}^b(\mathsf{PRF})$ such that:

$$\epsilon_{\mathsf{PrfCtr}}(\mathcal{A}[q_b, q_g]) = \epsilon_{\mathsf{Prf}}(\mathcal{B}[q_b + q_g + j_0]).$$

The proof is in the full paper. Intuitively, we have a perfect reduction because, in all cases, the specialized game still samples a single fresh block for each $j\|n$ for a single purpose, and returns a value computed from that block.

In the next section, once b holds and the MAC has been idealized, we will use two oracles that further idealize encryption and decryption:

Oracle $\mathsf{EvalEnx}'(j\|n, p)$	**Oracle** $\mathsf{EvalDex}'(j\|n, c)$
if $j \leq j_0$ **return** $\bot$	**if** $j \leq j_0$ **return** $\bot$
if $T[j\|n] \neq \bot$ **return** $\bot$	**if** $T[j\|n] = (p, c)$ for some p
if b' $c \xleftarrow{\$} \mathsf{byte}^{\|p\|}$	**return** p
else $c \xleftarrow{\$} \mathsf{EvalEnx}(j\|n, p)$	**else return** $\bot$
$T[j\|n] \leftarrow (p, c)$	
return c	

When b' holds, encryption samples c instead of $o = p \oplus c$, and records the pair (p, c) instead of just $p \oplus c$; and decryption simply performs a table lookup. This step is valid provided the block at $j\|n$ is used for encrypting a single p and decrypting the resulting c. The oracles enforce this restriction dynamically (on their second lines) whereas our code enforces it statically, using type-based preconditions on EvalEnx or EvalDex implied by the AEAD invariant of §V.

Verified Implementation Lemma 3 and the subsequent step are not currently verified by typing. (Still, note that the sampling of c instead of o is justified by F*'s probabilistic semantic and could be verified using the relational typing rule for sample in RF* [4])

We use an idealized PRF module with two idealization flags (for b and for b') that directly corresponds to the specialized game $\mathsf{PrfCtr}^{b,b'}$ parametrized by a Cipher module that implements real AES128, AES256, and ChaCha20 (depending on an algorithmic parameter alg) and by a MAC module. The separation of the PRF domain is enforced by typing: depending on alg, j_0, j, b, and b', its range includes keys, blocks, and pairs (p, c).

V. From MAC and PRF to AEAD

We implement the two main AEAD constructions used by TLS 1.3 and modern ciphersuites of TLS 1.2. We show that their composition of a PRF and a one-time MAC yields a standard notion of AEAD security. Our proof is generic and carefully designed to be modular and TLS-agnostic: we share our AEAD code between TLS 1.2 and 1.3, and plan to generalize it for other protocols such as QUIC.

AEAD functionality Our authenticated encryption with associated data (AEAD) has a real interface of the form

```
val ℓn: nat                    (* fixed nonce length *)
val ℓa: n:nat{n < 2^32}        (* maximal AD length *)
val ℓp: n:nat{n < 2^32}        (* maximal plaintext length *)
val cipherlen: n:nat{n ≤ ℓp} → Tot nat
type nonce = lbytes ℓn
type ad = bbytes ℓa
type plain = bbytes ℓp
type cipher = bytes

val decrypt: key → nonce → ad → c:cipher →
  ST (option (p:plain{length c = cipherlen (length p)}))
val encrypt: k:key → n:nonce → a:ad → p:plain →
  ST (c:cipher{length c = cipherlen (length p)})
```

with two functions to encrypt and decrypt messages with associated data of variable lengths, and types that specify the cipher length as a function of the plain length. (We omit declarations for keys, similar to those for PRFs in §IV.)

*Definition 6 (*Aead *security):* Let $\epsilon_{\mathsf{Aead}}(\mathcal{A}[q_e, q_d, \ell_p, \ell_a])$ be the advantage of an adversary that makes at most q_e Encrypt and q_d Decrypt queries on messages and associated data of lengths at most ℓ_p and ℓ_a in the game:

Game $\mathsf{Aead}^b(\mathsf{AEAD})$
$C \leftarrow \varnothing$
$k \xleftarrow{\$} \mathsf{AEAD.keygen}()$
return $\{\mathsf{Encrypt}, \mathsf{Decrypt}\}$

Oracle $\mathsf{Encrypt}(n, a, p)$
if $C[n] \neq \bot$ **return** $\bot$
if b $c \xleftarrow{\$} \mathsf{byte}^{\mathsf{cipherlen}(|p|)}$
else $c \leftarrow \mathsf{AEAD.encrypt}(k, n, a, p)$
$C[n] \leftarrow (a, p, c)$
return c

Oracle $\mathsf{Decrypt}(n, a, c)$
if b
  if $C[n] = (a, p, c)$ *for some* p
    return p
  return $\bot$
else
  $p \leftarrow \mathsf{AEAD.decrypt}(k, n, a, c)$
  return p

Our definition generalizes AE in §II; it has a richer domain with plaintext and associated data of variable lengths; a function cipherlen from plaintext lengths to ciphertext lengths; and nonces n. It similarly maintains a log of encryptions, indexed by nonces. Crucially, Encrypt uses the log to ensure that each nonce is used at most once for encryption.

Generic AEAD Construction Given a PRF and a compatible MAC, AEAD splits plaintexts into blocks which are then blinded by pseudo-random one-time pads generated by calling PRF on increasing counter values, as shown in §IV. (Blocks for MAC keys and the last mask may require truncation.)

To authenticate the ciphertext and associated data, the construction formats them into a single 16-byte-aligned buffer (ready to be hashed as polynomial coefficients as described in §III) using an encoding function declared as val encode: bbytes ℓ_p × bbytes ℓ_a → Tot bbytes $(\ell_p + \ell_a + 46)$ and implemented (in pseudo-code) as

Function $\mathsf{encode}(c, a)$
return $\mathsf{pad}_{16}(a) \,\|\, \mathsf{pad}_{16}(c)$
  $\|\, \mathsf{length}_8(a) \,\|\, \mathsf{length}_8(c)$

Function $\mathsf{pad}_{16}(b)$
$r, b_1, \ldots, b_r \leftarrow \mathsf{split}_{16}(b)$
return $b \,\|\, \mathsf{zeros}(16 - |b_r|)$

where the auxiliary function $\mathsf{split}_\ell(b)$ splits the bytestring b into a sequence of r non-empty bytestrings, all of size ℓ, except for the last one which may be shorter. (that is, if $r, b_1, \ldots b_r \leftarrow \mathsf{split}_b(\ell)$, then $b = b_1 \,\|\, \cdots \,\|\, b_r$.); where $\mathsf{zeros}(\ell)$ is the bytestring of ℓ zero bytes; and where $\mathsf{length}_8(n)$ is the 8-byte representation of the length of n. Thus, our encoding adds minimal zero-padding to a and c, so that they are both 16-bytes aligned, and appends a final 16-byte encoding of their lengths.

Recall that the domain of MAC messages is restricted by the wellformed predicate. We now define wellformed b = ∃ (c:cipher) (a:ad). b = encode c a and typecheck the property listed in §III that ensures injectivity of the polynomial encoding.

The rest of the AEAD construction is defined below, using an operator $otp \,\overline{\oplus}\, p$ that abbreviates the expression $\mathsf{truncate}(otp, |p|) \oplus p$, and a function untag_{16} that separates the ciphertext from the tag.

Function $\mathsf{keygen}()$
$k \xleftarrow{\$} \mathsf{PRF.keygen}()$; $k_0 \leftarrow \varepsilon$
if j_0
  $o \leftarrow \mathsf{PRF.eval}(k, 0^{\ell_b})$
  $k_0 \leftarrow \mathsf{truncate}(o, \mathsf{MAC}.\ell_{k_0})$
return $k_0 \| k$

Function $\mathsf{encrypt}(K, n, a, p)$
$(k_0, k) \leftarrow \mathsf{split}_{\ell_{k_0}}(K)$; $c \leftarrow \varepsilon$
$k_1 \leftarrow \mathsf{PRF.eval}(k, j_0 \| n)$
$k_m \leftarrow \mathsf{truncate}(k_0 \| k_1, \mathsf{MAC}.\ell_k)$
$r, p_1, \ldots, p_r \leftarrow \mathsf{split}_{\ell_b}(p)$;
for $j = 1..r$
  $otp \leftarrow \mathsf{PRF.eval}(k, j_0 + j \| n)$
  $c \leftarrow c \,\|\, (otp \,\overline{\oplus}\, p_j)$
$t \leftarrow \mathsf{MAC.mac}(k_m, \mathsf{encode}(c, a))$
return $c \| t$

Function $\mathsf{decrypt}(K, n, a, c)$
$(k_0, k) \leftarrow \mathsf{split}_{\ell_{k_0}}(K)$; $p \leftarrow \varepsilon$
$k_1 \leftarrow \mathsf{PRF.eval}(k, j_0 \| n)$
$k_m \leftarrow \mathsf{truncate}(k_0 \| k_1, \mathsf{MAC}.\ell_k)$
$(c, t) \leftarrow \mathsf{untag}_{16}(c)$
$m \leftarrow \mathsf{encode}(c, a)$
if $\neg \mathsf{MAC.verify}(k_m, m, t)$
  return $\bot$
$r, c_1, \ldots, c_r \leftarrow \mathsf{split}_{\ell_b}(c)$;
for $j = 1..r$
  $otp \leftarrow \mathsf{PRF.eval}(k, j_0 + j \| n)$
  $p \leftarrow p \,\|\, (otp \,\overline{\oplus}\, c_j)$
return p

The main result of this section is that it is Aead-secure when PRF is Prf-secure and MAC is MMac1-secure:

Theorem 2 (AEAD construction): Given $\mathcal{A}$ against Aead, we construct $\mathcal{B}$ against Prf and $\mathcal{C}$ against MMac1, with:

$$\epsilon_{\mathsf{Aead(AEAD)}}(\mathcal{A}[q_e, q_d, \ell_p, \ell_a]) \leq \epsilon_{\mathsf{Prf(PRF)}}(\mathcal{B}[q_b]) + \epsilon_{\mathsf{MMac1(MAC)}}(\mathcal{C}[\ell_p + \ell_a + 46, q_d, q_e + q_d])$$

where q_b (the number of distinct queries to the PRF) satisfies:

$$q_b \leq j_0 + q_e \left(1 + \left\lceil \frac{\ell_p}{\ell_b} \right\rceil\right) + q_d$$

Proof sketch: The proof is in the full paper; it relies on the $\mathsf{PrfCtr}^{b,b'}$ and $\mathsf{MMac1}^b$ idealizations; it involves a sequence of transformations from Aead^0 to Aead^1 that inline successively more idealizations. Therefore, we introduce a parametric game $\mathsf{AeadCtr}(X)$ for any game X that returns EvalKey, EvalEnx, EvalDex, Mac, and Verify oracles:

Game $\mathsf{AeadCtr}(X)$
$(\mathsf{EvalKey}, \mathsf{EvalEnx}, \mathsf{EvalDex}, \mathsf{Mac}, \mathsf{Verify}) \leftarrow \mathsf{X}()$
return {Encrypt, Decrypt}

Oracle $\mathsf{Encrypt}(n, a, p)$
if $C[n] \neq \perp$ **return** $\perp$
$\mathsf{EvalKey}(n)$; $c \leftarrow \varepsilon$
$r, p_1, \ldots, p_r \leftarrow \mathsf{split}_{\ell_b}(p)$
for $j = 1..r$
  $c \leftarrow c \,\|\, \mathsf{EvalEnx}(j_0 + j \| n, p_j)$
$c \leftarrow c \,\|\, \mathsf{Mac}(n, \mathsf{encode}(c, a))$
$C[n] \leftarrow (a, p, c)$
return c

Oracle $\mathsf{Decrypt}(n, a, c)$
$c, t \leftarrow \mathsf{untag}_{16}(c)$
$\mathsf{EvalKey}(n)$
if $\neg\mathsf{Verify}(n, \mathsf{encode}(c, a), t)$
  return $\perp$
$r, c_1, \ldots, c_r \leftarrow \mathsf{split}_{\ell_b}(c)$; $p \leftarrow \varepsilon$
for $j = 1 \ldots r$
  $p \leftarrow p \,\|\, \mathsf{EvalDex}(j_0 + j \| n, c_j)$
return p

When X is obtained from PrfCtr^0 and $\mathsf{MMac1}^0$ we have a game that is equivalent to Aead^0. We first switch to PrfCtr^1 to get random MAC keys and then idealize $\mathsf{MMac1}^1$. When X is obtained from PrfCtr^1 and $\mathsf{MMac1}^1$ ciphertexts are authenticated and we can switch to $\mathsf{PrfCtr}^{1,0}$ and then to $\mathsf{PrfCtr}^{1,1}$. At this stage the PRF table contains randomly sampled ciphertext blocks and decryption corresponds to table lookup in this table. This is ensured on the code by our AEAD invariant. ■

Verified Implementation We outline below the idealized interface of our main AEAD^b module built on top of (the idealized interfaces of) $\mathsf{PrfCtr}^{b,b'}$ and $\mathsf{MMac1}^b$, both taken as cryptographic assumption, and documented by the games with the same names on paper. We focus on types for encryption and decryption:

```
abstract type key (* stateful key, now containing the log *)
val log: memory → key →
  Spec (seq (nonce × ad × cipher × plain)
val keygen : unit → ST (k:key)
  (ensures b ⇒ log k = ∅)
val encrypt: k:key → n:nonce → a:ad → p:plain → ST (c:cipher)
  (requires b ⇒ lookup_nonce n (log k) = None)
  (ensures (b ⇒ log k′ = log k ++ (n,a,c,p)))
val decrypt: k:key → n:nonce → a:ad → c:cipher →
  ST (o:option plain)
  (ensures b ⇒ o = lookup (n,a,c) (log k))
```

As in §II, we have a multi-instance idealization, with a log for each instance stored within an abstract, stateful key; and we provide a proof-only function log to access its current contents in logical specifications. Hence, key generation allocates an empty log for the instance; encryption requires that the nonce be fresh and records its results; and decryption behaves exactly as a table lookup, returning a plaintext if, and only if, it was previously stored in the log by calling encryption with the same nonce and additional data.

This step of the construction is entirely verifiable by typing. To this end, we supplement its implementation with a precise invariant that relates the AEAD log to the underlying PRF table and MAC logs. For each entry in the log, we specify the corresponding entries in the PRF table (one for the one-time MAC key, and one for each block required for encryption) and, for each one-time MAC key entry, the contents of the MAC log (an encoded message and the tag at the end of the ciphertext in the AEAD log entry). By typing the AEAD code that implements the construction, we verify that the invariant is preserved as it completes its series of calls to the PRF and MAC idealized interfaces. Hence, although our code for decryption does *not* actually decrypt by a log lookup, we prove that (when b holds) its results always matches the result of a lookup on the current log. As usual, by setting all idealization flags to false, the verified code yields our concrete TLS implementation.

Security bounds Theorem 2 can be specialized to provide precise security bounds for the various AEAD ciphersuites:

Construction	$\epsilon_{\mathsf{Aead}}(\mathcal{A}[q_e, q_d, \ell_p, \ell_a]) \leq$
AES128-GCM, AES256-GCM	$\epsilon_{\mathsf{Prf}}\left(\mathcal{B}\left[q_e\left(1 + \frac{\ell_p}{16}\right) + q_d + 1\right]\right) + \frac{q_d}{2^{128}} \cdot \left(\frac{\ell_p + \ell_a + 46}{16}\right)$
ChaCha20-Poly1305	$\epsilon_{\mathsf{Prf}}\left(\mathcal{B}\left[q_e\left(1 + \frac{\ell_p}{64}\right) + q_d\right]\right) + \frac{q_d}{2^{103}} \cdot \left(\frac{\ell_p + \ell_a + 46}{16}\right)$

VI. From AEAD to Stream Encryption (StAE)

TLS requires stream encryption: message fragments must be received and processed in the order they were sent, thereby defeating attempts to delete or re-order network traffic. To this end, encryption and decryption use a local sequence number to generate distinct, ordered nonces for AEAD.

In practice, it is difficult to prevent multiple honest servers from decrypting and processing the same 0-RTT encrypted stream. Since decryption is now stateful, we must generalize our model to support multiple parallel decryptors for each encryptor. In our security definitions, we thus add a genD oracle to generate new decryptors (with local sequence numbers set to zero) from a given encryptor.

Otherwise, the stateful construction is quite simple: TLS 1.3 combines the sequence number with a static, random 'initialization vector' (IV) in the key materials to generate pairwise-distinct nonces for encrypting fragments using AEAD. In contrast, TLS 1.2 nonces concatenate the static IV with a per-fragment explicit IV that is sent alongside the ciphertext on the network (except for ciphersuites based on ChaCha20-Poly1305 which follow the TLS 1.3 nonce format). Some TLS 1.2 implementations incorrectly use uniformly random explicit IVs [15]. This is much inferior to using the sequence number because of the high collision risk on 64 bits. Therefore, in our implementation, we use the following nonce construction:

$$n = \begin{cases} bigEndian_8(seqn) \,\|\, iv_4 & \text{for AES-GCM in TLS 1.2} \\ bigEndian_{12}(seqn) \oplus iv_{12} & \text{otherwise} \end{cases}$$

where the indices indicate lengths in bytes. The use of longer static IVs in TLS 1.3 is a practical improvement, as (informally) it acts as an auxiliary secret input to the PRF and may improve multi-user security [7]. This is particularly clear for ChaCha20, where the key, nonce, and counter are just copied side by side to the initial cipher state.

We easily verify (by typing) that both constructions are injective for $0 \leq seqn < 2^{64}$, which is required (also by typing) to meet the 'fresh nonce' pre-condition for calling AEAD encryption. Formally, the state invariant for StAE encryption is that $0 \leq seqn < 2^{64}$ and the underlying AEAD log has an entry for every nonce n computed from a sequence number smaller than $seqn$.

StAE functionality A stream authenticated encryption functionality StAE implements the following interface:

```
type seqn_t = UInt64.t
val q_e: seqn_t (* maximal number of encryptions *)
val cipherlen: n:nat{ n ≤ ℓ_p } → Tot nat (* e.g. ℓ_p + MAC.ℓ_t *)

type role = E | D
abstract type state (r:role)
val seqn: mem → state r → Spec seqn_t
val gen: unit → ST (s:state E) (ensures seqn s' = 0)
val genD: state E → ST (s:state D) (ensures seqn s' = 0)
val encrypt: s:state E → ad → p:plain →
  ST (c:cipher{length c = cipherlen (length p)})
  (requires seqn s < q_e) (ensures seqn s' = seqn s + 1)
val decrypt: s:state D → ad → c:cipher →
  ST (o:option (p:plain{length c = cipherlen (length p)}))
  (requires seqn s < q_e)
  (ensures seqn s' = if o = None then seqn s else seqn s + 1)
```

We omit type declarations for plain, cipher and ad as they are similar to AEAD. For TLS, the length of additional data ℓ_a can be 0 (TLS 1.3) or 13 (TLS 1.2) and the length of IVs ℓ_{iv} is 12. Compared to previous functionalities, the main change is that keys are replaced by states that embed a 64-bit sequence number. Accordingly, in this section we assume that at most 2^{64} fragments are encrypted. The stateful function gen initializes the encryptor state used by the encryption algorithm, while genD initializes a decryptor state used by the decryption algorithm. The stateful encrypt and decrypt functions require that the sequence number in the key state does not overflow (seqn s < q_e) and ensure that it is incremented (only on success in the case of decryption). In pseudo-code, authenticated stream encryption is constructed as follows:

Function gen()
$k \xleftarrow{\$} \mathsf{AEAD.keygen}()$
$iv \xleftarrow{\$} \mathsf{byte}^{\ell_{iv}}$
return $\{k \leftarrow k;$
$iv \leftarrow iv;\ seqn \leftarrow 0\}$

Function genD(s)
return $\{k \leftarrow s.k;$
$iv \leftarrow s.iv;\ seqn \leftarrow 0\}$

Function encrypt(s, a, p)
$n \leftarrow \mathsf{nonce}(s.iv, s.seqn)$
$c \xleftarrow{\$} \mathsf{AEAD.encrypt}(s.k, n, a, p)$
$s.seqn \leftarrow s.seqn + 1$
return c

Function decrypt(s, a, c)
$n \leftarrow \mathsf{nonce}(s.iv, s.seqn)$
$p \leftarrow \mathsf{AEAD.decrypt}(s.k, n, a, c)$
if $(p = \bot)$ **return** $\bot$
$s.seqn \leftarrow s.seqn + 1$
return p

*Definition 7 (*Stae*):* Let $\epsilon_{\mathsf{Stae}}(\mathcal{A}[q_e, q_d, \ell_p, \ell_a])$ be the advantage of an adversary $\mathcal{A}$ that makes q_e encryption queries and q_d decryption queries in the game below.

Game Stae^b(StAE)
$s \xleftarrow{\$} \mathsf{StAE.gen}()$
$D \leftarrow \varnothing \quad E \leftarrow \varnothing$
return {GenD, Encrypt, Decrypt}

Oracle GenD(d)
if $(D[d] \neq \bot)$ **return** $\bot$
$D[d] \leftarrow \mathsf{StAE.genD}(s)$

Oracle Encrypt(a, p)
if b
$c \leftarrow \mathsf{byte}^{\mathsf{cipherlen}(|p|)}$
else
$c \leftarrow \mathsf{StAE.encrypt}(s, a, p)$
$E[s.seqn - 1, a, c] \leftarrow p$
return c

Oracle Decrypt(d, a, c)
if $(D[d] = \bot)$ **return** $\bot$
if b
$p \leftarrow E[D[d].seqn, a, c]$
if $(p \neq \bot)$
$D[d].seqn \leftarrow D[d].seqn + 1$
else
$p \leftarrow \mathsf{StAE.decrypt}(D[d], a, c)$
return p

The game involves a single encryptor, a table of decryptors D, and a log of encryptions E. For brevity, it relies on the stateful encryptor and decryptors specified above, e.g. encrypts increments $s.seqn$ and Encrypt records the encryption with sequence number $s.seqn - 1$. (Equivalently, it could keep its own shadow copies of the sequence numbers.) In contrast with AEAD, decryption only succeeds for the current sequence number of the decryptor.

Our definition corresponds most closely to level-4 (stateful) LHAE of [16]. In both definitions the requirement is that decrypt only successfully decrypted a prefix of what was sent. A difference is that we do not require real decryption to continue rejecting ciphertexts upon decryption failure. We also leave length-hiding and stream termination to §VII.

*Theorem 3 (*Stae *perfectly reduces to* Aead*):* Given $\mathcal{A}$ against Stae, we construct $\mathcal{B}$ against Aead with

$$\epsilon_{\mathsf{Stae}}(\mathcal{A}[q_e, q_d, \ell_p, \ell_a]) = \epsilon_{\mathsf{Aead}}(\mathcal{B}[q_e, q_d, \ell_p, \ell_a]).$$

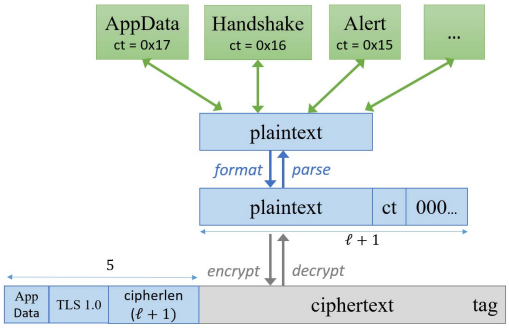

Figure 3. Constructing a TLS 1.3 record fragment

VII. TLS Content Protection: Length-Hiding Stream Encryption

We are now ready to use stream encryption for protecting TLS 1.3 traffic, which consists of a sequence of protocol-message fragments, each tagged with their content type, while hiding their content, their type, and their actual size before padding. The steps taken by the record layer to construct encrypted fragments are depicted in Figure 3, with apparent size ℓ after padding. The last line adds the (unprotected) record header; for backward compatibility, it pretends to be a TLS 1.0 AppData record irrespective of its actual encrypted content type. On the other hand, TLS does not attempt to hide the record boundaries (as e.g. SSH) so we do not expect indistinguishability from random for the resulting record.

Formatting: Content Type and Length Hiding Encryption and decryption rely on formatting and padding functions over a type fragment indexed by a length ℓ indicating the public *maximal length* of its content, specified as follows:

```
type len = n:nat {n ≤ 2^14} (* valid record length in TLS *)
type fragment (ℓ:len) = {ct:byte; data:bbytes ℓ}
val parse: ℓ:len → lbytes (ℓ+1) → Tot (option (fragment ℓ))
val format: ℓ:len → f:fragment ℓ → Tot (p:lbytes (ℓ+1))
  (ensures parse ℓ p = Some f)
```

These functions must be carefully implemented to prevent any side channel. We also construct and parse records into headers and payloads using functions

```
val parse_record: r:record → Tot (option (n:nat × c:lbytes n))
val format_record: n:nat → c:lbytes n → Tot (r:record)
  (ensures parse_record r = Some (n,c))
```

These function specifications suffice to establish our theorems below. We now give the concrete format function for TLS 1.3:

$$\frac{\textbf{Function format}(\ell : \text{len}, f : \text{fragment } \ell)}{f.data \,\|\, [f.ct] \,\|\, pad_0(\ell - |f.data|)}$$

where $\text{pad}_0\ n$ is the string of n 0x00 bytes. We verify the post-condition of format by typing. We omit the corresponding parse function and the code for processing headers.

The implementation of parse and format, and the converse function for parsing a bytestring into a fragment value, require precautions to avoid leaking the actual contents length using side-channels. The code for processing headers does *not* depend on the fragment, only on its length after padding.

Stream Closure As explained in §VI, stream integrity ensures that decrypted traffic is a prefix of encrypted traffic. Complementarily, the TLS record layer relies on well-defined *final fragments*, specified as a predicate `val final: fragment ℓ → Tot bool`, to ensure that no further encryptions are performed on a stream after sending such a fragment.

For LHAE, we extend the stateful key of StAE to record the termination status of the stream, which can be queried with the predicate `val closed: mem → state r → Spec bool`. Furthermore, we extend the post-condition of encryption to ensure that the state s′ after encrypting fragment f satisfies `closed s′ = final f`. Therefore, upon receiving a final fragment, the decryptor is guaranteed to have received the whole data stream.

LHSE Construction and Game The construction is:

Function encrypt(s, ℓ, f)
if closed(s) **return** $\perp$
$p \leftarrow$ format(ℓ, f)
$c \leftarrow$ StAE.encrypt$(s, [], p)$
if (final f) $s \leftarrow$ *closed*
return format_record(ℓ, c)

Function decrypt(s, r)
if closed(s) **return** $\perp$
$\ell, c \leftarrow$ parse_record(v)
$p \leftarrow$ StAE.decrypt$(s, [], c)$
$f \leftarrow$ parse(ℓ, p)
if $(f \neq \perp \wedge$ final $f)$ $s \leftarrow$ *closed*
return f

with the same state as StAE—we omit the unmodified functions for generating encryptors and decryptors. When a final fragment is sent or received, we erase the StAE state.

The TLS 1.3 construction uses empty associated data, relying on implicit authentication of the underlying key and sequence number. (Our code also supports the older TLS 1.2 construction, which uses 13 bytes of associated data in total, obtained by appending the protocol version and the content type to the sequence number of stream encryption.)

*Definition 8 (*Lhse*):* Given LHSE, let $\epsilon_{\text{Lhse}}(\mathcal{A}[q_e, q_d])$ be the advantage of an adversary $\mathcal{A}$ that makes q_e encryption queries and q_d decryption queries in the game below.

Game Lhse^b(LHSE)
$s \xleftarrow{\$}$ Lhse.gen()
$D \leftarrow \varnothing$; $F \leftarrow \varnothing$
return {GenD, Encrypt, Decrypt}

Oracle GenD(d)
if $(D[d] = \perp)$
  $D[d] \leftarrow$ LHSE.genD(s)

Oracle Encrypt(ℓ, f)
if b
  $r \leftarrow$ LHSE.encrypt$(s, \ell, f_{\text{final}(f)})$
else
  $r \leftarrow$ LHSE.encrypt(s, ℓ, f)
$F[s.seqn - 1, r] \leftarrow f$
return v

Oracle Decrypt(d, v)
if $(D[d] = \perp)$ **return** $\perp$
$s_d \leftarrow D[d]$
if b
  if closed(s_d) **return** $\perp$
  $f \leftarrow F[s_d.seqn, r]$
  if $(f \neq \perp)$ $s_d.seqn$++
  if $(f \neq \perp \wedge$ final $f)$
    $s_d \leftarrow$ *closed*
else
  $f \leftarrow$ LHSE.decrypt(s_d, v)
return f

where f_0 (respectively, f_1) is a fragment (respectively, a final fragment), with fixed content type and data 0^ℓ.

The game logs the encryption stream in F, indexed by fragment sequence numbers and ciphertexts. It has an oracle for creating decryptors; it stores their state in a table D, indexed by some abstract d chosen by the adversary. It does not model stream termination, enforced purely by typing the stream content.

*Theorem 4 (*Lhse *perfectly reduces to* Stae*):*
Given $\mathcal{A}$ against Stae, we construct $\mathcal{B}$ against Aead with

$$\epsilon_{\mathsf{Lhse}}(\mathcal{A}[q_e, q_d]) = \epsilon_{\mathsf{Stae}}(\mathcal{B}[q_e, q_d, 2^{14}+1, \ell_a])$$

where ℓ_a is 0 for TLS 1.3 and 13 for TLS 1.2.

Multi-Stream LHSE In the next section (as in our interface above), we use a multi-instance Lhse game, defined below.

$\mathbf{Game}\ \mathsf{Multi}(\mathsf{Lhse}^b)$
$E \leftarrow \varnothing;\ \mathbf{return}\ \{\mathsf{Gen}, \mathsf{GenD}, \mathsf{Encrypt}, \mathsf{Decrypt}\}$

$\mathbf{Oracle}\ \mathsf{Gen}(i)$
$\mathbf{if}\ (E[i] = \bot)\ E[i] \xleftarrow{\$} \mathsf{Lhse}^b()$

$\mathbf{Oracle}\ \mathsf{GenD}(i, d)$
$\mathbf{if}\ (E[i] = \bot)\ E[i] \xleftarrow{\$} \mathsf{Lhse}^b()$
$E[i].\mathsf{GenD}(d)$

$\mathbf{Oracle}\ \mathsf{Encrypt}(i, \ell, f)$
$\mathbf{if}\ (E[i] = \bot)\ \mathbf{return}\ \bot$
$\mathbf{return}\ E[i].\mathsf{Encrypt}(\ell, f)$

$\mathbf{Oracle}\ \mathsf{Decrypt}(i, d, v)$
$\mathbf{if}\ (E[i] = \bot)\ \mathbf{return}\ \bot$
$\mathbf{return}\ E[i].\mathsf{Decrypt}(d, v)$

For every fresh index i passed to Gen, we spawn an instance of Lhse and we record its state and oracle in table E. In all other cases, the oracles above now look up the shared instance at i and forward the call to the instance oracle.

Security bounds for TLS Table I gives the concrete bounds by ciphersuites, setting ℓ_p to $2^{14}+1$ and ℓ_a to 0 (or 13 for TLS 1.2). ChaCha20 uses a Davies-Meyer construction and is considered a good PRF. For AES-GCM ciphersuites, blocks are relatively small (16 bytes) so we incur a loss of $\frac{q_b^2}{2^{129}}$ by the PRP/PRF switching lemma [6], intuitively accounting for the probability of observing collisions on ciphertext blocks and inferring the corresponding plaintext blocks are different. As observed e.g. by Luykx and Paterson [34], this factor limits the amount of data that can be sent securely using AES-GCM.

Based on their suggestion to send at most $2^{24.5}$ fragments with the same key (itself based on a proof by [9] for the UF-1CMA security of GHASH that avoids the full PRF-PRP switching loss), our implementation may automatically trigger TLS re-keying after sending $2^{24.5}$ fragments. This strategy results in the bound in the last row, which no longer depends quadratically on q_e and thus achieves a poor man's form of beyond birthday bound security.

Ciphersuite	$\epsilon_{\mathsf{Lhse}}(\mathcal{A}[q_e, q_d]) \leq$
General bound	$\epsilon_{\mathsf{Prf}}(\mathcal{B}[q_e(1 + \lceil (2^{14}+1)/\ell_b \rceil) + q_d + j_0])$ $+\ \epsilon_{\mathsf{MMac1}}(\mathcal{C}[2^{14}+1+46, q_d, q_e+q_d])$
ChaCha20-Poly1305	$\epsilon_{\mathsf{Prf}}(\mathcal{B}\left[q_e\left(1 + \left\lceil \frac{(2^{14}+1)}{64} \right\rceil\right) + q_d\right]) + \frac{q_d}{2^{93}}$
AES128-GCM AES256-GCM	$\epsilon_{\mathsf{Prp}}(\mathcal{B}[q_b]) + \frac{q_b^2}{2^{129}} + \frac{q_d}{2^{118}}$ where $q_b = q_e(1 + \lceil (2^{14}+1)/16 \rceil) + q_d + 1$
AES128-GCM AES128-GCM	$\frac{q_e}{2^{24.5}}\left(\epsilon_{\mathsf{Prp}}(\mathcal{B}[2^{34.5}]) + \frac{1}{2^{60}} + \frac{1}{2^{56}}\right)$ with re-keying every $2^{24.5}$ records (counting q_e for all streams, and $q_d \leq 2^{60}$ per stream)

Table I
SUMMARY OF SECURITY BOUNDS FOR THE TLS AEAD CIPHERSUITES.

VIII. THE TLS 1.3 RECORD PROTOCOL

Figure 4 presents the TLS 1.3 protocol from draft-19, focusing only on how it drives the record layer. In particular, this presentation ignores most of the details of the handshake. Figure 1 in Section I illustrates the sub-protocol streams from the point of view of the client.

The client sends the `ClientHello` message in cleartext, and may then immediately install a fresh 0-RTT key k_0^c in the record layer and use it to encrypt a stream of *early data*.

The server receives this message and, if it accepts 0-RTT, also installs the 0-RTT key k_0^c and decrypts the client's data. Otherwise, it discards this first stream. In parallel, the server sends a `ServerHello` message that allows both parties to derive encryption keys k_h^c and k_h^s for the handshake messages, and k_1^c and k_1^s for application data. The server installs k_h^s in the record and uses it to encrypt a first stream of handshake messages, ending with a finished message that triggers the installation of key k_1^s. If the server supports 0.5-RTT, it may immediately start using k_1^s for sending application data.

Once 0-RTT stream is complete (signaled by an end-of-early-data message) and after processing the `ServerHello`, the client installs the handshake keys k_h^c and k_h^s for encryption and decryption. It completes the handshake by sending its own encrypted stream, ending with a finished message, and installs the application traffic keys k_1^c and k_1^s.

Upon completing the handshake, the server also installs k_1^c for decryption. After this point, the connection is fully established and both parties use the installed application traffic keys for all content types: AppData, Alert, and even Handshake messages (such as `KeyUpdate`).

Later, the client (or the server) can terminate their current output stream by sending either a `KeyUpdate` handshake message or a close-notify alert message. In the first case, it installs and starts using the next application traffic key k_2^c (then k_3^c, etc.). The server (or the client) responds accordingly, with a `KeyUpdate` or a close-notify. In the first case, it installs and starts using the next traffic keys k_2^c and k_2^s. In the second case, the connection is closed.

In all cases, each party uses a single stream at a time in each direction, for sending and receiving all content types, and each stream ends with a distinguished message that

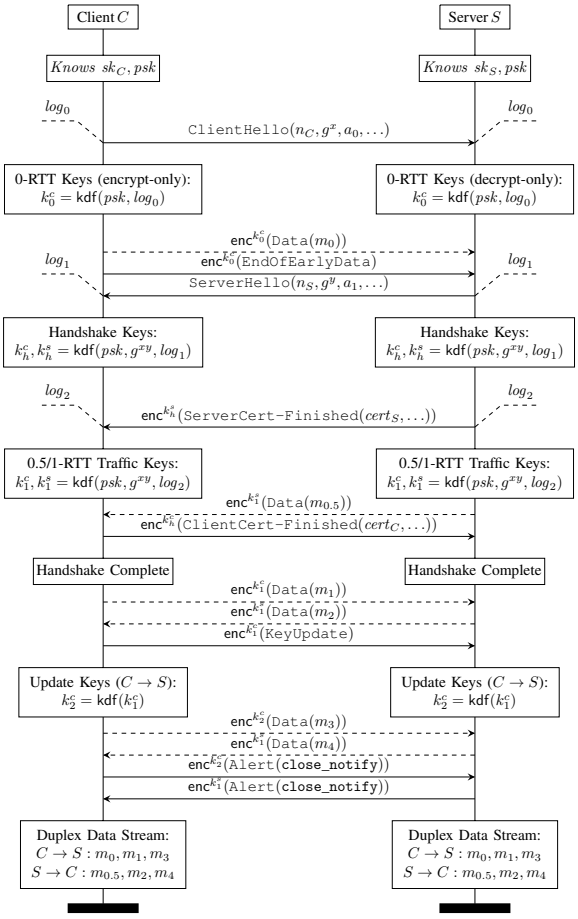

Figure 4. TLS 1.3 draft-19 message exchange, seen from the viewpoint of the Record Protocol. Dotted arrows represent zero or more (encrypted) application data fragments. Each key stands for an instance of LHSE.

clearly indicates its final fragment. 0-RTT data ends with an end-of-early-data message; encrypted handshake messages in both directions end with the finished message; 0.5 and 1-RTT data streams end with a key update or a close-notify alert. This precaution ensures that any truncations at the end of a stream will cause a connection failure, rather than continuing with the next stream.

Performance/Security Trade-Offs. 0-RTT and 0.5-RTT significantly decrease communications latency, but they yield weaker application security. 0-RTT traffic has weaker forward secrecy and is subject to replays: if multiple servers may accept the connection and (as usual) do not share an anti-replay cache, then they may all receive and process (prefixes of) the same early traffic data. This motivates our model with multiple decryptors, and also requires the server application to defer some effects of early-data processing till handshake completion. Also, since data is sent before ciphersuite negotiation, the client may use relatively weak algorithms (or broken implementations) that the server would otherwise have a chance to refuse.

0.5-RTT incurs similar, lesser risks as the server sends data before the handshake completes. The server is subject to 0-RTT truncation attacks if it starts sending data before receiving the client's end of early data. Also, if the server relies on a client signature, it should not send sensitive data before handshake completion. In contrast with 0-RTT, sending 0.5-RTT traffic is a local configuration issue for the server; the client receives 0.5-RTT data after completing the handshake and does not distinguish it from 1-RTT data.

TLS 1.2 is routinely deployed with 'FalseStart', which is similar to 0.5-RTT but in the other direction, the client may locally decide to start sending encrypted application data as soon as it can compute the keys, before handshake completion. This places additional trust in the client's ciphersuite whitelist, inasmuch as sensitive data may be sent before confirming their correct negotiation with the server.

A Minimal Record Game Next, we present a simplified, more liberal model of the Record that seeks to abstract away from the details of how the connection evolves. This facilitates the statement of a standalone 'record-layer' theorem, but our approach similarly applies to our full F⋆ implementation integrated with miTLS, which carefully keeps track of the sequence of keys, as outlined at the end of this section.

We abstract the state of the connection by a context bitstring; as the handshake progresses, we concatenate more relevant handshake parameters to the context. For instance, after `ClientHello`, the context consists of the client's nonce n_C and its proposed ciphersuites and key exchange values; after `ServerHello`, it additionally contains the server nonce n_S, algorithm choice, key exchange value, etc.

Instead of modeling duplex channels between clients and servers, we consider separate sequences of streams in each direction. Our game (Figure 5) models re-keying and context extension for a sequence of streams (all in the same direction), covering 0-RTT, 0.5-RTT, and 1-RTT traffic, relying on the multi-instance game $\mathsf{SE} = \mathsf{Multi}(\mathsf{Lhse})$ (see §VII).

The game has oracles Init and InitD for generating multi-stream encryptors and decryptors in their initial state, indexed by n and m, respectively. We assume that their arguments determine the record algorithm. Their state consist of a current context, a current stream number j, and a local map I from stream numbers to the value of the context when they were installed. We use variables ctx, j, and I to refer to the fields of $E[m]$ and $D[n]$, respectively.

Oracles Extend and ExtendD allow the local context to be extended (concatenated) with new information at any time.

Oracles Install and InstallD install an LHSE instance (allocating it if it does not exist) for encryption and decryption,

Game $\mathsf{Record}^b(\mathsf{Lhse}^b(\mathsf{LHSE}))$

$E \leftarrow \varnothing \quad D \leftarrow \varnothing$

$\mathsf{SE.Gen}, \mathsf{SE.GenD}, \mathsf{SE.Encrypt}, \mathsf{SE.Decrypt} \xleftarrow{\$} \mathsf{Multi}(\mathsf{Lhse}^b)$

return {Gen, Extend, Install, Encrypt, GenD, ExtendD, InstallD, Decrypt}

Oracle $\mathsf{Init}(n)$

if $E[n] = \perp$
  $E[n] \leftarrow \{ctx \leftarrow n; j \leftarrow 0; I \leftarrow \varnothing\}$

Oracle $\mathsf{InitD}(m, ctx_0)$

if $D[m] = \perp$
  $D[m] \leftarrow \{ctx \leftarrow ctx_0; j \leftarrow 0; I \leftarrow \varnothing\}$

Oracle $\mathsf{Extend}(n, \delta)$

if $E[n]$ *exists*
  $ctx \leftarrow ctx + \delta$

Oracle $\mathsf{ExtendD}(m, \delta)$

if $D[m]$ *exists*
  $ctx \leftarrow ctx + \delta$

Oracle $\mathsf{Install}(n)$

if $E[n]$ exists with $I[j] = \perp$
  $I[j] \leftarrow (ctx, j)$
  $\mathsf{SE.Gen}(I[j])$

Oracle $\mathsf{InstallD}(m)$

if $D[m]$ *exists with* $I[j] = \perp$
  $I[j] \leftarrow (ctx, j)$
  $\mathsf{SE.GenD}(I[j])$

Oracle $\mathsf{Encrypt}(n, \ell, f)$

if $E[n]$ *exists with* $I[j] \neq \perp$
  $v \leftarrow \mathsf{SE.Encrypt}(I[j], \ell, f)$
  if (*final* f) $j \leftarrow j + 1$
  return v

Oracle $\mathsf{Decrypt}(m, v)$

if $D[m]$ *exists with* $I[j] \neq \perp$
  $f \leftarrow \mathsf{SE.Decrypt}(I[j], d, v)$
  if $f \neq \perp \wedge$ *final* f
    $j \leftarrow j + 1$
  return f

Figure 5. The TLS 1.3 Record Game

respectively. Recall that calls to SE.Gen are memoized, so that an encryptor and a decryptor share the same stream if and only if they agree on the stream sequence number and context.

Oracles Encrypt and Decrypt apply encryption and decryption to the currently-installed stream. Some fragments are final: they terminate the stream and signal the need to install a new stream before continuing.

*Definition 9 (*Record*):* Let $\epsilon_{\mathsf{Record}}(\mathcal{A}[q_e, q_d, q_i])$ be the advantage of an adversary $\mathcal{A}$ that makes at most q_e encryption queries and q_d decryption queries for each of the q_i LHSE instances created using install queries in the game of Figure 5.

*Theorem 5 (*Record *reduces to* Lhse*):*

$$\epsilon_{\mathsf{Record}}(\mathcal{A}[q_e, q_d, q_i]) \leq q_i \; \epsilon_{\mathsf{Lhse}}(\mathcal{B}[q_e, q_d]).$$

Our game complies with the idealized interface for LHSE and relies on its conditional idealization. If $b = 0$, then the oracles operate purely on local state, and simply implement a real sequence of encrypted streams, under the control of the record state machine. If $b = 1$, then we get perfect authentication of (a prefix of) the whole sequence of streams of fragments. (This property is verified by typing our idealized record implementation.) The ctx field of encryptors and decryptor represents their implicitly authenticated shared context: unless there is an encryptor with a matching context, the ideal encryption log is empty hence decryption will fail. In particular, as soon as the context includes `ServerCert-Finished`, and thus the fresh TLS nonces n_C and n_S, we know that there is at most one encryptor and one decryptor.

More precisely, consider encryptor and decryptor states $E[n]$ and $D[m]$. If $E[n].I[j] = D[m].I[j]$ then also $E[n].I[j'] = D[m].I[j']$ for any $j' < j$. Thus, for instance, when $D[m]$ receives a final fragment, we know that $E[n]$ and $D[m]$ agree on the whole sequence of communicated fragment for the first j streams. By Theorem 5 these guarantees also hold for the real record for any game adversary $\mathcal{A}$, except with probability $\epsilon_{\mathsf{Record}}(\mathcal{A})$.

Application to 0-RTT We briefly show how to control our game to model 0-RTT and 0.5-RTT. For 0-RTT, the client is the encryptor and the server is the decryptor. Both use the encryptor index n as initial context, representing the content of `ClientHello`, notably the fresh client random n_C. Conversely, the decryptor index m (including the fresh server random n_S) is *not* included in the initial context of InitD. As both parties install their first stream ($j = 0$) for 0-RTT, this reflects that the underlying TLS key derivation (k_0^c in Figure 4) depends only on client-side information. Thus, although 0-RTT traffic is protected, it may be decrypted by multiple server instances with different indexes m.

Calls to ExtendD and Extend reflect handshake communications in the other direction, as the `ServerCert-Finished` stream is sent and received, causing ctx to be extended with (at least) m. Afterwards, as the two parties successively install streams for the TLS keys k_h^c, k_1^c, k_2^c, ..., successful decryption guarantees agreement on a context that includes the pair n, m. Thus, in this usage of our Record game at most one server will successfully decrypt the first encrypted handshake fragment from the client, and from this point all streams are one-to-one.

Application to 0.5-RTT The server is the encryptor, the client the decryptor and, since they both have initial access to the first message exchange, we may select as index n that includes the client hello and server hello messages and implicitly authenticate the pair n_C, n_S. Thus, there is at most one honest client decryptor for 0.5-RTT and, from the client's viewpoint, successful decryption of the first handshake fragment ensures agreement on this context. Still (at least from the record's viewpoint) the server is not guaranteed there is a matching decryptor until it receives `ClientCert-Finished` in the other direction and transitions to 1-RTT.

Verified Implementation for miTLS (Outline) Our TLS Record implementation supports sequences of streams for the full protocol described in Figure 4 and its TLS 1.2 counterpart.

Stream Sequences As described in the game above, it

maintains a current stream for each direction, and it receives 'extend' and 'install' commands from the handshake protocol as the connection gets established. Its indexes (ctx in the game) consist of a summary of the handshake context available at the time of key derivation (always including the algorithms to use). In contrast with our game, which models all communications in a single direction, our code supports 'duplex' communications. This is necessary, for instance, for synchronizing key updates and connection closure between clients and servers. Our code also maintains a small (type-based) state machine that controls the availability of the current streams for sending application data.

Re-keying and Corruption The state machine enforces a limit on the maximum number of fragments that can be sent with one key to prevent sequence number overflows and account for the birthday bound weakness of AES-GCM. On key updates we delete old keys and model the corruption of individual streams using leak and coerce functions for keys. This is in keeping with the static corruption modeling of MITLS, e.g. to account for insecure handshake runs.

Fragment API Our code for the record is parameterized by a type of abstract application-data plaintexts, indexed by a unique stream identifier and an apparent fragment length. Type abstraction guarantees that, if the stream identifier is *safe* (a property that depends on the handshake and key derivation process), the idealized TLS implementation never actually accesses their actual length and contents, a strong and simple confidentiality property.

Our API has a configuration to control 0-RTT and 0.5-RTT as the connection is created. In particular, 0-RTT plaintexts have their own type, (indexed by an identifier that describe their 0-RTT context) which should help applications treat it securely. Conversely, 0.5-RTT is modeled simply by enabling earlier encryption of 1-RTT traffic.

Message API Our code also has a higher-level API with messages as (potentially large) bytestrings instead of individual fragments. As usual with TLS, message boundaries are application specific, whereas applications tend to ignore fragment boundaries. Nonetheless, our code preserves apparent message boundaries, never caches or fragments small messages, and supports message length-hiding by taking as inputs both the apparent (maximal) size ℓ_{max} of the message and its private (actual) size ℓ_m. It has a simple fragmentation loop, apparently sending up to 2^{14} bytes at each iteration, starting with $\ell_{max} - \ell_m$ bytes of padding follows by the actual data. (This ensures that an application that waits for the whole message never responds before receiving the last fragment.) We do not model de-fragmentation on the receiving end; our code delivers fragments as they arrive in a buffer provided by the application for reassembling its messages.

The correctness and security of this construction on top of the fragment API is verified by typing, essentially abstracting sequences of fragments into sequences of bytes for the benefit of the application, and turning close-notify alert fragments into end-of-files. (See also [22] for a cryptographic treatment of fragmentation issues.)

IX. Experimental evaluation

We evaluate our reference implementation of the TLS record layer both qualitatively (going over the verified goals of the various modules and how they relate to the games presented in the paper, and checking that our implementation interoperates with other TLS libraries) and quantitatively (measuring the verification and runtime performance).

Verification evaluation Table II summarizes the modular structure of our code, and evaluates the verification costs and the extracted OCaml and C implementations. Since proofs and implementations are tightly interleaved in F⋆ source code, it is difficult to accurately measure a precise source overhead. The reported annotation percentages figure in the table are rough manual estimates, but they can be compared to the size of extracted implementations.

Most of the verification burden comes from the security proof of AEAD (totaling approximately 4,500 lines of annotation out of a total of about 5,500 lines of F⋆) and the functional correctness proof of the MAC implementations (totaling over 4,000 lines of annotations and lemmas). For the latter, we extended F⋆ with a new big number library to verify the low-level implementations of various mathematical operations (such as point multiplication on elliptic curves or multiplication over finite fields) using machine-sized integers and buffers. We use it to prove the correctness of the polynomial computations for Poly1305 and GHASH.

Current limitations Our record layer implementation is part of miTLS in F⋆: a larger, partially verified codebase that intends to provide a secure, efficient implementation for both TLS 1.2 and TLS 1.3. We leave a complete verification of the TLS 1.3 handshake and its integration with our code as future work.

Our implementation also includes the first miTLS component implemented in a lower level fragment of F⋆ that enables its extraction to C code. In contrast, the rest of miTLS is still extracted to OCaml and, until it is similarly adapted, relies on an unverified OCaml/C wrapper to call our extracted C code. Currently, this runtime transition is done at the level of the AEAD interface, enabling us to switch to other generic cryptographic providers more easily (e.g. to compare performance with OpenSSL). Hence, the TLS-specific stateful encryption in StreamAE is verified on top of an idealized AEAD interface that slightly differs from the one exported by our idealized Crypto.AEAD construction. For instance, the former represents fragments as sequences of bytes, whereas the latter uses a lower level memory

Module Name	Verification Goals	LoC	% annot	ML LoC	C LoC	Time
StreamAE	Game StAE^b from §VI	318	40%	354	N/A	307s
AEADProvider	Safety and AEAD security (high-level interface)	412	30%	497	N/A	349s
Crypto.AEAD	Proof of Theorem 2 from §V	5,253	90%	2,738	2,373	1,474s
Crypto.Plain	Plaintext module for AEAD	133	40%	95	85	8s
Crypto.AEAD.Encoding	AEAD encode function from §V and injectivity proof	478	60%	280	149	708s
Crypto.Symmetric.PRF	Game PrfCtr^b from §IV	587	40%	522	767	74s
Crypto.Symmetric.Cipher	Agile PRF functionality	193	30%	237	270	65s
Crypto.Symmetric.AES	Safety and correctness w.r.t pure specification	1,254	30%	4,672	3,379	134s
Crypto.Symmetric.ChaCha20		965	80%	296	119	826s
Crypto.Symmetric.UF1CMA	Game $\mathsf{MMac1}^b$ from §III	617	60%	277	467	428s
Crypto.Symmetric.MAC	Agile MAC functionality	488	50%	239	399	387s
Crypto.Symmetric.GF128	$GF(128)$ polynomial evaluation and GHASH encoding	306	40%	335	138	85s
Crypto.Symmetric.Poly1305	$GF(2^{130}-5)$ polynomial evaluation and Poly1305 encoding	604	70%	231	110	245s
Hacl.Bignum	Bignum library and supporting lemmas for the functional correctness of field operations	3,136	90%	1,310	529	425s
FStar.Buffer.*	A verified model of mutable buffers (implemented natively)	1,340	100%	N/A	N/A	563s
Total		15,480	78%	12,083	8,795	1h 41m

Table II
MODULES IN OUR VERIFIED RECORD LAYER IMPLEMENTATION

model and represents fragments as mutable I/O buffers, hence the transition requires copying fragments between representations.

Interoperability Our record implementation supports both TLS 1.3 and 1.2 and exposes them through a common API. We have tested interoperability for our TLS 1.2 record layer with all major TLS implementations. For TLS 1.3 draft-14, we tested interoperability with multiple implementations, including BoringSSL, NSS, BoGo, and Mint, at the IETF96 Hackathon. For draft-18, we tested interoperability with the latest version of Mint at the time of writing. In all cases, our clients were able to connect to interoperating servers using an ECDHE or PSK_ECDHE key exchange, then to exchange data with one of the following AEAD algorithms: AES256-GCM, AES128-GCM, and ChaCha20-Poly1305. Similarly, our servers were able to accept connections from interoperating clients that support the above ciphersuites.

Performance We evaluate the performance of our record layer implementation at two levels. First, we compare our implementation of AEAD encryption extracted to C using an experimental backend for F⋆ to OpenSSL 1.1.0 compiled with the no−asm option, disabling handwritten assembly optimizations. Our test encrypts a random payload of 2^{14} bytes with 12 bytes of constant associated data. We report averages over 3,000 runs on an Intel Core E5-1620v3 CPU (3.5GHz) on Windows 64-bit.

	Crypto.AEAD	OpenSSL
ChaCha20-Poly1305	13.67 cycles/byte	9.79 cycles/byte
AES256-GCM	584.80 cycles/byte	33.09 cycles/byte
AES128-GCM	477.93 cycles/byte	28.27 cycles/byte

Our implementation is 17 to 18 times slower than OpenSSL for AES-GCM and about 30% slower for ChaCha20-Poly1305. Note that the performance of custom assembly implementations can be significantly better. OpenSSL with assembly can perform ChaCha20-Poly1305 in about one cycle per byte and can do AES128-GCM and AES256-GCM in less than half a cycle per byte.

Next, we measure the throughput of our record layer integrated into miTLS by downloading one gigabyte of random data from a local TLS server. We compare two different integration methods: first, we extract the verified record layer in OCaml, and compile it alongside the OCaml-extracted miTLS. Then, we build an F⋆ interface to the C version of our record implementation and call it from miTLS. We compare these results with the default AEAD provider of miTLS (based on OpenSSL 1.1.0 with all optimizations, including hardware-accelerated AES), and curl (which uses OpenSSL for the full TLS protocol).

	OCaml	C	OpenSSL	curl
ChaCha20-Poly1305	167 KB/s	183 MB/s	354 MB/s	440 MB/s
AES256-GCM	68 KB/s	5.61 MB/s	398 MB/s	515 MB/s
AES128-GCM	89 KB/s	5.35 MB/s	406 MB/s	571 MB/s

We observe that miTLS is not a limiting factor in these benchmarks as its performance using the OpenSSL implementation of AEAD encryption is comparable to that of libcurl.

Unsurprisingly, the OCaml version of our verified implementation performs very poorly. This is due to the high overhead of both memory operations and arithmetic computations in the OCaml backend of F⋆ (which uses garbage-collected lists for buffers, and arbitrary-precision zarith integers). The C extracted version is over 30,000 times faster, but remains two orders of magnitude slower than the hardware-optimized assembly implementations in OpenSSL for AES. For ChaCha20-Poly1305, we achieve 50% of the assembly-optimized OpenSSL throughput.

Although our code is optimized for verification and modularity rather than performance, we do not believe that

we can close the performance gap only by improving F⋆ code for hardware-accelerated algorithms such as AES-GCM—instead, we intend to selectively link our F⋆ code with assembly code proven to correctly implement a shared functional specification. We leave this line of research for future work.

REFERENCES

[1] N. J. AlFardan and K. G. Paterson, "Lucky thirteen: Breaking the TLS and DTLS record protocols," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 526–540.

[2] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir, "Verifiable side-channel security of cryptographic implementations: Constant-time MEE-CBC," in *23rd International Conference on Fast Software Encryption, FSE 2016*, 2016, pp. 163–184.

[3] C. Badertscher, C. Matt, U. Maurer, P. Rogaway, and B. Tackmann, "Augmented secure channels and the goal of the TLS 1.3 record layer," in *9th International Conference on Provable Security, ProvSec 2015*, 2016, pp. 85–104.

[4] G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy, and S. Zanella-Béguelin, "Probabilistic relational verification for cryptographic implementations," in *41st Annual ACM Symposium on Principles of Programming Languages, POPL 2014*, 2014, pp. 193–206.

[5] M. Bellare and P. Rogaway, "The security of triple encryption and a framework for code-based game-playing proofs," in *Advances in Cryptology – EUROCRYPT 2006*, 2006, pp. 409–426.

[6] ——, "Code-based game-playing proofs and the security of triple encryption," Cryptology ePrint Archive, Report 2004/331, 2004, http://eprint.iacr.org/2004/331.

[7] M. Bellare and B. Tackmann, "The multi-user security of authenticated encryption: AES-GCM in TLS 1.3," in *Advances in Cryptology – CRYPTO 2016*, 2016, pp. 247–276.

[8] D. J. Bernstein, "The Poly1305-AES message-authentication code," in *12th International Workshopo on Fast Software Encryption, FSE 2005*, 2005, pp. 32–49.

[9] ——, "Stronger security bounds for Wegman-Carter-Shoup authenticators," in *Advances in Cryptology – EUROCRYPT 2005*, 2005, pp. 164–180.

[10] K. Bhargavan and G. Leurent, "On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN," Cryptology ePrint Archive, Report 2016/798, 2016, http://eprint.iacr.org/2016/798.

[11] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub, "Implementing TLS with verified cryptographic security," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 445–459.

[12] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, , A. Pironti, and P.-Y. Strub, "Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 98–113.

[13] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella-Béguelin, "Proving the TLS handshake secure (as it is)," Cryptology ePrint Archive, Report 2014/182, 2014, http://eprint.iacr.org/2014/182/.

[14] H. Böck, "Wrong results with Poly1305 functions," https://mta.openssl.org/pipermail/openssl-dev/2016-March/006413, 2016.

[15] H. Böck, A. Zauner, S. Devlin, J. Somorovsky, and P. Jovanovic, "Nonce-disrespecting adversaries: Practical forgery attacks on GCM in TLS," Cryptology ePrint Archive, Report 2016/475, 2016, http://eprint.iacr.org/2016/475.

[16] C. Boyd, B. Hale, S. F. Mjølsnes, and D. Stebila, "From stateless to stateful: Generic authentication and authenticated encryption constructions with application to TLS," in *Topics in Cryptology – CT-RSA 2016*, 2016, pp. 55–71.

[17] C. Cremers, M. Horvat, S. Scott, and T. van der Merwe, "Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication," in *2016 IEEE Symposium on Security and Privacy*, 2016, pp. 470–485.

[18] B. Dowling, M. Fischlin, F. Günther, and D. Stebila, "A cryptographic analysis of the TLS 1.3 handshake protocol candidates," in *22nd ACM Conference on Computer and Communications Security*, 2015, pp. 1197–1210.

[19] ——, "A cryptographic analysis of the TLS 1.3 draft-10 full and pre-shared key handshake protocol," http://eprint.iacr.org/2016/081, 2016.

[20] T. Duong and J. Rizzo, "Here come the ⊕ ninjas," Available at http://nerdoholic.org/uploads/dergln/beast_part2/ssl_jun21.pdf, May 2011.

[21] M. J. Dworkin, "Recommendation for block cipher modes of operation: Galois/Counter mode (GCM) and GMAC," National Institute of Standards & Technology, Tech. Rep. SP 800-38D, 2007.

[22] M. Fischlin, F. Günther, G. A. Marson, and K. G. Paterson, "Data is a stream: Security of stream-based channels," in *Advances in Cryptology - CRYPTO 2015*, 2015, pp. 545–564.

[23] M. Fischlin, F. Günther, B. Schmidt, and B. Warinschi, "Key confirmation in key exchange: A formal treatment and implications for TLS 1.3," in *2016 IEEE Symposium on Security and Privacy*, 2016, pp. 197–206.

[24] C. Fournet, M. Kohlweiss, and P. Strub, "Modular code-based cryptographic verification," in *18th ACM Conference on Computer and Communications Security, CCS 2011*, 2011, pp. 341–350.

[25] F. Giesen, F. Kohlar, and D. Stebila, "On the security of TLS renegotiation," in *2013 ACM Conference on Computer and Communications Security, CCS 2013*, 2013, pp. 387–398.

[26] P. Gutmann, "Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)," IETF RFC 7366, 2014.

[27] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk, "On the security of TLS-DHE in the standard model," in *Advances in Cryptology – CRYPTO 2012*, 2012, pp. 273–293.

[28] T. Jager, J. Schwenk, and J. Somorovsky, "On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption," in *22nd ACM Conference on Computer and Communications Security*, 2015, pp. 1185–1196.

[29] M. Kohlweiss, U. Maurer, C. Onete, B. Tackmann, and D. Venturi, "(de-) constructing TLS 1.3," in *Progress in Cryptology–INDOCRYPT 2015*. Springer, 2015, pp. 85–102.

[30] H. Krawczyk, "LFSR-based hashing and authentication," in *Advances in Cryptology – CRYPTO 1994*, 1994, pp. 129–139.

[31] ——, "The order of encryption and authentication for protecting communications (or: how secure is SSL?)," Cryptology ePrint Archive, Report 2001/045, 2001, http://eprint.iacr.org/2001/045.

[32] H. Krawczyk and H. Wee, "The OPTLS protocol and TLS 1.3," Cryptology ePrint Archive, Report 2015/978, 2015, http://eprint.iacr.org/2015/978.

[33] H. Krawczyk, K. G. Paterson, and H. Wee, "On the security of the TLS protocol: A systematic analysis," in *Advances in Cryptology – CRYPTO 2013*, 2013, pp. 429–448.

[34] A. Luykx and K. G. Paterson, "Limits on authenticated

encryption use in TLS," Personal webpage: http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf, 2015.
[35] D. McGrew and J. Viega, "Flexible and efficient message authentication in hardware and software." Unpublished draft. Available online at http://www.cryptobarn.com/.
[36] D. McGrew, "An interface and algorithms for authenticated encryption," IETF RFC 5116, 2008.
[37] B. Möller, T. Duong, and K. Kotowicz, "This POODLE Bites: Exploiting The SSL 3.0 Fallback," Available at https://www.openssl.org/~bodo/ssl-poodle.pdf, 2014.
[38] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF protocols," IETF RFC 7539, 2015.
[39] K. G. Paterson, T. Ristenpart, and T. Shrimpton, "Tag size does matter: Attacks and proofs for the TLS record protocol," in *Advances in Cryptology – ASIACRYPT 2011*, 2011, pp. 372–389.
[40] J. Rizzo and T. Duong, "The CRIME Attack," September 2012.
[41] J. Salowey, A. Choudhury, and D. McGrew, "AES Galois Counter Mode (GCM) cipher suites for TLS," IETF RFC 5288, 2008.
[42] P. Sarkar, "A trade-off between collision probability and key size in universal hashing using polynomials," Cryptology ePrint Archive, Report 2009/048, 2009, http://eprint.iacr.org/2009/048.
[43] V. Shoup, "On fast and provably secure message authentication based on universal hashing," in *Advances in Cryptology – CRYPTO 1996*, 1996, pp. 313–328.
[44] B. Smyth and A. Pironti, "Truncating TLS connections to violate beliefs in web applications," Inria, Tech. Rep. hal-01102013, Oct. 2014. [Online]. Available: https://hal.inria.fr/hal-01102013
[45] J. Somorovsky, "Systematic fuzzing and testing of TLS libraries," in *23rd ACM Conference on Computer and Communications Security, CCS 2016*, 2016.
[46] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin, "Dependent types and multi-monadic effects in F*," in *43nd ACM Symposium on Principles of Programming Languages, POPL 2016*, 2016, pp. 256–270.
[47] R. Święcki, "ChaCha20/Poly1305 heap-buffer-overflow," CVE-2016-7054, 2016.
[48] P. Swire, J. Hemmings, and A. Kirkland, "Online privacy and ISPs: ISP access to consumer data is limited and often less than access by others," Georgia Tech, Tech. Rep., 2016.
[49] D. Wagner and B. Schneier, "Analysis of the SSL 3.0 protocol," in *2nd USENIX Workshop on Electronic Commerce, WOEC 1996*, 1996, pp. 29–40.