

Towards a Provably Secure Implementation of TLS 1.3

Benjamin Beurdouche Karthikeyan Bhargavan Antoine Delignat-Lavaud Cédric Fournet Samin Ishtiaq
Markulf Kohlweiss Jonathan Protzenko Nikhil Swamy Santiago Zanella-Béguelin Jean Karim Zinzindohoué

Abstract—We report ongoing work towards a verified reference implementation of TLS 1.3 in the F* programming language. Our code supports selected ciphersuites for all versions of TLS from 1.0 to 1.3. It is being developed on <http://github.com/mitls> as the next version of MITLS, written in F* instead of F#. We intend to prove a strong, joint cryptographic security theorem for TLS 1.3 clients and servers, even when they run alongside older versions of the protocol with weaker security. Our verification approach adapts and extends that of MITLS in several ways: (1) we adopt an idiomatic stateful style that promises higher performance than the pure functional style of MITLS; (2) we seek to prove stronger security properties with fewer ad hoc assumptions; (3) we rely on advanced F* type inference to substantially reduce both the code base and the typed proof annotations. This paper describes our implementation architecture, our new composite state machine for TLS 1.0–1.3, and our target security theorem. By the time of the TRON workshop, we will present concrete verification results for our implementation. As far as we are aware, these will be the first cryptographic proofs for an implementation of TLS 1.3, and *a fortiori* for an implementation that is backward compatible with older versions of the protocol. We anticipate that our effort will identify early problems and offer implementation guidelines for other TLS libraries.

I. INTRODUCTION

The new version of the TLS protocol [8] is drawing a lot of attention, not only from major Web companies (who would like to improve performance while getting rid of obsolete cryptographic constructions) but also, perhaps less usually, from the academic community, in the form of early analyses of the protocol [15, 11, 10, 9], and even direct contributions to its design [12].

The latest TLS 1.3 draft (revision 10 at the time of writing) defines a protocol that is quite different from previous versions of TLS. Handshake messages after the `Hello`s are now encrypted using ephemeral, unauthenticated keys in order to mitigate pervasive monitoring of Internet traffic, and a completely new “zero-round-trip” (0-RTT) handshake mode has been introduced. The record protocol has also changed significantly, moving away from the error-prone mac-then-encrypt pattern towards a new authenticated encryption (AEAD) interface. Session and connection management has also been modified, with the removal of renegotiation, the merging of resumption with pre-shared key (PSK) cipher suites, and the introduction of post-handshake client authentication.

Because of all these major changes, a lot of effort is currently being invested by the security research community into modeling and proving correct the new key exchange, record, and key schedule of TLS 1.3. While these goals are certainly important, we chose in this paper to focus on less

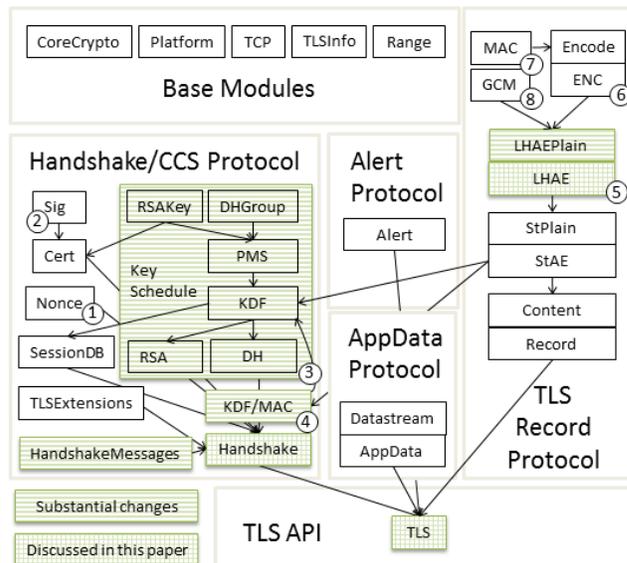


Fig. 1: miTLS Architecture for TLS 1.0-1.3

obvious problems that we argue, based on our experience implementing and analyzing previous versions of TLS, may cause issues in future implementations of the standard:

Application Interface The generally adopted design of the TLS interface is to transparently replace networking functions such as `connect`, `accept`, `read`, `write`, and `close` (even though this is not explicit in any official specification of TLS). In previous work [5], we have shown that misunderstandings of the security guarantees of various TLS functions can lead to surprising attacks. For instance, applications often assume that TLS protects the integrity of entire messages, when in fact, until a successful call to `close`, only the integrity of a prefix of the sent data is protected. Thus, early side effects on received data can be unsafe (as demonstrated by the cookie cutter attack [5]).

In addition, applications are responsible for managing sessions and tracking changes in their context, for instance when renegotiation is used for TLS client authentication, and thus cannot be completely oblivious to the TLS state (see [6, 5]). We argue that this problem becomes even more acute in TLS 1.3, with the introduction of 0-RTT client data (which has weaker security guarantees than normal data, in particular with respect with forward secrecy and re-playability) and late client authentication. These new features force applications

to be more aware of the status of their TLS connections. For example, an HTTPS server should not perform a state-changing operation in response to a 0-RTT request.

Composite state machine and downgrade resilience Due to its multiple co-existing versions and the broad range of algorithm configurations it supports, TLS has a track record of failing to ensure that handshakes between honest clients and servers succeed only when the strongest parameters supported by both peers are used. In a previous paper [2], we have shown that incorrect implementations of the protocol state machine may allow an attacker to confuse message used in related cipher suites, such as ephemeral and non-ephemeral Diffie-Hellman, or export-grade and regular RSA key exchange (a.k.a. the FREAK attack), and thus, downgrade the security of the connection. Again, we fear that this class of problem may re-occur in TLS 1.3, because of the introduction of many large new branches in the overall state machine.

Furthermore, several other downgrade attacks (ranging from the old version rollback of SSL 3.0 [16] to the recent Logjam attack [1]) are caused by weaknesses in the protocol design. Therefore, we seek to prove the resilience of TLS 1.3 against downgrade attacks by proposing a new security model and evaluating the new downgrade countermeasures in TLS 1.3 in a paper currently under submission to another conference. A confidential draft of a paper (currently under review) detailing our efforts is available at [7].

Our approach We follow the *verified reference implementation* [3, 4] approach (combining software security, protocol security, and cryptographic security within a common framework) that we successfully applied to previous versions of the protocol in miTLS. Properties of variables, as well as assumptions and guarantees of functions, are expressed as *logical refinements* on their types (for instance, by giving the type $x:\text{int} \rightarrow y:\text{int}\{y>x\}$ to the function `fun x \rightarrow x+1`), and by showing that all logical constraints are satisfied with the help of an SMT solver during static typechecking by F^* [14] (the old version of miTLS used F7). Critically, type-based verification is *modular*: once all the types of the functions within a module have been verified, they can be used by other modules without having to re-prove their goals.

The modular architecture of miTLS is outlined in Fig. 1; edges indicate that the target module depends on calls to functions in the source module to type check. We highlight the modules substantially affected by changes in the standard, as well as those that are discussed in more detail in this paper.

Type-based verification of cryptographic constructions consists of a series of *idealization steps*. The numbers in Fig. 1 indicate the order of idealization. For instance, in step 3 the master secret (or in TLS 1.3 the finished secret) derivation must be idealized before idealization of the *KDF/MAC* module used to compute the finished message. In TLS 1.2 the latter module is also used to derive subsequent record keys, while this is no longer the case in TLS 1.3. Each step is conditioned by cryptographic assumptions and typing conditions, to ensure its computational soundness; it enables us to replace a concrete

module implementation by a variant with stronger security properties; this variant can then be re-typechecked, to show that it implements a stronger *ideal interface*, which in turn enables further steps. Finally, we conclude that the idealized variant of our TLS implementation is both perfectly secure (by typing) and computationally indistinguishable from our concrete TLS implementation.

Returning to TLS 1.3, we claim that miTLS addresses the new application interface challenge, as it must. By design, it clearly exposes all necessary pre-conditions of its main API functions, and the precise guarantees that are achieved by its security theorem, in order to typecheck. While this means that a well-typed application on top of miTLS has to manage some information from the transport layer (which we try to keep to the bare minimum), it is also less likely to make dangerous decisions (such as confusing 0-RTT and 1-RTT data).

Outline §II gives an high-level overview of the new F^* interface of miTLS and its intended security. §III presents the new client and server composite state machine for the handshakes of all versions of TLS (including 0-RTT) and presents a summary of our efforts to model and improve downgrade resilience. §IV discusses ongoing work of porting miTLS to F^* , and illustrates the new stateful verification features by showing how to model length-hiding authenticated encryption in F^* . The latest code, papers, and details are available from <https://mitls.org>.

II. DEFINING SECURITY FOR THE TLS 1.3 API

We outline our new application programming interface (API) for miTLS. Surprisingly, the various TLS standards leave the design of this API to each implementation, with mixed results. Our API is designed to enable a precise formal, statement of security for applications that use TLS. We refer to earlier papers for a detailed presentation of our previous API for miTLS [4, 6]. Although the two APIs are similar in principle, the newer one is more general (notably to cover 0-RTT) and reflects the additional precision of stateful verification with F^* , illustrated in §IV.

Our API strives to hide the internal complexity of TLS; for instance, it hides all key materials, except for long-term keys in certificates. It also abstracts over all handshake messages, and most of its state machine, only letting the application know when handshakes successfully complete. At the same time, our API exposes the flexibility of TLS, notably its cryptographic agility, with support for multiple versions, ciphersuites, and extensions, for 0-RTT, for resumption and renegotiation. It also supports concurrent sessions and connections, as one would expect from any TLS library.

We define application security for TLS by providing a precise, type-based specification of its API. Informally, this corresponds to defining a much simpler, ideal functionality for this API that relies on shared state between clients and servers instead of cryptography for securing communications.

The main object of the API is a connection (Fig. 3) which represents a stateful endpoint of the protocol—either a client or a server. The application uses connections to read and write

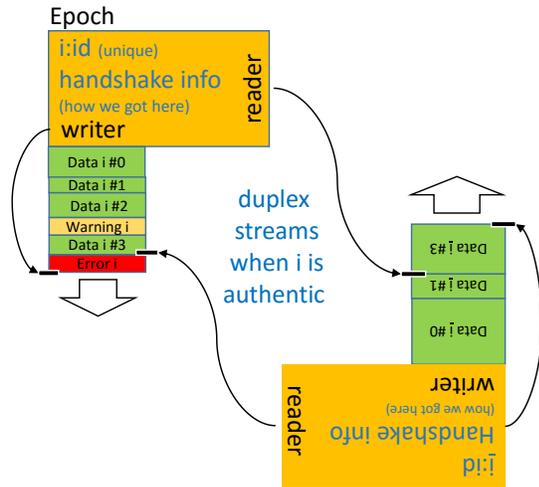


Fig. 2: Ideal state for Streams and Epochs

messages. Each connection goes through a sequence of epochs (Fig. 2), with a new epoch whenever a handshake completes. Each epoch in turn includes two stateful streams, for outgoing and incoming traffic, respectively.

Streams. We first define ideal streams (similar to one-way private channels) that carry sequences of messages from a *writer* to a *reader*. These messages range over application data fragments (in green) and TLS alerts. A stream may be terminated by a closure signal or a final, fatal alert (in red). Each stream has a unique index (i in the figures) that records information on the underlying credentials and algorithms used to derive the keys and protect messages. The actual message contents is given an *abstract type* indexed by i . Intuitively, this means that each stream carries messages of a different type, and that access to their contents can be restricted depending on i . (Hence, type safety prevents mTLS from sending application data in the clear, or on the ‘wrong’ stream.) In particular, the index controls whether the stream provides security, as follows.

- If i provides authentication, then our typed API guarantees that (1) the writer appends, at the end of the stream, application data sent by the application and alert generated by TLS; (2) the reader sequentially reads a *prefix* of the stream contents.
- If i provides confidentiality, then type abstraction ensures that our implementation cannot access the message contents—this is similar to the modelling of semantic encryption whereby secrets are replaced with dummy values in the ideal functionality.

The conditions under which i actually yields integrity and confidentiality depends on the details of key derivation in the handshake. For simplicity, this presentation also omits a discussion of message sizes—our implementation supports length-hiding by additionally indexing application-data messages with their size range, and adding padding such that only the range (not the actual message length) is leaked.

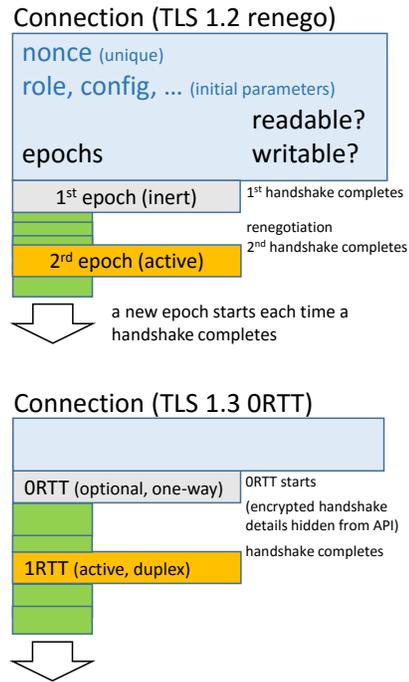


Fig. 3: Ideal state for Connections

Epochs. TLS traffic is structured into successive *epochs*, each supporting duplex communications, that is, including a stream for outgoing messages and a stream for incoming messages. In the figure, we write i and \bar{i} for their indexes, respectively. If these indexes provide authenticity, then there must be a peer epoch that includes the other ends of these streams (represented here upside-down). Fig. 2 represent the state of these streams by attaching a log of outgoing messages to each epoch and a pointer to the outgoing messages of the peer epoch. For example, our epoch has just sent a fatal error, which has not been received yet by its peer.

Connections. Finally, the ideal state of a TLS connection, depicted in Fig. 3, consists of a log of epochs and two flags, *readable* and *writable*, that indicates whether the application can read and write in this state. Only the last epoch is *active* and can be used for sending and receiving messages. The streams in earlier epochs are frozen.

We emphasize that this ideal state is used to specify the overall integrity of the sequence of communications over successive epochs. From a verification viewpoint, this is *ghost state*. Our concrete TLS implementation only keeps the current keys for outgoing and incoming traffic.

For TLS before 1.3, a new epoch starts whenever a handshake complete. For example, in the figure the connection exchanged data on a first epoch, established by a full handshake, and is now exchanging data on a second epoch, established by a renegotiation. Inasmuch as successive handshakes may involve different certificates, some of these epochs may be secure, while others are insecure. Moreover, for earlier versions of TLS, a connection may share secure epochs with different peer connections.

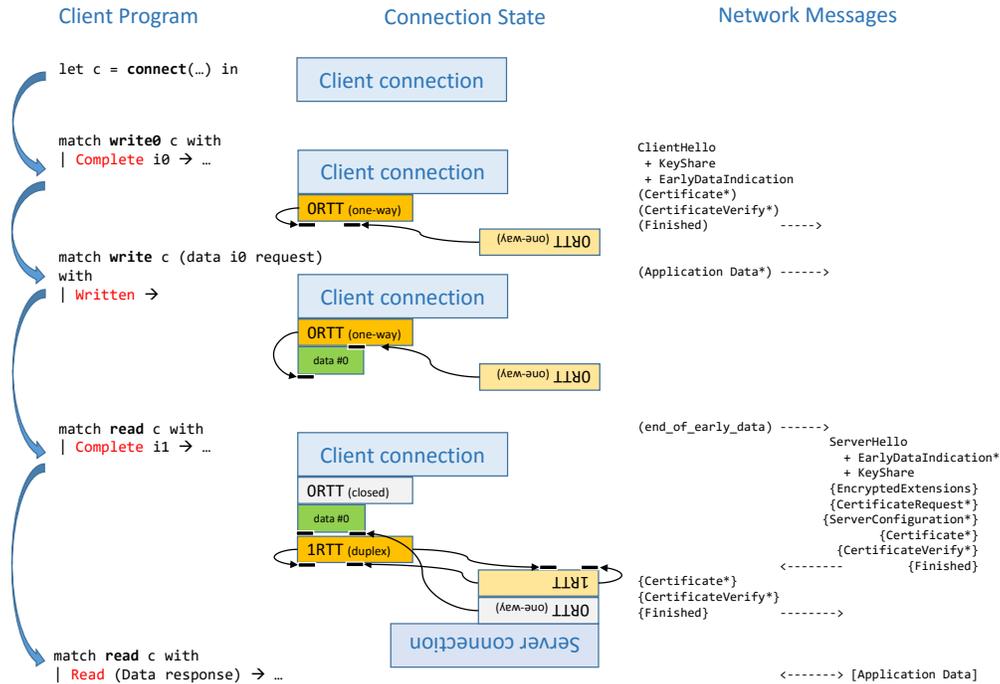


Fig. 4: Sample 0-RTT client run

For TLS 1.3, we have a main, ‘1-RTT’ epoch, established by the main handshake, optionally preceded by a ‘0-RTT’ epoch that only has one stream (from the client to the server). Similarly, those two epochs have different indexes, and may provide different levels of security.

Sample Application Code using 0-RTT. Fig. 4 illustrates a typical sequence of calls performed by a client that sends a (single-fragment) request in the 0-RTT epoch then waits for the responses in the 1-RTT epoch. For simplicity, the code handles only one result for each call to `MITLS`. The figure displays the client code (left), the successive states of its connection (center), and the corresponding TLS network traffic (right).

The client first creates a fresh connection `c` by calling `mitls connect` with its choice of parameters, requiring here TLS 1.3 with 0-RTT. It then calls `write0` on `c` to let TLS internally begin the handshake and return the index `i0` to be used for 0-RTT. The client then *explicitly* turns its request into a message indexed by `i0`, thereby conditioning the security of the request to the security provided by `i0`, and calls `write` to send it in the current, 0-RTT epoch. The client then calls `read`, which triggers the end of 0-RTT; TLS internally sends `end_of_early_data`, waits for the server’s messages flight up to `Finished`, and completes the 1-RTT handshake. Although most of the handshake is encrypted using an internal, ‘handshake epoch’, this is irrelevant for the application, hence hidden by our API. Finally, the client gets the index `i1` of the 1-RTT epoch for application data, and is ready to read the response at that index.

An interesting question for the application is whether the completion of 1-RTT acknowledges receipt of all 0-RTT

traffic. We believe it is the case e.g. if both handshakes are secure and the server waits for `end_of_early_data` before sending its `Finished` message.

III. STATE MACHINES AND DOWNGRADE RESILIENCE

One of the key challenges for deploying TLS 1.3 is that it interoperates with legacy peers that support older versions of the protocol. Thus, it is not enough to implement and verify (say) the TLS 1.3 handshake in isolation. We need a security theorem that provides strong guarantees for TLS 1.3 peers even if they support older protocol versions and weaker ciphersuites. Furthermore, such TLS peers would likely use the same certificates (and other pre-shared credentials) across all versions, opening the way to potential state machine bugs, version downgrades, and cross-protocol attacks.

We briefly describe the composite handshake state machine we are implementing and verifying in `MITLS`, and discuss how we prove that TLS 1.3 prevents downgrading to older protocol versions. Conversely, we do not discuss our more detailed internal API between handshake and record-layer modules.

A. Handshake State Machine for TLS 1.0-1.3

The composite state machine implemented by TLS 1.3 clients is depicted in Figure 5. (We omit the dual server state machine.) The handshake begins with the client in an idle state (`ClientInit`) in some previous (possibly null) *epoch*. The client sends a `ClientHello` and waits for the `ServerHello`. If the `ClientHello` asked for TLS 1.3 then it contains the client’s ephemeral public key shares in an extension, and the client remembers the corresponding private values in



Fig. 5: Composite client state machine for TLS 1.0-1.3, covering full and abbreviated handshakes, client authentication, and 0-RTT early data. Messages labeled with * may be skipped depending, for example, on the negotiated handshake *mode*.

the variable *eph*. Moreover, a TLS 1.3 client may send 0-RTT client authentication messages and application data while waiting for the *ServerHello*.

Once the *ServerHello* is received, the client knows the negotiated handshake *mode*, which includes the protocol version, the key exchange method (RSA / (EC)DHE / PSK or resumption), authenticated encryption algorithm, and whether its 0-RTT messages were accepted.

In TLS 1.0-1.2 full handshakes, the client then processes the remaining server flight of messages (up to *ServerHelloDone*), sends its own flight (up to *ClientFinished*) and then waits for the server's final flight (*ClientFinishedSent*). At this point the client has computed the next epoch's keys, and installed outgoing record encryption keys (after *ClientCCS*). When the client receives the server's next flight (up to *ServerFinished*), it installs incoming record keys (after *ServerCCS*) and completes the handshake, returning to *ClientInit* with the new epoch.

Abbreviated handshakes in TLS 1.0-1.2 are simpler; after the *ServerHello*, the client can immediately compute the new epoch's keys from the resumed session's master secret. It receives the server's flight (up to *ServerFinished*) and sends its own flight (up to *ClientFinished*) to complete

the handshake.

The TLS 1.3 1-RTT handshake is shown on the left of Figure 5. The client turns on handshake encryption in both directions by installing new record keys; it then receives the server's flight (up to *ServerFinished*) and sends its own flight (up to *ClientFinished*). It then installs new application data encryption keys in both directions before completing the epoch and returning to *ClientInit*.

For brevity, the state machine elides additional conditions that would be needed to fully define valid handshake traces. For example, renegotiation is deprecated in TLS 1.3; hence, one can only go round the state machine once for each connection. There are many restrictions on which messages must appear in each handshake mode and which are truly optional. For example, in the regular (EC)DHE 1-RTT handshake, the *ServerCertificate* and *ServerCertificateVerify* are required. In PSK handshakes, these messages are optional. If the server wishes to support 0-RTT in subsequent handshakes, it may send a *ServerConfiguration*. If the server desires client authentication it sends a *CertificateRequest*, and then the client may respond with a *Certificate* and *CertificateVerify*.

Our implementation of this state machine shares its message processing code between different branches, but it carefully separates the cryptographic processing of each handshake mode, both to enable their modular verification and in order to prevent state machine bugs that plague TLS implementations. Verification of our implementation is ongoing and we will report on its progress at the TRON workshop. In the remainder of this section, we focus on a specific aspect of this verification—preventing version downgrades from TLS 1.3 to TLS 1.0-1.2.

B. Downgrade resilience for TLS 1.3

TLS has suffered from downgrade attacks since its earliest versions [16], and despite built-in protections (such as exchanging MACs over the full handshake transcript), new downgrade attacks continue to appear. A typical example is the recent Logjam attack [1]. Suppose a client and server both prefer to use TLS 1.2 in combination with a ciphersuite such as `DHE-RSA-AES256-GCM-SHA384` with a 2048-bit Diffie-Hellman group. Under normal circumstances, a TLS connection between the client and server will negotiate this ciphersuite. However, if a server also offers an insecure 512-bit Diffie-Hellman group from the `DHE-EXPORT` ciphersuite, and the client does not enforce a minimum group size, then an adversary can force both of them to downgrade to this mode. The transcript MACs do not help, since they are calculated with keys derived from the 512-bit Diffie-Hellman shared secret. Logjam demonstrates that TLS versions up to 1.2 are not downgrade resilient since *the algorithms used for downgrade protection may themselves be downgraded*. Can we prevent this weakness from affecting TLS 1.3?

In [7] (currently under submission, but we plan to release a public version by TRON), we formally define our notion of downgrade resilience and evaluate various protocols, including TLS 1.2 and 1.3 against this definition. In particular, we prove that TLS 1.3 augmented with recent downgrade countermeasures prevents version downgrades. In the following, we informally summarize its results on TLS.

If we consider TLS 1.3 in isolation, many downgrade attacks are prevented by the new protocol. Server signatures in TLS 1.3 cover the full transcript, and hence they cover the full client and server hello messages. This foils many of the downgrade attacks on TLS 1.2 including Logjam: as long as the client only accepts strong signature and hash algorithms and honest public keys from the server, it cannot be downgraded to a weaker ciphersuite. Moreover, the signature enforces agreement on the chosen ciphersuite. Hence, draft 11 of TLS 1.3 appears to provide strong downgrade protection for the ciphersuite.

However, TLS 1.3 clients and servers also support TLS 1.2, then the network attacker can simply downgrade the version to TLS 1.2 and mount the same attacks as before. All the downgrade attacks on TLS 1.2 will then be inherited by TLS 1.3. Another similar version downgrade attack is also possible, by relying on the fallback mechanism implemented by many web browsers. The attacker stops TLS 1.3 connections and allows only TLS 1.2 connections to go through. Even if the

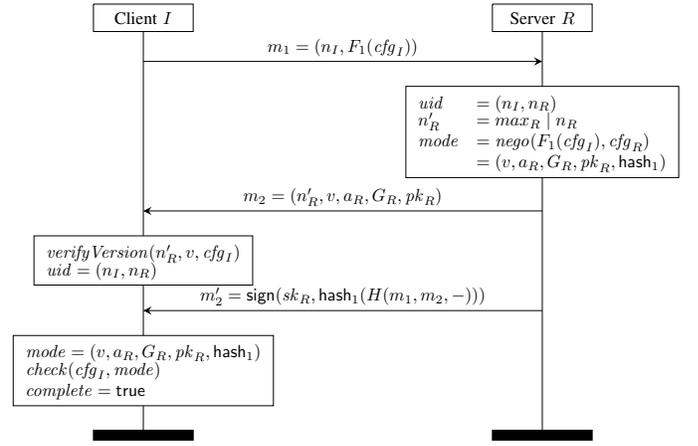


Fig. 6: TLS13-sub: A version downgrade fix for TLS 1.3

endpoints implement the fallback protection mechanism [13], the attacker may be able to use one of the downgrade attacks in TLS 1.2 (such as Logjam) to break the protection.

We prevent these attacks by proposing a version downgrade countermeasure (shown in Fig. 6), which is in the process of incorporation into the next TLS 1.3 draft. TLS 1.3 servers always include their highest supported version number (max_R) in the server nonce, even when they choose a lower version such as TLS 1.0. Therefore, if a TLS 1.3 client receives a `ServerHello` message that sets the negotiate version to TLS 1.0, even though the server nonce indicates that the server supports TLS 1.3, then the client will refuse the connection. Tampering with the server nonce will be detected by the server signature, which includes the server nonce in all versions of TLS. Hence, this countermeasure protects TLS 1.3 connections from being downgraded as long as either the client or the server supports only signature-based ciphersuites in all versions of TLS. The draft paper [7] contains a full proof of downgrade resilience for the simplified sub-protocol depicted in Figure 6.

If the client and server both support non-signature key exchanges such as RSA in TLS 1.0-1.2, then this downgrade protection does not apply. However, we note that TLS 1.3 servers are already encouraged not to support RSA key exchanges due to other cross-protocol attacks [10].

IV. PROGRAMMING AND VERIFYING TLS 1.3 IN F*

The `MITLS` code is carefully structured and programmed to enable its modular cryptographic verification by typing. Prior versions of `MITLS` were programmed in F# and verified using F7, a refinement typechecker. The mostly functional subset of F# supported by F7 dictated that `MITLS` be programmed in state-passing style, leading both to inefficiencies as well as an indirect, axiom-heavy style of specification.

To address these limitations, we are now programming and verifying `MITLS` in F*, a new verification-oriented dialect of ML. F* differs from F7 in several ways.

- F* is a full-fledged programming language, rather than a verification tool for F#, which enables a tighter integra-

tion of code and specification. Programs writtend in F* and verified by its typechecker are then compiled, via multiple backends, to either OCaml or F# for execution.

- The logic provided by F* for specifying and verifying programs is significantly more expressive. At its core, F* is a dependently typed language whose expressiveness is comparable to interactive theorem provers like Coq. Beyond what is expressible in Coq, F* provides effectful computations with specifications that can speak about effects via a Hoare-like program logic with pre- and post-conditions. The support for effects allows us to eschew the use of the purely functional style mandated by F7.
- Like F7, F* provides support for semi-automated proofs by integrating its typechecker with an SMT solver, although unlike F7’s first-order logic, F* SMT integration also covers its dependently typed, higher-order logic. Beyond SMT-based automation, F* also allows for interactive proofs provided by the user in cases where proofs are too intricate to be discovered automatically by SMT.

In the remainder of this section, we give a flavor of the style of verification we are now using. We focus on discussing a subset of the length-hiding authenticated encryption (LHAE) functionality in TLS, showing the use of shared stateful logs for the ideal functionality relating readers and writers. While this particular case is simple and easily explainable in the format of this short paper, the pattern it uses generalizes to other levels of the TLS record layer, including our precise modeling of streams, epochs, and connections outlined in §II.

A model of LHAE in F* There are three main ingredients in our methodology for type-based verification of cryptographic protocols. First, we use logical refinements to capture invariants of the data structures used in a protocol. Second, we rely on type abstraction to hide representations of confidential data and ensuring its secrecy. Finally, we model security using games, modeling the adversary as a client program that interacts with our verified functionality via its typed interface—hence, each function exposed in the interface defines an oracle. Our theorem guarantees in particular that, for any such adversary, the invariants of our functionality are preserved. To confirm that our verification result meaningfully captures properties of interest, one need only review the invariants provided by the interface.

To model LHAE, we first define the type of keys, $\text{key } i$, where the parameter (the index i) identifies various protocol parameters, including, e.g., the choice of algorithms being used. By explicitly mentioning the index i in the type of keys, we ensure that keys with different protocol parameters can never be confused.

```
private type key i = b:bytes{length b=keySize (alg i)}
```

(When parameters such as i are used only for specification purposes, they are erased by F* after typechecking.) The definition of $\text{key } i$ is a refinement type of the form $x:t\{\phi\}$, which can be thought of as restricting the type t to those elements x that validate the formula ϕ . The definition of $\text{key } i$ states that it is a sequence b of concrete bytes whose length matches

the keySize for the algorithm, $\text{alg } i$, corresponding to i —this level of details matters in our model of key derivation. By marking the type `private`, we indicate that clients of this module cannot directly access the raw bytes of a key.

Following a similar pattern, we define (below) types $\text{iv } i$ for static initialization vectors and $\text{cipher } i$ for ciphertexts. Since ciphers are not secret, the type $\text{cipher } i$ is not marked `private`. Also, rather than pinpointing the size of the cipher text, the refinement in $\text{cipher } i$ constrains its length—this matters in our model of block ciphers.

```
private type iv i = b:bytes{length b=staticIVSize (alg i)}
type cipher i = c:bytes{ validCipherlength i c }
```

As explained in §II, our functionality is parameterized by an abstract type $\text{plain } i \text{ rg}$ of secret plaintexts represented as bytestrings whose lengths fits in the range rg . The type plain , like key and iv is abstract, ensuring that the raw bytes of the plaintext is not accessible to the adversary. We also define the type $\text{!plain } i \text{ c}$, for (partially) *length-hidden* plaintexts, a function of the identifier i and, importantly, the ciphertext.

```
type !plain i c = plain i (cipherRangeClass i (length c))
```

Intuitively, $\text{!plain } i \text{ c}$ is the type of successful decryptions, whose range is $\text{cipherRangeClass } i \text{ (length } c)$, an interval determined by the choice of algorithm and the length of the (public) ciphertext c . As such, an adversary observing the ciphertext learns some information about the length of the plaintext, i.e., that it falls somewhere in the aforementioned interval.

We now turn to the ideal functionality of authenticated encryption by modeling the reader and writer using a shared log of entries.

Ideally, instead of encrypting a plaintext, the writer encrypts some dummy values, and appends an entry to the log that maps the resulting ciphertext to the message; and instead of decrypting a ciphertext, the reader does a table lookup in the log to retrieve the correspondong plaintext, or returns a decryption error. Crucially, these operations can be performed without accessing the abstract plaintext. An entry in the shared log is a pair $\text{Entry } c \text{ p}$ containing the ciphertext and length-hidden plain text associated with that cipher.

```
type entry i = ! Entry: c:cipher i → p:!plain i c → entry i
```

The shared state of the reader and writer oracles is described by the types $\text{state } i \text{ Reader}$ and $\text{state } i \text{ Writer}$, respectively. In each case, the state contains the key, the iv , some internal state (e.g., for stream ciphers), and the shared log.

```
type reader i = state i Reader
and writer i = state i Writer
and state i rw = ! State:
  region:rid
  → peer:rid{disjoint region peer}
  → key:key i
  → iv :iv i
  → internal:rref region internal_state
  → log:rref (if rw=Reader then peer else region) (seq (entry i))
  → state i rw
```

To keep separate the state of multiple instances of the LHAE functionality, and also to separate the internal state of the

reader and writer from their shared state, we make use of F*'s library for separating the heap into regions for verification purposes. In particular, we associate two disjoint regions, `region` and `peer` with each instance of LHAE—the region of the reader is the writer's peer and vice versa. The type `rref region state` indicates that the internal state of a reader or writer resides in its region. Likewise, the shared log has type `rref (if rw=Reader then peer else region) (seq (entry i))`, indicating that it always resides in the writer's region.

To encrypt a plaintext `p`, the writer calls `encrypt i w r p`. The type of `encrypt`, shown below, requires that `w` be a suitable writer for the identifier `i`, and that `p` be a plaintext whose length fits into a single TLS fragment. (Fragmentation occurs before calling `encrypt`.)

```
val encrypt: i:gid
  → w:writer i
  → r:range{fst r = snd r ∧ snd r ≤ TLSPlaintext_frag_max}
  → p:plain i r
  → ST (cipher i)
(ensures (fun h0 c h1 →
  modifies {w.region} h0 h1 ∧
  validCipherLength i c ∧
  sel h1 w.log = sel h0 w.log ++ Entry c p))
```

When satisfying these pre-conditions, the specification of `encrypt i w r p`, via its `ensures`-clause, guarantees that the initial heap `h0`, the resulting ciphertext `cipher i`, and final heap `h1` are related in the following way: (1) `h0` and `h1` may differ at most in the contents of the writer's region `w.region`; (2) the length of the ciphertext is valid; and (3) the shared log is extended with a single entry that maps the returned ciphertext `c` to the plaintext `p`.

To decrypt, dually, we call `decrypt i r c`, with some decryption key `reader i` and ciphertext `cipher i` suitable for `i`.

```
val decrypt: i:gid
  → r:reader i
  → c:cipher i
  → ST (option (lhplain i c))
(ensures (fun h0 res h1 →
  modifies {r.region} h0 h1 ∧
  (authld i ⇒
    let log = sel h0 r.log in
    (∀ j. ¬matches c log[j] ∧ res=None )
    ∨ (∃ j. matches c log[j] ∧ res=Some log[j].p))))
```

The specification above ensures that `decrypt` returns a result `res : option (lhplain i c)` related to the initial memory `h0` and the final memory `h1` as follows: (1) `h0` and `h1` differ at most in the contents of `r`'s region—notably, the log is unchanged; (2) if the instance identified by `i` provides integrity protection (i.e., we have `authld i`), then either the log does not contain `c` and the decryption returns `res=None`, or the log returns a correct plaintext recorded in the log. (In both cases, the integer `j` ranges over positions in the log.) In combination, the postconditions of `encrypt` and `decrypt` fully specify an ideal functionality that operates on the shared log instead of operating on actual plaintexts. In turn, those postconditions suffice to verify the record layer and, ultimately, our new API for TLS 1.0–1.3.

REFERENCES

- [1] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann, “Imperfect forward secrecy: How Diffie-Hellman fails in practice,” in *22nd ACM Conference on Computer and Communications Security*, 2015, pp. 5–17.
- [2] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. Zinzindohoue, “A Messy State of the Union: Taming the Composite State Machines of TLS,” in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 535–552.
- [3] K. Bhargavan, C. Fournet, R. Corin, and E. Zalescu, “Cryptographically verified implementations for tls,” in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 459–468. [Online]. Available: <http://doi.acm.org/10.1145/1455770.1455828>
- [4] 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.
- [5] 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 *IEEE Symposium on Security & Privacy 2014 (Oakland'14)*. IEEE, 2014.
- [6] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and S. Zanella-Béguelin, “Proving the TLS handshake secure (as it is),” in *Advances in Cryptology – CRYPTO 2014*, 2014, pp. 235–255.
- [7] K. Bhargavan, C. Brzuska, C. Fournet, M. Green, M. Kohlweiss, and S. Zanella-Béguelin, “Downgrade resilience in key-exchange protocols,” 2015, unpublished, under review. Confidential draft available at http://www.mitls.org/wsgi/md/papers/downgrade_resilience_draft.pdf.
- [8] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.3,” Internet Draft, 2015.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] H. Krawczyk and H. Wee, “The OPLS protocol and TLS 1.3,” *Manuscript, September*, 2015.
- [13] B. Moeller and A. Langley, “TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks,” IETF RFC 7507, 2015.
- [14] 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 *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Jul. 2016, available at <https://www.fstar-lang.org>.
- [15] B. Tackmann, “Augmented secure channels and the goal of the TLS 1.3 record layer,” in *Provable Security: 9th International Conference, ProvSec 2015, Kanazawa, Japan, November 24-26, 2015, Proceedings*, vol. 9451. Springer, 2016, p. 85.
- [16] 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.