DOI:10.1145/3023357

A Messy State of the Union: Taming the Composite State Machines of TLS

By Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue

Abstract

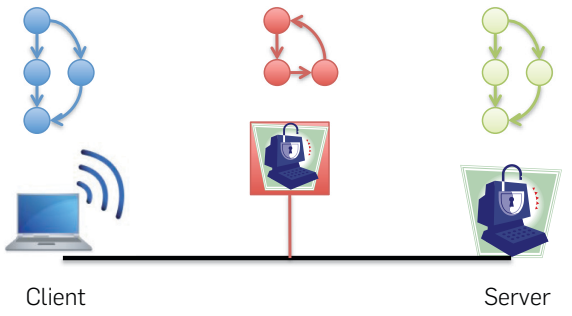
The Transport Layer Security (TLS) protocol supports various authentication modes, key exchange methods, and protocol extensions. Confusingly, each combination may prescribe a different message sequence between the client and the server, and thus a key challenge for TLS implementations is to define a composite state machine that correctly handles these combinations. If the state machine is too restrictive, the implementation may fail to interoperate with others; if it is too liberal, it may allow unexpected message sequences that break the security of the protocol. We systematically test popular TLS implementations and find unexpected transitions in many of their state machines that have stayed hidden for years. We show how some of these flaws lead to critical security vulnerabilities, such as FREAK. While testing can help find such bugs, formal verification can prevent them entirely. To this end, we implement and formally verify a new composite state machine for OpenSSL, a popular TLS library.

1. TRANSPORT LAYER SECURITY

Transport Layer Security (TLS),[13] previously known as Secure Sockets Layer (SSL), is a standard cryptographic protocol widely used to secure communications for the web (HTTPS), email, and wireless networks. Figure 1 depicts the common usage of TLS and its threat model. Following the protocol, a client and a server exchange messages to establish a secure channel across an insecure network. Meanwhile, a network attacker can intercept these messages, tamper with them, and inject new messages to confuse the two. Additionally, the attacker may control some malicious clients and servers that are free to deviate from the protocol. The goal of TLS is to ensure the integrity and confidentiality of data exchanged between honest clients and servers, despite the best efforts of attackers.

TLS offers a large choice of cryptographic algorithms and protocol features to accommodate the needs of diverse applications. Each TLS connection consists of a channel establishment protocol, called the *handshake*, followed by a transport protocol, the *record*. During the handshake, the client and server negotiate which algorithms and features they wish to use. For example, the client and server may be authenticated with certificates, or with pre-shared keys, or may remain anonymous; the key exchange may use Ephemeral Diffie–Hellman or RSA Encryption; the record protocol may encrypt sensitive application data using AES-GCM or RC4.

Figure 1. TLS threat model: network attacker aims to subvert client–server exchange.



If a connection uses a secure key exchange and a strong record encryption scheme, security against network attackers can be reduced to the security of these building blocks. Indeed, recent works provide cryptographic proofs for some of the key exchange methods[7, 16, 19, 22] and encryption schemes[27] used in TLS. However, not all of choices offered by TLS have been proved secure; in fact, many of them are obsolete and some are even known to be broken. Still, TLS client and servers continue to support old protocol versions, extensions, and ciphersuites for interoperability reasons. For example, TLS 1.0 offered several deliberately weakened ciphersuites to comply with US export regulations at the time. These ciphersuites were explicitly deprecated in TLS 1.1 but are still supported by mainstream implementations.

Even if the client and server support weak protocol modes, the TLS handshake is designed to negotiate and execute the strongest protocol that they both support. Hence, if *one* party is configured to accept only strong parameters, then its connections are expected to be secure, even if its peer supports other weaker modes. However, this guarantee depends on the implementation correctly composing different protocol modes, a task that is surprisingly tricky.

1.1. Composing protocol state machines

Each TLS client and server implements a state machine that keeps track of the protocol being run: which messages

The original version of this paper was published in *IEEE Symposium on Security and Privacy*, 2015, pages 535–552.

have been sent and received, what cryptographic materials have been computed, which messages are expected next, etc. The state machine for each individual ciphersuite is specified in the standard, but the task of writing a composite state machine for multiple ciphersuites is left to each implementation.

Figure 2 depicts a simplified TLS handshake for some (fictional) ciphersuite, as seen from the viewpoint of the client. On the left, the client first sends a `Hello` message containing a list of supported ciphersuites to the server. The server chooses a ciphersuite and responds with two protocol messages, `A` and `B`, to establish a session key for this ciphersuite. The client completes the handshake by sending a `Finished` message to confirm knowledge of the session key. At the end of the handshake, both the client and server can be sure that they have the same key and that they agree on the ciphersuite. Now suppose we wish to support a new ciphersuite, such that the client receives a different pair of messages, `C` and `D`, between `Hello` and `Finished`. To reuse our well-tested code for processing `Hello` and `Finished`, it is tempting to extend the client state machine to receive either `A` or `C`, followed by either `B` or `D`. This naive composition implements both ciphersuites, but it also enables unintended sequences, such as `Hello`; `A`; `D`; `Finished`. In TLS, clients and servers authenticate the full message sequence at the end of the protocol (in the `Finished` messages) and, since no honest server would send `D` after `A`, allowing extra sequences at the client may seem harmless.

However, a client that accepts this message sequence is actually running an unknown handshake protocol, with *a priori* no security guarantees. In our example, the code for processing `D` expects to run after `C` has been received. If `C` contains the server's signature, then accepting `D` without `C` may allow a crucial authentication step to be bypassed. Furthermore, the code for processing `D` may accidentally use memory that should have been initialized while processing `C`. Such memory safety bugs can lead to dangerous attacks such as HeartBleed,[a] which exposed the server's internal state and private keys to remote attackers.

1.2. Testing for state machine flaws

In Section 2, we describe a methodology for systematically testing whether a TLS client or server correctly implements the protocol state machine. We find that many popular TLS implementations exhibit composition flaws like those described above, and consequently accept unexpected message sequences. While some flaws are benign, others lead to critical vulnerabilities that a network attacker can exploit to bypass TLS security. In Section 3, we show how a network attacker can impersonate a TLS server to a buggy client, either by simply skipping messages (SKIP) or by factoring the server's export-grade RSA key (FREAK). These attacks were responsibly disclosed and have led to security updates in major web browsers, servers, and TLS libraries.

1.3. Formally verifying state machine code

We have seen that the security of a TLS implementation depends crucially on its correct implementation of the protocol state machine. Testing can help find some bugs, but once these have been fixed, how can we be sure that the code does not have other hidden flaws? We advocate the use of formal verification to prove the absence of any state machine flaws. In Section 4, we present a new state machine implementation for OpenSSL that supports all commonly enabled ciphersuites, versions, and extensions. Using Frama-C,[11] we verify that our code conforms with a logical specification of the TLS protocol state machine.

1.4. Online materials

We refer to https://mitls.org/pages/attacks/SMACK for additional details, including security testing tools, a summary of vulnerability disclosures and security updates, our state machine code for OpenSSL, and related verification work on TLS.

2. TESTING THE TLS STATE MACHINE

The TLS standard[14] does not define a state machine. Instead, it specifies a collection of message sequences, one for each handshake protocol mode. Other specifications add new ciphersuites, authentication methods, or protocol extensions; they typically define their own message sequences, reusing the message formats and mechanisms of TLS, and it is left to the implementation to design a state machine that can account for all these sequences.

2.1. A TLS state machine

We propose a reference state machine for TLS by adopting and extending the one used in the miTLS verified implementation,[6] based on a careful reading of the standard. Figure 3 depicts a simplified version of this state machine, which can be read from the viewpoint of the client or the server. Each state refers to the last message sent or received; messages prefixed by `Client` are sent by the client; those prefixed by `Server` are sent by the server. Transitions, shown as black arrows, indicate the order in which these messages are expected. When two transitions are possible, each is labeled by the condition under which it is allowed. (Dotted arrows are flawed transitions; they will be explained in Section 3.) The state machine depicted here covers the common usages of TLS on the web, a small but important subset of the full protocol. The figure only shows message sequences; it does not detail message contents, local states, or cryptographic

[a] https://heartbleed.com.

Figure 2. Unsafe composition of two protocol state machines.

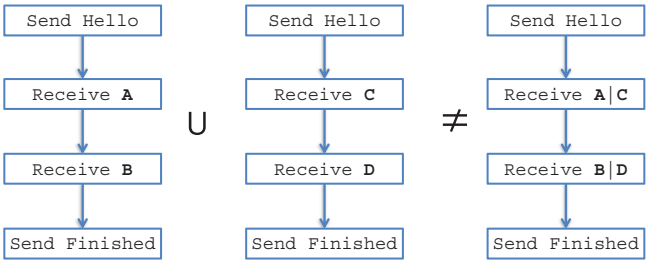
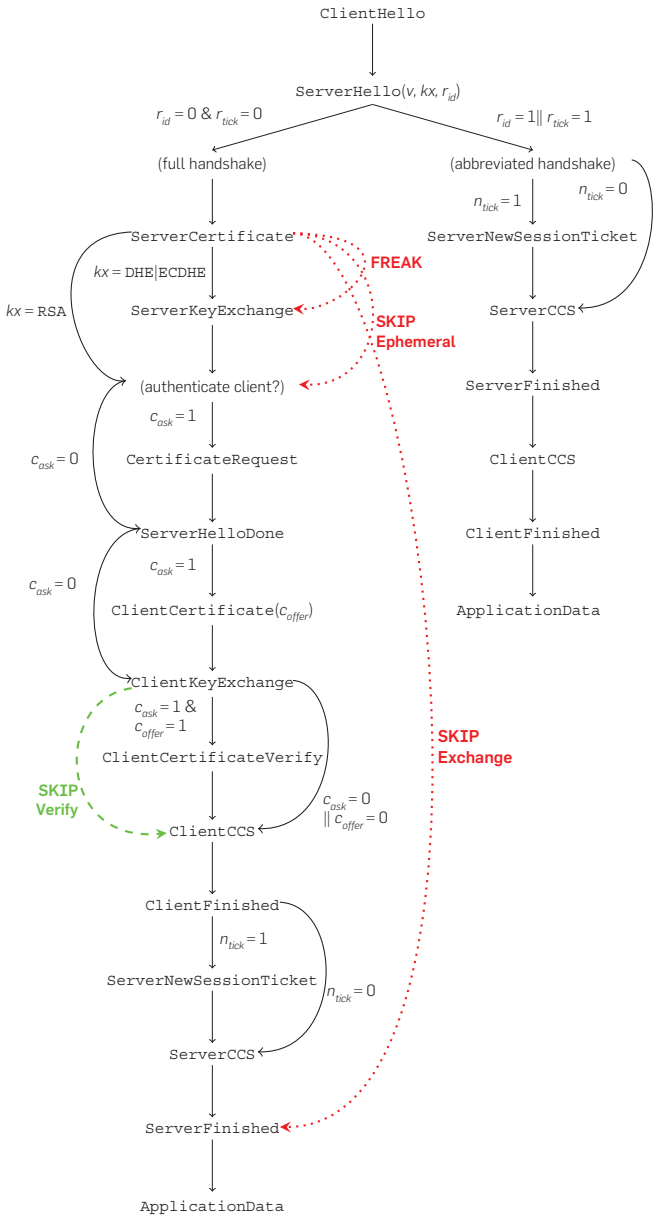


Figure 3. State machine for commonly used TLS configurations. Paths in the graph represent valid message sequences. Each node indicates the last message sent or received. Black arrows indicate the order in which these messages are expected; labels specify conditions under which the transition is allowed. Dashed arrows on the left show example incorrect transitions found in mainstream TLS servers; dotted arrows on the right show incorrect transitions found in TLS clients. The executed message sequence depends on the negotiated protocol version $v \in$ {TLSv1.0, TLSv1.1, TLSv1.2}; key exchange $kx \in$ {RSA, DHE, ECDHE}, and optional features such as fast session resumption (r_{id}, r_{tick}), client authentication (c_{ask}, c_{offer}), and session tickets (n_{tick}).



computations.

Each TLS connection begins with either a full handshake or an abbreviated handshake (also called a resumption). Full handshakes consist of four message flights: the client first sends a `ClientHello`; the server responds with a series of messages from `ServerHello` to `ServerHelloDone`; the client then sends messages ending with `ClientFinished`; and the server completes the handshake by sending messages ending with `ServerFinished`. The `ServerHello` includes negotiated protocol parameters, such as the version (v) and key exchange method (kx), that determine the rest of the handshake. The `Finished` messages include transcripts of all prior handshake messages, authenticated using the new keys established by the handshake. Before sending them, the client and the server send a change cipher spec (`CCS`) message to signal the use of the new keys. Once the handshake is complete, the client and the server may exchange streams of `ApplicationData` messages.

The server is identified by a certificate sent in the `ServerCertificate` message. In ephemeral Diffie–Hellman handshakes, the server proves knowledge of the certificate's private key by signing the `ServerKeyExchange` that carries its Diffie–Hellman public key. In the `RSA` handshake, instead, it uses the private key to decrypt the `ClientKeyExchange` message. In both key exchanges, upon receiving a `CertificateRequest` from the server, the client may optionally send a `ClientCertificate` and use its private key to sign the message transcript so far in the `ClientCertificateVerify`. The state variables c_{ask} and c_{offer} track whether client authentication was requested and accepted.

Abbreviated handshakes rely on shared secrets established in a previous full handshake. The server may store secrets either in a server-side cache or in a client-side session ticket. The server sends a `ServerHello` indicating which kind of resumption will be used, and then goes straight to `ServerCCS` and `ServerFinished`. The client immediately completes the handshake by sending its `ClientCCS` and `ClientFinished` messages.

To what extent does our reference state machine correspond to the one implemented in any given TLS library? For miTLS, we have a type-based proof that its code conforms to this state machine. Next, we investigate mainstream implementations, such as OpenSSL, by systematically testing for deviations from our state machine.

2.2. Generating deviant traces

We first describe the message sequences we use to search for state machine bugs. Let σ be a sequence of messages, m a message, and $\sigma; m$ their concatenation. We write $\sigma \leq \tau$ when σ is a prefix of τ. We write $m \sim m'$ when m and m' have the same message type, but different parameters; for instance when both are `ServerHello` messages selecting different ciphersuites. We also lift $\sim$ from messages to traces. Let V be the set of prefixes of valid traces allowed by the state machine outlined in Figure 3. A *deviant trace* is a minimal invalid trace: $\sigma; m$ is deviant when $\sigma \in V$ but $\sigma; m \notin V$.

Deviant traces are useful for systematically detecting bugs, because a compliant implementation should accept σ and then reject m. If it accepts m, it has a bug. This does not necessarily mean that it has an exploitable security vulnerability: an exploit may involve several carefully crafted messages after the deviant trace. Hence, once we identify an implementation accepting a deviant trace, we look into its source code to learn more about the

cause of the state machine bug.

The set of deviant traces is large (and even infinite unless we bound the number of renegotiations allowed), so we automatically generate a representative, finite subset using three heuristic rules:

Skip If $\sigma; m; n \in V$ and $\delta = \sigma; n \notin V$, test δ. Thus, for every prefix of a valid sequence, we skip a message if it is mandatory. For example, `ClientHello; ServerHello(kx=DHE); ServerKeyExchange` is a trace that skips the `Certificate` message. (Pragmatically, we also skip several messages within flights, but not their last messages, as otherwise the peer is deadlocked.)

Hop Let $\sigma; m \in V$ and $\sigma'; n \in V$. If $\sigma \sim \sigma'$, $m \neq n$, and $\delta = \sigma; n \notin V$, test δ. Thus, if two valid traces have the same prefix, up to their parameters, and they differ on their next messages, we create a deviant trace from the context of the first trace and the next message of the second trace. For example, `ClientHello; ServerHello(kx=RSA); Certificate; ServerKeyExchange` is a trace that sends an unexpected `ServerKeyExchange` by hopping from RSA to Diffie–Hellmann key exchanges.

Repeat If $\sigma; m; \sigma' \in V$ and $\delta = \sigma; m; \sigma'; m \notin V$, test δ. Thus, we resend any message that appears in a valid trace at any subsequent invalid position. For example, `ClientHello; ServerHello; ...; ServerHelloDone; Client-Hello` is a trace where the `ClientHello` message is repeated in the middle of a handshake, making it invalid.

An advantage of generating deviant traces from these rules is that, when a trace is accepted by an implementation, it is relatively simple to track the corresponding state machine bug by manual code review. We also experimented with randomly generated deviant traces, but their manual interpretation was more time-consuming and hence less effective.

2.3. Running deviant traces with FlexTLS

As can be expected, generating arbitrary sequences of well-formed messages is hard. By design, each message in a protocol depends on previously exchanged values, and must pass many basic checks before being accepted by the state machine—after all, TLS implementations are meant to comply with the protocol. At the very least, we need to provide reasonable defaults for any missing values, for instance when keys are needed to format a message and yet the peer's input to the key derivation is not available yet.

To this end, we develop FlexTLS, a tool for scripting and prototyping plausible TLS message sequences. To send and receive messages, FlexTLS relies on miTLS. Using this robust, verified TLS library helped us to significantly reduce false positives due, for instance, to malformed messages or incorrect cryptographic processing.

FlexTLS promotes a succinct and purely functional state-passing style, where each line of code typically corresponds to a message being sent or received. Sending messages out-of-order is as simple as reordering lines in the script. FlexTLS handles most of the complexity internally, filling in reasonable defaults for any missing values. For example, if the script sends a `Finished` message immediately after a `ServerHello` message, bypassing the full handshake, FlexTLS would still derive default well-formed connection keys based on empty key exchange values (see Ref.[3] for more detailed examples of FlexTLS scripts).

For each deviant trace, we generate a FlexTLS client or server script that tests its peer by executing the message sequence, which ends by sending a deviant message. According to the standard, the peer should then send an alert (usually `unexpected_message`) and close the connection. If a non-alert message is received, or the peer does not respond, we assume it wrongly accepted the message, and we flag the trace for further investigation. Not all the TLS implementations we tested support all the scenarios and ciphersuites considered in our traces, and some had unusual error behavior, so we instrumented our scripts to automatically classify peer behavior as correct, unsupported, or wrong. For flagged traces, we manually reviewed the code of the TLS peer, and wrote more detailed FlexTLS scripts by hand to expose and exploit the state machine flaw.

3. IMPLEMENTATION FLAWS AND ATTACKS

Using FlexTLS, we tested several mainstream open-source TLS clients and servers for state machine flaws. To ensure maximal support across implementations, we restricted our tests to use TLS 1.0 with RSA and DHE ciphersuites. Table 1 summarizes our experimental results for OpenSSL, GnuTLS, NSS, SecureTransport, Java, Mono, and CyaSSL. Of these, OpenSSL is widely used on servers and on Android phones; NSS is used in many web browsers including Firefox and some versions of Chrome and Opera; SecureTransport is used on Apple devices. Mono and CyaSSL do not support DHE key exchanges, so they are tested on a smaller set of deviant traces. CyaSSL and SecureTransport sometimes tear down the TCP connection when they reject a message, instead of sending a fatal alert as prescribed in the standard, so we filtered out

Table 1. Running deviant traces against mainstream TLS implementations

Library	Key exchange		Traces	Bugs
OpenSSL 1.0.1j	Client	`RSA, DHE`	83	3
	Server	`RSA, DHE`	94	6
GnuTLS 3.3.9	Client	`RSA, DHE`	83	0
	Server	`RSA, DHE`	94	2
SecureTransport 55471.14	Client	`RSA, DHE`	83	3
NSS 3.17	Client	`RSA, DHE`	83	9
Java 1.8.0_25	Client	`RSA, DHE`	71	6
	Server	`RSA, DHE`	94	46
Mono 3.10.0	Client	`RSA`	35	32
	Server	`RSA`	38	34
CyaSSL 3.2.0	Client	`RSA`	41	19
	Server	`RSA`	47	20

such results, and only counted the traces that expose real state machine bugs.

Each bug found by our method corresponds to an unexpected transition in the state machine. For example, Figure 3 shows four bugs we found in various libraries. Extra transitions allowed by clients are depicted as dotted arrows on the right, and those allowed by servers as dotted arrows on the left. Not all such transitions lead to attacks, but in the rest of this section we show how these four transitions can be exploited by an attacker to break the core security guarantees of TLS.

3.1. SKIP exchange (server impersonation)

Our first vulnerability enabled a network attacker to attack TLS clients that used the Java, CyaSSL, or Mono libraries. Our tests found that these client libraries were willing to accept handshakes where the server skips the `ServerCCS` message, thereby disabling encryption for incoming application data. While this is clearly an implementation flaw, it cannot be exploited in isolation; it only becomes an attack when it is combined with a second bug. We also found that Java and CyaSSL clients allowed the server to skip the `ServerKeyExchange` message in Diffie–Hellman exchanges. Since this message normally contains a signature for server authentication, by skipping it, a network attacker can impersonate any server.

Suppose a Java client *C* wants to connect to some trusted server *S* (e.g., PayPal). A network attacker *M* can hijack the TCP connection and impersonate *S*, without any actual interaction with *S*, by sending *S*'s certificate, skipping all messages, notably `ServerKeyExchange` and `ServerCCS`, and directly sending `ServerFinished`. Hence, *M* bypasses the authenticated key exchange: it can now send unencrypted data to *C*, and *C* will interpret it as secure data from *S*.

Practically exploiting the attack required just a bit more attention to implementation details. The Java and CyaSSL client state machines are so liberal that they allow almost all server messages to be skipped. When they receive the `ServerFinished` message, they authenticate it using an uninitialized master secret (since the key exchange was never performed). The Java client uses an empty master secret, a bytestring of length 0, which *M* can easily compute. The CyaSSL client compares the received authenticator with an uninitialized block of memory, so *M* can simply send a bytestring of 12 zeroes, and this will work against any client executed with fresh memory.

In effect, a network attacker can impersonate an arbitrary TLS server *S*, such as PayPal, to any Java or CyaSSL client. Even if the client carefully inspects the received certificate, it will find it to be perfectly valid for *S*. Hence, the security guarantees of TLS are completely broken. Furthermore, all the (supposedly confidential and authenticated) traffic between *C* and *M* is sent in the clear without any protection.

3.2. SKIP verify (client impersonation)

Our tests showed that OpenSSL, CyaSSL, and Mono allow a malicious client to skip the optional `CertificateVerify` message, even after sending a client certificate to authenticate itself. Since the skipped message normally carries the signature proving ownership of that certificate, this bug leads to a client impersonation attack, as follows.

Suppose a malicious client *M* connects to a Mono server *S* that requires client authentication. *M* can then impersonate any client *C* at *S* by running a regular handshake with *S*, except that, when asked for a certificate, it provides *C*'s client certificate instead, and then it skips the `ClientCertificateVerify` message. The server accepts the connection, incorrectly authenticating the client as *C*, allowing *M* to read and write sensitive application data belonging to *C*.

The attack works against Mono as described above, but requires more effort to succeed against other libraries: against OpenSSL, it works only for static Diffie–Hellman certificates, which are rarely used in practice; against CyaSSL, it requires the client to also skip the `ClientCCS` message and then send zeroes in the `ClientFinished` message (like in Section 3.1).

As a result, any attacker can connect to (say) a banking website that uses TLS client certificates to authenticate users. If the website use Mono or CyaSSL, the attacker can login as any user on this website, as long as it knows the user's public certificate. The attack also works if the website uses OpenSSL and allows static Diffie–Hellman certificates.

3.3. SKIP ephemeral (forward secrecy downgrade)

In some settings, a powerful adversary may be able to force a server to reveal its private key (see, e.g., Ref.[27]) and thus impersonate the server in future connections. Still, we would like to ensure that prior connections to the server (before the private key was revealed) remain secret. This property, commonly called *forward secrecy*, is achieved by the DHE and ECDHE ciphersuites in TLS, whereas RSA, DH, and ECDH ciphersuites do not offer this property.

Forward secrecy is particularly important for web browsers that implement the TLS "False Start" feature.[20] These browsers start sending encrypted application data to the server before the handshake is complete. Since the server's chosen ciphersuite (and, in some cases, even the server's identity) has not been authenticated yet, this early application data need the additional protection of forward secrecy.

However, our tests found that NSS and OpenSSL clients allow the server to skip the `ServerKeyExchange` message even in DHE and ECDHE handshakes, which require this message. In such cases, these clients try to use the static key provided in the server certificate as key exchange value, thereby falling back to the corresponding DH and ECDH ciphersuites, without forward secrecy.

Suppose a client based on NSS *C* (such as Firefox) connects to a website *S* authenticated by an ECDSA certificate (such as Google) using an ECDHE ciphersuite. A network attacker *M* can suppress the `ServerKeyExchange` message from *S* to *C*. The client then computes the session secrets using the static elliptic curve key of the server certificate, but still believes it is running ECDHE with forward secrecy, and immediately start sending sensitive application data (such as cookies or passwords) because of False Start. Although the connection never completes (as the client and server detect the message suppression at the end of the handshake), the attacker can capture this False Start encrypted data. As a result, assuming

it eventually obtains the server's private key, the attacker will be able to decrypt this data, thereby breaking forward secrecy.

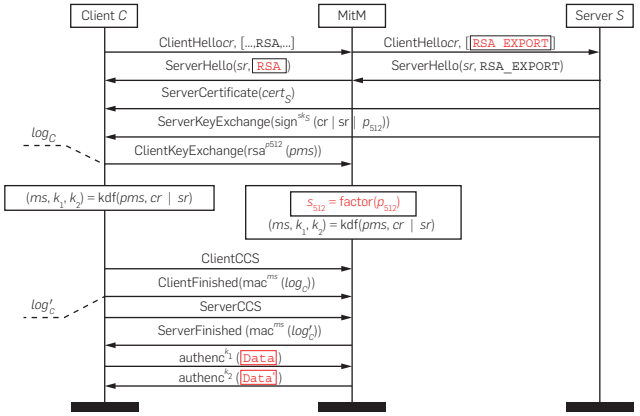
3.4. HOP to RSA_EXPORT (server impersonation)

In compliance with US export regulations before 2000, SSL and TLS 1.0 include several ciphersuites that deliberately use weak keys and are marked as eligible for export. For example, several RSA_EXPORT ciphersuites require that servers send a ServerKeyExchange message with an ephemeral RSA public key (modulus and exponent) whose modulus does not exceed 512 bits. RSA keys of this size were first factorized in 1999[9] and with advancements in hardware are now considered broken. In 2000, export regulations were relaxed, and in TLS 1.1 these ciphersuites were explicitly deprecated. Consequently, mainstream web browsers no longer offer or accept export ciphersuites. However, TLS libraries still include legacy code to handle these ciphersuites, and some servers continue to support them. We show that this legacy code causes a downgrade attack from RSA to RSA_EXPORT.

Our tests showed that OpenSSL, SecureTransport, and Mono accepted ServerKeyExchange messages even during regular RSA handshakes, in which such messages should never be sent. Upon receiving this message, the client would fallback to RSA_EXPORT by accepting the (signed) 512-bit RSA key in the message and using it instead of the full-size public key in the server certificate. This flaw leads to a man-in-the-middle attack, called FREAK, depicted in Figure 4.

Suppose a client *C* wants to connect to a server *S* using RSA, but the server *S* still supports some RSA_EXPORT ciphersuites. *M* intercepts *C*'s RSA handshake to *S* and responds to *C* with *S*'s certificate. In parallel, *M* connects to *S* using RSA_EXPORT and ensures that the client and server nonces on the two connections are the same. Now, *M* forwards *S*'s ServerKeyExchange to *C* and, due to the state machine flaw, *C* accepts this message and overwrites the server's public key with the weaker 512-bit RSA key in this message. Assuming *M* can factor this key (to obtain the private exponent), it can compute the connection keys and complete the connection, hence impersonating *S* at *C*.

Figure 4. FREAK attack: a man-in-the-middle downgrades a connection from RSA to RSA_EXPORT. Then, by factoring the server's 512-bit export-grade RSA key, the attacker can hijack the connection, while the client continues to think it has a secure connection to the server.



FREAK: Factoring 512-bit RSA keys. The main challenge that remains for the attacker is to factor the 512-bit modulus to recover the ephemeral private key during the handshake. First, we observe that 512-bit factorization is now solvable in hours. Second, we note that since computing ephemeral RSA keys on-the-fly can be quite expensive, many implementations of RSA_EXPORT (including OpenSSL) allow servers to precompute, cache, and reuse these public keys for the lifetime of the server (typically measured in days). Hence, the attacker does not need to break the key during the handshake; it can download the key, break it offline, then exploit the attack above for days.

After the disclosure of the vulnerability described above, we collaborated with other researchers to explore its real-world impact. The ZMap team[15] used internet-wide scans to estimate that more than 25% of HTTPS servers still supported RSA_EXPORT, a surprisingly high number. We downloaded the 512-bit ephemeral keys offered by many prominent sites and Nadia Heninger used CADO-NFS[b] on Amazon EC2 cloud instances to factor these keys within hours. We then built a proof-of-concept attack demo that showed how a man-in-the-middle could impersonate any vulnerable website to a client that exhibited the RSA_EXPORT downgrade vulnerability. The attack was dubbed FREAK—factoring RSA_EXPORT keys.

We independently tested other TLS implementations for their vulnerability to FREAK. Microsoft SChannel and IBM JSSE also allowed RSA_EXPORT downgrades. Earlier versions of BoringSSL and LibreSSL had inherited the vulnerability from OpenSSL, but they had been recently patched independently of our discovery. In summary, at the time of its disclosure, our server impersonation attack was effective on any client that used OpenSSL, SChannel, SecureTransport, IBM JSSE, or older versions of BoringSSL and LibreSSL. The resulting list of vulnerable clients included most mobile web browsers (Safari, Android Browser, Chrome, BlackBerry, Opera) and a majority of desktop browsers (Chrome, Internet Explorer, Safari, Opera).

3.5. Summary and responsible disclosure

We systematically tested eight TLS libraries including MITLS, found serious state machine flaws in six of them, and were able to mount ten practical attacks, including eight impersonation attacks that break the core security guarantees of TLS.

Almost all implementations allowed some handshake messages to be skipped even if they were required for the current key exchange. We believe that this misbehavior results from a naive composition of handshake state machines. Notably, several implementations allowed CCS messages to be skipped. Considering our attacks as well as the recent Early CCS attack on OpenSSL,[c] we note that the handling of CCS messages in TLS state machines is particularly error-prone and deserves close attention. Many implementations (OpenSSL, Java, Mono) also allowed messages to be repeated.

[b] http://cado-nfs.gforge.inria.fr/.
[c] http://ccsinjection.lepidum.co.jp.

We reported all the bugs presented in this paper to the various TLS libraries. They were acknowledged and several patches were developed in consultation with us. We then reran FLEXTLS to test whether they fixed the state machine bugs. All of the exploitable bugs we found have now been fixed, but other seemingly benign state machine flaws remain unfixed, and deserve closer analysis in future work.

4. A VERIFIED STATE MACHINE FOR OPENSSL

Systematic state-machine testing uncovers dangerous bugs, but does not guarantee that all flaws have been found and eliminated. Instead, it would be valuable to formally prove that a given state machine implementation complies with the TLS standard. Since new ciphersuites and protocol versions are continuously added to TLS implementations, it would be even better if we could set up an automated verification framework that could be maintained and systematically used to prevent regressions.

The MITLS implementation[6] uses refinement types to verify that its handshake implementation is correct with respect to a logical state machine specification. Furthermore, it establishes a strong security theorem: a TLS connection between a MITLS client and server is a secure channel, unless one of the low-level cryptographic primitives used by the connection is broken. However, it only covers RSA and DHE ciphersuites and only applies to carefully written F# code.

In this section, we investigate whether we could achieve a similar, if less ambitious, verification result for the state machine implemented by the popular OpenSSL TLS library, which is written in C and covers many more protocol versions, extensions, and ciphersuites than MITLS.

4.1. A new state machine for OpenSSL

The client and server state machines in OpenSSL are coded as loops with large *switch* statements, with one case for each message in the protocol. A series of functions implement the individual messages: each *ssl3_send_** function constructs and sends a message; each *ssl3_get_** function receives and processes a message. These functions maintain the current state in a shared *SSL* data structure with about 100 mutable fields.

The state machine code in OpenSSL has evolved over 17 years to incorporate new protocol versions, ciphersuites, and extensions, resulting in surprisingly complex handling of optional messages and subtle dependencies on various state variables. The current structure makes it difficult to verify whether this code conforms to its intended state machine. Indeed, the flaws in Table 1 indicate that it does not.

We propose a new state machine for OpenSSL that makes the allowed message sequences more explicit and easier to verify. In addition to the full *SSL* data structure used by the messaging functions, we maintain a separate *STATE* data structure (see Figure 5) with just the elements that control state transitions: the role (client or server); the protocol version; the key exchange method; the client authentication mode; flags for resumption and renegotiation; the last message received; and the message sequence so far. By default, each element is initially set to a special *UNDEFINED* value.

The core of our state machine is a single function, *ssl3_next_message*, which takes as arguments the current *SSL* and *STATE* structures, the next message to send or receive, its direction, and its content type. This function enforces the state machine on all incoming and outgoing messages. For incoming messages, it checks that the transition is enabled, and then calls the corresponding message handler in legacy code; that code may in turn send some messages, causing our *ssl3_next_message* function to be called in the outgoing direction. For outgoing messages, it similarly checks that the transition is enabled and then calls the usual OpenSSL *ssl_send_** functions.

Our state machine is coded in about 500 lines, supplemented by about 250 lines of simple message parsing functions that can extract message types, protocol versions, and key exchange methods, from various handshake messages.

Figure 5. A new state machine for OpenSSL: the STATE data structure encodes the current state; *ssl3_next_message* encodes allowed transitions.

```
typedef struct state {
  Role role;        // r ∈ {Client, Server}
  PV version;       // v ∈ {SSLv3, TLSv1.0, TLSv1.1, TLSv1.2}
  KEM kx;           // kx ∈ {DH*, ECDH*, RSA*}
  Auth client_auth; // (c_ask, c_offer)
  int resumption;   // (r_id , r_tick)
  int renegotiation; // = 1 if renegotiating
  int ntick;        // = 1 if ticket expected

  Msg_type last_message; // previous message type
  unsigned char* log;    // handshake messages so far
  unsigned int log_length;
} STATE;
```

```
int ssl3_next_message(SSL* ssl, STATE *st,
      unsigned char* msg, int msg_len,
       int direction,  unsigned char content_type);
```

4.2. Experimental evaluation

To test our new state machine, we deployed it as an inline reference monitor alongside the legacy OpenSSL state machine. Our function *ssl3_next_message* is called before sending or receiving any message, but it does not itself call any message handlers. Instead, it maintains the *STATE* data structure and logs whether the next message violates the state machine. We use this variant of OpenSSL in two ways. First, by running standard interoperability tests for against peers running OpenSSL and other TLS implementations, we check that our new code does not reject valid message sequences. Using this method, we found and fixed some early bugs in our state machine. Second, by running it against deviant FLEXTLS peers, we check that our code logs an error for all the deviant traces presented in Section 2.

4.3. Formal verification

To gain further confidence in our state machine, we formalize our reference TLS state machine as an inductive predicate *isValidState* over the current *STATE* structure. The predicate holds if and only if the message sequence seen so

far is allowed by the state machine. We then specify that this predicate must be maintained as an invariant by our *ssl3_next_message* function.

To mechanically verify that our state machine implementation complies with its *isValidState* specification, we use the C verification tool Frama-C.[11] We annotate our code with logical assertions and requirements in Frama-C's specification language, called ACSL, including 460 lines of first-order logic to define *isValidState*. To verify our state machine code, we ran Frama-C to generates proof obligations for multiple SMT solvers. We used Alt-Ergo to discharge some obligations and Z3 for others, for a total verification time of 30 min. Technically, verification also involves memory invariants, to ensure that our code maintains separation between its private state and the rest of OpenSSL, and 900 lines of lemmas to facilitate the proof. (We formally assume that the rest of OpenSSL does not interfere with our code; verifying their full codebase is well beyond the scope of this work.)

4.4. Discussion

Predicates such as *isValidState* are logical encodings of our state machines. They are inspired by the simpler log predicates used in the cryptographic verification of miTLS.[6] The properties they capture depend only on the TLS specification; they omit any implementation details, and are even independent of their programming languages.

Although our logical specification is almost as long as the code we verified, we found verification useful in several ways. First, in addition to our state invariant, we prove memory safety for our code, a mundane but important goal for C programs. Second, our predicates provide an independent specification of the state machine, and verifying that they agree with the code helped us find bugs, especially regressions due to the addition of new features to the machine. Third, our logical formulation of the state machine allows us to prove theorems about its precision. For example, we used the Coq proof assistant to formally establish that the message sequence stored in *STATE* is unambiguous, that is, if the sequences in two valid state are the same, then the rest of the states must be the same as well. This property is a key lemma for proving the security of TLS, inasmuch as the message transcripts (not the states they encode) are authenticated at the end of the handshake.

Still, our verification result is far from a miTLS-style security theorem for OpenSSL. We proved that our state machine for OpenSSL is functionally correct, but we did not, for example, verify the cryptographic constructions or the full message processing code. We could attempt to extend our results to a larger fragment of OpenSSL that implements all important protocol features; verifying all this code may be feasible but remains a daunting task.

An intermediate goal may be to verify the code in OpenSSL for a single strong ciphersuite, such as *TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256*. We would then need to prove that, no matter which other ciphersuites are supported, if the client and server choose this ciphersuite, then the resulting connection is secure. To achieve even this limited security theorem, we must overcome several challenges. The first step, which we have already accomplished, is to prove that the state machine correctly implements the message sequence for this ciphersuite, and that this sequence cannot be confused with that of another ciphersuite. The second step is to prove that it is safe to share the long-term signing keys used in our ciphersuite with other, unverified ciphersuites. This property is problematic for current versions of TLS, but is expected to hold for TLS 1.3.[14] The third step is to show that the session secrets of our verified ciphersuite are cryptographically independent from any other ciphersuite. This property should hold for connections that use TLS 1.3, and also for those that use the TLS extended master secret extension.[4]

In summary, by verifying its state machine, we have taken a first step toward an OpenSSL security theorem, but many problems remain before we can verify mainstream libraries that include legacy code, insecure ciphersuites, and obsolete protocol versions. Partly as a result of our work, the state machine in next major version OpenSSL 1.1.0 was rewritten from scratch, with the goal of making it simpler, stricter, and easier to validate. We hope that with similar efforts in the rest of the codebase, all of OpenSSL will one day become amenable to formal verification.

5. RELATED WORK

5.1. TLS attacks

The reader is advised to refer to Soghoian and Stamm[24] for a broad survey of previous attacks on TLS and its implementations; here, we discuss here only closely related work.

Wagner and Schneier[28] describe various attacks against SSL 3.0, and their analysis has proved prescient for many attacks on TLS, including the state machine flaws discussed in this paper. For instance, they present an early cross-ciphersuite attack (predating[23]) that rely on confusing ephemeral RSA handshakes with ephemeral Diffie–Hellman. They also anticipate some of our message skipping attacks by pointing out that, in MAC-only ciphersuites, the attacker can bypass authentication by skipping CCS messages.

In parallel with our work, de Ruiter and Poll[12] apply machine learning techniques to reverse engineer the state machines of several TLS libraries and discover flaws like the ones described in this paper. Their technique is able to reconstruct abstract state machines even for closed-source libraries, whereas our method focuses on testing conformance to the standard and uncovering concrete exploits.

Jager et al.[17] identify a class of backwards compatibility attacks on protocol implementations that support both strong and weak algorithms, showing for instance how a side-channel attack on RSA decryption in TLS servers can be exploited to mount a cross-protocol attack on server signatures.[18] FREAK, our downgrade attack on export RSA ciphersuites, can also be seen as a backwards compatibility attack. Inspired by FREAK, Logjam[1] is a downgrade attack that exploits a protocol-level ambiguity between the DHE and export DHE ciphersuites. Whereas FREAK relied on a state machine flaw, Logjam relies on the widespread acceptance of weak Diffie–Hellman groups in TLS clients.

Another class of TLS vulnerabilities stems from the incorrect composition of TLS sub-protocols for renegotiation,[26] alerts,[6] and resumption.[8] These flaws may be partly blamed on the state machine being underspecified in the standard—the last two were discovered while designing and verifying the state machine of miTLS.

5.2. TLS verification

Cryptographers have developed proofs for DHE,[16] RSA,[19] and PSK[22] key exchanges run in isolation; they apply to the TLS design, but not its implementations.

Bhargavan et al.[6, 7] proved that composite RSA and DHE are jointly secure in the miTLS implementation, programmed in F# and verified using refinement types.

Several works extract formal models from TLS implementations and analyze them with automated protocol verification tools. Bhargavan et al.[5] extract and verify ProVerif and CryptoVerif models from an F# implementation of TLS. Chaki and Datta[10] verify the SSL 2.0/3.0 handshake of OpenSSL using model checking and find several known rollback attacks. Avalle et al.[2] verify Java implementations of the TLS handshake protocol using ProVerif.

Others analyze TLS libraries for programming bugs. Lawall et al.[21] use the Coccinelle framework to detect incorrect checks on values returned by the OpenSSL API, and Frama-C has been used to verify parts of PolarSSL.

6. CONCLUSION

While security analyses of TLS primarily focused on flaws in fixed cryptographic constructions, the state machines that control the flow of protocol messages in their implementations have escaped scrutiny. Using a combination of automated testing and manual source code inspection, we discovered serious flaws in several TLS implementations. These flaws predominantly arise from the incorrect composition of the multiple ciphersuites and authentication modes supported by TLS.

Considering the impact and prevalence of these flaws, we advocate a principled programming approach for protocol implementations that includes systematic testing against unexpected message sequences (a form of directed fuzzing) as well as formal proofs of correctness for critical components.

Although current TLS implementations are far from perfect, upcoming improvements in the protocol and progress in verification tools let us hope that the security verification of mainstream TLS libraries will soon be within reach.

Acknowledgments

The authors would like to thank Matthew Green, Nadia Heninger, Santiago Zanella-Béguelin, the ZMap team, and the CADO-NFS team for their help with evaluating and exploiting FREAK. We thank the developers of OpenSSL, SChannel, SecureTransport, NSS, BoringSSL, Oracle JSSE, CyaSSL, and Mono for their rapid response to our disclosures. Bhargavan, Beurdouche, and Delignat-Lavaud were supported by the ERC Starting Independent Researcher Grant no. 259639 (CRYSP). C

References

1. Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J.A., Heninger, N., Springall, D., Thomé, E., Valenta, L., VanderSloot, B., Wustrow, E., Zanella-Béguelin, S., Zimmermann, P. Imperfect forward secrecy: How Diffie–Hellman fails in practice. In *ACM CCS* (2015), 5–17.
2. Avalle, M., Pironti, A., Pozza, D., Sisto, R. JavaSPI: A framework for security protocol implementation. *Int. J. Sec. Softw. Eng. 2* (2011), 34–48.
3. Beurdouche, B., Delignat-Lavaud, A., Kobeissi, N., Pironti, A., Bhargavan, K. FlexTLS: A tool for testing TLS implementations. In *USENIX Workshop on Offensive Technologies (WOOT)* (2015).
4. Bhargavan, K., Delignat-Lavaud, A., Pironti, A., Langley, A., Ray, M. Transport Layer Security (TLS) session hash and extended master secret extension. IETF RFC 7627, 2014.
5. Bhargavan, K., Fournet, C., Corin, R., Zălinescu, E. Verified cryptographic implementations for TLS. *ACM TISSEC 15*, 1 (2012), 1–32.
6. Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P. Implementing TLS with verified cryptographic security. In *IEEE S&P (Oakland)* (2013), 445–459.
7. Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.-Y., Zanella-Béguelin, S. Proving the TLS handshake secure (as it is). In *CRYPTO* (2014), 235–255.
8. Bhargavan, K., Lavaud, A.D., Fournet, C., Pironti, A., Strub, P.-Y. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE S&P (Oakland)* (2014), 98–113.
9. Cavallar, S., Dodson, B., Lenstra, A., Lioen, W., Montgomery, P., Murphy, B., te Riele, H., Aardal, K., Gilchrist, J., Guillerm, G., Leyland, P., Marchand, J., Morain, F., Muffett, A., Putnam, C., Zimmermann, P. Factorization of a 512-bit RSA modulus. In *EUROCRYPT* (2000), 1–18.
10. Chaki, S., Datta, A. ASPIER: An automated framework for verifying security protocol implementations. In *IEEE CSF* (2009), 172–185.
11. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B. Frama-C. In *Software Engineering and Formal Methods* (2012), 233–247.
12. de Ruiter, J., Poll, E. Protocol state fuzzing of TLS implementations. In *USENIX Security* (2015), 193–206.
13. Dierks, T., Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.2. IETF RFC 5246, 2008.
14. Dierks, T., Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.3. Internet Draft, 2014.
15. Durumeric, Z., Wustrow, E., Halderman, J.A. ZMap: Fast Internet-wide scanning and its security applications. In *USENIX Security* (2013), 605–620.
16. Jager, T., Kohlar, F., Schäge, S., Schwenk, J. On the security of TLS-DHE in the standard model. In *CRYPTO* (2012), 273–293.
17. Jager, T., Paterson, K.G., Somorovsky, J. One bad apple: Backwards compatibility attacks on state-of-the-art cryptography. In *NDSS* (2013).
18. Jager, T., Schwenk, J., Somorovsky, J. On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption. In *ACM CCS* (2015), 1185–1196.
19. Krawczyk, H., Paterson, K.G., Wee, H. On the security of the TLS protocol: A systematic analysis. In *CRYPTO* (2013), 429–448.
20. Langley, A., Modadugu, N., Moeller, B. Transport Layer Security (TLS) False Start. IETF RFC 7918, 2010.
21. Lawall, J., Laurie, B., Hansen, R.R., Palix, N., Muller, G. Finding error handling bugs in OpenSSL using Coccinelle. In *European Dependable Computing Conference* (2010), 191–196.
22. Li, Y., Schäge, S., Yang, Z., Kohlar, F., Schwenk, J. On the security of the pre-shared key ciphersuites of TLS. In *Public-Key Cryptography* (2014), 669–684.
23. Mavrogiannopoulos, N., Vercauteren, F., Velichkov, V., Preneel, B. A cross-cprotocol attack on the TLS protocol. In *ACM CCS* (2012), 62–72.
24. Meyer, C., Schwenk, J. Lessons learned from previous SSL/TLS attacks – A brief chronology of attacks and weaknesses. IACR Cryptology ePrint Archive, Report 2013/049, 2013.
25. Paterson, K.G., Ristenpart, T., Shrimpton, T. Tag size does matter: Attacks and proofs for the TLS record protocol. In *ASIACRYPT* (2011), 372–389.
26. Ray, M., Dispensa, S. Renegotiating TLS, 2009.
27. Soghoian, C., Stamm, S. Certified lies: Detecting and defeating government interception attacks against SSL. In *Financial Cryptography* (2012), 250–259.
28. Wagner, D., Schneier, B. Analysis of the SSL 3.0 protocol. In *USENIX Workshop on Electronic Commerce* (1996), 29–40.

Benjamin Beurdouche and Karthikeyan Bhargavan ({benjamin.beurdouche, karthikeyan.bhargavan, antoine.delignat-lavaud, alfredo.pironti}@inria.fr), INRIA.

Alfredi Pironti (alfredo@pironti.eu), IOActive.

Antoine Delignat-Lavaud, Cédric Fournet, and Markulf Kohlweiss ({antdl, fournet, markulf}@microsoft.com), Microsoft Research.

Pierre-Yves Strub (pierreyves.strub@imdea.org), IMDEA Software Institute.

Jean-Karim Zinzindohoué (jean-karim.zinzindohoue@inria.fr), INRIA & Ecole des Ponts, ParisTech.