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

Benjamin Beurdouche*, Karthikeyan Bhargavan*, Antoine Delignat-Lavaud*, Cedric Fournet†, Markulf Kohlweiss†, Alfredo Pironti*, Pierre-Yves Strub‡, Jean Karim Zinzindohoue*
*INRIA Paris-Rocquencourt †Microsoft Research ‡IMDEA Software Institute

***Abstract*—Implementations of the Transport Layer Security (TLS) protocol must handle a variety of protocol versions and extensions, authentication modes and key exchange methods, where each combination may prescribe a different message sequence between the client and the server. We address the problem of designing a robust *composite* state machine that can correctly multiplex between these different protocol modes. We systematically test popular open-source TLS implementations for state machine bugs and discover several critical security vulnerabilities that have lain hidden in these libraries for years (they are now in the process of being patched). We argue that these vulnerabilities stem from incorrect compositions of individually correct state machines. We present the first verified implementation of a composite TLS state machine in C that can be embedded into OpenSSL and accounts for all its supported ciphersuites. Our attacks expose the need for the formal verification of core components in cryptographic protocol libraries; our implementation demonstrates that such mechanized proofs are within reach, even for mainstream TLS implementations.**

I. Transport Layer Security

The Transport Layer Security (TLS) protocol [15] is widely used to provide secure channels in a variety of scenarios, including the web (HTTPS), email, and wireless networks. Its popularity stems from its flexibility; it offers a large choice of ciphersuites and authentication modes to its applications.

The classic TLS threat model considered in this paper is depicted in Figure 1. A client and server each execute their end of the protocol state machine, communicating across an insecure network under attacker control: messages can be intercepted, tampered, or injected by the attacker. Additionally, the attacker controls some malicious clients and servers that can deviate from the protocol specification. The goal of TLS is to guarantee the integrity and confidentiality of exchanges between honest clients and servers, and prevent impersonation and tampering attempts by malicious peers.

TLS consists of a channel establishment protocol called the *handshake* followed by a transport protocol called the *record*. If the client and server both implement a secure handshake key exchange (e.g. Ephemeral Diffie-Hellman) and a strong transport encryption scheme (e.g. AES-GCM with SHA256), the security against the network attacker can be reduced to the security of these building blocks. Recent works have exhibited cryptographic proofs for various key exchange methods used in the TLS handshakes [19, 22, 25] and for commonly-used record encryption schemes [28].

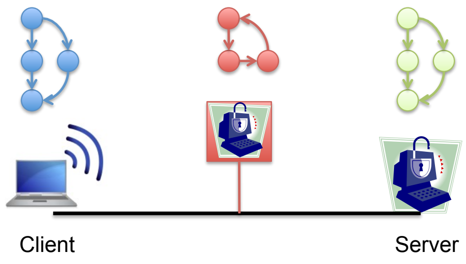

Fig. 1. Threat Model: network attacker aims to subvert client-server exchange.

Protocol Agility TLS suffers from legacy bloat: after 20 years of evolution of the standard, it features many versions, extensions, and ciphersuites, some of which are no longer used or are known to be insecure. Accordingly, client and server implementations offer much agility in their protocol configuration, and their deployment often support insecure ciphersuites for interoperability reasons. The particular parameters of a specific TLS session are negotiated during the handshake protocol. Agreement on these parameters is only verified at the very end of the handshake: both parties exchange a MAC of the transcript of all handshake messages they have sent and received so far to ensure they haven't been tampered by the attacker on the network. In particular, if *one* party only accepts secure protocol versions, ciphersuites, and extensions, then any session involving this party can only use these secure parameters regardless of what the peer supports.

Composite State Machines Many TLS ciphersuites and protocol extensions are specified in their own standards (RFCs), and are usually well-understood in isolation. They strive to re-use existing message formats and mechanisms of TLS to reduce implementation effort. To support their (potential) negotiation within a single handshake, however, the burden falls on TLS implementations to correctly compose these different protocols, a task that is not trivial.

TLS implementations are typically written as a set of functions that generate and parse each message, and perform the relevant cryptographic operations. The overall message sequence is managed by a reactive client or server process that sends or accepts the next message based on the protocol parameters negotiated so far, as well as the local protocol configuration. The composite state machine that this process must implement is not standardized, and differs between implementations. As explained below, mistakes in this state machine can lead to disastrous misunderstandings.

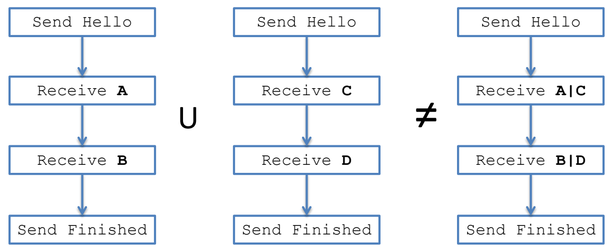

Fig. 2. Incorrect union of exemplary state machines.

Figure 2 depicts a simple example. Suppose we have implemented a client for one (fictional) TLS ciphersuite, where the client first sends a `Hello` message, then expects to receive two messages `A` and `B` before sending a `Finished` message. Now the client wishes to implement a new ciphersuite where the client must receive a different pair of messages `C` and `D` between `Hello` and `Finished`. To reuse the messaging code for `Hello` and `Finished`, it is tempting to modify the client state machine so that it can receive either `A` or `C`, followed by either `B` or `D`. This naive composition implements both ciphersuites, but it also enables some unintended sequences, such as `Hello`; `A`; `D`; `Finished`.

One may argue that allowing more incoming message sequences does not matter, since an honest server will only send the right message sequence. And if an attacker injects an incorrect message, for instance by replacing message `B` with message `D`, then the mismatch between the client and server transcript MAC ensures that the handshake cannot succeed. The flaw in this argument is that, meanwhile, a client that implements `Hello;A;D;Finished` is running an unknown handshake protocol, with *a priori* no security guarantees. For example, the code for processing `D` may expect to run after `C` and may accidentally use uninitialized state that it expected `C` to fill in. It may also leak unexpected secrets received in `A`, or allow some crucial authentication steps to be bypassed.

In Sections III and IV we systematically analyze the state machines implemented by various open source TLS implementations and we find that many of them exhibit such composition flaws and accept unexpected message sequences. In Section V, we show that these flaws lead to critical vulnerabilities where, for example, a network attacker can fully impersonate any server towards a vulnerable client.

Verified Implementations Security proofs for TLS typically focus on clients and servers that support a single, fixed message sequence, and that *a priori* agree on their security goals and mechanisms, e.g. mutual authentication with Diffie-Hellman, or unilateral authentication with RSA. Recently, a verified implementation called MITLS [6] showed how to compose proofs for various modes that may be dynamically negotiated by their implementation. However, mainstream TLS implementations compose far more features, including legacy insecure ciphersuites. Verifying their code seems infeasible.

We ask a limited verification question, separate from the cryptographic strength of ciphersuites considered in isolation. Let us suppose that the individual message processing functions in OpenSSL for unilaterally authenticated ECDHE in TLS 1.0 are correct. Can we then prove that, if a client or server negotiates a configuration, then its state machine faithfully implements the correct message sequence processing for that key exchange? In Section VI we present a verified implementation of a state machine for OpenSSL that accounts for all its ciphersuites.

Given that cryptographers proved ECDHE secure in isolation, what are the additional requirements on the set of ciphersuites supported by an implementation to benefit from this cryptographic security? Conversely, if they deviate from the correct message sequence, are there exploitable attacks?

Contributions In this paper,

- we define a composite state machine for the commonly implemented modes of TLS, based on the standard specifications (§II);
- we present tools to systematically test mainstream TLS implementations for conformance (§III);
- we report flaws (§IV) and critical vulnerabilities (§V) we found in these implementations;
- we develop a verified state machine for OpenSSL, the first to cover all of its TLS modes (§VI).

Our state machine testing framework FLEXTLS is built on top of MITLS [6], and benefits from its functional style and verified messaging functions. Our OpenSSL state machine code is verified using Frama-C [14], a framework for the static analysis of C programs against logical specifications written in first-order logic. All the attacks discussed in this paper were reported to the relevant TLS implementations; they were acknowledged and various critical updates are being tested. Until these updates are released, the results in this paper should be treated as confidential.

II. The TLS State Machine

Figure 3 depicts a simplified high-level state machine that captures the sequence of messages that are sent and received from the beginning of a TLS connection up to the end of the first handshake. It only covers commonly used ciphersuites and it does not detail message contents, local state at client and server, or cryptographic computations.

Message Sequences Messages prefixed by `Client` are sent from client to server; messages prefixed by `Server` are sent by the server. Arrows indicate the order in which these messages are expected; labels on arrows define conditions under which the transition is allowed.

Each TLS connection begins with either a full handshake or an abbreviated handshake (also called session resumption). In the full handshake, there are four flights of messages: the client first sends a `ClientHello`, the server responds with a series of messages from `ServerHello` to `ServerHelloDone`. The client then sends a second flight culminating in `Client-Finished` and the server completes the handshake by sending a final flight that ends in `ServerFinished`. Before sending their respective `Finished` message, the client and the server send a change cipher spec (`CCS`) message to signal that the new keys established by this handshake will be used to protect subsequent messages (including the `Finished` message). Once the handshake is complete, the client and the server may exchange streams of `ApplicationData` messages.

Abbreviated handshakes skip most of the messages by relying on session secrets established in some previous full handshake. The server goes from `ServerHello` straight to `ServerCCS` and `ServerFinished`, and the client completes the handshake by sending its own `ClientCCS` and `ClientFinished`.

In most full handshakes (except for anonymous key exchanges), the server *must* authenticate itself by sending a certificate in the `ServerCertificate` message. In the DHE|ECDHE handshakes, the server demonstrates its knowledge of the certificate's private key by signing the subsequent `ServerKeyExchange` containing its ephemeral Diffie-Hellman public key. In the RSA key exchange, it instead uses the private key to decrypt the `ClientKeyExchange` message. When requested by the server (via `CertificateRequest`), the client may optionally send a `ClientCertificate` and use the private key to sign the full transcript of messages (so far) in the `ClientCertificateVerify`.

Negotiation Parameters The choice of what sequence of messages will be sent in a handshake depends on a set of parameters negotiated within the handshake itself:

- the protocol version (v),
- the key exchange method in the ciphersuite (kx),
- whether the client offered resumption with a cached session and the server accepted it ($r_{id} = 1$),
- whether the client offered resumption with a session ticket and the server accepted it ($r_{tick} = 1$),
- whether the server wants client authentication ($c_{ask} = 1$),
- whether the client agrees to authenticate ($c_{offer} = 1$),
- whether the server sends a new session ticket ($n_{tick} = 1$).

A client knows the first three parameters (v, kx, r_{id}) explicitly from the `ServerHello`, but can only infer the others ($r_{tick}, c_{ask}, n_{tick}$) later in the handshake when it sees a particular message. Similarly, the server only knows whether or how a client will authenticate itself from the content of the `ClientCertificate` message.

Implementation Pitfalls Even when considering only modern protocol versions TLSv1.0|TLSv1.1|TLSv1.2 and the most popular key exchange methods RSA|DHE|ECDHE, the number of possible message sequences in Figure 3 is substantial and warns us about tricky implementation problems.

First, the order of messages in the protocol has been carefully designed and it must be respected, both for interoperability and security. For example, the `ServerCCS` message must occur just before `ServerFinished`. If it is accepted too early or too late, the client enables various server impersonation attacks. Implementing this message correctly is particularly tricky because `CCS` messages are not officially part of the handshake: they have a different content type and are not included in the transcript. So an error in their position in the handshake would not be caught by the transcript MAC.

Second, it is not enough to implement a linear sequence of sends and receives; the client and server must distinguish between truly optional messages, such as `ServerNewSessionTicket`, and messages whose presence is fully prescribed by the current key exchange, such as `ServerKeyExchange`. For example, we will show in Section V that accepting a `ServerKeyExchange` in RSA or allowing it to be omitted in ECDHE can have dire consequences.

Third, one must be careful to not prematurely calculate session parameters and secrets. Traditionally, TLS clients set up their state for a full or abbreviated handshake immediately after the `ServerHello` message. However, with the introduction of session tickets, this would be premature, since only the next message from the server would tell the client whether this is a full or abbreviated handshake. Confusions between these two handshake modes may lead to serious attacks.

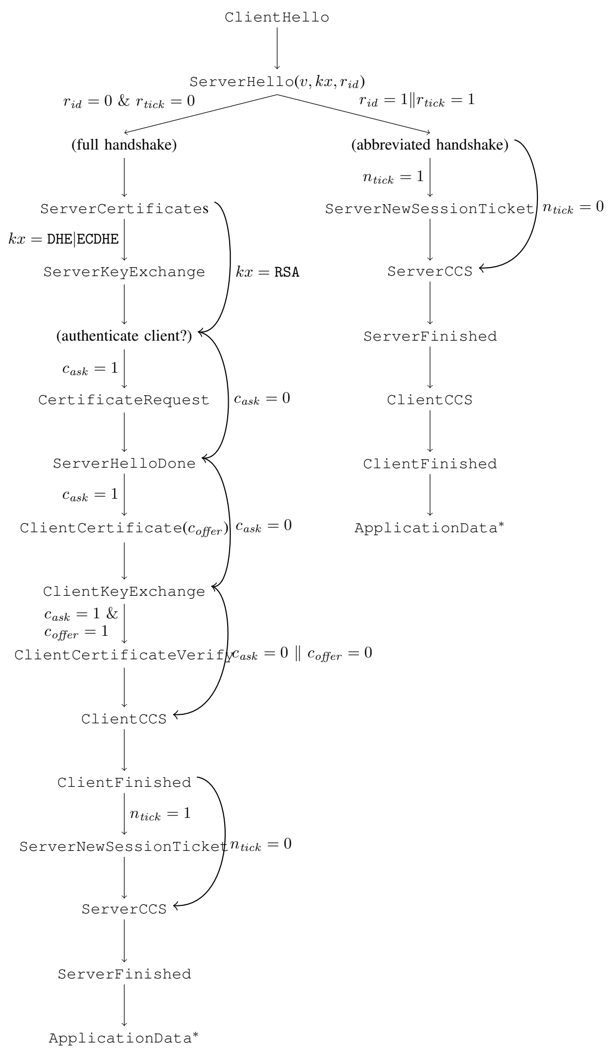

Fig. 3. State machine for commonly used TLS configurations: Protocol versions $v =$ TLSv1.0|TLSv1.1|TLSv1.2. Key exchanges $kx =$ RSA|DHE|ECDHE. Optional feature flags: resumption using server-side caches (r_{id}) or tickets (r_{tick}), client authentication (c_{ask}, c_{offer}), new session ticket (n_{tick}).

Other Versions, Extensions, Key Exchanges Typical TLS libraries also support other protocol versions such as SSLv2 and SSLv3 and related protocols like DTLS. At the level of detail of Figure 3, the main difference in SSLv3 is in client authentication: an SSLv3 client may decline authentication by not sending a ClientCertificate message at all. DTLS allows a server to respond to a ClientHello with a new HelloVerifyRequest message, to which the client responds with a new ClientHello.

TLS libraries also implement a number of ciphersuites that are not often used on the web, like static Diffie-Hellman (DH) and Elliptic Curve Diffie-Hellman (ECDH), anonymous key exchanges (DH_anon, ECDH_anon), and various pre-shared key ciphersuites (PSK, RSA_PSK, DHE_PSK, SRP, SRP_RSA). Figure 8 in the appendix displays a high-level TLS state machine for all these ciphersuites for TLSv1.0|TLSv1.1|TLSv1.2. Modeling the new message sequences induced by these ciphersuites requires additional negotiation parameters like PSK hints (c_{hint}) and static Diffie-Hellman client certificates ($c_{offer} = 2$).

Incorporating renegotiation, that is multiple TLS handshakes on the same connection, is logically straightforward, but can be tricky to implement. At any point after the first handshake, the client can go back to ClientHello (the server could send a HelloRequest to request this behavior). During a renegotiation handshake, ApplicationData can be sent under the old keys until the CCS messages are sent.

In addition to session tickets [10], another TLS extension that modifies the message sequence is called *False Start* [23]. Clients that support the False Start extension are allowed to send early ApplicationData as soon as they have sent their ClientFinished without waiting for the server to complete the handshake. This is considered to be safe as long as the negotiated ciphersuite is forward secret (DHE|ECDHE) and uses strong record encryption algorithms (e.g. not RC4). False Start is currently enabled in all major web browsers and hence is also implemented in major TLS implementations like OpenSSL, NSS, and SecureTransport.

Analyzing Implementations We wrote the state machines in Figures 3 and 8 by carefully inspecting the RFCs for various versions and ciphersuites of TLS. How well do they correspond to the state machines implemented by TLS libraries? We have a definitive answer for MITLS, which implements RSA, DHE, resumption, and renegotiation. The type-based proof for MITLS guarantees that its state machine conforms to a logical specification that is similar to Figure 3, but more detailed.

In the rest of the paper, we will investigate how to verify whether mainstream TLS implementations like OpenSSL conform to Figure 8. In the next section, we begin by systematically testing various open source TLS libraries for deviations from the standard state machine.

III. Testing Implementations with FlexTLS

To explore the state-machine behavior of existing TLS implementations, we send sequences of TLS messages to the tested implementations and we observe their reaction. For valid protocol sequences, the peer should proceed normally with the protocol execution; for sequences containing unexpected messages, the peer should report an error, typically by sending an unexpected_message alert.

```
// Ensure we use RSA
let ch = {defaultClientHello with ciphersuites =
    Some([TLS_RSA_WITH_AES_128_CBC_SHA]) } in
let st,nsc,ch = ClientHello.send(st,ch) in
let st,nsc,sh = ServerHello.receive(st,ch,nsc) in
let st,nsc,cert = Certificate.receive(st,Client,nsc) in
let st,shd = ServerHelloDone.receive(st) in
let st,nsc,cke = ClientKeyExchange.sendRSA(st,nsc,ch) in
let st,_ = CCS.send(st) in
let st = State.installWriteKeys st nsc in
let log = ch.payload @| sh.payload @| cert.payload @| shd.
      payload @| cke.payload in
let st,cf = Finished.send(st,nsc,logRole=(log,Client)) in
let st,_,_ = CCS.receive(st) in
let st = State.installReadKeys st nsc in
let log = log @| cf.payload in
let st,sf = Finished.receive(st,nsc,(log,Server)) in
st
```

Fig. 4. A normal RSA key exchange scripted with FlexTLS.

Generating arbitrary sequences of valid TLS messages is not a trivial task, as (by protocol design) the content of each message typically depends on previously exchanged values. For example, the master secret value needed to compute the Finished message depends on both client and server randomness, and at least one of the two is freshly generated by the implementation under test. In our experience, modifying a TLS library to execute non-standard message sequences can be awkward and error prone. After all, TLS implementations are designed to comply with the protocol and reject bad traces.

For these reasons, we have developed FlexTLS, a tool for scripting and prototyping TLS scenarios in F#. To send and receive TLS messages, FlexTLS uses the MITLS library, a verified reference implementation of TLS. MITLS was developed in a modular, functional, state-passing style, with an emphasis on clarity rather than performance, and we found it easy to reuse its core modules for cryptography and message parsing. In addition, using verified messaging libraries improves the robustness of FlexTLS and reduces false positives due to, for example, malformed or incorrectly parsed messages.

FlexTLS scripting Figure 4 presents FlexTLS by example, using a client script for a normal RSA key exchange with no client authentication. For each handshake message, FlexTLS provides a class equipped with *send* and *receive* functions, and a record that holds its parsed contents. For example, the ClientHello message record contains a *ciphersuites* field; the user may set its value before sending, or read its value after receiving. In addition, FlexTLS keeps some internal connection state (including for instance the connection keys and sequence numbers) in a state variable, *st*, passed from one call to the other. Finally, each handshake also prepares the next *security context*, to be installed after exchanging CCS messages; FlexTLS reflects its evolution using another state variable, `nsc`.

Sending messages out-of-order with FLEXTLS is usually as simple as reordering lines in a script. FLEXTLS handles most of the complexity internally, notably by filling in any missing values, inasmuch as the protocol specification does not indicate which values to use out of order. For example, if the user creates a script that sends a `Finished` message immediately after a `ServerHello` message, which value should be used for the master secret? One may pick an empty (null) pre-master secret and combine it with the client and server random to get the master secret; or one may use an empty (null) master secret; or one may fill the master secret with an array of zeros of the right length. FLEXTLS produces context-dependent default values that are expected to work in most of the cases; yet, it is designed to let the user easily override these defaults. For example, the master secret of a next security context `nsc` can be set by the user to an array of 48 zeros by adding the following lines:

let *keys* = {*nsc.keys* **with** *ms* = *Array.zeroCreate* 48} **in**
let *nsc* = {*nsc* **with** *keys* = *keys*} **in** ...

Searching for deviant traces Next, we define valid and deviant traces. Let σ be a sequence of protocol messages, m a protocol message, and $\sigma; m$ their concatenation. We let $\sigma \leq \tau$ denote that 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, possibly with different ciphersuites. We also lift $\sim$ from messages to traces. Let *Valid* be the set of valid traces allowed by the state machine described in figure 3, closed under the prefix relation. A deviant trace is a minimal invalid trace:

- $\sigma; m$ is *deviant* when $\sigma \in$ *Valid* but $\sigma; m \notin$ *Valid*.

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

The set of deviant traces is rather large (and even infinite unless we bound the number of renegotiations allowed), so we automatically generate a representative, finite subset according to three heuristic rules that proved the most effective:

Skip If $\sigma; m; n \in$ *Valid* and $\delta = \sigma; n \notin$ *Valid*, test δ. That is, for every prefix of a valid message sequence, we skip a message if it is mandatory. For example, `ClientHello; ServerHello(DHE); ServerKeyExchange` is a trace where the `Certificate` message has been skipped. In practice, we find it useful to allow even a sequence of messages to be skipped, but to get reliable feedback from the peer we don't skip the final message of a flight, that is, `ClientHello`, `ServerHelloDone`, `ClientFinished`, or `ServerFinished`.

Hop Let $\tau = \sigma; m \in$ *Valid* and $\tau' = \sigma'; n \in$ *Valid*. If $\sigma \sim \sigma'$, $m \neq n$, and $\delta = \sigma; n \notin$ *Valid*, test δ. That is, if two valid traces have the same prefix, up to their parameters, and they differ on their next message, we create a deviant trace from the context of the first trace and the next message of the second trace.
This can be seen as hopping from one state machine trace to another, or as a way to skip optional protocol messages that may be required in some other context.
For example, `ClientHello(noResumption); ServerHello; ServerCCS` is a trace that hops into a session resumption trace, even if the client asked to start a full handshake. `ClientHello; ServerHello(RSA); Certificate; ServerKeyExchange` is a trace that sends an unexpected `ServerKeyExchange` by hopping from an RSA to a DHE trace.

Repeat If $\tau = \sigma; m; \sigma' \in$ *Valid* and $\delta = \tau; m \notin$ *Valid*, test δ. That is, for every prefix of a valid message sequence, we take any message that has appeared before and send it again if this results in a deviant trace. For example, `ClientHello; ServerHello ... ServerHelloDone; ClientHello` is a trace where the `ClientHello` message is repeated in the middle of a handshake, making it invalid.

A trace such as `ClientHello; ServerHello(DHE); Certificate; ServerHelloDone` that skips the optional `ServerKeyExchange` message can be generated by both the *Skip* and *Hop* policies, so we just consider the set of traces produced by any rule. Moreover, we only consider traces that begin with a `ClientHello; ServerHello` prefix, as all the implementations we tested require these first messages.

The main advantage of generating deviant traces according to such well-defined rules is that when a trace is accepted by an implementation, it is relatively simple to identify the corresponding state machine bug, which helps guide our subsequent manual code inspection. We also tried randomly generating deviant traces but manually interpreting their results was more time consuming and hence less effective.

Automated testing We partition the subset of deviant traces in server-executed and client-executed traces, according to the sender of the last message. We generate a FLEXTLS script for every deviant trace, and we run this script against a target implementation. Each FLEXTLS-generated script ends its deviant trace by sending an illegal message and then waiting for an alert from the peer. Indeed, the correct peer behavior against a deviant trace is to return an alert (usually `unexpected_message`) as soon as the deviant message is received. If a non-alert message is received, we flag that trace as detecting a state machine bug that requires further investigation. If the peer does not respond within a timeout, we assume that it accepted the trace and is waiting for further messages, and flag the trace for investigation.

Unfortunately, not all the TLS implementations we tested

TABLE I. Testing results for mainstream TLS implementations

Library	Mode	Version	Kex	Traces	Flags
OpenSSL 1.0.1j	Client	TLS 1.0	RSA, DHE	83	3
OpenSSL 1.0.1j	Server	TLS 1.0	RSA, DHE	94	6
OpenSSL 1.0.1g	Client	TLS 1.0	RSA, DHE	83	4
OpenSSL 1.0.1g	Server	TLS 1.0	RSA, DHE	94	14
GnuTLS	Client	TLS 1.0	RSA, DHE	83	0
GnuTLS	Server	TLS 1.0	RSA, DHE	94	2
SecureTransport	Client	TLS 1.0	RSA, DHE	83	3
NSS	Client	TLS 1.0	RSA, DHE	83	9
Java	Client	TLS 1.0	RSA, DHE	71	6
Java	Server	TLS 1.0	RSA, DHE	94	46
Mono	Client	TLS 1.0	RSA	35	32
Mono	Server	TLS 1.0	RSA	38	34
CyaSSL	Client	TLS 1.0	RSA	41	19
CyaSSL	Server	TLS 1.0	RSA	47	20

support all the scenarios and ciphersuites we test. For example, the Mono and CyaSSL implementations do not support DHE key exchange. In our experiments, such scenarios fail early—typically at the `Hello` messages, before reaching the deviant message—so we flag them instead as unsupported. Pragmatically, we instrument all our FlexTLS scripts so that they automatically classify peer behavior on each trace as either correct, or unsupported, or buggy.

Experimental results We tested the client and server sides of the following mainstream implementations: OpenSSL 1.0.1g and 1.0.1j; GnuTLS 3.3.9; NSS 3.17; Secure Transport 55471.14; Java 1.8.0_25; Mono 3.10.0; CyaSSL 3.2.0. Our results are reported in table I. All tests were run enforcing TLS 1.0, which ensures maximum support across different implementations. We ran only the RSA and DHE ciphersuites, since they were most commonly implemented.

We observe that both Mono and CyaSSL do not support DHE key exchange, and they do not accept an empty `ClientCertificate` message, hence they have been tested on a smaller number of traces.

CyaSSL and Secure Transport tear down the TCP connection when a deviant trace is detected; this is in contrast with the TLS specification, which prescribes to send a fatal alert to the peer. For this reason, our tool automatically flagged all traces when testing these implementations. We filtered out deviant traces that were correctly recognized, but for which the TCP connection had been torn down, and in the table we report traces that expose real state machine bugs.

We find more state machine issues in the older OpenSSL 1.0.1g version compared to 1.0.1j, which is not surprising since the former had known state machine issues that were fixed in the subsequent version.

Turning bugs into exploits In the next two sections, we will use these results to uncover state machine flaws and concrete attacks against these implementations. Once we find an attack, typically by inspecting the code and running targeted experiments with FlexTLS, we write our exploit as a FlexTLS scenario and use it as a demo to communicate with the implementors of the TLS library.

Our automated testing technique is a form of protocol-aware state machine fuzzing. Although effective, it is not complete, and every trace it flags requires further manual inspection of the source code to assess the severity of a state machine bug.

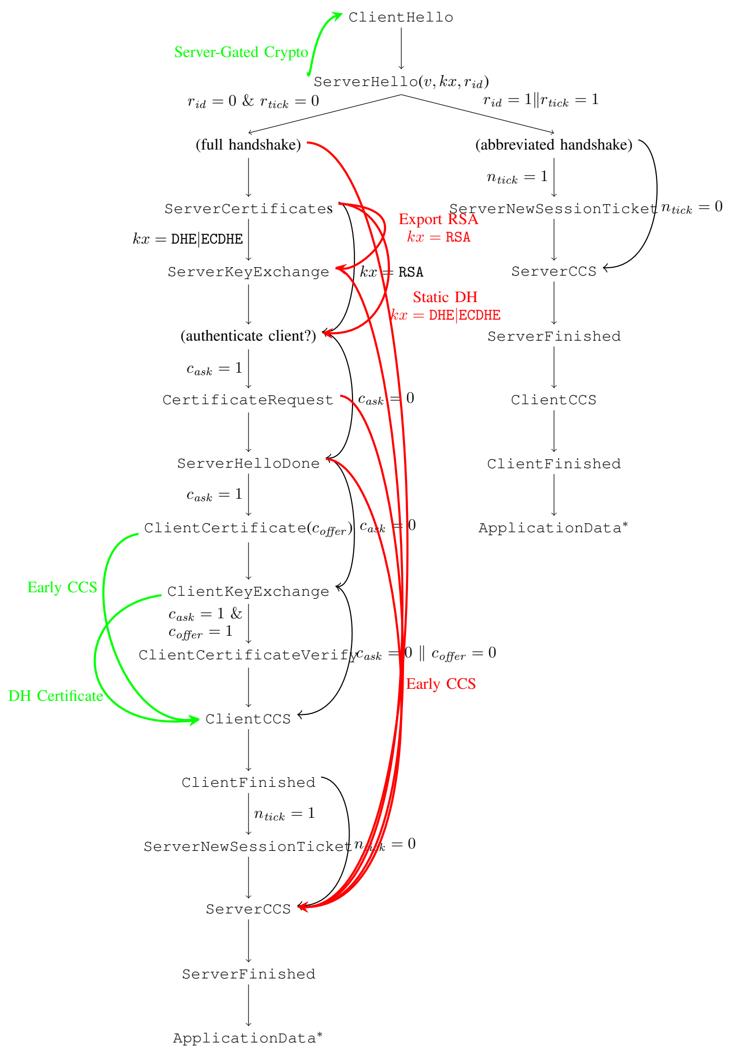

Fig. 5. OpenSSL Client and Server State machine for HTTPS configurations. Unexpected transitions: client in red on the right, server in green on the left

We chose a set of traces that, in our experience, were likely to expose security critical bugs. Independently, we wrote specific scenarios in FlexTLS to experiment with message content tampering and fragmentation, and could rediscover known attacks, such as the `ClientHello` fragmentation rollback attack on OpenSSL (CVE-2014-3511).

IV. State Machine Flaws in TLS Implementations

We now report the result of our systematic search for state-machine bugs in major TLS implementations, before analyzing their security impact in §V.

IV-A Implementation Bugs in OpenSSL. OpenSSL is the most widely-used open source TLS implementation, in particular on the web, where it powers HTTPS-enabled websites served by the popular Apache and nginx servers. It is also the most comprehensive: OpenSSL supports SSL versions

2 and 3, and all TLS and DTLS versions from 1.0 to 1.2, along with every ciphersuite and protocol extensions that has been standardized by the IETF, plus a few experimental ones under proposal. As a result, the state machines of OpenSSL are the most complex among those we reviewed, and many of its features are not exerted by our analysis based on the subset shown in Figure 3.

Running our tests from Section III reveal multiple unexpected state transitions that we depict in Figure 5 and that we investigate by careful source code inspection below:

Early CCS This paragraph only applies to OpenSSL versions 1.0.1g and earlier. Since `CCS` is technically not a handshake message (e.g. it does not appear in the handshake log), it is not controlled by the client and server state machines in OpenSSL, but instead can (incorrectly) appear at any point after `ServerHello`. Receiving a `CCS` message triggers the setup of a record key derived from the session key; because of obscure DTLS constraints, OpenSSL allows derivation from an uninitialized session key.

This bug was first reported by Masashi Kikuchi as CVE-2014-0224. Depending on the OpenSSL version, it may enable both client and server impersonation attacks, where a man-in-the-middle first setups weak record keys early, by injecting `CCS` messages to both peers after `ServerHello`, and then let them complete their handshake, only intercepting the legitimate `CCS` messages (which would otherwise cause the weak keys to be overwritten with strong ones).

DH Certificate OpenSSL servers allow clients to omit the `ClientCertificateVerify` message after sending a Diffie-Hellman certificate, because such certificates cannot be used for signing. Instead, since the client share of the Diffie-Hellman exchange is taken from the certificate's public key, the ability to compute the pre-master secret of the session demonstrates to the server ownership of the certificate's private exponent.

However, we found that sending a `ClientKeyExchange` along with a DH certificate enables a new client impersonation attack, which we explain in Section V-B.

Server-Gated Crypto (SGC) OpenSSL servers have a legacy feature called SGC that allows clients to start over a handshake after receiving a `ServerHello`. Further code inspection reveals that the state created during the first exchange of hello messages is then supposed to be discarded completely. However, we found that some pieces of state that indicate whether some extensions had been sent by the client or not can linger from the first `ClientHello` to the new handshake.

Export RSA In legacy export RSA ciphersuites, the server sends a signed, but weak (at most 512 bits) RSA modulus in the `ServerKeyExchange` message. However, if such a message is received during a handshake that uses a stronger, non-export RSA ciphersuite, the weak ephemeral modulus will still be used to encrypt the client's pre-master secret. This leads to a new downgrade attack to export RSA that we explain in Section V-C.

Static DH We similarly observe that OpenSSL clients allow the server to skip the `ServerKeyExchange` message when a DHE or ECDHE ciphersuite is negotiated. If the server certificate contains, say, an ECDH public key, and the client does not receive a `ServerKeyExchange` message, then it will automatically rollback to static ECDH by using the public key from the server's certificate, resulting in the loss of forward-secrecy. This leads to an exploit against False Start that we describe in Section V-D.

IV-B Implementation Bugs in JSSE.

The Java Secure Socket Extension (JSSE) is the default security provider for a number of cryptographic functionalities in the Oracle and OpenJDK Java runtime environments. Sometimes called *SunJSSE*, it was originally developed by Sun and open-sourced along with the rest of its Java Development Kit (JDK) in 2007. Since then, it has been maintained by OpenJDK and Oracle. In the following, we refer to code in OpenJDK version 7, but the bugs have also been confirmed on versions 6 and 8 of both the OpenJDK and Oracle Java runtime environments.

On most machines, whenever a Java client or server uses the *SSLSocket* interface to connect to a peer, it uses the TLS implementation in JSSE. In our tests, JSSE clients and servers accepted many incorrect message sequences, including some where mandatory messages such as `ServerCCS` were skipped. To better understand the JSSE state machine, we carefully reviewed its source code from the OpenJDK repository.

The client and server handshake state machines are implemented separately in *ClientHandshaker.java* and *Server Handshaker.java*. Each message is given a number (based on its *HandshakeType* value in the TLS specification) to indicate its order in the handshake, and both state machines ensure that messages can only appear in increasing order, with two exceptions. The `HelloRequest` message (n^{o}0) can appear at any time and the `ClientCertificateVerify` (n^{o}15) appears out of order, but can only be received immediately after `ClientKeyExchange` (n^{o}16).

Client Flaws To handle optional messages that are specific to some ciphersuites, both client and server state machines allow messages to be skipped. For example, *ClientHandshaker* checks that the next message is always greater than the current state (unless it is a `HelloRequest`). Figure 6 depicts the state machine implemented by JSSE clients and servers, where the red arrows indicate the extra client transitions that are not allowed by TLS. Notably:

- JSSE clients allow servers to skip the `ServerCCS` message, and hence disable record-layer encryption.
- JSSE clients allow servers to skip any combination of the `ServerCertificate`, `ServerKeyExchange`, `ServerHelloDone` messages.

These transitions lead to the server impersonation attack on Java clients that we describe in Section V-A.

Server Flaws JSSE servers similarly allow clients to skip messages. In addition, they allow messages to be repeated due to another logical flaw. When processing the next message, *ServerHandshaker* checks that the message number is either greater than the previous message, or that the last message was a `ClientKeyExchange`, or that the current message is a `ClientCertificateVerify`, as coded below:

```
void processMessage(byte type, int message_len)
```

```
            throws IOException {
    if ((state > type)
        && (state != HandshakeMessage.ht_client_key_exchange
            && type != HandshakeMessage.ht_certificate_verify)) {
        throw new SSLProtocolException(
         "Handshake message sequence violation,
          state = " + state + ", type = " + type);
        }
    ... /* Process Message */
}
```

There are multiple coding bugs in the error-checking condition. The first inequality should be >= (to prevent repeated messages) and indeed this has been fixed in OpenJDK version 8. Moreover, the second conjunction in the if-condition (&&) should be a disjunction (||), and this bug remains to be fixed. The intention of the developers here was to address the numbering inconsistency between `ClientCertificateVerify` and `ClientKeyExchange` but instead this bug enables further illegal state transitions (shown in green on the left in Figure 6):

- JSSE servers allow clients to skip the `ServerCCS` message, and hence disable record-layer encryption.
- JSSE servers allow clients to skip any combination of the `ClientCertificate`, `ClientKeyExchange`, `ClientCertificateVerify` messages, although some of these errors are caught when processing the `ClientFinished`.
- JSSE servers allow clients to send any number of new `ClientHello` `ClientCertificate`, `ClientKeyExchange`, or `ClientCertificateVerify` messages after the first `ClientKeyExchange`.

We do not demonstrate any concrete exploits that rely on these server transitions in this paper, but we observe that by sending messages in carefully crafted sequences an attacker can cause the JSSE server to get into strange, unintended, and probably exploitable states similar to the other attacks in this paper.

IV-C IMPLEMENTATION BUGS IN OTHER IMPLEMENTATIONS. More briefly, we summarize the flaws that our tests found in other TLS implementations.

NSS Network Security Services (NSS) is a TLS library managed by Mozilla and used by popular web browsers like Firefox, Chrome, and Opera. NSS is typically used as a client and by inspecting our test results and the library source code, we found the following unexpected transitions:

- NSS clients allow servers to skip `ServerKeyExchange` during a `DHE` (or `ECDHE`) key exchange; it then treats the key exchange like static `DH` (or `ECDH`).
- During renegotiation, NSS clients accept `ApplicationData` between `ServerCCS` and `ServerFinished`.

The first of these leads to the attack on forward secrecy described in Section V-C. The second breaks a TLS secure channel invariant that `ApplicationData` should only be accepted encrypted under keys that have been authenticated by the server. It may be exploitable in scenarios where server certificates may change during renegotiation [see e.g. 9].

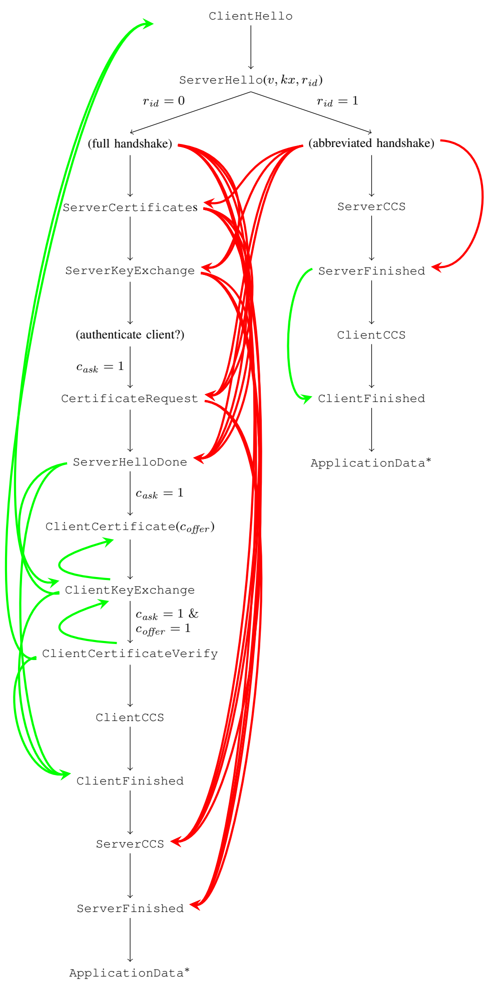

Fig. 6. JSSE Client and Server State Machines for HTTPS configurations. Unexpected transitions: client in red on the right, server in green on the left.

Mono Mono is an open source implementation of Microsoft's .NET Framework. It allows programs written for the .NET platform to be executed on non-Windows platforms and hence is commonly used for portability, for example on smartphones. Mono includes an implementation of .NET's *SslStream* interface (which implements TLS connections) in *Mono.Security. Protocol.Tls*. So, when a C# client or server written for the .NET platform is executed on Mono, it executes this TLS implementation instead of Microsoft's SChannel implementation.

We found the following unexpected transitions:

- Mono's TLS clients and servers allow the peer to skip the `CCS` message, hence disabling record encryption.
- Mono's TLS servers allow clients to skip the `ClientCertificateVerify` message even when a `ClientCertificate` was provided.
- Mono's TLS clients allow servers to send new `ServerCertificate` messages after `ServerKeyExchange`.

The second flaw leads to the client impersonation attack described in Section V-B.

The third allows a *certificate switching* attack, whereby a malicious server M can send one `ServerCertificate` and, just before the `ServerCCS`, send a new `ServerCertificate` for some other server S. At the end of the handshake, the Mono client would have authenticated M but would have recorded S's certificate in its session.

CyaSSL The CyaSSL TLS library (sometimes called yaSSL or wolfSSL) is a small TLS implementation designed to be used in embedded and resource-constrained applications, including the yaSSL web server. It has been used in a variety of popular open-source projects including MySQL and lighthttpd. Our tests reveal the following unexpected transitions, many of them similar to JSSE:

- Both CyaSSL servers and clients allow their peers to skip the `CCS` message and hence disable record encryption.
- CyaSSL clients allow servers to skip many messages, including `ServerKeyExchange` and `ServerHelloDone`.
- CyaSSL servers allow clients to skip many messages, notably including `ClientCertificateVerify`.

The first and second flaws above result in a full server impersonation attack on CyaSSL clients (Section V-A). The last results in a client impersonation attack on CyaSSL servers (Section V-B).

SecureTransport The default TLS library included on Apple operating system is called SecureTransport, and it was recently made open-source. The library is used primarily by web clients on OS X and iOS, including the Safari web browser. We found two unexpected behaviors:

- SecureTransport clients allow servers to send `CertificateRequest` before `ServerKeyExchange`.
- SecureTransport clients allow servers to send `ServerKeyExchange` even for `RSA` key exchanges.

The first violates a minor user interface invariant in `DHE` and `ECDHE` handshakes: users may be asked to choose their certificates a little too early, before the server has been authenticated. The second flaw can result in a rollback vulnerability, as described in Section V-D.

GnuTLS The GnuTLS library is a widely available open source TLS implementation that is often used as an alternative to OpenSSL, for example in clients like *wget* or SASL servers. Our tests on GnuTLS revealed only one minor deviation from the TLS state machine:

- GnuTLS servers allow a client to skip the `ClientCertificate` message entirely when the client does not wish to authenticate.

miTLS and others We also ran our tests against the MITLS implementation and did not find any deviant trace. miTLS is a verified implementation of TLS and is therefore very strict about the messages it generates and accepts.

We did not run systematic analyses with closed-source TLS libraries such as Microsoft's SChannel, because our analysis technique required repeatedly looking through source code to interpret errors (or sometimes the absence of errors). In general, we believe our method is better suited to developers who wish to test their own implementations, rather than to analysts who wish to perform black-box testing of closed-source code.

V. Attacks on TLS Implementations

We describe a series of attacks on TLS implementations that exploits their state machine flaws. We then discuss disclosure status and upcoming patches for various implementations.

V-A Early Finished: Server Impersonation (Java,CyaSSL). 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 as follows, without needing any interaction with S:

1) C sends `ClientHello`
2) M sends `ServerHello`
3) M sends `ServerCertificate` with S's certificate
4) M sends `ServerFinished`, by computing its contents using an empty master secret (length 0)
5) C treats the handshake as complete
6) C sends `ApplicationData` (its request) *in the clear*
7) M sends `ApplicationData` (its response) *in the clear*
8) C accepts M's application data as if it came from S

A FlexTLS script that implements this scenario is given in figure 9, in the appendix.

Impact At the end of the attack above, C thinks it has a secure connection to S, but is in fact connected to M. Even if C were to carefully inspect the received certificate, it would find a perfectly valid certificate for S (that anyone can download and review). Hence, the security guarantees of TLS are completely broken. An attacker can impersonate *any* TLS server to a JSSE client. Furthermore, all the (supposedly confidential and authenticated) traffic between C and M is sent in the clear without any protection.

Why does it work? At step 4, M skips all the handshake messages to go straight to `ServerFinished`. As we saw in the previous section, this is acceptable to the JSSE client state machine.

The only challenge for the attacker is to be able to produce a `ServerFinished` message that would be acceptable to the client. The content of this message is a message authentication code (MAC) applied to the current handshake transcript and keyed by the session master secret. However, at this point in the state machine, the various session secrets and keys have not yet been set up. In the JSSE *ClientHandshaker*, the *masterSecret* field is still *null*. It turns out that the TLS PRF function in SunJSSE uses a key generator that is happy to accept a null *masterSecret* and treat it as if it were an empty array. Hence, all M has to do is to use an empty master secret and the log of messages (1-3) to create the finished message.

If M had sent a `ServerCCS` before `ServerFinished`, then the client C would have tried to generate connection keys based on the *null* master secret, and that the key generation functions in SunJSSE *do* raise a null pointer exception in this case. Hence, our attack crucially relies on the Java client allowing the server to skip the `ServerCCS` message.

Attacking CyaSSL The attack on CyaSSL is very similar to that on JSSE, and relies on the same state machine bugs, which allow the attacker to skip handshake messages and the `ServerCCS`. The only difference is in the content of the `ServerFinished`: here M does not compute a MAC, instead it sends a byte array consisting of 12 zeroes.

In CyaSSL (which is written in C), the expected content of the `ServerFinished` message is computed whenever the client receives a `ServerCCS` message. The handler for the `ServerCCS` message uses the current log and master secret to compute the transcript MAC (which in TLS returns 12 bytes) and stores it in a pre-allocated byte array. The handler for the `ServerFinished` message then simply compares the content of the received message with the stored MAC value and completes the handshake if they match.

In our attack, M skipped the `ServerCCS` message. Consequently, the byte array that stores the transcript MAC remains uninitialized, and in most runtime environments this array contains zeroes. Consequently, the `ServerFinished` message filled with zeroes sent by M will match the expected value and the connection succeeds.

Since the attack relies on uninitialized memory, it may fail if the memory block contains non-zeroes. In our experiments, the attack always succeeded on the first run of the client (when the memory was unused), but sometimes failed on subsequent runs. Otherwise, the rest of the attack works as in Java, and has the same disastrous impact on CyaSSL clients.

V-B SKIP VERIFY: CLIENT IMPERSONATION (MONO, CYASSL, OPENSSL). Suppose a malicious client M connects to a Mono server S that requires client authentication. M can then impersonate any user u at S as follows:

1) M sends `ClientHello`
2) S sends its `ServerHello` flight, requesting client authentication by including a `CertificateRequest`
3) M sends u's certificate in its `ClientCertificate`
4) M sends its `ClientKeyExchange`
5) M skips the `ClientCertificateVerify`
6) M sends `ClientCCS` and `ClientFinished`
7) S sends `ServerCCS` and `ServerFinished`
8) M sends `ApplicationData`
9) S accepts this data as authenticated by u

Hence, M has logged in as u to S. Even if S inspects the certificate stored in the session, it will find no discrepancy.

At step 5, M skipped the only message that proves knowledge of the private key of u's certificate, resulting in an impersonation attack. Why would S allow such a crucial message to be omitted? The `ClientCertificateVerify` message is required when the server sends a `Certificate-Request` and when the client sends a non-empty `Client-Certificate` message. Yet, the Mono server state machine considers `ClientCertificateVerify` to be always optional, allowing the attack.

Attacking CyaSSL The CyaSSL server admits a similar client impersonation attack.

The first difference is that M must also skip the `ClientCCS` message at step 6. The reason is that, in the CyaSSL server, the handler for the `ClientCCS` message is the one that checks that the `ClientCertificateVerify` message was received. So, by skipping these messages we can bypass the check altogether.

The second difference is that M must then send a `Client-Finished` message that contains 12 zeroes, rather than the correct MAC value. This is because on the CyaSSL server, as on the CyaSSL client discussed above, it is the handler for the `ClientCCS` message that computes and stores the expected MAC value for the `ClientFinished` message. So, like in the attack on the client, M needs to send zeroes to match the uninitialized MAC on the CyaSSL server.

The server accepts the `ClientFinished` and then accepts unencrypted data from M as if it were sent by u. We observe that even if CyaSSL were more strict about requiring `ClientCertificateVerify`, the bug that allows `ClientCCS` to be skipped would still be enough to enable a man-in-the middle to inject application data attributed to u.

Attacking OpenSSL In the OpenSSL server, the `Client-CertificateVerify` message is properly expected whenever a client certificate has been presented, except when the client sends a static Diffie-Hellman certificate. The motivation behind this design is that, in static DH ciphersuites, the client is allowed to authenticate the key exchange by using the static DH key sent in the `ClientCertificate`; in this case, the client then skips both the `ClientKeyExchange` and `Client-CertificateVerify` messages. However, because of a bug in OpenSSL, client authentication can be bypassed in two cases by confusing the static and ephemeral state machine composite implementation.

In both the static DH and ephemeral DHE key exchanges, the attacker M can send an honest user u's static DH certificate, then send its own ephemeral keys in a `ClientKey-Exchange` and skip the `ClientCertificateVerify`. The server will use the ephemeral keys from the `Client-KeyExchange` (ignoring those in the certificate), and will report u's identity to the application. Consequently, an attacker is able to impersonate the owner of any static Diffie-Hellman certificate at any OpenSSL server.

V-C SKIP SERVERKEYEXCHANGE: FORWARD SECRECY ROLLBACK (NSS, OPENSSL). To counter strong adversaries who may be able to compromise the private keys of trusted server certificates [32], TLS clients and servers are encouraged to use forward secret ciphersuites such a DHE and ECDHE, which guarantee that messages encrypted under the resulting session keys cannot be decrypted, even if the client and server certificates are subsequently compromised. In particular, forward secrecy is one of the conditions for enabling False Start [23] in some browsers.[1]

Suppose an NSS or OpenSSL client C is trying to connect to a trusted server S. We show how a man-in-the-middle attacker M can force C to use a (non-forward secret) static key exchange (DH|ECDH) even if both C and S only support ephemeral ciphersuites (DHE|ECDHE).

1) C sends `ClientHello` with only ECDHE ciphersuites
2) S sends `ServerHello` picking an ECDHE key exchange with ECDSA signatures

[1]https://bugzilla.mozilla.org/show_bug.cgi?id=920248

3) S sends `ServerCertificate` containing S's ECDSA certificate
4) S sends `ServerKeyExchange` with its ephemeral parameters but M intercepts this message and prevents it from reaching C
5) S sends `ServerHelloDone`
6) C sends `ClientKeyExchange`, `ClientCCS` and `ClientFinished`
7) C sends `ApplicationData` d to S
8) M intercepts d and closes the connection

When the attacker suppresses the `ServerKeyExchange` message in step 4, the client should reject the subsequent message since it does not conform to the key exchange. Instead, NSS and OpenSSL will rollback to a non-ephemeral ECDH key exchange: C picks the static public key of S's ECDSA certificate as the server share of the key exchange and continues the handshake.

Since M has tampered with the handshake, it will not be able to complete the handshake: C's `ClientFinished` message is unacceptable to S and vice-versa. However, if False Start is enabled, then, by step 7, C would already have sent `ApplicationData` encrypted under the new (non forward-secret) session keys.

Consequently, if an active network attacker is willing to tamper with client-server connections, it can collect False Start application data sent by clients. The attacker can subsequently compromise or compel the server's ECDSA private key to decrypt this data, which may contain sensitive authentication credentials and other private information.

V-D Inject ServerKeyExchange: RSA_EXPORT Flashback (OpenSSL, SecureTransport). Due to US export regulations before 2000, SSL version 3 and TLS version 1 include several ciphersuites that use sub-strength 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 [12] and with advancements in hardware are now considered broken. Consequently, since export regulations were relaxed, mainstream web browsers no longer offer or accept export ciphersuites. However, TLS implementations still include legacy code to handle these ciphersuites, and some servers continue to support them. We show that this legacy code causes a client to "flashback" from RSA to RSA_EXPORT.

Suppose a client C wants to connect to a trusted server S using RSA, but the server S also supports some RSA_EXPORT ciphersuites. Then a man-in-the-middle attacker M can fool C into accepting a weak RSA public key for S, as follows:

1) C sends `ClientHello` with an RSA ciphersuite
2) M replaces the ciphersuite with an RSA_EXPORT ciphersuite and forwards the `ClientHello` message to S
3) S sends `ServerHello` for an RSA_EXPORT ciphersuite
4) M replaces the ciphersuite with an RSA ciphersuite and forwards the `ServerHello` message to C
5) S sends `ServerCertificate` with its strong (2048-bit) RSA public key, and M forwards the message to C
6) S sends a `ServerKeyExchange` message containing a weak (512-bit) ephemeral RSA public key (modulus N), and M forwards the message to C
7) S sends a `ServerHelloDone` that M forwards to C
8) C sends its `ClientKeyExchange`, `ClientCCS` and `ClientFinished`
9) M factors N to find the ephemeral private key. M can now decrypt the pre-master secret from the `Client-KeyExchange` and derive all the secret secrets
10) M sends `ServerCCS` and `ServerFinished` to complete the handshake
11) C sends `ApplicationData` to S and M can read it
12) M sends `ApplicationData` to C and C accepts it as coming from S

At step 6, C receives a `ServerKeyExchange` message even though it is running an RSA ciphersuite, and this message should be rejected. However, because of a state machine composition bug in both OpenSSL and SecureTransport, this message is silently accepted and the server's strong public key (from the certificate) is replaced with the weak public key in the `ServerKeyExchange`.

The main challenge that remains for the attacker M is to be able to factor the 512-bit modulus and recover the ephemeral private key. First, we observe that 512-bit factorization is currently solvable at most in weeks, and the hardware is rapidly getting better. Second, we note that since generating ephemeral RSA keys on-the-fly can be quite expensive, many implementations of RSA_EXPORT (including OpenSSL) allow servers to pre-generate, cache, and reuse these public keys for the lifetime of the server (typically measured in days). Hence, if the attacker cannot break the key during the handshake, he may have several days to retry the attack.

V-E Summary and Responsible Disclosure. Of the eight TLS implementation we tested, we found serious state machine flaws in six, and were able to exploit them and mount nine different attacks, including five impersonation attacks that break the stated authentication guarantees of TLS.

Almost all of the implementations allowed some messages to be skipped even though they were required for the current handshake. This results from a naive composition of handshake state machines and is the primary source of attacks.

Three implementations (Java, Mono, CyaSSL) incorrectly allowed the `CCS` messages to be skipped. Considering also the recent Early CCS attack on OpenSSL, we conclude that the handling of `CCS` messages in TLS state machines is an important weak point.

Many implementations (OpenSSL, Java, Mono) allowed messages to be repeated. We leave the exploration of exploits based on these flaws for future work.

We reported all the flaws and attacks presented in this paper to the various TLS libraries. They were acknowledged and patches are in development in consultation with us. We briefly report on their status below, and will be better able to describe them in the final version of this paper.

- OpenSSLhas released an update (1.0.1k) that fixes our reported flaws.
- Oracle and OpenJDK have released an update fixing our

reported flaws as part of the January 2014 critical patch update for all versions of Java.

- Mono is testing an update to *Mono.Security.Tls*.
- CyaSSL has released a security update (3.3.0).
- NSS has an active bug report (id 1086145) on forward secrecy rollback and a fix is expected for Firefox 35.
- SecureTransport is testing an update.

Since some of these updates have not been released at the current time, the attacks in this paper should be treated as confidential.

VI. A VERIFIED STATE MACHINE FOR OPENSSL

Implementing composite state machines for TLS has proven to be hard and error-prone. Systematic state machine testing can be useful to uncover bugs but does not guarantee that all flaws have been found and eliminated. Instead, it seems valuable to formally verify 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 more valuable to 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. 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, proof for the full state machine of OpenSSL using the Frama-C verification tool.

OpenSSL Clients and Servers The OpenSSL client and server state machines for SSLv3 and TLSv1.0-TLSv1.2 are implemented in *ssl/s3_clnt.c* and *ssl/s3_srvr.c*, respectively. Both state machines maintain a data structure of type *SSL* that has about 94 fields, including negotiation parameters like the version and ciphersuite, cryptographic material like session keys and certificates, running hashes of the handshake log, and other data specific to various TLS extensions.

Both the client and the server implement the state machine depicted in Figure 8 as an infinite loop with a large switch statement, where each case corresponds to a different state, roughly one for each message in the protocol. Depending on the state, the switch statement either calls a *ssl3_send_∗* function to construct and send a message or calls a *ssl3_get_∗* function to receive and process a message.

For example, when the OpenSSL client is in the state *SSL3_ST_CR_KEY_EXCH_A*, it expects to receive a ServerKeyExchange, so it calls the function *ssl3_get_key_exchange*(*s*). This function in turn calls *ssl3_get_message* (in *s3_both.c*) and asks to receive *any* handshake message. If the received message is a ServerKeyExchange, it processes the message; otherwise it assumes that the message was optional and returns control to the state machine which transitions to the next state (to try and process the message as a CertificateRequest). If the ServerKeyExchange message was in fact required, it may only be discovered later when the client tries to send the ClientKeyExchange message.

Due to their complex handling of optional messages, it is often difficult to understand whether the OpenSSL client or server correctly implements the intended state machine. (Indeed, the flaws discussed in this paper indicate that they do not.) Furthermore, the correlation between the message sequence and the *SSL* structure (including the handshake hashes) is easy to get wrong.

A new state machine We propose a new state machine structure for OpenSSL that makes the allowed message sequences more explicit and easier to verify.

In addition to the full *SSL* data structure that is maintained and updated by the OpenSSL messaging functions, we define a separate data structure that includes only those elements that we need to track the message sequences allowed by Figure 8:

```
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 ntick; // n_tick
  int renegotiation; // reneg = 1 if renegotiating

  Msg_type last_message; // previous message type
  unsigned char∗ log; // full handshake log
  unsigned int log_length;
} STATE;
```

The structure contains various negotiation parameters: a *role* that indicates whether the current state machine is being run in a client or a server, the protocol version (v in Figure 8), the key exchange method (kx), the client authentication mode (c_{ask}, c_{offer}), and flags that indicate whether the current handshake is a resumption or a renegotiation, and whether the server sends a ServerNewSessionTicket. We represent each field by an **enum** that includes an *UNDEFINED* value to denote the initial state. The server sets all the fields except *client_auth* immediately after ServerHello. The client must wait until later in the handshake to set *resumption*, *client_auth* and *ntick*.

To record the current state within the handshake, the structure keeps the type of the last message received. It also keeps the full handshake log as a byte array. We use this array to verify our invariants about the state machine, but in production environments it will be replaced by running hashes of the log.

The core of our state machine is in one function:

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

This function takes the current state (*ssl*,*st*), the next message to send or receive *msg*, the content type (handshake/CCS/alert/application data) and direction (outgoing/incoming) of the message. Whenever a message is received by the record layer, this function is called. It then executes one step of the state machine in Figure 8 to check whether the incoming message is allowed in the current state. If it is, it calls the corresponding message handler, which processes the message and may in turn want to send some messages by calling *ssl3_next_message* with an outgoing message. For an outgoing message, the function again checks whether it is allowed by the state machine before writing it out to the record layer. In other words, *ssl3_next_message* is called on all incoming and outgoing messages. It enforces the state machine and maintains the handshake log for the current message sequence.

We were able to reuse the OpenSSL message handlers (with small modifications). We write our own simple message parsing functions to extract the handshake message type, to extract the protocol version and key exchange method from the `ServerHello`, and to check for empty certificates.

Experimental Evaluation We tested our new state machine implementation in two ways.

First, we checked that our new state machine does not inhibit compliant message sequences for ciphersuites supported by OpenSSL. To this end, we implemented our state machine as an inline reference monitor. As before, the function *ssl3_get_message* is called whenever a message is to be sent or received. However, it does not itself call any message handlers; it simply returns success or failure based on whether the incoming or outgoing message is allowed. Other than this modification, messages are processed by the usual OpenSSL machine. In effect, our new state machine runs in parallel with OpenSSL on the same traces.

We ran this monitored version of OpenSSL against various implementations and against OpenSSL itself (using its inbuilt tests). We tested that our inline monitor does not flag any errors for these valid traces. (In the process, we found and fixed some early bugs in our state machine.)

Second, we checked that our new state machine does prevent the deviant trace presented of Section III. We ran our monitored OpenSSL implementation against a FLEXTLS peer running deviant traces and, in every case, our monitor flagged an error. In other words, OpenSSL with our new state machine would not flag any traces in Table I.

Logical Specification of the State Machine To gain further confidence in our new state machine, we formalized the allowed message traces of Figure 8 as a logical invariant to be maintained by *ssl3_next_message*. Our invariant is called *isValidState* and is depicted in Figure 7.

The initial state is specified by the predicate *StateAfterInitialState*, which requires that the state structure be properly initialized. The predicate *isValidState* says that the current state structure should be consistent with either the initial state or the expected state after receiving some message; it has a disjunct for every message handled by our state machine.

For example, after `ServerHelloDone` the current state *st* must satisfy the predicate *StateAfterServerHelloDone*. This predicate states that there must exist a previous state *prev* and a new (*message*), such that the following holds:

- *message* must be a `ServerHelloDone`,
- *st→last_message* must be *S_HD* (a *Msg_type* denoting `ServerHelloDone`),
- *st→log* must be the concatenation of *prev→log* and the new *message*,
- and for each incoming edge in the state machine:
  - the previous state *prev* must an allowed predecessor (a valid state after an allowed previous message),
  - if the previous message was `CertificateRequest` then *st→client_auth* remains unchanged from *prev →client_auth*; in all other cases it must be set to *AUTH_NONE*
  - (plus other conditions to account for other ciphersuites)

Predicates like *StateAfterServerHelloDone* can be directly encoded by looking at the state machine and do not have to deal with implementation details. Indeed, our state predicates look remarkably similar to (and were inspired by) the *log predicates* used in the cryptographic verification of MITLS [6]. The properties they capture depend only on the TLS specification; except for syntactic differences, they are independent of the programming language.

Verification with Frama-C To mechanically verify that our state machine implementation satisfies the *isValidState* specification, we use the C verification tool Frama-C.[2]. We annotate our code with logical assertions and requirements in Frama-C's specification language, called ACSL. For example, the logical contract on the inline monitor variant of our state monitor is written as follows (embedded within a /∗@ ... @∗/ comment).

We read this contract bottom-up. The main pre-condition (**requires**) is that the state must be valid when the function is called (*isValidState*(*st*)). (The OpenSSL state *SSL* is not used by the monitor.) The post-condition (**ensures**) states that the function either rejects the message or returns a valid state. That is, *isValidState* is an invariant for error-free runs.

Moving up, the next block of pre-conditions requires that the areas of memory pointed to by various variables do not intersect. In particular, the given *msg*, state *st*, and log *st→log* , must all be disjoint blocks of memory. This pre-condition is required for verification. In particular, when *ssl3_next_message* tries to copy *msg* over to the end of the *log*, it uses *memcpy*, which has a logical pre-condition in Frama-C (reflecting its input assumptions) that the two arrays are disjoint.

The first set of pre-conditions require that the pointers given to the function be valid, that is, they must be non-null and lie within validly allocated areas of memory that are owned by the current process. These annotations are required for Frama-C to prove memory safety for our code: that is, all our memory accesses are valid, and that our code does not accidentally overrun buffers or access null-pointers.

From the viewpoint of the code that uses our state machine (the OpenSSL client or server) the preconditions specified here require that the caller provide *ssl3_next_message* with validly allocated and separated data structures. Otherwise, we cannot give any functional guarantees.

Formal Evaluation Our state machine is written in about 750 lines of code, about 250 lines of which are message processing functions. This is about the same length as the current OpenSSL state machine.

The Frama-C specification is written in a separate file and takes about 460 lines of first-order-logic to describe the state

[2]http://frama-c.com

```
predicate isValidState(STATE *state) =
  StateAfterInitialState(state)    ||
  StateAfterClientHello(state)    ||
  StateAfterServerHello(state)    ||
  StateAfterServerCertificate(state)   ||
  StateAfterServerKeyExchange(state)   ||
  StateAfterServerCertificateRequest(state)  ||
  StateAfterServerHelloDone(state)   ||
  StateAfterClientCertificate(state)   ||
  StateAfterClientKeyExchange(state)  ||
  StateAfterClientCertificateVerify(state)  ||
  StateAfterServerNewSessionTicket(state)  ||
  StateAfterServerCCS(state)    ||
  StateAfterServerFin(state)    ||
  StateAfterClientCCS(state)    ||
  StateAfterClientFin(state)   ||
  StateAfterClientCCSLastMsg(state)   ||
  StateAfterClientFinLastMsg(state)  ;

predicate StateAfterInitialState(STATE *state) =
  state→version == UNDEFINED_PV &&
  state→role == UNDEFINED_ROLE &&
  state→kx == UNDEFINED_CS &&
  state→last_message == UNDEFINED_TYPE  &&
  state→log_length == 0 &&
  state→client_auth == UNDEFINED_AUTH &&
  state→resumption == UNDEFINED_RES &&
  state→renegotiation == UNDEFINED_RENEG &&
  state→ntick == UNDEFINED_TICK;

predicate StateAfterServerHelloDone(STATE *st) =
  ∃ STATE *prev, unsigned char *message,
    unsigned int len, int direction;
      isServerHelloDone(message,len,handshake) &&
      st→last_message == S_HD &&
      HaveSameStateValuesButClientAuth_E(st, prev) &&
      MessageAddedToLog_E(st, prev, message, len) &&
      ( (StateAfterServerCertificate(prev) &&
          st→kx == CS_RSA &&
          st→client_auth == NO_AUTH)
      || (StateAfterServerKeyExchange(prev) &&
          (st→kx == DHE || st→kx == ECDHE) &&
          st→client_auth == NO_AUTH)
      || (StateAfterServerCertificateRequest(prev) &&
          (st→kx == DHE || st→kx == ECDHE
            || st→kx = CS_RSA) &&
          st→client_auth == s→client_auth)
      || .... /* other ciphersuites */
      );
```

Fig. 7. Logical Specification of State Machine (Excerpt)

```
/*@
  requires \valid(st);
  requires \valid(msg+(0..(len−1)));
  requires \valid(st→log+(0..(st→log_length+len−1)));

  requires \separated(msg+(0..(len−1)),
                      st+(0..(sizeof(st)−1)));
  requires \separated(msg+(0..(len−1)),
                      st→log+(0..(st→log_length + len−1)));
  requires \separated(st+(0..(sizeof(st)−1)),
                      st→log+(0..(st→log_length+len−1)));

  requires isValidState(st)
  ensures (isValidState(st) && \result == ACCEPT)
          || \result == REJECT;
  @*/
int ssl3_next_message(SSL* s, STATE *st,
      unsigned char* msg, int len,
      int direction, unsigned char content_type);
```

machine. To verify the code, we ran Frama-C which generates proof obligations for multiple SMT solvers. We used Alt-Ergo to verify some obligations and Z3 for others (the two solvers have different proficiencies). Verifying each function took about 2 minutes, resulting in a total verification time of about 30 minutes.

Technically, to verify the code in a reasonable amount of time, we had to provide many annotations (intermediate lemmas) to each function. The total number of annotations in the file amounts to 900 lines. Adding a single annotation often halves the verification time of a function. Still, our code is still evolving and it may be possible to get better verification times with fewer annotations.

One may question the value of a logical specification that is almost as long as the code being verified (460 lines is all we have to trust). What, besides being declarative, makes it a better specification than the code itself? And at that relative size, how can we be confident that the predicates themselves are not as buggy as the code?

We find our specification and its verification useful in several ways. First, in addition to our state invariant, we also prove memory safety for our code, a mundane but important goal for C programs. Second, our predicates provide an alternative 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 can use off-the-shelf interactive proof assistants for deriving more advanced properties.

To illustrate this point, using the Coq proof assistant, we formally establish that the valid logs are unambiguous: equal logs imply equal states:

```
theorem UnambiguousValidity: ∀ STATE *s1, *s2;
  (isValidState(s1) && isValidState(s2)
   && LogEquality(s1,s2))
   ==> HaveSameStateValues_E(s1,s2);
```

This property is a key lemma for proving the security of TLS, inasmuch as the logs (not the states they encode) are authenticated in `Finished` messages at the end of the handshake. Its proof is similar to the one for the unambiguity of the logs in miTLS. However, the Frama-C predicates are more abstract, they better capture what makes the log unambiguous, and they cover a more complete set of ciphersuites.

VII. Towards Security Theorems for OpenSSL

In the previous section, we verified the functional correctness of our state machine for OpenSSL (a refinement) and proved that our logical specification is unambiguous (a consistency check). We did not, however, prove any integrity or confidentiality properties. How far are we from a security theorem for OpenSSL?

Traditional cryptographic proofs for TLS focus on *single ciphersuite security*. They prove, for example, that the mutually-authenticated DHE handshake is secure when used with a secure record protocol [19]. One may attempt to extend these formal results to the fragment of OpenSSL that implements them, but this would still be thousands of lines of code. Our experience in verifying our small state machine in C suggests that verifying all this code might be feasible, but nevertheless remains a daunting task.

The miTLS verified implementation securely composes several *DHE* and *RSA* ciphersuites in TLS [6] and guarantees connection security when a ciphersuite satisfying a cryptographic strength predicate (α) is negotiated. Their proof technique requires that the code for *all* supported ciphersuites be verified to guarantee that connections with different ciphersuites (but possibly the same long-term keys and short-term session secrets) cannot confuse one another. Even if this verified code could be ported over to C, verifying all the remaining ciphersuites supported by OpenSSL seems infeasible.

A more practical goal may be to target 1-out-of-k ciphersuite security. Suppose we can verify, with some concerted effort, all the messaging functions for some strong ciphersuite in OpenSSL (e.g. *TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256*). The goal is then to prove that, no matter which other ciphersuites are supported, if the client and server choose this ciphersuite, then the resulting connection is secure. This could for instance be captured in a multi-ciphersuite version of the widely used *authenticated and confidential channel establishment* (ACCE) definition [19, 22]). Bergsma et al. [4] give such a definition, but require all ciphersuites to be secure. One could instead define an α-ACCE notion with a strength predicate à la miTLS that only guarantees channel security when the strong ciphersuite is negotiated.

The first step to prove this property is to show that the OpenSSL state machine correctly implements our chosen ciphersuite, and that message sequences for this ciphersuite are disjoint from all other supported ciphersuites. These are indeed the properties we have already proved.

The second hurdle is to show that the use of the same long-term signing key in different ciphersuites is safe. In current versions of TLS, this is a difficult property to guarantee because of the possibility of cross-protocol attacks [26]. Indeed, these attacks are the main reason why Bergsma et al. [4] found it difficult to transfer their multi-ciphersuite security results for SSH over to TLS. The core problem is that the `ServerKeyExchange` message in TLS requires a server signature on one of many ambiguous formats. However, the new format of this message in TLS 1.3 [16] is designed to prevent these attacks, and may make 1-out-of-k ciphersuite security proofs easier.

The third challenge is to show that the session secrets of our verified ciphersuite are cryptographically independent from any other ciphersuite. Current versions of TLS do not guarantee this property, and indeed the lack of context-bound session secrets can be exploited by man-in-the-middle attacks [9]. However, the recently proposed session-hash extension [7] guarantees that the master secret and connection keys generated in connections with different ciphersuites will be independent when their logs are unambiguous as guaranteed by the *UnambiguousValidity* theorem. We believe that this extension would significantly simplify our verification efforts.

To summarize, our proofs about the OpenSSL state machine are an important first step toward a security theorem, but many open problems remain to achieve a verified TLS implementation that includes legacy code for insecure ciphersuites.

VIII. Related Work

Attacks Wagner and Schneier [34] discuss various attacks in the context of SSL 3.0, and their analysis has proved prescient for many attacks. For instance, they predicted the cross-ciphersuite attack of Mavrogiannopoulos et al. [26] by observing that the ephemeral key exchange parameters signed by TLS servers mostly contain random data that could be misinterpreted. The omission of the `ChangeCipherSpec` message from the handshake transcript is also mentioned, and indeed, a recent and serious attack against OpenSSL (CVE-2014-0224) relies on an attacker being able to change the point at which `CCS` is received.

Attacks on the incorrect composition of various TLS protocol modes include the renegotiation [30, 31], Alert [6], and Triple Handshake [9] attacks. Those flaws can be blamed in part to the state machine being underspecified in the standard—the last two attacks were discovered while designing the state machine of miTLS.

Cryptographic attacks target specific constructions used in TLS such as RSA encryption [11, 21, 27] and MAC-then-Encrypt [33, 28, 1].

Code Analyses Lawall et al. [24] use the Coccinelle framework to detect incorrect checks on values returned by the OpenSSL API. Pironti and Jürjens [29] generate a provably correct TLS proxy that intercepts invalid protocol messages and shuts down malicious connections. *TrustInSoft* advertises the PolarSSL verification kit and its use of Frama-C.[3]

Chaki and Datta [13] verify the SSL 2.0/3.0 handshake of OpenSSL using model checking of fixed configurations and found rollback attacks. Jürjens [20], Avalle et al. [3] verify Java implementations of the handshake protocol using

[3]http://trust-in-soft.com/polarssl-verification-kit/

logical provers. Goubault-Larrecq and Parrennes [18] analyze OpenSSL for reachability properties using Horn clauses.

Bhargavan et al. [5] extract and verify ProVerif and CryptoVerif models from an F# implementation of TLS. Dupressoir et al. [17] use the VCC general purpose C verifier to prove the security of C implementations of security protocols, but they do not scale their methodology to the TLS handshake.

Proofs Cryptographers primarily developed proofs of specific key exchanges when running in isolation: DHE [19], RSA [22], PSK [25]. Two notable exceptions are: Bhargavan et al. [6, 8] proved that composite RSA and DHE are jointly secure in an implementation written in *F#* and verified using refinement types. Bergsma et al. [4] analyze the multi-ciphersuite security of SSH using a black-box composition result but fall short of analyzing TLS because of cross-protocols attacks. Almeida et al. [2] prove computational security and side channel resilience for machine code implementing cryptographic primitives, generated from EasyCrypt.

IX. Conclusion

While security analyses of cryptographic implementations focused on flaws in specific protocol constructions, the state machines that control their flow of messages have escaped scrutiny. Using a simple but systematic state-machine exploration, we discovered serious flaws in most 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 (fuzzing) as well as formal proofs of correctness for critical components. Current TLS implementations are far from perfect, but we hope that improvements in the protocol and in the available verification technology will bring their formal automated verification within reach.

Online Materials

Our attack scripts, test trace generators, and OpenSSL state monitor can be obtained from:

https://smacktls.com
(username:Oakland, password:Referee)

References

[1] N. J. AlFardan and K. G. Paterson. Lucky thirteen: breaking the TLS and DTLS record protocols. In *IEEE S&P*, 2013.

[2] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In *ACM CCS*, pages 1217–1230, 2013.

[3] M. Avalle, A. Pironti, D. Pozza, and R. Sisto. JavaSPI: A framework for security protocol implementation. *International J. of Secure Software Engineering*, 2:34–48, 2011.

[4] F. Bergsma, B. Dowling, F. Kohlar, J. Schwenk, and D. Stebila. Multi-ciphersuite security of the Secure Shell (SSH) protocol. In *ACM CCS*, 2014.

[5] K. Bhargavan, C. Fournet, R. Corin, and E. Zălinescu. Verified Cryptographic Implementations for TLS. *ACM TISSEC*, 15(1):1–32, 2012.

[6] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptographic security. In *IEEE S&P*, 2013.

[7] K. Bhargavan, A. Delignat-Lavaud, A. Pironti, A. Langley, and M. Ray. Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension. IETF Internet Draft, 2014.

[8] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella-Béguelin. Proving the TLS handshake secure (as it is). In *CRYPTO*, 2014.

[9] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti, and P.-Y. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE S&P (Oakland'14)*, 2014.

[10] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) Extensions. IETF RFC 3546, 2003.

[11] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on RSA encryption standard PKCS #1. In *CRYPTO'98*, pages 1–12, 1998.

[12] S. Cavallar, B. Dodson, A. Lenstra, W. Lioen, P. Montgomery, B. Murphy, H. te Riele, K. Aardal, J. Gilchrist, G. Guillerm, P. Leyland, J. Marchand, F. Morain, A. Muffett, C. Putnam, and P. Zimmermann. Factorization of a 512-bit rsa modulus. In *EUROCRYPT 2000*, Lecture Notes in Computer Science, pages 1–18. 2000.

[13] S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. In *IEEE CSF*, 2009.

[14] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C. In *Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.

[15] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. IETF RFC 5246, 2008.

[16] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. Internet Draft, 2014.

[17] F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann. Guiding a general-purpose C verifier to prove cryptographic protocols. *Journal of Computer Security*, 22(5):823–866, 2014.

[18] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *VMCAI*, pages 363–379, 2005.

[19] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. On the security of TLS-DHE in the standard model. In *CRYPTO*, 2012.

[20] J. Jürjens. Security analysis of crypto-based java programs using automated theorem provers. In *ASE'06*, pages 167–176, 2006.

[21] V. Klima, O. Pokorny, and T. Rosa. Attacking RSA-based sessions in SSL/TLS. In *CHES*, pages 426–440, 2003.

[22] H. Krawczyk, K. G. Paterson, and H. Wee. On the security of the TLS protocol: A systematic analysis. In *CRYPTO*, 2013.

[23] N. M. Langley, A. and B. Moeller. Transport Layer Security (TLS) False Start. Internet Draft, 2010.

[24] J. Lawall, B. Laurie, R. R. Hansen, N. Palix, and G. Muller. Finding error handling bugs in OpenSSL using coccinelle. In *EDCC'10*, 2010.

[25] Y. Li, S. Schäge, Z. Yang, F. Kohlar, and J. Schwenk. On the security of the pre-shared key ciphersuites of tls. In *Public-Key Cryptography (PKC'14)*. 2014.

[26] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel. A cross-protocol attack on the TLS protocol. In *ACM CCS*, 2012.

[27] C. Meyer, J. Somorovsky, E. Weiss, J. Schwenk, S. Schinzel, and E. Tews. Revisiting SSL/TLS implementations: New bleichenbacher side channels and attacks. In *USENIX Security*, 2014.

[28] K. G. Paterson, T. Ristenpart, and T. Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In *ASIACRYPT*, 2011.

[29] A. Pironti and J. Jürjens. Formally-based black-box monitoring of security protocols. In *International Symposium on Engineering Secure Software and Systems*, page 79–95, 2010.

[30] M. Ray and S. Dispensa. Renegotiating TLS, 2009.

[31] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. TLS renegotiation indication extension. IETF RFC 5746, 2010.

[32] C. Soghoian and S. Stamm. Certified lies: Detecting and defeating government interception attacks against SSL. In *FC*. 2012.

[33] S. Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ... In L. R. Knudsen, editor, *EUROCRYPT*, pages 534–546, 2002.

[34] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *USENIX Electronic Commerce*, 1996.

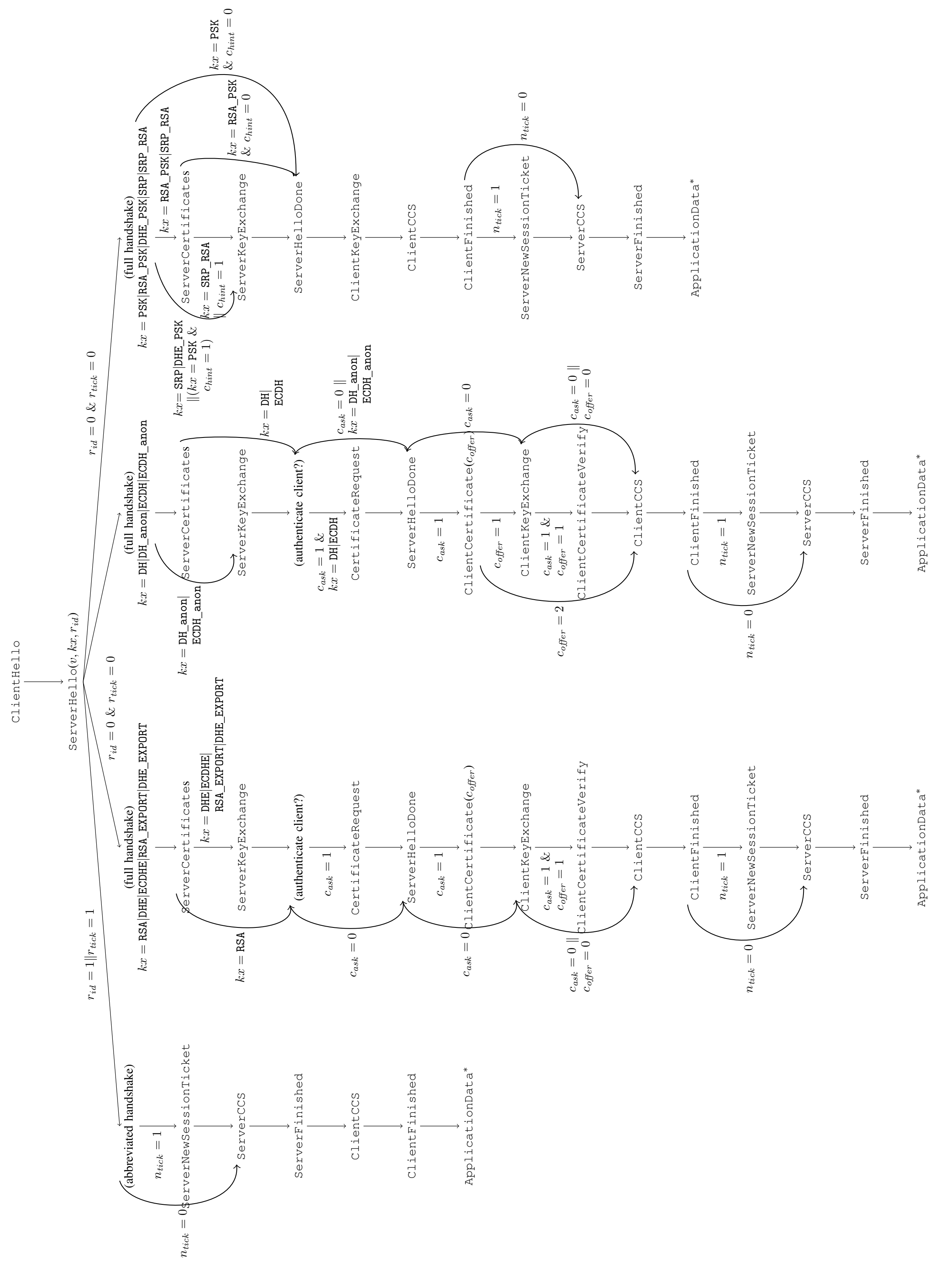
ClientHello
ServerHello(v, kx, r_{id})
$r_{id} = 1 \| r_{tick} = 1$
$r_{id} = 0$ & $r_{tick} = 0$
(abbreviated handshake)
(full handshake) $kx =$ RSA|DHE|ECDHE|RSA_EXPORT|DHE_EXPORT
(full handshake) $kx =$ DH|DH_anon|ECDH|ECDH_anon
(full handshake) $kx =$ PSK|RSA_PSK|DHE_PSK|SRP|SRP_RSA
$n_{tick} = 1$
$n_{tick} = 0$
ServerNewSessionTicket
ServerCertificates
ServerKeyExchange
(authenticate client?)
CertificateRequest
ServerHelloDone
ClientCertificate(c_{offer})
ClientKeyExchange
ClientCertificateVerify
ClientCCS
ClientFinished
ServerCCS
ServerFinished
ApplicationData*
$kx =$ RSA
$kx =$ DHE|ECDHE| RSA_EXPORT|DHE_EXPORT
$c_{ask} = 1$
$c_{ask} = 0$
$c_{ask} = 1$ & $c_{offer} = 1$
$c_{ask} = 0 \|$ $c_{offer} = 0$
$kx =$ DH_anon| ECDH_anon
$kx=$ SRP|DHE_PSK $\|(kx =$ PSK & $c_{hint} = 1)$
$kx =$ DH| ECDH
$c_{ask} = 1$ & $kx =$ DH|ECDH
$c_{ask} = 0 \|$ $kx =$ DH_anon| ECDH_anon
$c_{offer} = 1$
$c_{offer} = 2$
$kx =$ RSA_PSK|SRP_RSA
$kx =$ SRP_RSA $\| c_{hint} = 1$
$kx =$ RSA_PSK & $c_{hint} = 0$
$kx =$ PSK & $c_{hint} = 0$

Fig. 8.   All message sequences for enabled ciphers in OpenSSL

```
(* Accept a TCP connection from the victim client *)
let st,cfg = Connection.serverAcceptTcp(listening_address, port) in

let st,nsc,ch = ClientHello.receive(st) in

(* Sanity check: our preferred ciphersuite is there *)
if not (List.exists (fun cs → cs = TLS_RSA_WITH_AES_128_CBC_SHA) (ClientHello.getCiphersuites ch)) then
    failwith "No suitable ciphersuite given"
else

(* Force our preferred ciphersuite when sending the ServerHello *)
let sh = { Constants.defaultServerHello with ciphersuite = Some(TLS_DHE_RSA_WITH_AES_128_CBC_SHA)} in
let st,nsc,sh = ServerHello.send(st,ch,nsc,sh) in

(* Send the certificate of a server we want to impersonate *)
let st, nsc, cert = Certificate.send(st, Server, chain, nsc) in

(* Compute the verify_data with the correct log and an empty master secret *)
let log = ch.payload @| sh.payload @| cert.payload in
let verify_data = Secrets.makeVerifyData nsc.si [||] Server log in

(* Jump straight to the Finished message *)
let st,fin = Finished.send(st,verify_data) in

(* Read sensitive data from the client in the clear... *)
let st, data = AppData.receive(st) in

(* ... and let the client accept some data *)
let st = AppData.send(st,
  "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n
  Content-Length: 43\r\n\r\n
  You are vulnerable to the EarlyFinished attack!\r\n") in
Tcp.close st.ns;
```

Fig. 9. FLEXTLS server code implementing the Early Finished attack on Java clients.