Towards a Systematic and Automatic Use of State Machine Inference to Uncover Security Flaws and Fingerprint TLS Stacks

Aina Toky Rasoamanana, Olivier Levillain, and Hervé Debar

Télécom SudParis, Samovar, Institut Polytechnique de Paris

Abstract. TLS is a well-known and thoroughly studied security protocol. In this paper, we focus on a specific class of vulnerabilities affecting TLS implementations, state machine errors. These vulnerabilities are caused by differences in interpreting the standard and correspond to deviations from the specifications, e.g. accepting invalid messages, or accepting valid messages out of sequence. We develop a systematic methodology to infer the state machines of major TLS stacks from stimuli and observations, and to study their evolution across revisions. We use the $L^\star$ algorithm to compute state machines corresponding to different execution scenarios. We reproduce several known vulnerabilities (denial of service, authentication bypasses), and uncover new ones. We also show that state machine inference is efficient and practical for integration within a continuous integration pipeline, to help find new vulnerabilities or deviations introduced during development.
With our systematic black-box approach, we study over 400 different versions of server and client implementations in various scenarios (protocol version, options). Using the resulting state machines, we propose a robust algorithm to fingerprint TLS stacks. To the best of our knowledge, this is the first application of this approach on such a broad perimeter, in terms of number of TLS stacks, revisions, or execution scenarios studied.

1 Introduction

TLS is a fundamental block of Internet security. The most recent version of the standard is TLS 1.3 [23]. It fixes many vulnerabilities uncovered in the last decade. Automata implementation errors represent one category of these. The RFC does not specify a reference automaton. Hence, implementers need to derive their state machine from the protocol messages descriptions and sequences. The complexity of the task is such that errors are easy.

Such vulnerabilities can be triggered by an attacker sending messages in an inappropriate order (e.g. EarlyCCS [18]) or skipping messages (e.g. SkipVerify [4], which bypasses server authentication by skipping the corresponding messages). In more complex cases, interfering with the state machine enables new cryptographic attacks (e.g. FREAK [4], Factoring RSA Export Keys). All major TLS stacks have been vulnerable to at least one such flaw in the last decade[1].

[1] e.g.: CVE-2014-0224, CVE-2014-6321, CVE-2015-0204, CVE-2015-0205

Our work focuses on black-box testing of TLS implementations, to better understand how they react to messages that diverge from an ideal message sequence. We use an active learning algorithm, L*, initially described by Angluin [2], and later adapted to Mealy machines [25], to infer the actual state machine through interactions with implementations. We then compare these state machines with the expected behavior of an ideal TLS stack. Despite the absence of a formal specification of such an ideal stack, a simple approximation of said ideal stack is to use so-called happy paths, which correspond to the expected message sequences for successful connections. A fully compliant stack should only contain happy paths and error transitions, leading to the end of the connection. Every other transition is deemed suspicious. Our contributions are the following:

- We propose an improved methodology to systematically analyze TLS stacks, both client- and server-side, in an rigorous, automatic and efficient way.
- We propose optimizations exploiting the determinism hypothesis used in L*.
- By applying our methodology to different versions of popular open source projects, we confirm already known security vulnerabilities.
- We also discover new implementation errors, including security-relevant ones.
- Our methodology spots differences in the implementation of error conditions, supporting the concept of state-machine-based fingerprinting of TLS stacks.

Our tools have been published on `gitlab.com` in two separate repository: one for the inference tool,[2] and one for the test bed infrastructure[3].

2 TLS in a Nutshell

A typical TLS 1.3 connection is shown on the left side of Fig. 1: the client sends a `ClientHello` message to advertise the ciphersuites, i.e. a set of cryptographic algorithms, it supports and to propose a key share using one of the algorithms it supports. If the client and the server agree on capabilities, the server selects a suitable ciphersuite, and sends its own key share in a `ServerHello` message.

Once the client and server have agreed on algorithms and a common session key, messages are protected using authenticated encryption. The server carries on with several messages, including its certificate chain (`Certificate`) and a signature over the exchanged messages proving its identity (`CertificateVerify`). The `Finished` messages confirm keys in both directions. Then, session keys are updated, and application data can be exchanged.

Of course, this transcript only represents a simple, typical situation. It represents a *happy path*, which does not take into account session resumption or the so-called 0-RTT mode. It also ignores common error cases, such as the impossibility for the client and the server to agree on a common ciphersuite.

From this description, we represent the expected behavior of a TLS 1.3 client with the state machine on the right side of Fig. 1. The happy path, in green,

[2] `https://gitlab.com/gaspians/pylstar-tls`
[3] `https://gitlab.com/gaspians/tls-test-bed`

starts with the client outputting a `ClientHello`, and leads to the `Finished` messages and the exchange of Application data. Outside of this happy path, all other messages (denoted `*`) lead to the sink state with a fatal alert. This figure is identical to the state machine inferred using our methodology on OpenSSL 3.0.1.

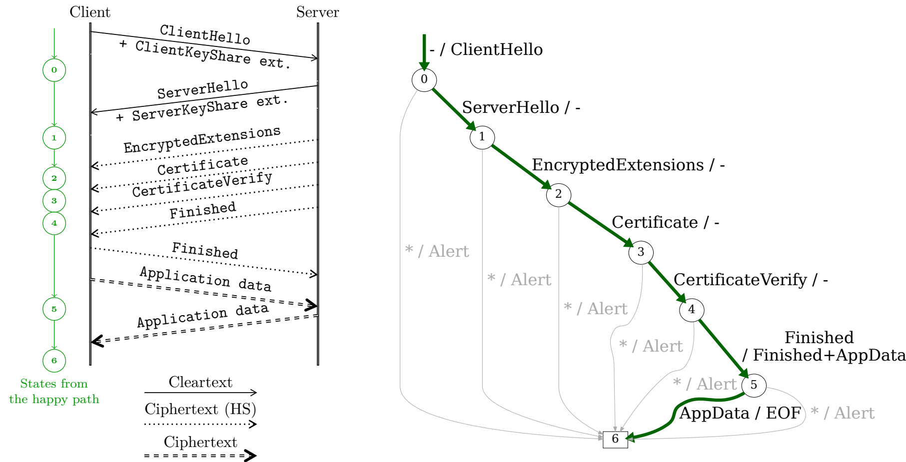


Fig. 1: A typical TLS 1.3 connection and the corresponding expected client state machine. On the right, transitions are labeled with the messages *sent to* / *received from* the client. The path in green is the expected flow described on the left, ending with a request (the `AppData` received from the client between states 4 and 5) and the answer (the `AppData` sent between states 5 and 6). A transition with `*` aggregates the behaviors for the remaining input messages.

3 Background on Model Learning

In 1987, Angluin proposed $L^\star$, an algorithm that infers a deterministic finite automaton using membership and equivalence queries [2]. This technique can be extended to extract the state machine of a protocol implementation using the Mealy machines representation, which can be seen as automata where transitions are labeled by both the messages sent and received, as shown on Fig. 1.

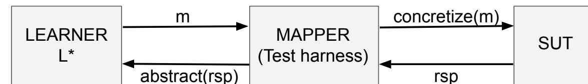


Fig. 2: Model learning setup.

$L^\star$ is an automated black-box technique driven by a LEARNER. Fig. 2 describes the experimental process. The analyzed implementation (a TLS client

or server) is the *System Under Test* (SUT). The LEARNER generates sequences of letters from a finite input vocabulary, where each letter represents an abstract protocol message (e.g. `ClientHello`) associated with its specific parameters. Interactions between the LEARNER and the SUT are mediated through a MAPPER, which transforms abstract letters into concrete protocol messages, and transforms the concrete answers back to abstract letters.

To infer the state machine, $L^\star$ populates an observation table using membership queries by collecting answers from the SUT to a series of message sequences. This step ends when the observation table is closed and consistent [2]. Then, it builds a hypothesis, i.e. a tentative state machine, and the LEARNER uses a so-called equivalence query to validate it. This query either confirms the hypothesis or exhibits a counter-example sequence where the hypothesis differs from the actual state machine. The counter-example is used to run the first step again to build a new hypothesis. This process is repeated until a hypothesis is validated.

Since the actual state machine is not known, equivalence queries do not really exist in practice, so we must approximate them. Several methods have been developed, such as W-method [9], Wp-method [14], Random Walk [20] and Distinguishing Bounds [21]. They use the same input vocabulary to create new message sequences that have not been used in the creation of the state machine. These sequences are executed both on the SUT using membership queries and on the hypothesis. In case the executions produce different results, the corresponding message sequence is a counter-example invalidating the hypothesis.

W-method or Wp-method have an exponential complexity in the size of the inferred automata, which was not reasonable in most cases[4]. We use the Random Walk to approximate equivalence queries, since it produced the best results, both in terms of performance and accuracy. We also cross-check our results using Distinguishing Bounds on the obtained unique state machines to benefit from its guarantees. Indeed, given a bound value B_{dist}, it guarantees that the obtained state machine will be accurate as soon as two states in the real state machine can be distinguished in at most B_{dist} steps.

4 Description and Implementation of our Platform

Fig. 3 illustrates how TLS implementations and our inference tool interact for a client inference. Server inference works in a similar way.

4.1 TLS Stacks

We create containers for more than 400 TLS stacks. Table 4 in App. A details the TLS stacks currently included in our platform.

For each stack, we reuse the tools or the example code available within the project to build and run a TLS client and/or a TLS server. Such pieces of code are representative of the way the libraries are used in practice.

[4] Some scenarios use many messages, which can produce state machines with many states. The maximum number of states in our experiments is 31.

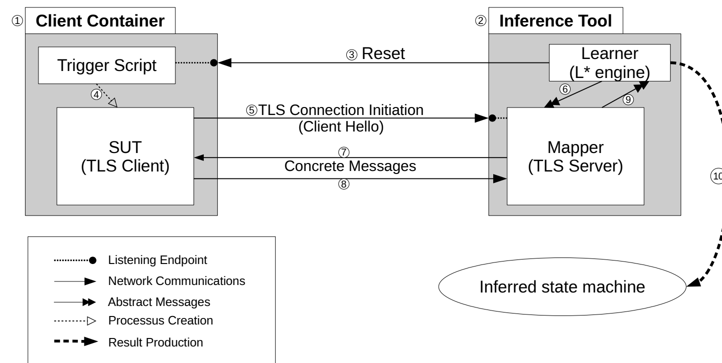


Fig. 3: TLS client inference cinematics. See App. A for a detailed explanation.

Each container is customized to select the protocol version and the ciphersuites, and to include the required cryptographic material (certificate, keys, trusted certification authority), which allows us to study different scenarios.

In addition to the "example" client, we create for each stack a second container, using `curl` dynamically linked with each stack. These curl-based images provide a unified interface across stacks, removing small differences in the example provided by the projects, e.g. missing certificate checks. All server-side examples include a functional and sufficiently customizable example for our needs.

4.2 Inference Tools

One major challenge with the L⋆ approach is that the Mapper used to concretize the abstract messages has to be flexible enough to send arbitrary messages at any state of execution of the protocol (even ones that would clearly be invalid). We thus need a modular and robust TLS stack to implement the Mapper. We use `scapy` [5], a Python-based network tool, to forge and decode packets. `scapy` allows us to easily build customized packets (e.g a `CertificateVerify` with a wrong signature).

To complete our setup, we choose `pylstar`, a Python-based implementation of L⋆, which allows for a straightforward connection with `scapy`. `pylstar` has previously been used to infer protocols used by malware with their Command & Control servers [7], as well as to study the behavior of HTTP/2 clients [8].

4.3 Assumptions

Deterministic SUTs The most important requirement for L⋆ is that the SUT behavior must be *deterministic*, relative to the selected input vocabulary. For a given stack and a given set of parameters, a given input abstract message sequence should always produce the exact same abstract output sequence.

TLS stacks behave deterministically, with a few exceptions. When a SUT takes too long to answer a stimulus, we can misinterpret its silence as the absence

of messages, whereas output messages were actually expected. This requires to get the timeout parameter right. In rare cases, an encrypted message can be misinterpreted as a cleartext message, which produces an unexpected response with a low probability. To avoid this, we always tag reception of encrypted packets that cannot be properly decrypted by `scapy` with a dedicated letter, `UnknownPacket`.

Similarly, OpenSSL 1.0.1d was a short-lived version with a known bug in the CBC encryption function. The defective function leads to the emission of malformed packets with a low probability, which could not be interpreted correctly deterministically. We chose to remove this particular image from our corpus.

Timeouts A common issue with L⋆ inference is the time required to produce a result. For a typical inference (a 10-state automaton, 15 input letters), we need to send thousands of sequences, with up to 10 letters. At each step, we must ensure we have received all the messages the SUT has sent. The usual solution is to wait for a long period of time before inferring "no response", which makes the inference slow. To improve the performance of our tools, we introduce heuristics to reduce the timeouts when possible.

5 Optimizations

Before discussing our optimizations in `pylstar`, we can already improve the performance by running in parallel several inferences. Indeed, since we use containers, running multiple instances of SUTs and inference tools is essentially free, so we can benefit from a multi-core architecture.

EOF is Final. When we receive a network error, which indicates that the SUT has shut down the communication channel, we can conclude that all subsequent messages will trigger the same signal (EOF), so it is not necessary to build and emit the corresponding messages.

In [24], de Ruiter and Poll actually proposed a similar improvement in the equivalence method implemented in `statelearner`, which resulted in measurable performance gains. By also applying the idea to the first phase of the algorithm (the membership queries), we further improve the performance.

Exploiting the Determinism. As discussed earlier, L⋆ relies on the fact that the SUT is deterministic. So we propose another optimization, which is a direct consequence of this assumption. During its execution, L⋆ often sends sequences that are extensions of already sent sequences. Let us assume that we have already observed that sending `A` to the SUT triggers two messages, `x` and `y`. When evaluating the input sequence `A B`, we can send `A`, read `x` and `y` *without waiting after the reception of* `y`, then send `B` and observe the answer using the timeout.

A restricted version of this optimization consists in skipping the timeout only when we know sending a message will not trigger any message back.

Evaluation. Table 1 describes the time required for a typical inference with different optimizations. We infer the TLS 1.2 server state machine for OpenSSL 1.1.1k (which contains 6 states) with 12 input messages and a 1-second timeout. The machine hosting the experiment is an 16-core AMD EPYC 7302P at 3GHz, with 128 GB of RAM and all the storage on SSDs.

	EOF optimization	
	Off	**On**
No anticipation	1,885 s (100 %)	1,598 s (85 %)
Skip timeouts on empty responses	1,081 s (57 %)	862 s (46 %)
Skip timeouts on all known responses	128 s (7 %)	77 s (4 %)

Table 1: Average time required to infer TLS 1.2 server state machine for OpenSSL 1.1.1k. Percentages are the fraction of the unoptimized time.

It appears both optimizations improve the overall performance, with a drastic improvement from the fully-fledged timeout anticipation. We ran similar experiments with `statelearner` (same timeout, same vocabulary), on the same hardware, and the time required to produce the (identical) state machine was 2,945 seconds.

Obviously, the time required for our inferences can vary, depending on the complexity of the SUT state machine (which can count as much as 30 states in some cases), the size of the input vocabulary (the scenario), and the speed of the SUT. The default timeout used is 1 second, but to get a stable inference, we must raise this value to 3 for several stacks.

For a 1400-experiment run (which took around 2 and a half hours overall, with 30 inferences in parallel), the average inference time was around 3 minutes, the median was 81 seconds, and the 10th and 90th percentiles were respectively 27 seconds and around 8 minutes.

6 Studied Scenarios and Vulnerabilities

A scenario is defined by the following information: (i) the role (client/server) and the configuration (protocol version, ciphersuites, etc.) of the SUT; (ii) the input vocabulary (the list of abstract messages) used during the inference; (iii) a set of expected path, which helps challenge the built state machine during the equivalence query phase; and (iv) a set of security properties to test on the resulting graph.

To identify bugs using learned model, we first identify RFCs violations and then we analyze whether these violations represent bugs with the following steps:

(i) color in green the happy paths representing the successful connections;
(ii) color in gray error transitions leading to sink state, which are expected;
(iii) color all remaining transitions in red since they are RFCs violations, and may correspond to vulnerabilities.

6.1 Client Scenarios

In these scenarios, the SUT is a client, running a given version of TLS. The client is configured with a trusted certification authority and is expected to check the certificate presented by the server. The inference tool acts as a server, with the following input vocabulary: `ServerHello`, `Certificate` messages (valid, empty, invalid — trusted but for the wrong domain —, and untrusted), other server-side Handshake messages, `ApplicationData` and `CloseNotify`.

In these scenarios, we ensure that the client only sends application data to a correctly authenticated server. We look for paths leading to `ApplicationData` messages and check for proper authentication.

Another area of interest is the presence of loops that could be used by an attacker to stall a client, enabling complex cryptographic attacks, such as the LogJam attack [1]. Since the goal of such attacks is to delay the completion of the TLS Handshake, we only focus on loops happening early in the connection.

6.2 Server Scenarios

In these scenarios, the SUT is a server, running a given version of TLS. The server can be configured to require mutual authentication (with regards to a given certification authority). The inference tool acts as a client, and uses the following vocabulary: different `ClientHello`s, various `Certificate` messages (empty, trusted, untrusted), other client-side Handshake messages, `ApplicationData` and `CloseNotify`. We also include alerts and unexpected messages such as server-side messages.

When client authentication is required, we want to ensure that the server properly authenticates the client. Only paths with a valid certificate and the corresponding signature should be accepted.

We are also interested in the presence of loops in server state machines, which could force the server to maintain an open connection indefinitely. For such denial of service attacks, we only focus on occurrences happening before encryption is activated; this way, the attacker only needs to spend very few resources to keep the channel open. Moreover, keeping the server in an early stage of the connection reduces the chances of something being logged. Note that these loops are different from the ones created through TCP segmentation or TLS `ClientHello` fragmentation, which would be limited by the length of the data to send.

6.3 Vulnerability Confirmation

These scenarios identify potential implementation issues, which need to be independently confirmed as security flaws. $L^\star$ is an algorithm that produces a state machine, which represents the behavior of the SUT. However, the produced state machine is only an approximation due to the (limited) set of abstract messages selected in the scenario and the equivalence query method used. We thus use

tools to independently check whether a potential security issue, uncovered by the inference, actually translates into a real security flaw.

For authentication bypass issues, we extract the potentially dangerous paths and replay them to the SUT, in a context where we do not have access to the authentication secret. If we can trigger the tested stack to emit Application Data, the flaw is confirmed.

For loops, we send precomputed packets to the SUT at a given pace (typically one message per minute), and for a given duration (e.g. several hours). If we can maintain the connection open, we have proof the loop can be weaponized.

7 Analysis of the Resulting State Machines

We analyze over 400 different versions of different TLS stacks using different client and server scenarios and get over 2,000 automata. App. B summarizes the vulnerabilities reproduced and discovered during our study.

7.1 Authentication Bypasses

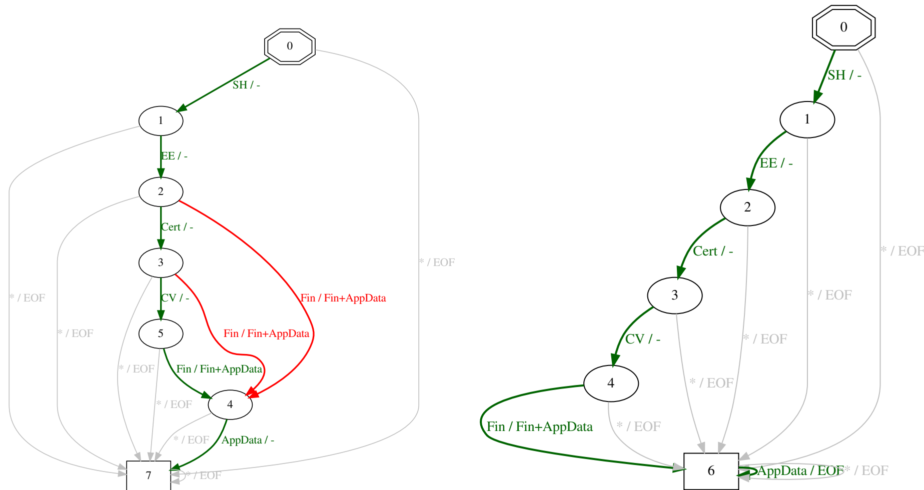
Server authentication bypasses in wolfSSL. Around 2015, authentication bypasses in state machines seemed to be pervasive in TLS stacks [4,24]. In 2020, CVE-2020-24613, an authentication bypass affecting wolfSSL TLS 1.3 client, caught our eye, and we decided to try and reproduce it using $L^\star$.

To this aim, we infer the state machines for wolfSSL TLS 1.3 clients, for different versions, with standard Handshake messages. Fig. 4a represents the state machine corresponding to wolfSSL 4.4, which is vulnerable to CVE-2020-24613. By skipping the `CertificateVerify` message, an attacker can bypass server authentication, and thus impersonate any server to a vulnerable client. The vulnerability was fixed in version 4.5, as can be seen on Fig. 4b, which corresponds to the inferred state machine for the patched version, using the same vocabulary.

However, in other scenarios, we also use a broader input vocabulary including an empty `Certificate` message, that should never be sent by the server. We could thus discover another vulnerability in wolfSSL, present in all versions at the time. As shown in Fig. 4c, instead of skipping the `CertificateVerify` message, the attacker can send an empty `Certificate` message, followed by a `CertificateVerify` message signed by an arbitrary RSA key[5]. This new bug was confirmed, reported as CVE-2021-3336, and fixed.

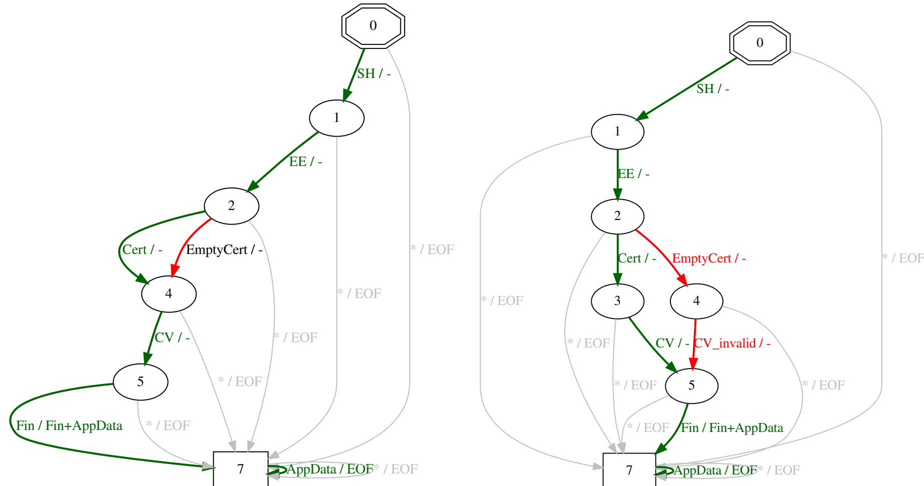
By adding new messages to the input vocabulary, we also discover another alternate path to reintroduce the initial bug. Fig. 4d shows that an attacker can send an empty `Certificate` message, followed by an invalid `CertificateVerify` message, containing an unknown signature algorithm and an arbitrary payload

[5] In our inference tool, sending a `Certificate` message selects the corresponding RSA key to be used in the subsequent `CertificateVerify`. For `EmptyCertificate`, the selected RSA key is a fresh key generated for the experiment.



(a) CVE-2020-24613, a server authentication bypass in wolfSSL TLS 1.3 clients, up to version 4.4. An attacker can impersonate any server to a vulnerable client by skipping the `CertificateVerify` message.

(b) CVE-2020-24613 fixed in version 4.5. With the same vocabulary used in Fig. 4a, the dangerous transitions have disappeared.



(c) CVE-2021-3336. Sending an empty `Certificate` message followed by an arbitrary `CertificateVerify`, allows server impersonation to a vulnerable client.

(d) CVE-2022-25638. Adding a completely invalid `CertificateVerify` message reintroduces a dangerous transition.

`SH`	:	`ServerHello`	`EE`	:	`Encrypted Extensions`
`Cert`	:	`Certificate`	`CV`	:	`CertificateVerify`
`Fin`	:	`Finished`	`AppData`	:	`ApplicationData`
`EOF`	:	End of the connection			

Fig. 4: Attacks against wolfSSL TLS 1.3 clients.

to bypass server authentication. This bug, identified as CVE-2022-25638, has been fixed in version 5.2.0.

All these attacks were reproduced by sending the identified transcript to the vulnerable SUTs. The program replaying the attack was not given access to the server private key, and we checked both wolfSSL and curl+wolfSSL stacks to make sure the authentication bypasses were real.

Other bypasses. In OpenSSL, different paths are incorrectly identified as invalid bypasses: the client seems to be accepting any certificate from the server. However, when we analyze a real TLS client using OpenSSL (the curl+OpenSSL stack), these dangerous paths disappear. Indeed, in our OpenSSL containers, TLS clients use the `s_client` application, which does not enforce any checks regarding the certificate[6].

We use the same approach to assess the quality of TLS servers authenticating clients. We get an issue in wolfSSL TLS 1.3 servers, as shown in Fig. 5, which is the transposition of CVE-2020-24613 to the server. By skipping the `CertificateVerify` message (and optionally the `Certificate` message), an attacker can bypass the authentication and impersonate any legitimate client. It is worth noting that the server correctly reject untrusted certificates and empty `Certificate` messages (since client authentication is required in this scenario). This bug, CVE-2022-25640, has been fixed in version 5.2.0.

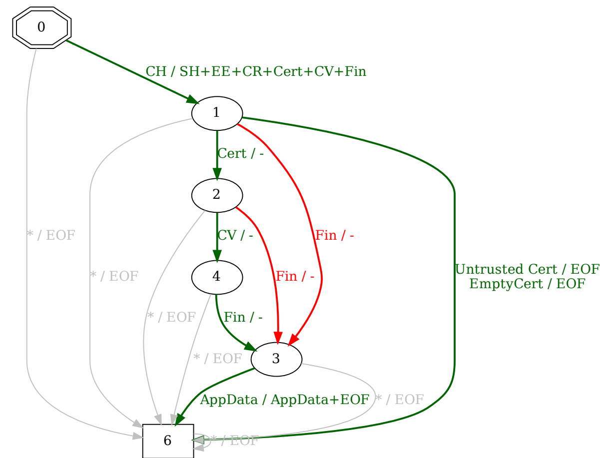


Fig. 5: CVE-2022-25640. In versions, up to 5.1.0, client authentication can be bypassed in wolfSSL TLS 1.3 servers, using the same idea as in CVE-2020-24613.

[6] It is possible to add options such as `-verifyCAfile` to the command line, but they do not end an unauthenticated handshake and merely produces a warning message

7.2 Loops in the Automata

As discussed in Sec. 6, exploiting loops in TLS state machines can be used to mount sophisticated cryptographic attacks [1]. Loops have also been considered as a potential vector for denial of service attacks (e.g. CVE-2020-12457). We thus identify such loops in our state machines, and to focus on those happening before messages are protected.

Analysis of a False Positive. The inferred state machine for wolfSSL TLS 1.2 server (all versions) seems to exhibit a loop on the initial state, tagged with the `NoRenegotiation` warning. However, when we repeatedly send such warnings to the SUT, the server actually closes the connection after 4 warnings. This situation exhibits the fact that $L^\star$ is only an approximation, which can not always capture behaviors happening very deep in the state machine. This justifies our approach, to always confirm potential vulnerabilities identified on the generated state machine.

Real bugs. After careful verification, we confirm several loops in different stacks, which are summarized in Table 2.

Stack	Scenario	Messages	Max. Time Between Msgs
erlang 24	1.0/1.2 Server	`NoRenegotiation` Alert or `ApplicationData`	> 1 hour*
fizz 22.01.24	1.3 Client	`ChangeCipherSpec`	> 1 hour
matrixssl 4.0 - 4.3	1.0/1.2 Server	`NoRenegotiation` Alert	≈ 40 seconds
NSS 3.15 - 3.78	1.0/1.2 Server	`NoRenegotiation` Alert	> 1 hour
OpenSSL $< 1.1.0$	1.0/1.2 Server	Empty `ApplicationData`	> 1 hour

* Erlang has a `Timeout` parameter that can thwart the attack. It was added to the official tutorial.

Table 2: Description of confirmed loops in TLS stacks.

For servers, loops can lead to Denial of Service attacks against TLS services, with very few resources. Indeed, the attacker can easily establish TCP connections and regularly send the right payload. Beyond the payload and the SUT identification (IP address and port), the attacker only needs to store, for each connection, the source port and the associated sequence numbers. With stacks keeping a connection alive for several minutes between packets (most probably because they do not enforce any kind of timeout), this represents a tiny amount of CPU, memory and network resources for the attacker. Moreover, distributing this attack is trivial. Finally, with vulnerable stacks, the attack can be run indefinitely and does not usually generate logs.

Beyond adding reasonable timeouts (both per-message and per-handshake) within the affected stacks, firewalls or other network devices should be used to detect and deter such extreme behavior. For the affected stacks, the issues have been reported and fixed when deemed relevant.

7.3 Unsolicited Client Authentication

TLS client authentication is an optional feature. The client can only present its certificate when the server sent a `CertificateRequest`. Servers may however be accommodating, and accept `Certificate` and `CertificateVerify` messages from the client, even when they were not solicited.

Such behavior may expose parts of the code that are not normally used. In 2014, a critical security flaw was found in Microsoft SChannel: a buffer overflow in the ECDSA signature check, triggered by client authentication, led to remote code execution. Accepting unsolicited client authentication messages made this obscure bug actually reachable in most deployments.

In our corpus, several versions of wolfSSL exhibit a similar behavior. Even if these paths do not necessarily lead to security issues, they should be removed, and considered bad practice, as they are a deviation from the specification.

8 TLS Stack Fingerprinting

We expect the state machines to be rather simple, as shown in Fig. 1, with less than 10 states, a restricted number of happy paths and the rest of the transitions consisting of fatal errors pointing towards the sink state. Yet, as surprising as it may seem, we observe that the produced state machines are actually richer, with up to 31 states, and that they are each specific to a given TLS stack[7].

The differences usually lie in variations among implementations about the error handling: different alert messages can be emitted. Several state machines sometimes accept unexpected messages and silently ignores them.

Using a method described by Shu and Lee [26], we can compute, for a given scenario, a set of input message sequences separating the different stacks we inferred. Then, we can compute the stack *fingerprints* as the answer on each stack to the distinguishing sequences.

Beyond revealing interesting differences in TLS stack internals, fingerprinting TLS stacks can help an attacker pinpoint, with a few message sequences, a given version (or a set of versions) of a TLS implementation to select an effective exploit against this particular target. This may also help identify the underlying TLS stack in network appliances.

Fingerprinting also allows to detect the presence of interception middleboxes that can be used for censorship. Indeed, such middleboxes may produce unique fingerprints, either at the message-level or at the state machine-level. It is also possible to look for discrepancies between the TLS stack and the application-layer stack to detect middleboxes, as described by Durumeric et al. [10].

8.1 Application to TLS 1.3 Servers

To illustrate our state-machine-based fingerprinting, Table 3 presents the classes we identify for the simple TLS 1.3 scenario with no client authentication.

[7] Of course, within a given project, successive versions may share the same automaton.

Stack	Versions	N	High-severity CVEs affecting the servers
erlang	24.0.3 - 24.2.1	9	*No high-severity CVE referenced*
GnuTLS	3.6.16 - 3.7.2	4	*2021-20231*, *2021-20232*
matrixssl	4.0.0 - 4.1.0	4	*2019-10914*, 2019-13470
	4.2.1 - 4.3.0	6	*No high-severity CVE referenced*
NSS	3.39 - 3.40	4	2019-17006, 2019-17007, 2020-12403, 2020-25648, 2021-43527
	3.41 - 3.78	4	*2019-17006*, *2019-17007*, *2020-12403*, *2020-25648*, *2021-43527*
OpenSSL	1.1.1a - 1.1.1n	4	*2020-1967*, *2020-1971*, *2021-3449*, *2021-3711*, *2022-0778*, 2022-1292
	3.0.0 - 3.0.2	4	*2022-0778*, *2022-1473*, 2022-1292
wolfSSL	3.15.5 - 4.0.0	7	2019-11873 and all the ones in the next row
	4.1.0 - 4.6.0	7	*2019-15651*, *2019-16748*, *2019-18840*, 2021-38597, 2022-25640
	4.7.0 - 4.8.1	7	*2021-38597*, 2022-25640
	5.0.0 - 5.1.1	7	2022-23408, 2022-25640
	5.2.0	6	*No high-severity CVE referenced*

Table 3: TLS 1.3 server stacks grouped by state machine. N is the number of states. CVEs in italic only affect part of the equivalence class.

Separating these 13 classes only requires sending 8 distinguishing sequences:

```
CloseNotify                    ClientHello, Certificate
ClientHello, Certificate       ClientHello, Finished, CloseNotify
ClientHello, ClientHello       ClientHello, EmptyCertificate, CertificateVerify
ClientHello, CloseNotify       ClientHello, EmptyCertificate, InvalidCertificateVerify
```

8.2 Advantages and Limitations of the Approach

We believe such fingerprints are rather robust, since they rely on the way TLS stacks handle messages at their core, and not on easily customizable parameters such as the list of supported ciphersuites.

However, there exists configuration parameters that can impact the structure of the state machine. We already handle several of them, such as server-requested client certificate authentication or TLS 1.3 middlebox compatibility (which consists in sending useless `ChangeCipherSpec` messages), but other features might affect the accuracy of our tool, such as the renegotiation mechanisms, which we leave to future work.

9 Related Work

State Machine Learning. Several methods have been used to analyze TLS implementations. In 2014, Kikuchi discovered the EarlyCCS vulnerability trying to prove state-machine-level properties using a proof assistant [18]. This approach does however not scale well, considering the huge work required to properly model the protocol.

Juraj Somorovsky presented TLS-Attacker [27], a framework for evaluating the security of TLS implementations. TLS-Attacker allows to forge customized

TLS message sequence. It was successfully used to uncover several vulnerabilities in TLS libraries such as OpenSSL, Botan and matrixssl.

On its own, TLS-Attacker does not do state machine learning. It was nevertheless used as the mapper by van Thoor et al. [28] with `statelearner` to infer TLS 1.3 state machines in 2018. With regards to our work, the study has several limitations: it only covers an internet draft of TLS 1.3, was only run on a few OpenSSL and wolfSSL servers, and included a less rich vocabulary.

Beurdouche et al. [4], developed a tool, FlexTLS, and proposed a method to test the behavior of mainstream TLS stacks against deviant traces consisting in removing or adding messages from valid traces. They uncovered many bugs in different TLS stacks, including the EarlyCCS vulnerability discussed above and the infamous FREAK attack (Factoring `RSA_EXPORT` Keys). Tarun et al. [29] also used FlexTLS on Microsft SChannel, and they found bugs and vulnerabilities, including loops as those described in Sec. 7.2. By comparison, our approach is more exhaustive, with regards to the used input vocabulary and under the assumption equivalence queries are properly approximate. Moreover, FlexTLS was not updated to be compatible with TLS 1.3.

De Ruiter and Poll used $L^\star$ in 2015 to infer TLS state machines for different TLS servers [24]. They discovered various anomalies and security issues. Our work builds on their results, since their study only covered server state machines and predates TLS 1.3.

Active learning methods have also been applied to other protocols and problems. In his thesis, Bossert developed `pylstar` and used it to reverse-engineer communication protocols between a malware and its server [7]. He also studied the behavior of HTTP/2 clients to allow for robust fingerprinting [8]. Fiterau-Brostean et al. applied model learning to SSH implementations [12] in 2017 and DTLS implementations [11] in 2020. In 2019, de Rasool et al. used `learnlib` (the library used by `statelearner`) to study Google's QUIC protocol [22].

Finally, Henrix et al. [15] explored parallelization and checkpointing to improve inference performance in `learnlib`. We did not investigate parallelism at the inference level since we could more easily parallelize our experiments with no complexity added. However, we believe checkpointing is promissing, and we plan to explore instrumented active learning in our future work, not only for performance improvements, but also to characterize more precisely the SUT's internals in dangerous states.

TLS Fingerprinting. To identify a TLS client, Husák et al. [16] used the list of ciphersuites proposed by the client to fingerprint TLS stacks. The idea has been generalized by Kotzias et al. and by Frolov and Wustrow [13, 19] to include other fields of the `ClientHello` to fingerprint the client. The method was applied successfully to detect malware, censorship circumvention tools and web browsers. Salesforce proposed two formats to capture the idea: JA3 for passive fingerprinting and JARM for active fingerprinting[8].

[8] `https://github.com/salesforce/ja3` and `https://github.com/salesforce/jarm`

Durumeric et al. [10] presented the impact of HTTPS interception on security. They identified the nature of the client by identifying a mismatch between the HTTPS User-Agent header and TLS client behavior (supported ciphersuites, declared extensions).

Janssen et al. [17] proposed an approach similar to ours to fingerprint TLS servers, with a tool called `tlsprint`,[9] based on state machines inferred with `statelearner`. However, the studied stacks are limited to `OpenSSL` and `mbedTLS` servers without TLS 1.3 support. Furthermore, we observed that `tlsprint` had a non-deterministic behavior against several OpensSSL stacks from our testbed.

We believe our work on state-machine fingerprinting can be more robust than ciphersuite-based fingerprinting, since the latter behavior can usually be configured, whereas the former is based on behaviors that are fundamentally representative of the studied stack.

10 Conclusion

Using our platform containing more than 400 stacks representing various versions of open source projects and our methodology, we could reproduce known bugs on TLS stacks, as well as uncover new implementation errors, including security vulnerabilities such as authentication bypasses or possible denial-of-service vectors. Moreover, since the state machine we infer are sufficiently precise to spot differences between implementation families, this supports the concept of state-machine-based fingerprinting, an alternative to the more classical approach based on ciphersuite-based fingerprinting, which offer a more robust characterization.

To the best of our knowledge, our work is the most extensive and systematic application of model learning to an important corpus of TLS implementations.

Overall, we believe that these deviations from the standard, even when they do not lead to exploitable security vulnerabilities, are detrimental to the overall quality of the implementation. They represent an unnecessary complexity that has been known to facilitate the introduction of security issues in the future when features are added. To reduce these deviations (and to limit fingerprinting opportunities), standards should produce more formal definitions of the expected state machines in future specifications.

Beyond TLS, other protocols could benefit from our methodology. In particular, lowering the time required to infer a state machine allow us to explore more complex protocols with a rich input vocabulary, such as the recently standardized QUIC protocol.

Since our tools have been published as open-source software, we hope our work can help build a common test-bed for the community where we can compare and improve different approaches and tools.

[9] `https://github.com/tlsprint/tlsprint`

References

1. Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J.A., Heninger, N., Springall, D., Thomé, E., Valenta, L., VanderSloot, B., Wustrow, E., Zanella-Béguelin, S., Zimmermann, P.: Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 5–17 (2015)
2. Angluin, D.: Learning Regular Sets from Queries and Counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
3. Aviram, N., Schinzel, S., Somorovsky, J., Heninger, N., Dankel, M., Steube, J., Valenta, L., Adrian, D., Halderman, J.A., Dukhovni, V., Käsper, E., Cohney, S., Engels, S., Paar, C., Shavitt, Y.: DROWN: Breaking TLS with SSLv2. In: 25th USENIX Security Symposium (2016)
4. Beurdouche, B., Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P., Zinzindohoue, J.K.: A Messy State of the Union: Taming the Composite State Machines of TLS. In: IEEE Symposium on Security and Privacy, SP. pp. 535–552 (2015)
5. Biondi, P.: Packet generation and network based attacks with Scapy. In: CanSecWest Applied Security Conference (2005)
6. Böck, H., Somorovsky, J., Young, C.: Return of bleichenbacher's oracle threat (ROBOT). In: 27th USENIX Security Symposium. pp. 817–849 (2018)
7. Bossert, G.: Exploiting Semantic for the Automatic Reverse Engineering of Communication Protocols. Ph.D. thesis, MATISSE (2014)
8. Bossert, G.: Comparison and attacks against HTTP2. In: Symposium sur la Sécurité des Technologies de l'Information et de la Communication (2016)
9. Chow, T.S.: Testing Software Design Modeled by Finite-State Machines. IEEE Trans. Software Eng. **4**(3), 178–187 (1978)
10. Durumeric, Z., Ma, Z., Springall, D., Barnes, R., Sullivan, N., Bursztein, E., Bailey, M., Halderman, J.A., Paxson, V.: The security impact of HTTPS interception. In: 24th Annual Network and Distributed System Security Symposium, NDSS (2017)
11. Fiterau-Brostean, P., Jonsson, B., Merget, R., de Ruiter, J., Sagonas, K., Somorovsky, J.: Analysis of DTLS Implementations Using Protocol State Fuzzing. In: 29th USENIX Security Symposium. pp. 2523–2540 (2020)
12. Fiterau-Brostean, P., Lenaerts, T., Poll, E., de Ruiter, J., Vaandrager, F.W., Verleg, P.: Model learning and model checking of SSH implementations. In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software. pp. 142–151 (2017)
13. Frolov, S., Wustrow, E.: The use of TLS in censorship circumvention. In: 26th Annual Network and Distributed System Security Symposium, NDSS (2019)
14. Fujiwara, S., von Bochmann, G., Khendek, F., Amalou, M., Ghedamsi, A.: Test Selection Based on Finite State Models. IEEE Trans. Software Eng. **17**(6), 591–603 (1991)
15. Henrix, M., Tretmans, J., Jansen, D., Vaandrager, F.: Performance improvement in automata learning. Master's thesis. Radboud University (2018)
16. Husák, M., Cermák, M., Jirsík, T., Celeda, P.: HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting. EURASIP J. Inf. Secur. **2016**, 6 (2016)
17. Janssen, E., Vaandrager, F., de Ruiter, J., Poll, E.: Fingerprinting TLS Implementations Using Model Learning. Master's thesis. Radboud University (2021)

18. Kikuchi, M.: How I discovered CCS Injection Vulnerability (CVE-2014-0224). `http://ccsinjection.lepidum.co.jp/blog/2014-06-05/CCS-Injection-en/index.html` (2014)
19. Kotzias, P., Razaghpanah, A., Amann, J., Paterson, K.G., Vallina-Rodriguez, N., Caballero, J.: Coming of age: A longitudinal study of TLS deployment. In: Proceedings of the Internet Measurement Conference, IMC. pp. 415–428 (2018)
20. László, L.: Random Walks on Graphs: A Survey, Combinatorics, Paul Erdos is Eighty. Bolyai Soc. Math. Stud. **2** (1993)
21. Radhakrishna, A., Lewchenko, N.V., Meier, S., Mover, S., Sripada, K.C., Zufferey, D., Chang, B.E., Cerný, P.: DroidStar: callback typestates for Android classes. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. pp. 1160–1170 (2018)
22. Rasool, A., Alpár, G., de Ruiter, J.: State machine inference of QUIC. CoRR **abs/1903.04384** (2019)
23. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard) (2018)
24. de Ruiter, J., Poll, E.: Protocol State Fuzzing of TLS Implementations. In: 24th USENIX Security Symposium. pp. 193–206 (2015)
25. Shahbaz, M., Groz, R.: Inferring Mealy Machines. In: FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5850, pp. 207–222 (2009)
26. Shu, G., Lee, D.: A formal methodology for network protocol fingerprinting. IEEE Trans. Parallel Distributed Syst. **22**(11), 1813–1825 (2011)
27. Somorovsky, J.: Systematic fuzzing and testing of TLS libraries. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 1492–1504 (2016)
28. van Thoor, J., de Ruiter, J., Poll, E.: Learning state machines of TLS 1.3 implementations. Bachelor thesis. Radboud University (2018)
29. Yadav, T., Sadhukhan, K.: Identification of Bugs and Vulnerabilities in TLS Implementation for Windows Operating System Using State Machine Learning. In: International Symposium on Security in Computing and Communication. pp. 348–362 (2018)

A Platform Architecture

In our platform, a TLS stack is defined as a container running at least one of the following scripts: `run_server`, which launches a TLS server, ready to be sollicited; `run_client`, which starts a so-called trigger server, a service listening to signals from the inference tool, so a TLS client can be spawned each time we want to test a message sequence. Table 4 lists the TLS stacks currently included.

Fig. 3 in Sec. 4 describes a typical run of our platform to infer a client state machine[10]. First, we start a client container running the trigger script (step 1). Then, we start our inference tool containing the $L^\star$ engine (the Learner) and the TLS Mapper (step 2).

Each time the algorithm needs to learn from the System Under Test (the TLS client) using a sequence of messages, it first resets the client (step 3), which

[10] Inferring a server works in a similar, but simpler, way. Indeed, we can simply start the server and have the inference tool open a connection for each sequence to test.

spawns a fresh TLS client in the client container (step 4). This client establishes a TCP connection to the TLS server within the harness (step 5) and sends its `ClientHello`. From now on, the L* engine drives the Mapper by transmitting abstract messages to send to the client (step 6). The harness concretizes those messages and sends them to the client (step 7). In return, the concrete answer from the client (step 8) are abstracted by the harness (step 9).

Steps 3 to 9 are repeated until the L* engine is able to produce a valid hypothesis regarding the client state machine, that is to generate an automaton accurately describing the client behavior (step 10).

Stack Name	Versions	Client	Server	Comments
OpenSSL	0.9.8m - 1.0.0t (41)	✓	✓	Only TLS 1.0
	1.0.1a - 1.1.0l (53)	✓	✓	Only TLS 1.0 and 1.2
	1.1.1a - 1.1.1n (14)	✓	✓	
	3.0.0 - 3.0.2 (3)	✓	✓	
curl+OpenSSL	1.0.0a - 1.0.0t (20)	✓		Only TLS 1.0
	1.0.1a - 1.1.0l (53)	✓		Only TLS 1.0 and 1.2
	1.1.1a - 1.1.1n (14)	✓		
	3.0.0 - 3.0.2 (3)	✓		
GnuTLS	3.6.16 - 3.7.2 (4)	✓	✓	
curl+GnuTLS	3.6.16 - 3.7.2 (4)	✓		
mbedtls	1.3.10 - 1.4 (17)	✓	✓	Only TLS 1.0
	2.0.0 - 3.0.0p1 (96)	✓	✓	Only TLS 1.0 and 1.2
wolfssl	3.12.0 - 3.14.4 (10)	✓	✓	Only TLS 1.0 and 1.2
	3.15.5 - 5.2.0 (20)	✓	✓	
curl+wolfssl	3.12.0 - 3.14.4 (10)	✓		Only TLS 1.0 and 1.2
	3.15.5 - 5.1.1 (20)	✓		
matrixssl	3.7.2 (1)		✓	Only TLS 1.0
	4.0.0 - 4.3.0 (7)		✓	
NSS	3.15 - 3.38	✓	✓	Only TLS 1.0 and 1.2
	3.39 - 3.78	✓	✓	
erlang	20.0 (1)		✓	Only TLS 1.0
	24.0.3 - 24.2.1 (2)		✓	
fizz	2021.02 - 2021.06	✓		Only TLS 1.3 Weekly snapshots

Table 4: List of TLS Stacks included in our Platform.

B List of the Studied Vulnerabilities

This appendix lists the vulnerabilities we studied during our work. New vulnerabilities uncovered during our study are tagged "New". Previously known vulnerabilities are tagged with one of the following status. "Not Reproduced" means we could not reproduce the issue, either because we did not include the

vulnerable stack or because of a limitation in our approach (e.g. the absence of a given abstract mesage); "Detected" means the infered state machine shows an unexpected transition related to the vulnerability; "Reproduced" means that the infered state machines provides evidence that the vulnerability is present and can be exploited, should the state machine be accurate.

Since we only focus on TLS 1.0 to 1.3 versions, we do not investigate several vulnerabilities such as DROWN [3], a cryptographic attack using flaws (including state machine bugs) in SSLv2 servers to recover TLS-encrypted plaintext.

B.1 Unexpected Loops

CVE #	Stack	Versions	Description	Status
2020-12457	wolfSSL	$\leq$ 4.4.0	Reproduced	TLS 1.2 server DoS
-	erlang	24.0	New	Default configuration allow for TLS server DoS
2022-25639	matrixSSL	4.0 - 4.3	New	TLS server DoS
-	fizz	2021 snapshots	New	Unexpected client loops
pending	NSS	3.15 - 3.78	New	TLS 1.0 to 1.2 server DoS

B.2 Authentication Bypasses

CVE #	Stack	Versions	Status	Comments
2014-0224	OpenSSL	$\leq$ 0.9.8za $\leq$ 1.0.0l $\leq$ 1.0.1h	Detected	EarlyCCS (unexpected CCS transitions)
2015-0204	OpenSSL	$\leq$ 0.9.8zc $\leq$ 1.0.0o $\leq$ 1.0.1j	Detected	FREAK (client- and server-side EXPORT RSA downgrade)
2015-0205	OpenSSL	$\leq$ 1.0.0p $\leq$ 1.0.1j	Not Reproduced	Client auth. bypass. Requires DH certificate support
2020-24613	wolfSSL	$\leq$ 4.4.0	Reproduced	TLS 1.3 server auth. bypass
2021-3336	wolfSSL	$\leq$ 4.6.0	New	TLS 1.3 server auth. bypass
2022-25638	wolfSSL	$\leq$ 5.1.0	New	TLS 1.3 server auth. bypass
2022-25640	wolfSSL	$\leq$ 5.1.0	New	TLS 1.3 client auth. bypass

B.3 Bleichenbacher Padding Oracles

The vulnerabilities described here affect TLS servers offering RSA key exchange (removed in TLS 1.3). At the state-machine level, a vulnerable stack exhibits a state where outgoing edges labeled with well-formed and wrongly-formed messages can be distinguished. Using a dedicated scenario including such malformed messages, we reproduced existing vulnerability, but we did not find any new bugs.

CVE #	Stack	Versions	Status	Comments
2016-0800	OpenSSL	$\leq$ 1.0.1t $\leq$ 1.0.2f	Not Reproduced	Requires SSLv2 messages
2016-6883	matrixSSL	$\leq$ 3.8.2	Reproduced	
2017-13099	wolfSSL	$\leq$ 3.12.2	Reproduced	ROBOT attack [6]
2017-1000385	Erlang	20.0	Reproduced	ROBOT attack [6]