USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

Towards Internet-Based State Learning of TLS State Machines

Marcel Maehren and Nurullah Erinola, *Ruhr University Bochum;* Robert Merget, *Technology Innovation Institute;* Jörg Schwenk, *Ruhr University Bochum;* Juraj Somorovsky, *Paderborn University*

https://www.usenix.org/conference/usenixsecurity25/presentation/maehren

This paper is included in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

Towards Internet-Based State Learning of TLS State Machines

Marcel Maehren[1], Nurullah Erinola[1], Robert Merget[2], Jörg Schwenk[1], and Juraj Somorovsky[3]

[1]Ruhr University Bochum
[2]Technology Innovation Institute
[3]Paderborn University

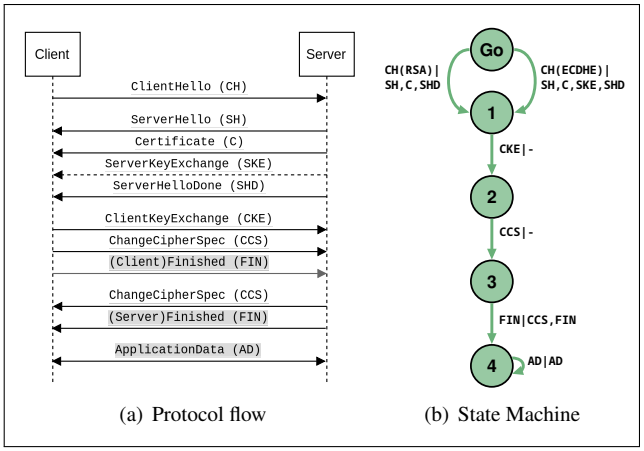
State machine learning extracts a Mealy state machine hypothesis from a given implementation. This approach was repeatedly used on open-source TLS implementations to find security vulnerabilities and bugs. Until now, TLS state learning has been conducted exclusively in controlled local environments, effectively avoiding various challenges, such as jitter, IDS interference, unknown network infrastructures (load balancers), timeouts, and most notably, *non-determinism* resulting from all these factors.

For the first time, we address these challenges by extending state learning beyond a controlled local environment and using it to learn TLS state machines over the Internet in a large-scale study. We improve the scope of state-of-the-art learning approaches by considering previously excluded features and directions, like ID-based session resumption, renegotiation, and CBC padding oracles. To enable a fully autonomous analysis of large numbers of servers, we develop novel techniques for dealing with large alphabets and automatically analyzing the retrieved Mealy automata.

We demonstrate the feasibility of our approach in a large-scale study across 7337 domains, successfully extracting 1304 state machine models. These models provide unique insights into the state machines deployed in the TLS ecosystem. Leveraging our automated analysis techniques, we uncovered a handshake transcript integrity vulnerability in Citrix NetScaler and the first CBC padding oracle vulnerabilities detected through state machine learning.

1 Introduction

Transport Layer Security (TLS) is one of the most prevalent protocols for securing communication over the Internet. It ensures authenticity, integrity, and confidentiality for application-layer protocols like HTTP, FTP, or IMAP. To establish a secure connection, a TLS *handshake* is executed. The TLS handshake follows a structured sequence of messages, starting with a *ClientHello* sent by the client and concluding with a *Finished* sent by the server (cf. Figure 1). A TLS state machine must carefully implement the TLS handshake flow and reject out-of-order messages. Incorrectly implementing the TLS state machine can lead to serious vulnerabilities, such as authentication bypasses [19] or cases where peers accept unencrypted application messages [12].



(a) Protocol flow (b) State Machine

Figure 1: Two happy flows in the (incomplete) Mealy state machine of a TLS 1.2 server. The difference in the happy flows results from the fact that for any `TLS-(EC)DHE` cipher suite the `ServerKeyExychange` message is mandatory, whereas in `TLS-RSA` and `TLS-(EC)DH` cipher suites it must be omitted.

A widely used technique to detect state machine vulnerabilities is *state machine learning (SML)* - also termed *state machine fuzzing*. SML is a technique where a machine called *learner* uses a special learning algorithm, such as L^* [3] or TTT [24], to establish hypotheses about the state machine of an implementation. The learning algorithm uses a small, fixed *alphabet* consisting of abstract *symbols* to formulate queries. A *mapper* must translate these symbols into concrete inputs that the *system under learning* (SUL) can respond to. The mapper likewise translates responses of the SUL into abstract symbols that allow the learner to identify *semantically* equal outputs. From the observed behavior of the SUL, the learning algorithm then forms a hypothesis of the state machine of the SUL. Each new hypothesis is then tested for

correctness and either rejected or kept. A simplified correct TLS 1.2 handshake (*happy flow*) as a state machine is shown in Figure 1.

State learning has already been applied to cryptographic protocols like TLS [5, 12, 19, 32, 42, 44], DTLS [19, 20], WPA [41, 42], SSH [20, 22], QUIC [17], TCP [17], or EDHOC [37]. In the above cases, the learning algorithm was carried out in a controlled environment, in a black- or grey-box scenario. In this scenario, the timeout and jitter are known and can be tuned, bugs can be analyzed via source code or binary review, the mapper can be adjusted during the research period to converge the model, and most importantly, the implementation acts deterministically or can be *changed* to do so. For example, in some approaches, the SUL is patched before learning to enforce determinism (see artifacts repository of [19, 20]). Such patches cannot be applied outside of a controlled environment, and problems around non-deterministic behavior must be solved differently.

While controlled environments offer advantages from a learner perspective, they are unable to capture the diversity of the real-world TLS ecosystem that consists of a vast range of implementations deployed in various configurations. For example, in the ROBOT attack [7], exploitable Bleichenbacher oracles were found in the wild, which did not exist in well-studied open-source libraries. A similar phenomenon was observed in CBC padding oracles [30] and elliptic curves [43]. These vulnerabilities could only be found by looking at deployed implementations in real environments. This leads us to the first research question.

> **RQ 1**: Is it possible to learn the state automaton of a TLS implementation deployed in an uncontrolled environment over the Internet at scale?

Challenges in Internet TLS State Learning When scanning servers outside our control, we do not know how they are configured and which features they support. Additionally, the TLS infrastructure on the Internet can be complex and, therefore, inherently non-deterministic. Non-determinism can arise in multiple ways. Examples include:

Non-hetero-gen load balancing: If servers in a load balancing setup are using different software (versions) or different configurations, from the outside, the behavior of the whole system appears non-deterministic.

System-load: Depending on the load of the SUL and the underlying network, the time it takes the server to receive, process, and respond can vary. In a complete black-box environment, the learner cannot differentiate between the SUL not wanting to respond (yet) or a delayed response, meaning that the learner has to employ a timeout when probing for a response.

Mapper Discrepancies: The mapping of symbols to protocol messages is not static. TLS requires dynamic computations as messages depend on random nonces and ephemeral keys. If there is a mismatch between the mapper and the SUL about the meaning of a given message, the mapper may translate messages non-deterministically.

Extending the Scope of State Machine Learning The vulnerabilities that can be found with state machine learning are highly dependent on the alphabet that is considered. The more symbols are in the alphabet, the higher the potential for vulnerabilities, as more potential states can be explored. At the same time, increasing the alphabet with more symbols directly increases learning time and the amount of queries required to create the model. If the state machine that can be explored with a given alphabet is too big, it is possible that learning does not terminate in a reasonable amount of time. In the context of learning in uncontrolled environments over the Internet, our learner should not disrupt the regular operation of the SUL and be mindful of the connections made. At the same time, we want to have a large alphabet to detect many vulnerabilities, resulting in a conflict. Even if large-scale state learning with an extensive input alphabet were successful, the challenge remains in analyzing the extracted state machines. Given the sheer size and complexity of the resulting models, manual analysis becomes infeasible, leading to the next research question.

> **RQ 2**: Is it possible to conduct state learning over the Internet using an extended input alphabet and automatically evaluate the resulting TLS state machines? Can this approach effectively detect unknown bugs by generating high-context results?

Challenges in Evaluating State Machines The main challenge in this context lies in creating a comprehensive test catalog tailored to the specific protocol. This requires a comprehensive understanding of the protocol and the identification of common implementation flaws and security vulnerabilities. A first step in this direction was made by Fiterău-Broştean et al. [20, 21]. They implemented a tool called SMBugFinder, which takes as input a state machine model of SUL and a catalog of *known* bug patterns for the protocol, specified as finite sub-automata. The tool found these bugs in the tested implementations by storing *known* bad patterns as deterministic finite automata (DFA). However, this approach may fail to detect new bug patterns. For unknown bugs, the DFA-based analysis for a single-state machine could take over 24 hours without guarantees of finding all violations, rendering such techniques infeasible for large-scale evaluations. We detail these shortcomings in Appendix B.

Methodology In this paper, we answer both research questions in the affirmative. To solve the issues encountered by *RQ 1*, we first do a TLS configuration scan with a regular TLS scanner before we perform the actual state learning. This allows us to configure our alphabet(s) only to contain symbols that fit the evaluated server implementation. We also

use the server's response times during the scan to calibrate our timeout. We then start the learning process. To deal with non-determinism, we employ a majority decision approach. If non-determinism occurs, we execute the inducing query (*word*) seven times and select the answer that was seen the most often as the intended answer.

RQ 2: To push the boundaries of state learning, we build our alphabet to test a diverse range of TLS features, including multiple different cipher suites, TLS 1.2 & TLS 1.3, ID-based session resumption, renegotiation, Bleichenbacher attacks, CBC padding oracle attacks, client authentication, alert messages, and heartbeat messages (see Table 8 of the appendix). To be able to apply our extensive alphabet and obtain meaningful results for a large number of targets, we apply different techniques. First, we propose a novel methodology to systematically partition the set of symbols into multiple alphabets and successively increasing in size towards our complete in-scope feature set (Section 5.2). Second, we propose a novel methodology for an automated analysis of obtained state machines (Section 5.6) to reveal non-compliant protocol flows and vulnerabilities. As the starting point of our analysis, we identify benign transitions in the state machine by running a predefined list of protocol-compliant message sequences – *happy flows* – for the different cipher suite classes. We identify deviations whenever a transition leaves the happy flow without terminating the session and whenever a transition leads back to the happy flow. In particular, transitions leading back to the happy flow – indicating the ability to successfully complete the handshake – may reveal potential new vulnerabilities, which we can analyze further. To ensure issues observed in the hypothesis exist in the actual state machine of the tested implementation, we propose a strategy for equivalence testing that incorporates happy flow paths, random queries, and queries derived from the automated analysis.

To avoid excessive queries, we consistently employ a *cache* and retain it across the different alphabets to answer previous queries from the learner immediately without performing any real connection. We adapt the proposal by de Ruiter and Poll [12] to leverage TCP-states and generalize it to increase the cache's efficiency through the concept of *sink states*. Specifically, we propose to also use the context of the current TLS session state to limit the learner from using certain symbols when they cannot reveal interesting information. We further incorporate timeout optimizations based on cached prefixes as proposed by Rasoamanana et al. [32]. We detail these optimizations in Section 5.4.

Findings To test our approach, we started our evaluation with a list of 7337 domains from open bug bounty programs[1] as these hosts are generally open to security research. From these hosts, we could successfully extract 1304 state machines. Many of these state machines deviated from the standards. These deviations are described in detail in Section 7.1, and include irregular behavior at the TCP layer (disregard of connection closure alerts, closing the TCP connection in the middle of a renegotiation), the mixture of application data and handshake messages, and the position of the CCS message. Among the most interesting findings, we detected two domains vulnerable to CBC padding oracle attacks and 11 domains that allow bypassing a central security guarantee of the TLS handshake, the *transcript integrity*. We further clustered obtained state machines based on their similarity (see Figure 5).

[1] https://github.com/arkadiyt/bounty-targets-data

Contributions:

- We performed the first TLS state machine learning (SML) scan in the wild.
- We used the largest SML alphabet on TLS so far (including ID-based session resumption, renegotiation, Bleichenbacher and padding oracle symbols).
- We devise an efficient algorithm for analyzing the resulting state automata, allowing us to detect deviations from the RFC-defined protocol flow.
- We found two padding oracle vulnerabilities and a transcript integrity vulnerability, which violates security assumptions about the TLS handshake.
- We provide unique insights into the TLS ecosystem.

2 Related Work

Analysis of TLS Implementations TLS implementations have already been evaluated in a controlled local environment utilizing SML [11, 12, 32, 42, 44]. All these works rely on manual analysis of the state machine model in order to find bugs. For DTLS, corresponding work was conducted by Fiterău-Broştean et al. [19, 20]. Other works focused on testing sequences designed to evaluate the state machine, but do not employ SML themselves [5, 40]. An overview and comparison to related work is presented in Table 1.

Active vs. Passive Learning Our scans are active and adaptive, which is the standard approach when the protocol specification is known. In contrast to this, Gascon et al. [23] and Comparetti et al. [10] use passive protocol traces to learn unknown protocol specifications.

Grey-Box Approaches Most papers on state machine learning use a relatively small, fixed alphabet. This naturally limits the coverage of the real state machine. Greybox approaches try to overcome this limitation. MACE [9] uses an interplay between the SUL and symbolic executions of the discovered state machine to extend the considered alphabet to explore more states automatically. AFLnet [31] uses binary instrumentation and reads internal states from the targeted binary to build a state-machine model for binary fuzzing. The created model is then used to steer the fuzzer towards interesting states to discover more code branches.

Paper	Protocol	Scope	Targets	SML	■/□	Q⚙	🔑	💓	🪪	PO	BB	↻	⟲	↓↑	#
[5]	TLS 1.0			✗	■	✗	✗	✗	✓	✗	✗	✗	✗	N.A.	N.A.
[12]	TLS ≤ 1.2			✓	■	✗	✗	✓	✓	✗	✗	✗	✗	12	28
[40]	TLS ≤ 1.2			✗	■	✗	✗	✓	✓	✓	✓	✗	✗	N.A.	N.A.
[11]	TLS ≤ 1.2			✓	■	✗	✗	✗	✓	✗	✗	✗	✗	7	29
[44]	TLS ≤ 1.2			✓	■	✗	✗	✗	✓	✗	✗	✗	✗	12	> 6
[19]	DTLS ≤ 1.2			✓	■🔧	✗	✓	✗	✓	✗	✗	✗	✗	17	13
[42]	TLS 1.2[a]			✓	□	✗	✗	✓	✓	✗	✗	✗	✗	15	8
[32]	TLS ≤ 1.3			✓	■	✗	✗	✗	✓	✗	✓	✗	✗	37	> 400
[20]	DTLS ≤ 1.2[b]			✓	■🔧	✓	✓	✗	✓	✗	✗	✗	✗	17	33
This Paper	TLS ≤ 1.3[c]			✓	■	✓	✗	✓	✓	✓	✓	✓	✓	60	1304

Lab Evaluation, Internet Evaluation, Server Implementations, Client Implementations, Black-Box, Gray-Box, Modified SUL, Automatic Analysis, Pre Shared Key, Heartbeat, Client Authentication, **PO** Padding Oracle Vectors, **BB** Bleichenbacher Vectors, Renegotiation Handshake, Session Resumption, Maximum Alphabet Size, **SML** State Machine Learning, # State Machines Learned [a] further evaluated WPA/2, [b] further evaluated SSH 2.0, [c] Where possible, we tested using TLS 1.3 and the next-highest TLS version. All hosts presented in our study (Section 7.1) supported at least TLS 1.2.

Table 1: Comparison to prior work on protocol state machine learning (SML) and fuzzing of (D)TLS implementations. The second block provides an overview of the scope and methodology while the third block focuses on the composition of the applied alphabet. New contributions in the context of applying SML to TLS have been highlighted in blue.

3 Background

TLS The TLS *handshake* is used to negotiate a set of cryptographic algorithms and to establish symmetric keys. These keys are then used by the TLS *record layer* to protect messages. In the handshake, the client and server exchange a fixed sequence of messages. This sequence depends on the TLS version, the cipher suite chosen by the server, and the optional selection of client authentication.

Figure 1 (a) shows an incomplete mealy machine with two possible sequences which cover a non-ephemeral (TLS-RSA) and an ephemeral (TLS-ECDHE) TLS 1.2 cipher suite. In the *ClientHello* and *ServerHello* messages, the client and server negotiate cryptographic parameters bundled in *cipher suites* and contained in extensions. Through the handshake messages *Certificate*, *ServerKeyExchange*, *ServerHelloDone* and *ClientKeyExchange*, the server authenticates itself, and both peers agree on secret keys for the record layer. A *ChangeCipherSpec* is a synchronization message that switches the record layer to encrypted mode. The two *Finished* messages contain a MAC of the transcript of exchanged messages and protect against man-in-the-middle attacks by guaranteeing *transcript integrity*. Encrypted application data can be sent in both directions. The *ServerKeyExchange* message is only sent if a TLS-DHE or TLS-ECDHE cipher suite has been negotiated.

In TLS 1.3, the server continues its first message flight after the *ServerHello* with an *EncryptedExtensions* message. As the name implies, the TLS record layer will already encrypt this message. To achieve this, both peers already include Diffie-Hellman key shares in their respective hello messages. After sending the *EncryptedExtensions* message, the server issues a *Certificate*, *CertificateVerify*, and *Finished* message. In TLS 1.3, the client concludes the handshake by sending its own *Finished* message. If no key share is provided by the client or an unsupported group was guessed, the server sends a *HelloRetryRequest* message. Once the client sends a new *ClientHello*, the handshake continues.

Both TLS versions offer a *session resumption* mechanism that utilizes previously negotiated keys to bootstrap a new handshake. TLS 1.2 allows a client to reference a previous session through a *session ID* field in the *ClientHello* or to use *session tickets* [38]. TLS 1.3 only supports the session ticket mechanism.

TLS 1.2 libraries can perform a *renegotiation*: Through a new handshake performed within the established secure channel, all symmetric keys get replaced. To obtain fresh keys in TLS 1.3, peers can send a *KeyUpdate* message that triggers a new key derivation based on the already negotiated shared secrets.

To indicate errors towards a peer, the TLS protocol defines *Alert* messages.

State Machine Learning SML allows to obtain finite-state models approximating the logical behavior of systems [24]. Ruiter and Poll [12] were the first to use it to find security vulnerabilities in TLS implementations. SML consists of four elements (Figure 2): a *learner*, a *mapper*, a *fuzzer* (for equivalence tests) and a *System Under Learning (SUL)*. The Learner is given an abstract *alphabet*, where each symbol represents some real-world input to the SUL. Sequences of symbols form *words*, which represent a (one-sided) message flow to be sent to the SUL. The mapper receives the individual symbols and translates them into meaningful input messages. In TLS, the mapper must keep track of the state of the connection to dynamically compute appropriate messages/inputs. The mapper likewise distinguishes the responses of the SUL for the learner.

Observing the responses to various words, the learner derives a *hypothesis* for the state machine. To test if this hypothesis is accurate, *equivalence tests* are performed. To do so, the fuzzer generates (random) words and compares the responses predicted by the hypothesis with the responses from the SUL (received through the mapper). If a counterexample is found, the counterexample is passed to the learner and the learner continues to refine the hypothesis through additional words. The learner terminates once a hypothesis is issued for which the equivalence tests could not find a counterexample. The result is a finite state machine, more precisely, a *Mealy machine*.

The TLS specification does not include a formal Mealy machine; RFC 8446 [34] only defines an informal one for TLS 1.3. The 'correct' behavior of a TLS implementation is, therefore, only implicitly defined through the text within the RFC. At the same time, as remarked by [28], the TLS specification contains some ambivalences, especially regarding the handling of alert messages. Figure 1 (b) shows an incomplete small Mealy machine of a TLS 1.2 server that only contains the sequence of states for a successful establishment of a TLS connection ("happy flow") and omits all error messages. The first part of the label is the input of the directed edge. The second part of the label is the output of the SUL, the sequence of messages sent by the TLS server. The response to an initial *ClientHello* depends on the cipher suite. Therefore, two different edges lead to the same state, as the SUL responds identically to succeeding inputs.

4 Challenges in Learning State Machines Over the Internet

Large-scale scans have proven very successful in uncovering impactful vulnerabilities in TLS implementations [2, 4, 6, 7, 8, 29, 30, 33, 36, 43]. With Internet-wide scans, it is possible to scan proprietary implementations and implementations in non-default but relevant configurations without buying software or hardware or manually setting up open-source software in different environments and configurations. At the same time, SML has proven to be a useful technique for finding high-impact vulnerabilities within (D)TLS implementations [11, 12, 19, 20, 32, 42, 44]. However, combining both approaches comes with its own set of challenges that need to be addressed.

Fully Blackbox When analyzing unknown implementations over the Internet, the target is always a black-box. This means that the configuration of the SUL is unknown and learning has to succeed without manual implementation adjustments and work very reliably. Once the evaluation starts, the learner and mapper cannot be adjusted anymore. In controlled environments, if learning does not converge in a reasonable/finite time, it is possible to analyze the implementation through code-review or log analysis to come up with a learning setup for which learning does converge. When scanning over the Internet, these adjustments are not possible and a *truly* black-box approach is required.

Non-Determinism Previous approaches applying SML to (D)TLS struggled with non-deterministic behavior and had to work around it with techniques like majority votes and/or by changing the implementation to make it more deterministic by changing internals like timeouts or the DTLS retransmission mechanism [19, 20]. Non-determinism issues are amplified when scanning over the Internet, as the scanner has no knowledge about the internals of the SUL. The SUL might use load-balancing techniques, which, combined with non-heterogenous deployments, results in non-deterministic behavior from the learner's perspective. Additionally, the mapper and implementation may conflict in the interpretation of a message sent out of the expected order, which can also result in non-determinism. When scanning over the Internet, network jitter is much higher than in a local environment, which makes calibrating to a proper timeout value that much more important.

Automatic Analysis When evaluating implementations in a controlled environment, only a limited number of implementations are usually scanned. This allows for manual debugging and manual analysis. When performing large-scale scans, the manual analysis does not scale, and a fully automatic approach is required. Additionally, since vulnerabilities cannot be investigated through source-code analysis, their presence has to be confirmed automatically, as state learning always only creates a hypothesis of the state machine and can never *guarantee* that the state machine accurately represents the implementation.

Limited Scope State machine vulnerabilities are limited to vulnerabilities that actually leave evidence in the extracted state machine. While increasing the input alphabet with more features seems like an easy adjustment to the limited scope of state machine learning, it increases the amount of queries required to learn a state machine. An increased alphabet comes with a polynomial increase in the number of queries. For example, for the TTT algorithm, the number of queries is $O(kn^2 + n \log m)$, where k is the alphabet size, n is the number of states of the SUL, and m is the maximum length of a considered counterexample.

Service Interruption Potential Performing many queries comes at the risk of performing an unintentional Denial of Service (DoS) attack on the tested server. SML can be quite heavy on the required number of connections and can far exceed the number of connections done in a typical TLS scan, which can interfere with the server's normal operation. It is, therefore, crucial that the number of connections made by the learner is kept at a minimum.

5 Methodology and Implementation

To tackle these challenges, we propose a multi-step approach, pictured in Figure 2.

Scope For our study, we consider TLS 1.2 and TLS 1.3. Additionally, we also consider different cipher suites. Instead of learning these features separately, we model them in one state machine to also potentially capture bugs manifesting from the interplay. To support renegotiation, we implement our mapper to reset the session transcript upon issuing a new *ClientHello* message while retaining session information required for secure renegotiation [35]. To add resumption to the scope, we added a distinct *ClientHello* message that attempts to resume a previous session by reflecting an old session ID. We further add a symbol to reset the TCP connection. This allows the learner to explore if partially established sessions can be resumed in a new connection. The introduction of session resumption to the state learner leads to a significant performance penalty, as most benign states get duplicated due to differences in when session resumption can be executed. This is best explained by examining the initial state: when starting a new message sequence, the learner has yet to establish a state that would allow for session resumption. This can be done by completing a full TLS handshake. Once the handshake has been completed, a connection reset will not lead back to the initial state but to a new initial state. When performing a new full handshake, the learner will transition through additional new states, which oftentimes behave exactly as the states traversed during the first handshake, except for the fact that the learner can now always reset and perform a session resumption.

To find CBC padding oracle vulnerabilities through state machine analysis, we extend the approach for Bleichenbacher vulnerabilities of Rasoamanana et al. [32] with attack vectors that manipulate the padding of an encrypted record. We selected our test vectors based on the findings of Böck et al. [7] and Merget et al. [30]. To explore the effect of various record content types, we sent these records as application data, handshake, change cipher spec, and heartbeat messages resulting in 16 padding oracle inputs for the alphabet.

Since prior TLS studies [7, 30] showed that some Bleichenbacher and padding oracle vulnerabilities can not be detected based on the TLS layer alone, but by looking at the TCP headers that close the connection, we consider the TLS messages received, the number of records, and the socket state of the underlying TCP connection in the output alphabet.

5.1 Feature Extraction ①

To perform state learning of a server over the Internet, we first use a classic TLS scanner to enumerate which features the TLS server supports. We analyze supported TLS versions, cipher suites, extensions, and other server features related to state machines. For every server, we attempt to learn a host-specific timeout. For our analysis, we first measure the latency of a host during the handshake for different considered cipher suites. We then use the worst latency observed as a baseline and apply a tolerance factor of 1.5. For our study, we perform the feature extraction based on the scan report generated by TLS-Scanner [1].

5.2 Alphabet Generation ②

Alphabet Selection Criteria We design our alphabets to include all protocol message types defined in the core specifications of TLS 1.2 and 1.3 [13, 34], as these are expected to be supported by all implementations and play a key role in progressing the session state. To examine side effects arising from shared code between TLS 1.2 and DTLS 1.2, we also include the *HelloVerifyRequest* message, which is the only protocol message unique to DTLS 1.2. Since the protocol flow varies slightly across different cipher suites and prior research has shown that certain vulnerabilities only emerge with specific combinations of key exchange and symmetric ciphers [7, 28, 30], we diversified our set of *ClientHello* messages to cover a range of key exchange algorithms and cipher types. We further incorporated protocol messages targeting edge cases, such as a *ClientHello* crafted to trigger a *HelloRetryRequest* response from the server, or a *ChangeCipherSpec* message that retains the old key set instead of updating it. To probe for known cryptographic attacks, we added test vectors for Bleichenbacher and padding oracle attacks. We chose these vulnerabilities in particular as previous studies found them to be widespread in TLS and detectable based on message response patterns [7, 30]. Finally, we included the *Heartbeat* message [39], which uses a distinct record content type and has been shown by de Ruiter and Poll [12] to trigger invalid state transitions in some implementations.

Partitioned Alphabets The number of queries needed to infer the state machine of an implementation is heavily influenced by the selected alphabet. Therefore, we leverage information about the supported TLS features to construct an optimized alphabet tailored to the SUL. Since we aim to evaluate remote hosts for which learning could fail at any point, we do not start with the full alphabet at once but increase the detail of the considered alphabet in multiple steps. We have four groups of alphabets (cf. Table 8 of the appendix):

1. **Happy Flow** This alphabet contains all symbols that are needed to complete a TLS handshake and to be able to send encrypted data on the TLS record layer. The goal of this alphabet is to quickly derive a fairly detailed Mealy machine hypothesis. It contains up to 5 different *ClientHello* symbols for TLS 1.2, which will be mapped to *ClientHello* messages containing a different, single cipher suite.

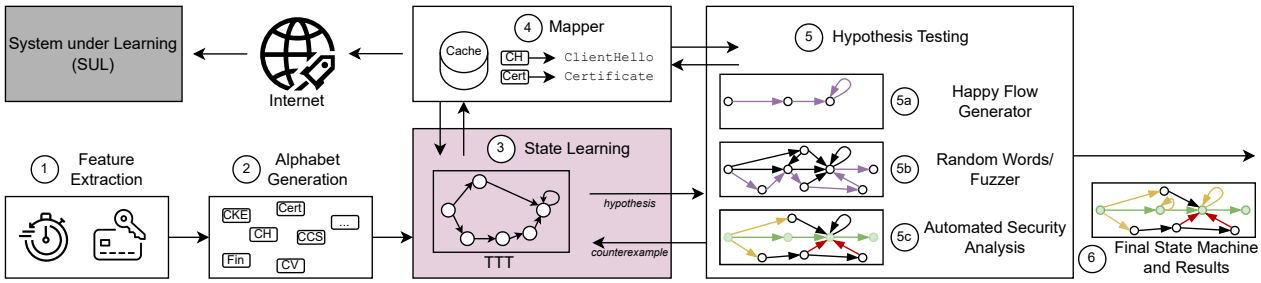


Figure 2: Our analysis framework. The different components are described in Section 5. Tools ① and ④ were adapted for this paper. Tools ② and ⑤ are novel.

2. **Extended** This alphabet contains *Happy Flow*, and adds symbols that explore *optional* paths within the TLS state machine (e.g., Certificate Verify). Additionally, it contains *reflection* symbols which map to messages that should never be sent to the server, but only to the client. This was motivated by CVE-2018-10933, where a message only intended to be sent to an SSH client caused an authentication bypass when sent to the server. We also add a symbol to reset the TCP connection.
3. **Vulnerabilities** This alphabet contains *Extended*, and adds 28 symbols that are translated in the mapper to 12 different Bleichenbacher and 16 different Padding Oracle attack vectors.
4. **All** This alphabet contains *Vulnerabilities*, and explores more unexpected inputs, such as a DTLS *HelloVerifyRequest* and an *EndOfEarlyData* message.

Before we start learning on a specific SUL, we remove all symbols that are related to unsupported features by the SUL, based on the output of feature extraction ①. E.g., if the SUL does not support any RSA-based cipher suites, we remove all Bleichenbacher symbols from the *Vulnerabilities* alphabet. For the initial learning, we use a single TLS 1.3 *ClientHello* message, as well as a *ClientHello* that (preferably) uses an RSA key exchange and CBC mode encryption. At the start of the extraction, we valued the insights gained by entirely different symbols more than the information gained by considering different cipher suites, but still wanted to explore the impact of the selected cipher suite on the state machine. For this purpose, once we finish scanning with the alphabets listed above, we restart the learning process with additional *ClientHello* symbols while keeping the cache (see Section 5.4). First, we include new *ClientHello* symbols that negotiate different key exchange algorithms (based on availability). Subsequently, we add *ClientHello* symbols corresponding to different symmetric cipher types.[2] Both times, when we increased the number of *ClientHello* symbols, we proceeded through all alphabets again.

[2]Note that we are using the RFC 5246 definition of `CipherType` for this, where AES+CBC belongs to type `BLOCK` while AES+GCM belongs to type `AEAD`.

Finally, we add an additional *heartbeat* symbol to the last and biggest alphabet, resulting in a maximum of 13 state machines returned by the TTT algorithm for each SUL. We treat the heartbeat symbols separately since some OpenSSL versions create a lot of obscure states when learning with heartbeat [12].

We provide a full list of the symbols used for each input alphabet along with a brief explanation in Appendix C.

5.3 State Learning ③

In this phase, we use the generated alphabets to start extracting a state machine model. This is done using the TTT algorithm [24]. The TTT algorithm maintains a compact representation of the machine model hypothesis in the form of a *discrimination tree*, which allows to efficiently test if a given *word* (= a sequence of symbols from the alphabet) fits within the hypothesis or not. To expand the machine model stepwise from the initial state to a realistic approximation of the real state machine, the TTT algorithm needs *counterexamples* (in the form of new words) from an external source (Figure 2, feedback loop from ⑤ to ③). For each counterexample that does not fit within the current state machine hypothesis, the algorithm has to refine its hypothesis. We describe our strategies to generate counterexamples in Section 5.5.

For our study, we use the TTT algorithm as implemented in the well-known LearnLib[3] library.

5.4 Mapper ④

The TTT algorithm ③, the equivalence test generator ⑤ and the automated security analysis ⑤c only operate on abstract input symbols. Translating these symbols into meaningful protocol messages for the *System under Learning* (SUL) is a critical aspect of SML performed by the mapper. In the case of TLS, this translation is dynamic: I.e., for each new query to the SUL, the messages must be recomputed to account for the state of the connection (nonces, hash values, signatures, MACs, ciphertexts).

[3]https://github.com/LearnLib/learnlib

5.4.1 Implementation

Our mapper implementation is based on TLS-Attacker [40], a TLS library that allows for arbitrary creation and manipulation of messages. At the same time, TLS-Attacker is able to dynamically create TLS messages which are both valid cryptographically and conform to the TLS specifications. This allows us to implement alphabet inputs that test specific parts of the error handling of an implementation (e.g, when generating records with malformed padding) or to test corner cases of the protocol. For example, we craft a TLS 1.3 *ClientHello* message that deliberately leaves out key exchange parameters to prompt the server for a *HelloRetryRequest* message. We further use the access to session parameters provided by TLS-Attacker to implement the transcript management required to support renegotiation.

5.4.2 Optimizations

Learning over the Internet is *slow*: There is latency, and we must be careful not to disrupt normal operations with a high query rate. Below, we discuss the strategies we employ to reduce the number of actual queries to the SUL. In Section 6, we present an evaluation on the impact of these optimizations.

Sink States We introduce artificial sink states to provide the learner with context about TCP and TLS. (1) TCP: Whenever the SUL closes the TCP connection, new queries asked by the learner can no longer change the state of the SUL since they do not reach it. De Ruiter and Poll [12] hence proposed to answer any symbols after a closed TCP connection with artificial TCP closed responses. In our case, closed TCP connections are not final, as studying session resumption requires us to be able to initiate new connections. Hence, we adapt their approach by introducing a sink state that is entered when the learner requests a symbol other than a connection reset after the TCP connection was closed. In this case, our mapper returns an *illegal learner transition* (ϕ) response without executing the query. From here on, all subsequent input symbols will also be replied with ϕ, effectively trapping the learner in its current state. (2) TLS: Besides closed sessions, we also propose to trap the learner when it attempts to send a session resumption *ClientHello* before a session cache has been established, when the server rejects the resumption and falls back to a full handshake, when resetting multiple times in a row, when sending padding oracle attack inputs before encryption is active, and when sending Bleichenbacher test vectors before RSA key exchange has been negotiated. Extending the concept of sink states in this manner enables us to guide the learner toward relevant queries while reducing execution time and minimizing unnecessary connections.

Cache Optimization Since the SUL is deterministic, a cache can be used to respond to queries that have been requested before [11]. We additionally use the cache for all cases where ϕ was returned before, as we know the learner can not leave the sink state anymore. Consequently, we only need to query the SUL when the sequence requested by the learner has not been cached entirely yet *and* is not known to lead to the sink state. Due to the strict definitions of valid message flows in the TLS specification, most sequences are expected to transition into the sink state quickly. Hence, this optimized cache results in a high speedup of the learning ③ and counterexample ⑤ validation since nearly all requests from these two components can be answered by the cache. Among the results of our study (see Section 7), the cache was able to respond to between 97.58% and 99.99% of all words requested during the alphabet learning process, with an average response rate of 99.79%.

Timeout Shortcuts We use the timeout optimization suggested by Rasoamanana et al. [32] that leverages the determinism of the SUL. Utilizing known responses to initial symbols, we can stop receiving as soon as expected messages have been received when we are forced to re-execute cached prefixes to avoid waiting the full timeout for a given flight.

5.4.3 Handling Non-Determinism

In a controlled local environment, timeouts in the SUL are the main source of non-determinism. When learning over the Internet, non-determinism is amplified, as Jitter and non-heterogeneous load-balancing can interfere with the learning process. Additionally, mapper inconsistencies can result in misunderstandings between the SUL and the mapper, which can result in non-deterministic answers from the SUL. As the TTT algorithm requires determinism, we employ the following strategies to combat the impact of these factors.

Majority Votes During learning, we detect non-determinism when a previously executed sequence of symbols yields a different response than before, i.e, it contradicts the information in our cache (*cache conflict*). For each cache conflict, we perform a *majority vote* that repeats the query in question seven times. We then overwrite the cache with the most common behavior of the peer thus invalidating the possibly incorrect cache entry from before. Subsequently, we restart TTT with the updated cache as the algorithm itself cannot handle conflicting responses. By capturing the most common response, we can mitigate the impact of infrequently occurring non-determinism.

Dynamic Timeouts During the state machine inference, we track the ratio of cache conflicts and confirmed cache entries. If multiple cache conflicts occur within recent queries, we increase the timeout. Note that an insufficient timeout may have caused multiple invalid cached entries. Once we increase the timeout, it is expected that other cache conflicts will occur solely because we now identify previously cached entries as incorrect. Hence, we ignore the first 100 send-receive cycles

for further timeout calibrations after a timeout adjustment to give the cache time to heal from the insufficient timeout before we start monitoring the timeout again.

Cache Fallbacks Inaccurate timeouts can cause the mapper to stop receiving when the peer is not done sending its flight of messages yet. Putting such incomplete responses into the cache poses a problem as subsequent queries may be served directly by the cache, causing the incomplete response to propagate further into the learner's state machine inference. This effect is somewhat mitigated by subsequent queries that repeat a cached prefix, as these prefixes must be executed again and can be used to validate previously established cache entries. This is not the case if we observed an empty response before and thus set an empty list of expected messages, as the receiving phase would essentially be skipped. As a trade-off between performance and error correction, we thus re-execute 25% of empty receives again with a full timeout to be able to identify inaccurate entries. Whenever we detect an inconsistent cache response. We fix the cache and then reset and restart the learning algorithm. Since the cache is already filled, the learner can quickly progress and, for the most part, only costs local computation time.

Omission from the data set As described before, we expect most non-determinism to arise from the network rather than from the analyzed implementation itself. If non-determinism is indeed frequently introduced by the implementation itself, the learner ultimately will not converge as each newly observed non-determinism requires us to restart TTT. Since the learner does not abort by itself in such cases, we set a time limit of 24 and 72 hours, respectively, for the completion of the first and second alphabet. The execution time of the initial two alphabets is limited based on observations from a preliminary scan, where learning either failed to converge early or proceeded without issues across all alphabets.

5.5 Equivalence Tests/Counterexamples (5)

We chose a combination of three different strategies to generate counterexamples for TTT.

First, we use words describing the happy flows in TLS 1.2 and TLS 1.3 as counterexamples (5a), similar to the seeding of traditional fuzzers. This enables the learner to quickly identify common states of an implementation at the beginning of the state machine inference. The happy flows that should be contained in the state machine are determined with the help of module (2).

Second, for each state in the hypothesis, we generate 42,000 random words that walk through the selected state and extend further past it (5b). This step can be seen as the 'fuzzing' part of the machine learning algorithm. This happens for each hypothesis in all runs of TTT. So if n_i is the number of hypothesis output by TTT for alphabet i, S_i the number of states, we generate $(\sum n_i \cdot S_i) \cdot 42,000$ random words. Without our highly optimized cache, this amount of fuzzing would be very difficult, even locally, and impossible over the Internet. Our choice of 42,000 random words per state is based on a preliminary scan (see Section 7). Finally, we use assumed non-compliances and vulnerabilities detected by our automated security analysis (Section 5.6) as additional counterexamples (5c). This is done to verify that the issue is actually present in the *implementation* and not solely the result of an *inaccurate hypothesis*. For symbol sequences that yield an unexpected response in the hypothesis, we can simply query the SUL to ascertain that the assumed response is accurate. For issues related to the transitions between states, we run extra random word queries to check whether paths that unexpectedly lead to the same state do, in fact, exhibit the same behavior and whether paths that are expected to lead to the same state actually diverge.

5.6 Automated Security Analysis of State Machine Graphs (5c)

In our hypothesis testing and after creating the final hypothesis, we want to automatically analyze the extracted Mealy machine for bugs and vulnerabilities. For this purpose, we created a new approach. We provide the reasons for not following the approach of Fiterău-Broştean et al. [20] in Appendix B. Our algorithm is illustrated in steps (a) to (e) in Figure 3. We use a simplified representation of the extracted mealy machines here. Real machines encountered in our evaluations contained up to 39 states.

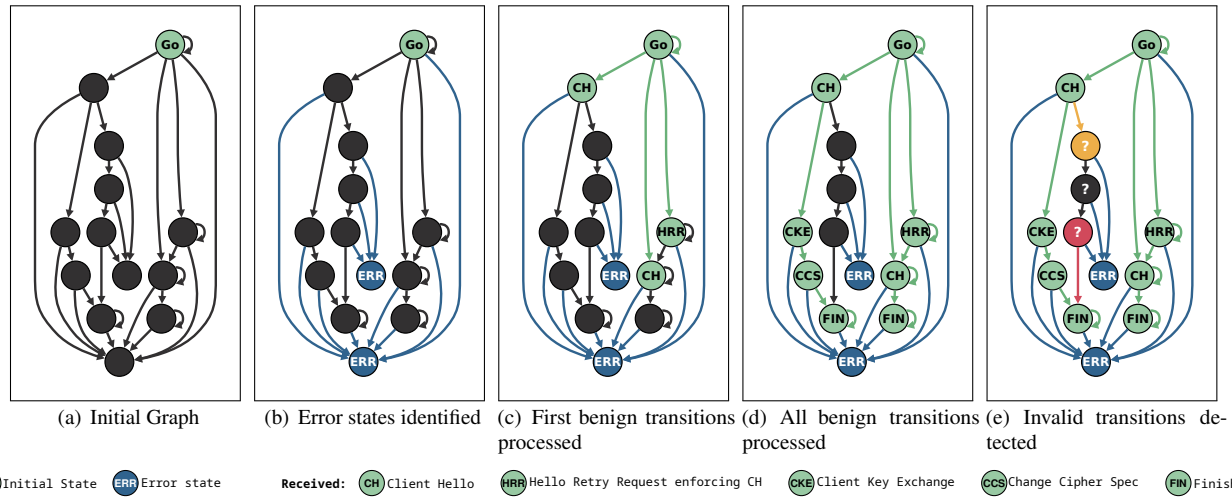
5.6.1 Valid Message Flows

Determining Error States (Figure 3 (b)) The first step in our evaluation is to identify *error states*. We consider all states that do not have any leaving edges[4] error states. Usually, an error state in TLS is expected to be equivalent to a closed TCP connection. However, some implementations, such as Amazon's s2n[5], deliberately keep the connection alive but unresponsive when errors have been detected. Using our error state definition, we can also cover the semantics of these implementations.

Determining Happy Flows (Figure 3 (c, d)) In the next step, we whitelist all edges and states that are part of a benign TLS connection (*happy flow*) based on the TLS RFCs. To do this, we walk through the state machine beginning from the starting state and use a list of rules called *symbol chains* to determine which edges and states are expected. We denote these edges and states as the *Happy Subgraph* (HSG). We define a *symbol chain* as a tuple of incoming and outgoing input symbols (e.g. '*ClientHello*, *ClientKeyExchange*'), as well as a context condition in which the rule is applicable. In the first step, we consider all symbol chains applicable to the starting

[4] Except for our *Reset Connection* symbol.

[5] https://github.com/aws/s2n-tls



(a) Initial Graph (b) Error states identified (c) First benign transitions processed (d) All benign transitions processed (e) Invalid transitions detected

Figure 3: Simplified illustration of our classification process that determines benign (green) and invalid (orange, red) edges and nodes in an extracted state machine. Blue edges and nodes denote detected error states. The labels of the benign handshake states indicate which messages have been sent to reach this state. For readability, we merge edges transitioning into the same successor state.

state. By doing so, we add the first edges and successor states to our HSG (green paths of (c) in Figure 3). We then assign labels to each newly added state of the HSG by considering the context from which we move to the respective state and the input and output symbols used in the transition. Additionally, we check that the output symbol is valid in response to the used input symbol. We then repeat this process for each new state of the HSG, marking valid edges in a given context, assigning labels to states we progress to, updating the context, and finally adding the edges and states we traversed to the HSG again. This process analyzes every edge at most once. If we reach an already visited state through different paths, we check if the session context of both paths matches. We report any discrepancies since a mismatch between the contexts means that the paths should traverse distinct states. [6] After this phase, all vertices that are part of a benign connection have a label associated with them, explaining their purpose and all edges that are expected to be present are whitelisted. We provide the list of symbol chains we applied in our artifact.

Determining Valid Self-Edges (Figure 3 (d)) Not all messages of a protocol affect the state of a session. For example, in TLS, a client may send an arbitrary number of application data records once the handshake has been concluded. All messages of the handshake, in contrast, are expected to cause a state change. When tracing the HSG, we hence also test if symbols that are intended to progress the session also result in a state change.

[6] The context comparison was not used to derive counterexamples during our scan.

5.6.2 Identifying Invalid Message Flows

Once we identified the subgraph containing the different *happy flows* and *error states*, we have a clear criterion on how to identify potential vulnerabilities. Any edge not leading to an error state that is not whitelisted is *unwanted*. However, the severity of an edge being present can vary significantly. We, therefore, further classify these unwanted edges based on the context we assigned during the creation of the HSG.

Unexpected Edges Leaving the HSG While tracing the HSG, we detect if a SUL rejects benign message sequences by transitioning into an error state. This behavior can be interpreted as an incompatibility (though it is not necessarily an RFC violation). We further detect illegal inputs which do not cause a termination of the session. This hints towards a too lax interpretation in the implementation's state machine.

Illegal Paths to HSG (Figure 3 (e)) Whenever there is an edge leading from a state *outside* the HSG to a state *within* the HSG, we know that there is a path from the initial state to the successful completion of the TLS handshake that *does not conform to any RFC*. E.g., in Figure 3 (e), there is an edge leading from the last of the three states labeled with a question mark (red) to the state 'Finished received'. Such paths may skip crucial cryptographic checks done on the happy flows and are thus *potentially dangerous*.

Ignored Messages We further test which non-optional messages are ignored by the SUL within a happy flow. These messages appear as edges that lead to the same node in the graph.

Unexpected States Whenever we leave the happy flow but do not enter an error state, we found an unnecessary state in the state machine. These states are potentially dangerous and require further manual analysis, as they allow attackers to influence the state of the TLS connection beyond what should be possible according to the protocol. If these states lead to exploitable behavior is hard to tell without extensive further testing or access to the implementation's source code.

Suspicious Alert Message We traverse various code paths while learning the state machine and trigger different error cases. Some of these error cases may be indicators for deeper issues in the implementation that cannot realistically be explored through state machine learning alone. Examples of this include alert messages that refer to a cryptographic error, like a `DECRYPTION_FAILURE` at a point in connection where the server should not be trying to decrypt anything. Another example includes `INTERNAL_ERROR` alerts, that according to the TLS specification, should only be sent in response to issues that are unrelated to the peers behavior. We search for this kind of behavior throughout all edges, even if they lead directly to an error state.

Identifying Bleichenbacher and Padding Oracle Vulnerabilities To identify Bleichenbacher and Padding Oracle vulnerabilities, we track if any of the attack-related inputs behave differently in a given state. We distinguish between two kinds of vulnerabilities: same-state or state-diverging vulnerabilities. A same-state vulnerability can be observed solely based on the response obtained when sending an attack input. The state-diverging vulnerability, on the other hand, may have an identical response for all attack inputs while showing distinct behaviors for subsequent messages, therefore leading to a different state. This case is especially interesting as it is more difficult to uncover using manually written tests.

6 Evaluation of Cache Optimizations

To assess the effect of the cache optimizations described in Section 5.4.2, we conducted a controlled evaluation for OpenSSL 3.4.0's `s_server` focusing on the queries we can save by employing the sink state optimizations and how often a cached prefix allows us to apply timeout shortcuts. The results across the thirteen alphabets are depicted in Table 2. As shown, between 36.3% and 58.7% of the responses provided by the cache stem from the sink-state optimization. For 78.98% to 84.7% of the times the learner waited for a response from the SUL, we could leverage cached prefixes to reduce the timeout. Note that the impact of our sink-state optimization drops back to around 36% in alphabets five and eleven due to our diversification of the *ClientHello* input sets, which introduce new handshake paths that are not covered by the cache yet. Across all alphabets, 38,277,436 queries could be replied by the cache, with 19,483,321 replies provided by the sink-state optimizations, resulting in an overall ratio of 50.90%. For 1,329,555 of 1,613,752 symbols (82.39%) in total sent to OpenSSL, timeout shortcuts could be applied.

	Cached			Responses	
↓a z	Total	Opt-S	Uncached	Total	Opt-T
1	375,747	36.80%	2,447	16,258	81.58%
2	1,749,046	49.07%	31,323	224,172	83.33%
3	1,038,261	58.70%	19,121	96,630	78.98%
4	1,446,303	56.86%	14,439	75,973	79.73%
5	374,154	37.24%	4,086	31,294	81.71%
6	1,789,695	45.96%	17,703	133,520	84.70%
7	2,227,571	54.09%	17,925	104,252	81.43%
8	3,114,955	53.30%	16,512	99,116	81.93%
9	374,207	36.30%	4,056	33,332	81.78%
10	2,046,359	45.55%	18,752	134,748	83.79%
11	8,029,991	51.69%	44,653	275,612	82.39%
12	7,561,042	51.75%	36,804	226,495	82.19%
13	8,150,105	50.26%	25,895	162,350	82.81%

Table 2: Comparison of the number of queries and the number of awaited responses during the learning of each of our 13 alphabets (↓a z). For cached queries, we report the ratio contributed by our sink-state optimizations (Opt-S). We further state the ratio of responses that could be captured with a reduced timeout (Opt-T). Note that the cache is retained across all alphabets. Hence, all subsequent alphabets benefit from previous queries.

To compare the impact on the actual execution time, we further ran our state learner for the first alphabet in four different configurations: with an unoptimized generic cache, with a cache employing our sink-state optimizations, with a cache employing timeout shortcuts, and with a cache employing both strategies. All experiments were performed on a virtual machine powered by an Intel Xeon E5-2640 CPU and with the same parameters as applied for our Internet scan. As shown in Table 3, the execution time ranges from 369 minutes (without optimizations) to 5 minutes (with both optimizations), highlighting the benefits of the optimizations. Disabling the sink state optimizations leads to an increase of 138,260 connections for the first alphabet. This is expected due to the loss of 37% of the cache efficiency (see Table 2). Employing timeout shortcuts alone already reduces the execution time from 369 minutes to 161, which indicates that a large share of the time is spent on capturing the SUL's responses.

7 Internet Evaluation

To evaluate our methodology on the Internet, we selected domains for our study from popular and open bug bounty programs provided by a GitHub repository that collects bug bounty domains [7], resulting in 7337 domains. To determine the parameters for our scan, we first performed a preliminary

[7] https://github.com/arkadiyt/bounty-targets-data

Setup	Connections	Execution Time
Unoptimized Cache	140,707	369 min, 53 sec
Cache Opt-T	140,707	161 min, 3 sec
Cache Opt-S	2,447	14 min, 45 sec
Cache Opt-S, Opt-T	2,447	5 min, 35 sec

Table 3: Comparison of connections and execution time required to complete the first alphabet in setups with sink-state optimizations (Opt-S) and timeout shortcuts (Opt-T) activated or deactivated. We applied our methodology from Section 7.2 to assert that all setups yielded the same effective state machines.

scan on the top 100 HTTPS domains of the Tranco list. We chose these domains as we expect them to be the least influenced by potential noise created by our connections spread out over time. We configured our equivalence tests to evaluate hypotheses with 60,000 random words per state. Our learner converged for 54 of the 100 domains. For 95% of equivalence tests, counterexamples could be found with less than 42,000 random words. We decided to use this value for our main study.

7.1 Results

Of the 7337 targeted domains, 1486 domains could not be scanned as they did not support TLS on port 443. Of the remaining domains, learning converged for 1285 (22%) hosts. For 2303 domains, learning was aborted before finishing the first alphabet, and for 2224, learning was aborted before finishing the second alphabet. Among these, 173 domains blocked our connection attempts, while the rest exceeded the time thresholds of 24 and 72 hours we set for the first two alphabets. Finally, 39 domains completed the first and second alphabets but did not converge afterward. Among the hosts for which the state learning has been aborted, only 19 finished at least the first *Vulnerabilities* alphabet that contained Bleichenbacher and Padding Oracle test vectors. Below, we present the results of these 19 incomplete and the 1285 completed domains. In our scans, these domains corresponded to 835 unique IP addresses.

Overview Our initial feature extraction determined TLS 1.2 support for 1286 (98.6%) of the considered domains, with 374 (28.7%) supporting only TLS 1.2. TLS 1.3 was deemed supported for 912 domains (69.9%), with 18 (1.4%) supporting exclusively the newest version. We were able to complete TLS 1.2 handshakes with 1269 domains and TLS 1.3 handshakes with 904 domains. 606 domains (46.5%) domains offered RSA key exchange, which allowed us to test for Bleichenbacher vulnerabilities, and 887 (68%) negotiated CBC cipher suites enabling padding oracle tests.

For 12 domains, we were unable to complete any handshakes. Five of these are due to servers that requested client authentication and rejected our self-signed certificate. Three domains, in contrast, requested authentication and accepted our certificate.[8] Only 47 domains accepted our client-initiated renegotiation handshake, and 436 accepted our ID-based resumption attempts.

The alphabets used to extract the final state machines ranged from 40 to 60 inputs with an average of 54.99 symbols. The number of states ranged from 4 to 39, with a mean of 21.22. Table 4 provides an overview of the distribution of applicable symbols. We provide statistics on the timeout adjustments made by our tool at runtime and the duration of our automated analysis in Table 6 and Table 7 of Appendix A. Additional statistics on the alphabets along with the collected datasets can be found in our artifact.

Property	Domains
Domains learned with TLS 1.2 *ClientHello*	1286
Domains with successful TLS 1.2 handshakes	1269
Domains learned with TLS 1.3 *ClientHello*	912
Domains with successful TLS 1.3 handshakes	904
No handshake succeeded	12
Any TLS 1.3 handshake succeeded, none TLS 1.2[a]	23
Any TLS 1.2 handshake succeeded, none TLS 1.3[a]	3
Domains learned with ECDHE *ClientHello*	1276
Domains learned with AEAD *ClientHello*[b]	1272
Domains learned with Block Cipher *ClientHello*[b]	887
Domains learned with RSA Key Exchange *ClientHello*	606
Domains learned with DHE *ClientHello*	159
Domains learned with Stream Cipher *ClientHello*[b]	2
Domains learned with ECDH *ClientHello*	1
Domains learned with DH *ClientHello*	1
Domains accepting renegotiation	47
Domains accepting resumption	436
Domains requesting client certificate	8
Certificate requested, any handshake succeeded	3
Certificate requested, no handshake succeeded	5

a: Only counting domains with both versions in alphabet

b: Classified based on RFC 5246 *CipherType* definition

Table 4: Overview of the frequency of TLS features supported by analyzed domains.

Table 5 provides an overview of the state machine issues we tracked according to our methodology presented in Section 5. As shown, few state machines contain concerning paths that leave the HSG (33), and only two contain paths returning back to it. Conversely, a larger share of 465 domains (35.7%) ignored unexpected messages, and 282 (21.6%) sent `Internal Error` alerts. Two domains showed a padding oracle vulnerability, but no domain was vulnerable to our Bleichenbacher vectors. The low number of vulnerabilities may be related to our selection of bug bounty websites that are expected to be

[8] While the provided self-signed certificate should be invalid for any server, it is unclear if any privileges could be gained using our test certificate.

Property	Domains
Benign inputs leading to error states	39
Edges leaving HSG without error	33
Illegal paths to HSG	2
Ignored messages	465
Suspicious alert messages	282
Padding oracles	2
Bleichenbacher vulnerability	0

Table 5: Overview of the state machine issues we aimed to evaluate as described in Section 5.

highly tested for issues like these.

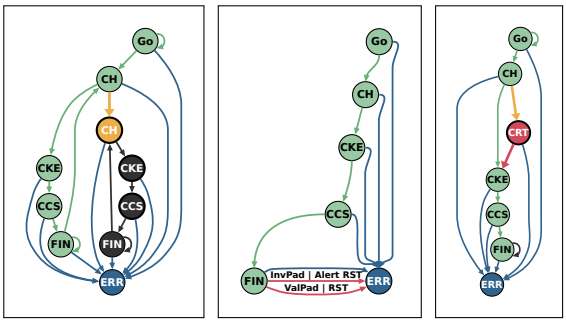
After our scan, we manually analyzed the impact of the reported findings from our automatic analysis with further manual tests.

7.1.1 Session Transcript Vulnerability

Our automated analysis reported 11 domains, which accepted two *ClientHello* messages in the handshake. Through manual testing, we found that the affected servers do not include the first *ClientHello* in the session transcript validated through the *Finished* message. Both *ClientHello*s do, however, affect the parameter negotiation. Generally, the integrity of the transcript is critical as it plays a major role in proving the security of TLS 1.2 [26, 27] (and TLS 1.3 [14, 15, 18]). If it is violated, these security guarantees do not hold anymore. In this specific case, we found that a Man-in-the-Middle attacker could exploit the server's behavior and inject additional extensions into a handshake through a crafted *ClientHello* sent before the benign *ClientHello* of a victim. The server would process the extension without leaving any trace of the attacker's *ClientHello* in the session transcript. Consequently, the victim could conclude the handshake with the server without realizing the session had been tampered with. Under specific circumstances, this could be used to conduct a renegotiation attack [33, 35] against the client. An excerpt of the affected state machines illustrating the invalid handshake flow is shown in (a) of Figure 4. According to our analysis, the affected domains use Citrix NetScaler. A similar issue was mentioned by Ruiter and Poll [12] in OpenSSL 1.0.1j. While the discovered issue is similar, the state machine of the analyzed NetScaler hosts is clearly distinct from OpenSSL's.

7.1.2 Padding Oracle Vulnerabilities

Our automated analysis reported two domains with CBC padding oracle vulnerabilities. In both cases, the servers react to messages with valid padding but an invalid MAC (`ValPad`) by issuing a *bad record mac* TLS alert. Conversely, messages with invalid padding (`InvPad`) only yield a TCP RST without any further TLS messages, as depicted in Figure 4 (b). This behavior can allow an attacker to break confidentiality based on the padding of manipulated records. This hints that SML can find complex padding oracles, as reported in ROBOT [7].



(a) NetScaler (b) Padding Oracle (c) Unsolicited Certificate

Figure 4: Illustration of notable findings. State labels and edge colors are used as stated in Figure 3. For (b), we denote the sent symbol and output as `<input>|<output>`.

7.1.3 Misc Findings

Ignored Unsolicited Client Certificates The automated analysis reported two domains that accepted a *Certificate* message sent by our learner even though no certificate had been requested by these domains. The analysis further reported that it was possible to conclude the handshake by subsequently sending a *ClientKeyExchange*, *ChangeCipherSpec*, and *Finished* message, leaving out the *CertificateVerify* message that is usually expected after sending a client certificate with signing capabilities (cf. Figure 4 (c)). While this separate handshake path existed in the extracted state machines, it was also possible to conclude the handshake following the benign message flow that does not contain the unsolicited *Certificate* message.

Additional Domains Accepting Multiple Handshake Messages We found three additional domains that accepted multiple TLS 1.2 *ClientHello* messages. In contrast to the NetScaler domains, however, these hosts included all messages in the session transcript. Even if an attacker is able to manipulate the session context by prefixing a victim's *ClientHello*, the handshake will thus eventually fail when the server sends its *Finished* messages as the transcripts of the peers do not align.

Aborted Renegotiation Handshakes We found that 38 domains accepted a renegotiation handshake but terminated the connection simultaneously. When receiving a *ClientHello* after an already completed handshake, these hosts would reply with the usual flight of handshake messages (from *ServerHello* to *ServerHelloDone*) but immediately also send a TCP FIN. While renegotiation is an optional feature, the RFC defines that a *no renegotiation* warning alert *should* be sent when rejecting it instead of initiating a partial handshake that terminates the connection.

Ignored Application Data During Handshake Our automated analysis reported that 43 domains tolerated application data sent during the handshake without sending any reply. To determine if the application data was silently ignored or passed to the application layer, we manually tested these domains by splitting an HTTP request and sending the first part during the handshake and the rest after the handshake. While these domains usually replied to HTTP requests, none replied to the split HTTP request, suggesting that the early application data was ignored. Since we do not have access to the internals of the implementation, we cannot entirely rule out that the bug is exploitable.

Ignored Post Handshake ChangeCipherSpec (CCS) Messages We found that 396 TLS 1.2 servers and 89 TLS 1.3 servers ignored a *ChangeCipherSpec* sent after a completed handshake. In TLS 1.2, additional CCS messages can only be sent in a renegotiation handshake. TLS 1.3 explicitly demands that servers treat a CCS sent after the handshake as an unexpected record type that must be met with a fatal alert. Although neither case poses a security vulnerability, they highlight an issue with compliance with the specification.

Unrejected Initial Messages For 17 domains, our analysis reported that various messages, such as *ChangeCipherSpec* and *ServerHello*, have been seemingly accepted when sent at the start of the TCP session. However, any subsequent messages led to a closed TCP connection without any records sent by the server. While messages such as *ChangeCipherSpec* may cause a premature key derivation in the server, we assume that these servers employ some sort of traffic filtering where only TCP traffic starting with a *ClientHello* record is relayed to the TLS implementation.

Ignored Close Notify For 59 domains, it was possible to send a *close_notify* alert signaling the end of the TLS session without changing the session state. Subsequently, it was still possible to send the same messages as before, both in and after a handshake, showing that these messages were simply ignored by the server.

Missing Key Update Support We found one server that supported TLS 1.3 but could not handle a key update request. Any key update message immediately terminated the TCP connection without any further TLS records.

7.1.4 Limitations

Due to a bug in TLS-Attacker, our TLS 1.3 *ClientHello* incorrectly did not retain the *client random* when we enforced a *HelloRetryRequest* before. The implementations of 87 domains (9.5%) verified this field and rejected our handshake attempts in these cases. Regular TLS 1.3 handshakes were not affected by this. Additionally, due to a bug in our mapper, majority vote queries have not been blocked by sink states (see Section 5.4.2). Consequently, 43 extracted state machines contained redundant nodes introduced solely because a ϕ was incorrectly not applied.

Excluded Hosts During the study, we excluded multiple hosts from the evaluation as they did not finish learning in the respective 24-hour and 72-hour timeframes. When comparing the statistics of successfully evaluated and aborted targets, we found that unsuccessful executions exhibit an increased count of cache conflicts and hypotheses. For example, 75% of successful targets finished the first alphabet within 158 or less hypotheses, whereas the 75% quantile among unsuccessful executions is at 2016. Similarly, 75% of the presented domains finished the first alphabet with 60 or less cache conflicts, compared to 798 among the unsuccessful ones. Both numbers hint towards a higher level of non-determinism, slowing down the learning and hindering the construction of a definitive hypothesis. Finally, we also found outliers where the smallest alphabet already resulted in up to 197 states in the hypothesis. We provide a comparison of the performance for successful and aborted domains in Appendix D. Additional statistics on our Internet study are included in our artifact.

7.2 Clustering State Machines

To provide insights into the state machine ecosystem and to give an impression of the seen diversity, we tried to cluster the extracted state machines but stress that further research in this direction is necessary.

Comparing the State Machines We base the comparison of state machines on a bisimulation approach considering paths of the shared alphabet of two state machines. We define the similarity as the ratio of matching paths and the total number of paths considered. To emphasize differences where one state machine accepts an input while the other rejects it and terminates the connection, we applied a weight equal to the size of their shared alphabet.

Creating a Graph Using the similarity scores, we created a graph representing the evaluated hosts. In this graph, each vertex represents a domain, while the edges between vertices represent the similarity between hosts by their respective similarity scores. Two vertices are connected if the similarity of their associated state machines exceeds the average similarity of all state machine comparisons. We then embed the graph in a two-dimensional plane using the ForceAtlas2 algorithm [25], as implemented in the Gephi[9] software. To provide some labeled data for reference, we added four additional state machines extracted in a local environment: OpenSSL versions 3.4.0, 1.1.1, 1.0.2a, and mbedTLS 3.6.0. We chose OpenSSL as it is used in various web applications and mbedTLS because we would not expect it to be used for HTTPS servers. In Figure 5, we highlight these state machines with bigger

[9] https://gephi.org/

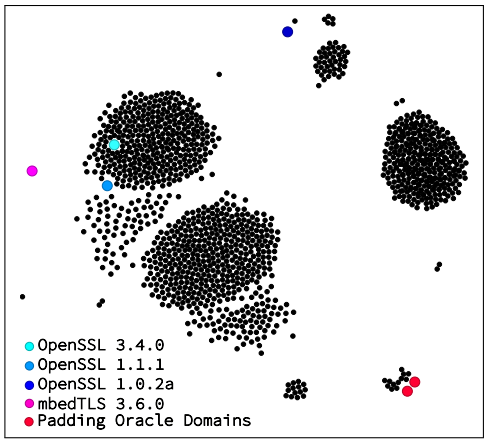


Figure 5: Exemplary clustering of domains based on similarity of their extracted state machines. As a fixed point for comparison, we added state machines extracted for the open-source libraries OpenSSL and mbedTLS.

and colored vertices. We further highlight the two hosts with padding oracle vulnerabilities. As depicted, we found multiple domains that behaved very similarly to OpenSSL 3.4.0. In comparison, the older OpenSSL 1.0.2a, released in 2015, shows little similarities to the analyzed domains.

8 Conclusion

In this work, we showed that combining large-scale analysis with SML is suited to gather unique insights into the TLS ecosystem. For 22% of TLS supporting targets, we were able to extract the state machine while considering the full scope of a typical TLS implementation. Among the domains in our study, we observed that state machine flaws are common, but from the outside often appear non-critical. In many cases, invalid messages are ignored rather than explicitly rejected, without causing observable deviations in the state machine or triggering distinct invalid handshake paths. For a definite answer regarding exploitability, access to the binary or source code is required. Our clustering experiments further suggest that most targets form larger groups with similar response patterns. However, among the more isolated domains, we identify issues that do impact protocol security, including padding oracle vulnerabilities and violations of session transcript integrity. Based on these findings, we conclude that common implementations provide mostly protocol-conforming implementations, while deeper flaws exist in a small number of outlier cases. These findings indicate that studying more hosts on the Internet may be valuable. At the same time, our study also hints towards the limitations of state machine learning. By considering session resumption in our study, we significantly worsened the performance of the extraction process. This is primarily due to the fact that session resumption inherently creates an internal state that is poorly representable in a mealy machine. At the same time, we could not extract the state machine of many hosts. While this can partially be explained by our self-set restriction for ethical scanning and limiting us to slow, dragged-out scans, non-determinism, and unexpected message handling prevented learning from converging.

A way forward for state-machine fuzzing in this environment could be to embrace the complexity and provide confidence measures for different areas in the state-machine and avoid exploring non-deterministic parts further automatically. Another direction for future work could be to analyze the relation of hosts on the Internet based on their state machines. While our results indicate clusters of hosts whose state machines are similar, it would be interesting to analyze whether these state machines can be mapped to known TLS libraries to create fingerprinting capability.

9 Ethics Considerations

Since obtaining a state machine representation requires many connections to a server, we configured our implementation to pause for at least 1.5 seconds after each connection. We further only scan each host with a single thread and do not utilize messages specifically aimed to cause buffer overflows, trigger known crashes of TLS libraries or to test for attacks that may result in the leakage of private keys or user data (e.g. we did not include alphabet symbols triggering buffer boundary violations, for example, for the Heartbleed attack [16]). We limited our study to hosts participating in bug-bounty programs as these are expected to be more resilient against non-conforming packets. When we detect that a peer does not respond to our queries for a prolonged time while our Internet connection appears healthy, we stop scanning the host to respect possible blacklisting. We provided a web page on our scanning IP that informed about the purpose of our scans and provided a contact address to request an opt-out. The test vectors we employed to detect padding oracle and Bleichenbacher vulnerabilities are not suited to extract keys of other TLS sessions or to obtain any user data.

Regarding the publication of anonymized state machines (see below), we believe that the potential insights gained from further studies outweigh any potential risks. If the state machines hint towards vulnerabilities not considered in the scope of this paper, a malicious actor would first have to derive queries suitable to identify a host on the Internet.

Responsible Disclosure We responsibly disclosed the identified padding oracle and transcript integrity vulnerabilities to the respective domain owners/software vendors and provided support to test for reported issues.

10 Open-Science

We provide the tools we developed and used in this work as open-source software under the Apache 2 license. Extracted

state machines will be openly available under pseudonyms with message fields identifying SULs (e.g., certificates) stripped. For long-term availability, we provide these artifacts on Zenodo.[10] For the artifact evaluation, we will undergo verification of the availability, functionality, and reproducibility.

Acknowledgements

We thank the anonymous reviewers and shepherd for their valuable feedback. We further thank Joeri de Ruiter, Matthias Geuchen, and Niklas Niere for their contributions to the state learner project. This research was partially supported by Germany's Excellence Strategy - EXC 2092 CASA - 390781972, the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 450197914, and by the German Federal Ministry of Research, Technology and Space (BMFTR) through the project KoTeBi.

References

[1] TLS-Scanner. https://github.com/tls-attacker/TLS-Scanner.

[2] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *ACM CCS 2015*, 2015. ACM Press.

[3] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[4] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS using SSLv2. In *USENIX Security 2016*, 2016. USENIX Association.

[5] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *2015 IEEE Symposium on Security and Privacy*, 2015. IEEE Computer Society Press.

[6] Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *ACM CCS 2016*, 2016. ACM Press.

[7] Hanno Böck, Juraj Somorovsky, and Craig Young. Return of Bleichenbacher's oracle threat (ROBOT). In *USENIX Security 2018*, 2018. USENIX Association.

[8] Marcus Brinkmann, Christian Dresen, Robert Merget, Damian Poddebniak, Jens Müller, Juraj Somorovsky, Jörg Schwenk, and Sebastian Schinzel. ALPACA: Application layer protocol confusion - analyzing and mitigating cracks in TLS authentication. In *USENIX Security 2021*. USENIX Association, 2021.

[9] Chia Yuan Cho, Domagoj Babic, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *USENIX Security 2011*, 2011. USENIX Association.

[10] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Krügel, and Engin Kirda. Prospex: Protocol specification extraction. In *2009 IEEE Symposium on Security and Privacy*, 2009. IEEE Computer Society Press.

[11] Joeri de Ruiter. A tale of the openssl state machine: A large-scale black-box analysis. In *Secure IT Systems: 21st Nordic Conference, NordSec 2016, Proceedings*, volume 10014 of *Lecture Notes in Computer Science*. Springer International Publishing, 2016.

[12] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *USENIX Security 2015*, 2015. USENIX Association.

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

[14] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In *ACM CCS 2015*, 2015. ACM Press.

[15] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol. *Journal of Cryptology*, 34(4):37, 2021.

[16] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The Matter of Heartbleed. In *Conference on Internet Measurement Conference, IMC*, 2014.

[17] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. Prognosis: closed-box analysis of network protocol implementations. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, 2021. Association for Computing Machinery.

[18] Marc Fischlin, Felix Günther, Benedikt Schmidt, and Bogdan Warinschi. Key confirmation in key exchange: A formal treatment and implications for TLS 1.3. In *2016 IEEE Symposium on Security and Privacy*, 2016.

[10] https://doi.org/10.5281/zenodo.15520932

IEEE Computer Society Press.

[19] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS implementations using protocol state fuzzing. In *USENIX Security 2020*. USENIX Association, 2020.

[20] Paul Fiterau-Brostean, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. Automata-based automated detection of state machine bugs in protocol implementations. In *NDSS 2023*, 2023. The Internet Society.

[21] Paul Fiterău-Broştean, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. Smbugfinder: An automated framework for testing protocol implementations for state machine bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, 2024. Association for Computing Machinery.

[22] Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. Model learning and model checking of ssh implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, 2017. Association for Computing Machinery.

[23] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Proceedings 11*. Springer, 2015.

[24] Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *Runtime Verification - 5th International Conference, RV 2014, Proceedings*, volume 8734 of *Lecture Notes in Computer Science*. Springer, 2014.

[25] Mathieu Jacomy, Tommaso Venturini, Sebastien Heymann, and Mathieu Bastian. Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. volume 9. Public Library of Science, 2014.

[26] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DHE in the standard model. In *CRYPTO 2012*, volume 7417 of *LNCS*, 2012. Springer Berlin Heidelberg, Germany.

[27] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. On the security of the TLS protocol: A systematic analysis. In *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, 2013. Springer Berlin Heidelberg, Germany.

[28] Marcel Maehren, Philipp Nieting, Sven Hebrok, Robert Merget, Juraj Somorovsky, and Jörg Schwenk. TLS-anvil: Adapting combinatorial testing for TLS libraries. In *USENIX Security 2022*, 2022. USENIX Association.

[29] Robert Merget, Marcus Brinkmann, Nimrod Aviram, Juraj Somorovsky, Johannes Mittmann, and Jörg Schwenk. Raccoon attack: Finding and exploiting most-significant-bit-oracles in TLS-DH(E). In *USENIX Security 2021*. USENIX Association, 2021.

[30] Robert Merget, Juraj Somorovsky, Nimrod Aviram, Craig Young, Janis Fliegenschmidt, Jörg Schwenk, and Yuval Shavitt. Scalable scanning and automatic classification of TLS padding oracle vulnerabilities. In *USENIX Security 2019*, 2019. USENIX Association.

[31] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020.

[32] Aina Toky Rasoamanana, Olivier Levillain, and Hervé Debar. Towards a systematic and automatic use of state machine inference to uncover security flaws and fingerprint TLS stacks. In *ESORICS 2022, Part III*, volume 13556 of *LNCS*, 2022. Springer, Cham, Switzerland.

[33] M. Ray and S. Dispensa. Authentication gap in TLS renegotiation, 2009.

[34] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, IETF, 2018.

[35] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746, IETF, 2010.

[36] Juliano Rizzo and Thai Duong. The CRIME attack. In *Ekoparty Security Conference*, 2012.

[37] Konstantinos Sagonas and Thanasis Typaldos. Edhoc-fuzzer: An edhoc protocol state fuzzer. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, 2023. Association for Computing Machinery.

[38] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077, IETF, 2008.

[39] R. Seggelmann, M. Tuexen, and M. Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520, IETF, 2012.

[40] Juraj Somorovsky. Systematic fuzzing and testing of TLS libraries. In *ACM CCS 2016*, 2016. ACM Press.

[41] Chris McMahon Stone, Tom Chothia, and Joeri de Ruiter. Extending automated protocol state learning for the 802.11 4-way handshake. In *ESORICS 2018, Part I*, volume 11098 of *LNCS*, 2018. Springer, Cham, Switzerland.

[42] Chris McMahon Stone, Sam L. Thomas, Mathy Vanhoef, James Henderson, Nicolas Bailluet, and Tom Chothia. The closer you look, the more you learn: A grey-box approach to protocol state machine learning. In *ACM*

CCS 2022, 2022. ACM Press.

[43] Luke Valenta, Nick Sullivan, Antonio Sanso, and Nadia Heninger. In search of CurveSwap: Measuring elliptic curve implementations in the wild. In *2018 IEEE European Symposium on Security and Privacy*, 2018. IEEE Computer Society Press.

[44] Tarun Yadav and Koustav Sadhukhan. Identification of bugs and vulnerabilities in tls implementation for windows operating system using state machine learning. In *Security in Computing and Communications*, 2019. Springer Singapore.

A Internet Evaluation Statistics

Statistic	Initial Timeouts (ms)	Timeout Adjustments
min	50	0
25%	50	2
50% (median)	98	6
75%	200	14
max	1200	78
average	156.83	9

Table 6: Statistics on timeout adjustments made by our tool based on observed cache conflicts for the 1285 completed domains: each adjustment added 10ms, with timeouts starting from 50ms (based on stable timeouts observed in prescans) and limited to 1200ms.

Statistic	Analysis Duration (ms)
min	333
25%	1335
50% (median)	1808
75%	2608
max	5497
average	2002.07

Table 7: Statistics on the duration of the automated analysis of the 1285 completed domains as measured on an AMD EPYC 7763.

B Shortcomings of Fiterău-Broştean et al. for Large-Scale Evaluations

Fiterău-Broştean et al. [20] introduced two approaches for automated analysis in the context of DTLS. They either used a DFA-based whitelist or a blacklist approach. In the blacklist approach, they used known or plausible state machine issues represented as DFA's, whereas for the whitelist approach, the defined a ground truth 'path' as a DFA and considered deviations as bugs. They then heuristically considered sequences of messages through the mealy machine and traced them through the considered DFA. If the DFA accepted, they considered a bug to be present. Both approaches have significant drawbacks. Their white-listing scales poorly in computation time. According to Fiterău-Broştean et al. [20], analysis of an individual state machine could take over 24 hours using this approach. At the same time, the blacklisting approach has the issue that it can only find *known* vulnerabilities, introducing false negatives into the evaluation. Consider the example from the paper in Figure 6.

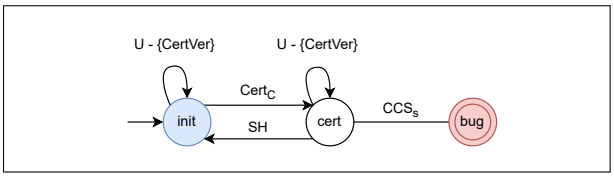


Figure 6: An example bug pattern from Fiterău-Broştean et al. [20]. The back edge from the `cert` state to the `init` state can lead to false negative results if the present bug causes the server to send an out-of-order *ServerHello* message.

The example tries to detect cases where the server accepts a client connection where the client did not present a *CertificateVerify* message. While this pattern can detect previous vulnerabilities like CVE-2020-2655, it fails to generalize. Consider the following (artificial) message flow:

```
(CH ->) (SH, C, CR, SHD <-),
        (C ->),
      (CKE ->),
        (CH ->),
(CKE ->) (SH, CCS, FIN <-)
```

In this example, the attacker deviated from the happy flow by sending a *ClientHello* and *ClientKeyExchange* message after we had already sent the *ClientKeyExchange* message, which triggered a hypothetical bug in the server to send a *ServerHello*, *ChangeCipherSpec*,*Finished* message. The *ServerHello* from the server confuses the bug pattern into believing that a new connection is getting established, which moves the DFA into the starting state again where it 'forgot' that the client already sent a *Certificate*, even though the state the server is in is clearly different from the start state. The back edge in the DFA is there to account for a mapper artifact (it is not related to renegotiation as claimed in the paper) and prevents the DFA from maintaining context, which creates a false negative.

Additionally, both approaches do not provide context and, therefore, fail to explain and accurately classify bugs. On a technical level, by only considering message sequences that visit a given state up to K times, their approach cannot guarantee that, even if a bug is present, the approach accurately detects it.

C Alphabet Sets

	Symbol	Sender	🤝 / 💬	Valid	Description
Happy Flow	0 - 5 Selected TLS 1.2 Client Hellos[a]	💻	🤝		Initiates a new TLS 1.2 handshake
	0 - 1 Selected TLS 1.3 Client Hello[a]	💻	🤝		Initiates a new TLS 1.3 handshake
	Certificate	★	🤝		Provides a trust chain to verify peer identity
	Client Key Exchange	💻	🤝		Provides the client's key exchange material
	Change Cipher Spec	💻	🤝		Signals sender's transition to negotiated key set
	Finished	★	🤝		Concludes and verifies handshake
	Hello Retry Enforcing Client Hello	🔍	🤝		A *ClientHello* modified to enforce a *HelloRetryRequest*
	Close Notify Warning Alert	★	★		Signals closure of the TLS session
	Application Data (HTTP)	★	💬		Application Data containing an HTTP request
Extended	Server Hello Done	🖥	🤝	ϟ	Terminates the first transmission of a TLS 1.2 server
	Server Key Exchange	🖥	🤝	ϟ	Provides the server's key exchange material
	Encrypted Extensions	🖥	🤝	ϟ	Provides the server's list of negotiated extensions in TLS 1.3
	New Session Ticket	🖥	🤝	ϟ	Issues a session ticket for stateless session resumption
	Hello Request	🖥	💬	ϟ	Asks the client to inititate a new handshake (*renegotiation*)
	Hello Retry Request	🖥	🤝	ϟ	Requests a key share for a different group from the client
	Certificate Verify[b]	★	🤝	ϟ	Authenticates a party through a signature over the transcript
	Key Update	★	💬		Signals a key refresh in TLS 1.3
	Resuming Client Hello	💻	🤝		A *ClientHello* designed to resume the most recently established session
	Dummy Change Cipher Spec	🔍	🤝	ϟ	A *ChangeCipherSpec* modified to retain the mapper's record layer state
	Reset Connection	🔍	★		Artificial input to start a new TCP connection
Vuln	12 Bleichenbacher Test Vectors	🔍	🤝	ϟ	Sends invalid PKCS#1v1.5 structures encrypted using RSA
	16 Padding Oracle Test Vectors	🔍	★	ϟ	Sends records encrypted with invalid block cipher padding
All	Certificate Request	🖥	🤝	ϟ	Requests a client to provide a certificate
	DTLS Hello Verify Request	🖥	🤝	ϟ	Provides a cookie to a DTLS client to avoid DoS attacks
	End of Early Data	💻	🤝	ϟ	Signals the end of 0-RTT early data sent by a client in TLS 1.3

💻 Client Message, 🖥 Server Message, 🔍 Artificial input to test error cases and corner cases of implementations, 🤝 Handshake Message, 💬 Post-Handshake Message, ★ Both, ϟ Never used RFC Compliant

a: The number of *ClientHello* inputs increases with each iteration. Not all hosts supported TLS 1.2 and 1.3. All hosts were tested with at least one *ClientHello* input and at most five TLS 1.2 and one TLS 1.3 input.

b: Since we use a self-signed certificate, the *CertificateVerify* message is expected to be rejected if strict client authentication is enabled.

Table 8: Overview of the main alphabet sets. Each subsequent set contains all inputs of the sets listed before. We further state which peer is intended to issue the message, where it appears (handshake, post-handshake, any time) and if this message is always non-compliant in our context (ϟ).

D Performance Comparison for Initial Alphabets

		min	25%	median	75%	max	average
CH_1_HappyFlow	States	2	7	9	9	16	8.36
	Hypotheses	2	5	31	158	3,454	123.87
	Queries Replied by Cache	83,974	423,352	907,320	2,572,937	278,606,819	4,000,033.98
	Queries Replied by SUL	7	541	1,760	3,675	37,434	2,692.9
	Symbols Used	336,545	2,195,518	4,287,020	11,847,074	1,317,893,626	18,649,320.90
	Received with Expectation	36	2,489	7,330	15,432	178,850	12,724.80
	Received Total	40	3,034	9,234	18,995	214,324	15,436.12
	Cache Conflicts	0	1	15	60	2,034	51.98
CH_1_Extended	States	4	14	15	19	30	15.57
	Hypotheses	3	95	264	640	14,045	562.72
	Queries Replied by Cache	167,695	8,658,513	22,006,064	58,568,008	839,996,752	48,756,511.94
	Queries Replied by SUL	939	26,739	48,048	70,691	147,734	51,099.30
	Symbols Used	758,924	44,440,894	108,897,878	281,722,099	4,107,762,442	237,882,777.32
	Received with Expectation	4,371	166,290	278,450	423,128	919,796	311,965.23
	Received Total	5,914	196,480	332,622	503,468	1,089,777	370,128.83
	Cache Conflicts	0	15	50	127	3,081	109.71
CH_1_HappyFlow	States	1	9	9	10	197	9.54
	Hypotheses	2	30	238	2,016	21,991	1,654.57
	Queries Replied by Cache	589	1,135,976	3,189,668	17,482,726	2,749,319,826	41,013,613.97
	Queries Replied by SUL	7	1,513	24,935	48,771	79,827	27,197.59
	Symbols Used	2,453	5,098,129	13,828,610	77,841,349	16,307,477,089	189,122,956.40
	Received with Expectation	36	7,512	45,280	259,822	721,301	132,344.26
	Received Total	42	8,919	57,725	325,793	822,719	165,228.25
	Cache Conflicts	0	12	86	798	4,842	754.0
CH_1_Extended	States	1	1	12	15	69	10.43
	Hypotheses	14	739	1,571	3,352	18,604	2,636.83
	Queries Replied by Cache	291,204	60,854,965	119,736,605	228,571,011	1,274,117,558	166,340,273.20
	Queries Replied by SUL	630	54,089	77,053	101,685	154,941	77,751.59
	Symbols Used	1,177,841	289,599,411	566,117,179	1,067,805,581	6,276,920,230	773,436,849.39
	Received with Expectation	4,082	303,861	421,342	597,476	1,228,715	454,394.47
	Received Total	4,780	364,639	506,620	708,302	1,363,254	540,779.32
	Cache Conflicts	1	133	297	673	7,063	565.9

Table 9: Comparison of the performance statistics for the state learning of the first two alphabets of the 1304 results presented in Section 7 (top) and the domains aborted in the first and second alphabet (bottom). Notably, the number of hypotheses and cache conflicts is significantly higher among the aborted domains. Note that the number of presented states for aborted domains was evaluated based on the latest hypothesis. Due to cache conflicts, the learner may have temporarily discarded assumed states again. Hence, the presented numbers only serve as a lower bound. We provide additional data on the performance as part of our artifact.