Using Active Automata Learning to Find Vulnerabilities in Network Stacks

Olivier Levillain[1], Aina Toky Rasoamanana[2], and Yohan Pipereau[3]

[1] Télécom SudParis
[2] Valeo
[3] Gandi

Network protocol implementations ("stacks") are pervasive in our modern systems. Indeed, we rely on various protocols on a daily basis, the most proeminent thereof being TLS. One of the problems with network stacks is that they can exhibit wrong transitions in their state machines, which can lead to security issues. This is especially true when protocols are specified using natural language, which encourages ambiguities and discrepancies between implementations.

In this paper, we present a black-box approach to study real-world implementations and their internal state machines. Our methodology relies on Active Automata Learning to infer the behavior of a given stack. Using this approach, we were able to reproduce existing bugs and uncover new vulnerabilities, including authentication bypasses in TLS and SSH.

1 Introduction

Information systems heavily rely on complex network protocols to work. Many such protocols are specified in documents written in natural language, such as RFCs published by the IETF. However, these specifications lack formalism and may contain ambiguities or be incomplete, which can lead to interesting bugs in various implementations. Thus, in this paper, we focus on the analysis of protocol implementations, and not on the specifications. The analysis of protocol implementations has been explored at length with the TLS protocol at least since 2014 and 2015, when major flaws were uncovered in various implementations.

Examples of such flaws include EarlyCCS (CVE-2014-0224), a vulnerability allowing a man-in-the middle attacker between a vulnerable OpenSSL client and a vulnerable OpenSSL server to force the use of a fixed, known, session secret. Several attacks against the state machine automata used in various SSL/TLS implementations were later published [7], including the infamous FREAK (CVE-2015-0204) (Factoring RSA Export Keys) vulnerability, where a man-in-the-middle attacker could trick a

vulnerable client into accepting a small RSA key (the so-called EXPORT feature, where a temporary 512-bit RSA key is used instead of the longer one) by sending an unexpected message.

In 2018, an interesting bug was found in the `libssh` [4] server implementation. During the user authentication phase, where a client is supposed to request user authentication, e.g. by sending its login and password, it was found that the client could simply send a `UserAuthSuccess` message (which should never be sent by the client) to convince the vulnerable server to accept the authentication.

These examples show that many stacks do not properly handle the state machine of the implemented protocols, which can lead to various consequences, including authentication bypasses, information leakage or denial of service.

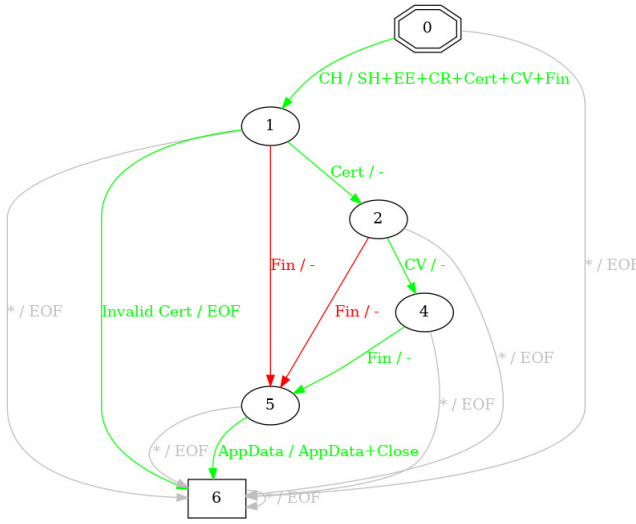

Fig. 1. CVE-2022-25640: a client authentication bypass in wolfSSL v5.1.0. Green paths represent the expected message paths, whereas the red transitions consist in shortcuts allowing an attacker to terminate the handshake without proving its identity to the server. CH is short for `ClientHello`, SH for `ServerHello`, EE for `EncryptedExtensions`, CR for `CertificateRequest`, Cert for `Certificate`, CV for `CertificateVerify`, and Fin for `Finished`.

Since 2019, we have been developing at Télécom SudParis (in the "Sécurité et Confiance Numérique" team) a rigorous and efficient methodology to learn the internal behavior of TLS and SSH network stacks using Active Automata Learning. This methodology allowed us to reproduce or

uncover interesting flaws. Figure 1 presents CVE-2022-25640 on wolfSSL, an example where client authentication can simply be bypassed by omitting the `CertificateVerify` (`CV` on the figure) message, which normally contains a signature proving the client's identity; by sending an early `Finished` in state 2, a malicious client can get to state 5 (the end of the handshake) without knowing the client's private key. In recent years, we successfully applied our approach to different protocols:

— TLS 1.2 [34] and 1.3 [33], which is a classical benchmark;
— OPC-UA [1], a protocol used in industrial control systems;
— SSH [24–26], the well-known secure shellprotocol.

We already obtained results that were published in academic conferences [31, 37] and that led to critical CVEs on different protocols (CVE-2022-25638 and CVE-2022-25640 for TLS, CVE-2023-26150 for OPC-UA and CVE-2025-14942 for SSH). We believe Active Automata Learning (AAL) is a powerful approach that should be improved and generalized as much as possible, to find bugs, but also to help developers improve their software or to explore fingerprinting opportunities.

After presenting the AAL framework and some background on the studied protocols, we will present an end-to-end automated pipeline to verify protocols. Then, we will present some of our results: the analysis of TLS and SSH stacks, and a proposal to help fingerprint implementations. Finally, we will discuss the related work and the future work.

2 The Active Automata Learning Framework

One of the few sources of information available when studying blackbox network protocol implementations are network traces. Network traces are a collection of intertwined messages emitted and received by a peer (e.g. wireshark traces). But, network traces contain many redundant patterns which makes analysis inefficient to look for bugs or vulnerabilities.

Can we propose a compact logical structure to represent all network traces? This is the challenge that automata learning tries to solve with two different approaches. In *passive learning*, it is not possible to exchange messages, thus, the logical structure must be extracted entirely from previous capture of network traces. On the contrary, *active learning* interacts with the target to dynamically append new traces to its knowledge base while building the logical structure. Active learning is more interesting since it can fill the blank in network traces by playing missing message sequences.

In this paper, we only discuss active learning. Yet, this approach requires to tackle new issues:

— **Can we discover a faithful logical structure while avoiding an infinite growth of network traces?**
— **When should the collection of network traces grow? stop to grow?**
— **What guarantees do we have on the final logical structure?**

2.1 Intuitive insight in AAL

In a seminal paper [6], Dana Angluin proposed the *Minimally Adequate Teacher (MAT) framework* to extract a model by exchanging messages with a System Under Learning (SUL). The MAT framework is an analogy

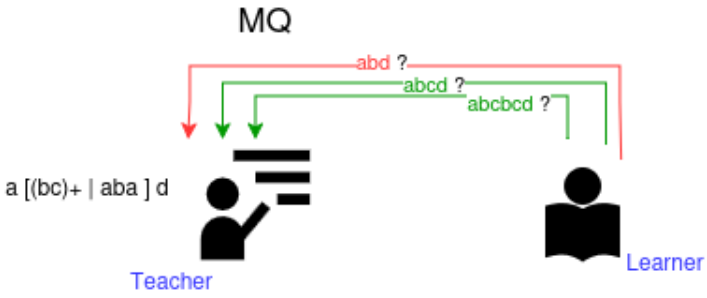

Fig. 2. MAT framework: Membership queries

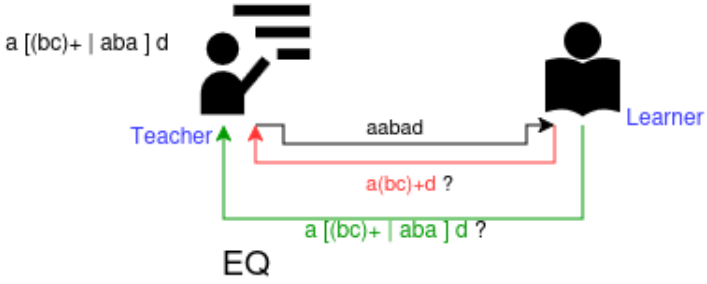

Fig. 3. MAT framework: Equivalence queries

of a classroom where a student (the learner) tries to learn a language by example. The student does not know the language and can only ask very simple questions like "Is the following sentence correct in your language?".

As an example, Figure 2 and 3 represent a scenario where a learner tries to learn the following regular expression $a[(bc) + |aba]d$ which is only known by the teacher. In Figure 2, red arrows represent words sent by the learner which are not in the language. Green arrows represent words which are in the language.

After gathering enough information, the learner attempts to submit a hypothesis regular expression, as described in Figure 3. In this example, the hypothesis is rejected by the teacher which provides the "aabad" counter-example. The learner fixes its hypothesis and perform another round of membership queries before submitting a new hypothesis.

2.2 Algorithm principles

In Section 2.1, we have reviewed the intuition behind the MAT framework used in Active Automata Learning. In the following paragraphs, we provide additional details about the algorithm and the components used in the algorithms. The following paragraphs can be skipped, and we recommend the reader in a hurry to resume reading with the background on protocols (see Section 3).

In the same paper [6], Dana Angluin proposed the L^* algorithm to infer a deterministic finite automata (e.g. a regular expression). L^* was later extended for Mealy Machines, an equivalent automaton model which better describes the ability to send and receive messages in networking protocols. Figure 1 is an example of a Mealy Machine where transitions are labeled with an input symbol and one or multiple output symbols. In this figure, the input and output symbol is separated by a slash i.e. '/'. New algorithms [20, 21, 35, 38] have been proposed to improve the time complexity of the original L^* algorithm.

In practice, network protocols separate the teacher into two independent components: the oracle and the SUL. Thus, the framework used for network protocols is made of:

— the *learner* in charge of sending and collecting messages;
— the *system under learning* (SUL), typically a network stack;
— the *oracle* which guides the inference towards exhaustive models.

Assumption. AAL algorithms rely on the assumption that the SUL can be modeled by a finite state automaton. This means that the system does not have an infinite number of states and can be modeled by a regular language. In general, this assumption is verified for most networking protocols.

Variables. The *learner* is given a selected set of messages known as *vocabulary*. In the context of Mealy Machines (i.e. networking protocols), the vocabulary contains the *input vocabulary* and *output vocabulary*. For the sake of clarity, in TLS 1.3, the learner could use the following input vocabulary `ClientHello`, `Finished`, `ApplicationData`, `Alert`. The output vocabulary could be the following `ServerHello`, `EncryptedExtensions`, `Certificate`, `CertificateVerify`, `Finished`, `ApplicationData`, `Alert`.

State identification. The main challenge of the learner is to rely exclusively on traces of input and output messages to *identify states*. Thus, *state identification* is at the heart of AAL. Initially the learner believes that there are as many states as input words. Then, it tries to reduce this number of states by identifying *equivalent states*. Two states are equivalent if for any input message sequence, the output message of the two states is strictly identical. Here is the catch; in practice *any input message* means that it should test message of arbitrary length.

Algorithm. The learner iteratively builds *hypothesis automaton* and proposes them to the *oracle* for validation. The oracle may either provide a counter-example to refine the automaton model or validate the hypothesis to terminate the inference.

The learner builds hypotheses by sending a sequence of input messages and by collecting the associated output messages. It is common to name *input word* a sequence of input symbol which represent a path on the finite state machine. There are different approaches to build a hypothesis depending on the selected AAL algorithm. The learner needs to maintain:

1. a collection of input message sequences leading to each state, known as *prefixes*;
2. a collection of distinguishing sequence, also known as *suffixes*, to remember how to prove that states are different.

For details about the algorithm, we invite you to read the L^* paper [6].

2.3 The oracle

The oracle is the component which answers equivalence queries presented in Figure 3. As a remainder, the learner sends a hypothesis automaton to the oracle which represents its understanding of the system. Then, the learner either expects a confirmation that the hypothesis is valid, in which case the algorithm terminates, or a counter-example to help the

learner fix its hypothesis, which leads to a new hypothesis proposed by the learner in the next iteration of the algorithm.

At this stage, something may be confusing. Remember, the initial problem is to find an algorithm to extract an automaton model which represents a system. Yet, the learning algorithm requires this "oracle" component which seem to require knowing the model. **How is that helpful?** There is a subtle difference between the goal of the learning algorithm and the oracle. On the one hand, the learning algorithm seeks to **find out the model** of the system by knowing only the system vocabulary. On the other hand, the oracle only needs to check the **conformance of the hypothesis model with the system** by using the vocabulary and the hypothesis model. To sum it up, the goal of the oracle is hard, but it is simpler than the goal of the learning algorithm.

Why is it hard to implement an oracle? In theory, an oracle is supposed to always be accurate. Thus, if the hypothesis is correct, the oracle must not return a counter-example. Conversely, if it is incorrect, it cannot validate the hypothesis. There are different ways for the hypothesis automaton to be incorrect. First, the automaton may contain an **output fault**. In this scenario, the output observed on the transition of the model does not correspond to the system. Simply exploring all transitions of the system is enough to detect an output fault. Second, the automaton may contain a **next-state fault**. This occurs when the destination state of the system is different from the destination state of the hypothesis. In order to detect these faults, we need a way to distinguish any two states of the automaton. It is achieved using a specific sequence of message called *distinguishing sequences*. Third, the automaton may be **incomplete** as it may be missing some states and transitions. It is the most problematic case because proving that the hypothesis is complete requires exploring paths of infinite length. Even worse, the time-complexity to discover a missing state grows exponentially with its distance (in symbols) to a known state.

How to implement an oracle in practice? In practice, the oracle needs to be approximated. It is possible to look for counter-examples using conformance testing, random strategies (e.g. random walk), binary analysis techniques, or human knowledge. In particular, for network protocols, it is common to rely on *conformance testing algorithms* [18] such as W method [12], WP method [17] or BDist [29]. Conformance testing algorithms rely on the automaton model to compute an optimal strategy to detect faults in the hypothesis. To tackle the infinite path length problem, conformance testing algorithms rely on an upper-bound exploration

parameter to prune the exploration space. Some states may never be found, but the algorithm can terminate.

How to balance between accuracy and speed? Lowering the exploration-bound speeds up the oracle, but it may cause the oracle to miss extra states. Thus, a proper balance must be found on exploration-bound parameters. There are two types of exploration-bound parameters:

1. The upper-bound on the *number of states* of the unknown system automaton (e.g. W method [12] and WP method [17]);
2. the upper-bound on the *maximum shortest separating sequence length* which roughly represents the similarity between states (e.g. BDist [29]).

In the case of network protocols, it is hard to guess the number of states of an implementation without prior knowledge. In particular, we have observed significant difference in the number of states across implementations of the same protocol. Thus, we rely on the more practical approach offered by the BDist equivalence method. Indeed, it is observed that for most network protocol implementations, the BDist parameter is less or equal to 4 in most cases, and may very rarely reach 7.

2.4 The Mapper

In practice, what is required to actually extract the model of a real-world implementation?

AAL only requires a simple translation component named *mapper*. The role of the mapper is to translate abstract vocabulary symbol (e.g. ClientHello, ServerHello, Alert...) to concrete binary messages. Thus, for each abstract symbol, a mapper simply implements two methods: **unparse** and **parse**. The unparse method reads an abstract input symbol (e.g. TLS1.3 ClientHello) and creates a corresponding binary message. The parse method reads a binary message and tries to find a corresponding abstract output symbol. Writing the mapper is often challenging for the following reasons:

— multiple binary messages correspond to a single abstract message (e.g. length, random fields are meaningless);
— the mapper relies on implementation choices (e.g. selected ciphersuites or extensions in TLS 1.3);
— the mapper is expected to produce concrete messages even for message sequences that may have no meaning from the protocol point of view.

2.5 Advantages of AAL

Since the learner and the SUL only interact through messages, AAL describes a black-box learning technique. Thus, it is very practical to infer closed-source implementations or hardware implementations of protocols.

Moreover, AAL offers strong guarantees about the model learnt. For instance, two different states of the learnt automaton are proved **observably different**. It can be verified manually after the inference by computing the shortest distinguishing sequence (i.e. suffix) between two states. Conversely, the main limitation of the model compared to the real implementation comes from some states being considered equivalent while they are in reality different. Fortunately, this over-approximation is tackled through a parameter of the oracle which results in a trade-off between inference duration and model accuracy, as seen previously. An over-approximation leads to long inference, while an under-approximation may result in missing interesting states. In practice, we rely on the BDist method [29] which offers an equivalent and more intuitive parameter named BDist. The BDist parameter defines the maximum length of the distinguishing sequence of messages between two different states. Intuitively, if a long BDist is required to distinguish two states, it means that these states will almost always return the same output messages for any input messages. Concretely, a BDist of 3 is generally a good trade-off to find most counter-examples on the implementations we studied.

2.6 Challenges of State Machine Inference

There are also some challenges and limits to the use of AAL.

Time complexity and slow inference. The time complexity of AAL algorithm is actually measured as the number of messages sent. In L^*, the complexity is $O(kn^2m)$ where k is the size of the vocabulary, n the number of states and m the longest counter-example. Newer algorithms have improved the original L^* complexity, however, in practice, these improvements are not very meaningful. Indeed, most of the inference duration is actually spent in the oracle which requires sending exponentially many messages to improve the accuracy of the model.

Timeout. The time complexity of AAL algorithm and oracles certainly contributes to the duration of the inference. When studying network protocols, the learner is also not aware of the reply latency of the SUL. Even worst, in some protocols like TLS, a stack may reply with a variable

number of messages (as shown in Figure 1). Thus, the learner needs to systematically wait for a configurable timeout to expire. This adds significant overhead to the duration of the inference.

Oracle: the speed-accuracy trade-off. One of the remaining challenge to run the inference is to define the oracle parameter which defines the accuracy of the model. We already mentioned that increasing accuracy requires to increase the number of messages sent exponentially. Even, if most protocols require a low accuracy to infer a complete model, we found some intractable patterns. For example, some implementations will tolerate a fixed number of alerts before closing the connection, which may require to unroll a long counter-example to find this case.

Mapper and vocabulary choice. A practical challenge comes from the implementation of the mapper and the selection of vocabulary. Since the size of the vocabulary contributes to the duration of the inference, meaningful messages must be selected. Writing a mapper for protocols which negotiate encryption often leads to consider undefined choices. For example, how can a TLS 1.3 client encrypt a `Finished` message if it has not derived any key? Should the message be sent in cleartext?

3 Background on protocols

3.1 TLS

TLS (Transport Layer Security), formerly known as SSL (Secure Sockets Layer) is a security protocol whose main security goal is to provide a secure channel between a client and a server, where the server is authenticated (the client can also be authenticated at the TLS level, but it is less common). The latest version of TLS is TLS 1.3, published in 2018 [33].

Figure 4 shows the expected flow for a TLS 1.3 connection, with a very limited set of messages (it is always possible to consider several variants of the same messages, or to include protocol extensions). Plain arrows represent cleartext messages and dotted lines represent encrypted messages. A client initiates the TLS handshake by sending `ClientHello` which contains supported cipher suites and it receives a `ServerHello` message in cleartext which contains the cipher suite for the handshake. In most cases, the `Hello` messages also contain a key exchange allowing the parties to derive a common secret. The `EncryptedExtension` message, introduced in TLS 1.3, supports sending encrypted protocol extensions (e.g. to choose

the application-layer protocol). The server sends its certificate in the `Certificate` message and proves it owns the private key by sending a `CertificateVerify` message containing a signature. The client can authenticate the server with the certificate authority (CA).

The `Finished` message switches from the handshake encryption key to a new key used for application data. In TLS 1.3, there exists another legitimate expected flow to support certificate-based client authentication where the client also sends `Certificate` and `CertificateVerify` messages before the `Finished` message.

3.2 SSH

SSH (Secure Shell) is a network protocol very commonly used for remote administration. It allows admins to securely connect to machines and appliances. There are many implementations of SSH, such as OpenSSH [2], which can be seen as the reference implementation, or wolfSSH [5], an open source project aiming at embedded devices. The normal flow for an SSH connection consists in chaining three different stages:

— the *Transport Layer* [26], which establishes a secure channel with encryption and integrity protection, and where the server is authenticated by the client;
— the *User Authentication Layer* [24], which consists, as its name suggest in authenticating the client to the server, usually using a password or a public key mechanism;
— the *Connection Layer* [25], which is used by application to open data channels to execute commands or establish network redirections.

Figure 5 represents the most common flow of messages between an SSH client and an SSH server. The SSH handshake starts with an exchange of banners with ASCII text to identify the networking stack. The handshake proceeds with the negotiation of the cryptographic algorithm to use for the key exchange and the key exchange itself.

The authentication phase begins by the client requesting the authentication service. In the normal flow, the server accepts to run the authentication service and the client proceeds with the authentication request. Each request defines the desired authentication method (typically password or public key) as a parameter.

Once authenticated, the client can open channels in the Connection Layer [4] and use `ChannelData` messages to run commands on the server.

[4] A client may actually open an arbitrarily number of connections, which would violate the assumption that SSH can be described by a finite state automaton. To avoid this situation when we consider the Connection Layer, we limit the number of channels.

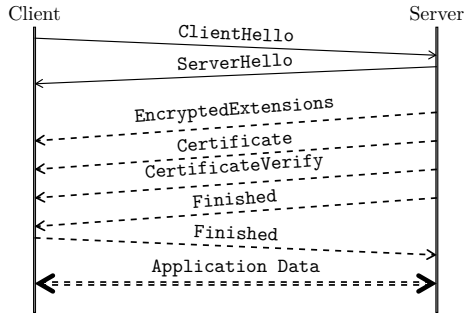

Fig. 4. Sequence diagram for one of TLS 1.3 expected flow

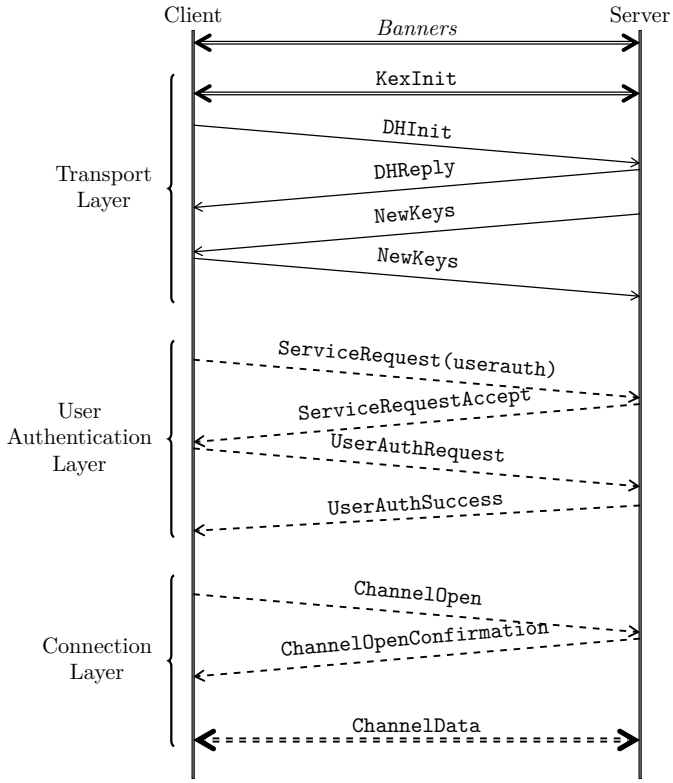

Fig. 5. Sequence diagram for SSH expected flow

SSH is clearly an order of magnitude more complex than TLS, with many more messages and, accordingly, many more states in the resulting state machines.

4 Automated pipeline

After reviewing the model extraction in Section 2, this section presents an automated pipeline for security analysis of a network protocol implementation.

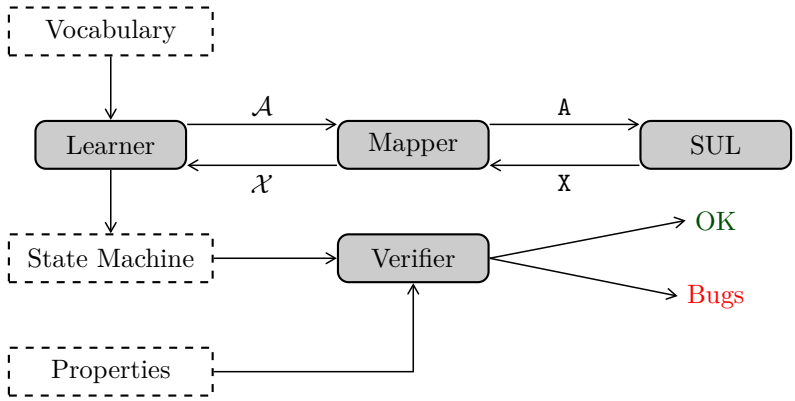

Fig. 6. Automated pipeline for security analysis

Figure 6 represents the entire pipeline for a given protocol with

— the *system under learning* (SUL) i.e. a network implementation of the protocol;

— the *AAL components* made of

1. a *protocol Mapper* to translate abstract symbols $\mathcal{A}$,$\mathcal{X}$ of the protocol to/from binary messages `A`,`X`,
2. a *Learner* to run the AAL algorithm given a vocabulary (i.e. $\mathcal{A}$,$\mathcal{X}$),

— the *Verifier* to identify bug or vulnerabilities in the implementation.

The pipeline can be entirely automated for a protocol, i.e. after selecting a protocol (e.g. TLS 1.3) it can be used on any implementation of the protocol. However, supporting a new protocol for the pipeline still requires manual work in particular to implement the *protocol mapper* and to write the *protocol property* rules for the verifier.

The pipeline begins with the user selecting a set of symbols from the mapper vocabulary. Then, the learner runs the AAL algorithm to extract the Mealy Machine which describes the behavior of the SUL. The Mealy Machine is stored in an intermediary file such as graphviz `.dot` file. After adapting the intermediary file to the verifier syntax, the verifier uses predefined protocol rules to try to identify vulnerabilities.

4.1 Server and Client Inference

Server Inference. Infering a server is straightforward: first, we start the server we want to study; then, for each message sequence the learner needs to run, the mapper opens a connection with the server and handles message sending and receiving.

By doing so, we actually make the implicit assumption that connections with the server are independant from one another, which is true in general. This is necessary since we expect the server behavior to be deterministic: the same message sequence should always produce the same answers. However, in some cases, the assumption is false and we need to restart the server between sequences, which can have a significant overhead. [5]

Client Inference. To infer a client, we need to instantiate a new client for each message sequence that we need to run. To this aim, the mapper has a listening socket ready, to act as a server, and it triggers a client connection from the target using a script. By construction, this means we get a fresh client for each new message sequence.

4.2 Verifier

There exists various programs designed to verify properties on state machines. We review the use of model checkers to verify security properties before presenting new alternative approaches.

Model checkers. Interestingly, verifying a state machine using a model checker may also take some time for very large state space (e.g. 10^{20} states and beyond [11]). The need to support a very large number of states is usually caused by the use of variables on transitions for extended automaton models (e.g. EFSM). Fortunately, the verification of security properties on Mealy Machine models of a networking protocol is fast.

[5] As OPC-UA servers are supposed to only keep a limited number of active sessions, they usually exhibit such a behavior; since the inference may sometimes open session and not close them properly, we may hit the limitation because of past sequences.

Model checkers such as Spin [19], nuSMV [13] can be used to write specifications in temporal logic (e.g. LTL, CTL). They also use a description of the model, usually with their own syntax and verify specification with the model. One of the interest of these model checkers is their ability to provide a counter-example whenever the specification does not comply with the model.

Let us give an example with CVE-2022-25640 presented in Figure 1 of Section 1. We propose to verify that the handshake always finishes successful if the certificate is exchanged and correct.

LTL formula. Let us start by defining the meaning of a successful handshake in TLS. A handshake is successful when both peers have successfully sent and received a "Finished" message. In our TLS mapper, an unsuccessful exchange of "Finished" message would result in the output symbol "EOF". A successful handshake corresponds to input="Fin" $\wedge$ output="Empty".

A careful reader will notice that logical properties depend on the implementation of the mapper. Thus, the syntax of messages must be equivalent between logical properties and the model (e.g. Finished $\neq$ Fin). Moreover, the meaning of messages is also important. In Figure 1, alert messages are merged with "End-Of-File" messages to simplify the automaton. This may break logical properties which detect a successful handshake with input="Fin" $\wedge \neg$output="Alert(decrypt_error)".

There are two main classes of temporal logic: *linear-time* (e.g. LTL) and *branching-time* (e.g. CTL). In order to verify safety properties for TLS implementations, the property must hold for any path of the automata. Thus, we specify our property in linear temporal logic (LTL) which is universal, i.e. the property holds for all paths.

LTL properties are commonly expressed in future modality describing how a property holds from present to future. However, it is often simpler and more concise to write safety properties in past modality. Indeed, past modality in LTL helps you write properties by reasoning **from effect to cause**. It is convenient to express safety patterns such as: "the sequence has been observed".

Let us consider the desirable effect $P1$ which is a successfully finished handshake and the cause $P2$ which is the validation of a client certificate. The property can be represented as $G(P1 \rightarrow Y\,P2)$ where:

— G (Globally) means that the property holds globally for all branches;
— Y (Yesterday) means that the property holds in the previous past instant.

Listing 1: nuSMV code for CVE-2022-25640

```
MODULE main

VAR
  state: {s0, s1, s2, s4, s5, s6};
  input: {"CH", "Cert", "CV", "Fin", "Invalid_Cert", "AppData" };
  output: {"SH+EE+CR+Cert+CV+Fin", "AppData+Close", "EOF", "Empty"};

ASSIGN
  init(state) := s0;
  next(state) :=
    case
      state=s0 & input="CH": s1;
      state=s0 & input!="CH": s6;

      state=s1 & input="Cert": s2;
      state=s1 & input="Fin": s5;
      state=s1 & input!="Cert" & input!="Fin": s6;

      state=s2 & input="CV": s4;
      state=s2 & input="Fin": s5;
      state=s2 & input!="CV" & input!="Fin": s6;

      state=s4 & input="Fin": s5;
      state=s4 & input!="Fin": s6;

      state=s5: s6;
      state=s6: s6;
    esac;

  next(output) :=
    case
      state=s0 & input="CH": "SH+EE+CR+Cert+CV+Fin";
      state=s0 & input!="CH": "EOF";

      state=s1 & input="Cert": "Empty";
      state=s1 & input="Fin": "Empty";
      state=s1 & input!="Cert" & input!="Fin": "EOF";

      state=s2 & input="CV": "Empty";
      state=s2 & input="Fin": "Empty";
      state=s2 & input!="CV" & input!= "Fin": "EOF";

      state=s4 & input="Fin": "Empty";
      state=s4 & input!="Fin": "EOF";

      state=s5 & input="AppData": "AppData+Close";
      state=s5 & input!="AppData": "EOF";

      state=s6: "EOF";
    esac;

LTLSPEC G( (input="Fin" & X(output="Empty")) ->
           Y ( (input="CV" & X(output="Empty")) &
            Y (input="Cert" & X(output="Empty")) ) )
```

Then, $P1 = (\text{input="Fin"} \wedge X\, \text{output="Empty"})$ means that the transition has a Finished input and Empty output. Because of our encoding of the Mealy Machine in nuSMV, $X(output = "Empty")$ means the output is triggered by the input at the same step.

As a conclusion, we have the following LTL formula with past temporal operator ending Listing 1. When we run it, nuSMV reports a single counter-example: $CH \cdot Fin \cdot CH$.

Mealy Verifier. Model checkers are powerful tools, but they require some effort to write proper logical formulas. Internally, this led to the implementation of the Mealy Verifier [39] focused on verifying protocol properties on Mealy machines.

The Mealy Verifier takes as input a `.dot` file which describe the state transition diagram and a set of properties.

Model checkers mostly try to refute a property by exhibiting the first counter-example. But they stop after finding this first counter-example which is not really practical for exhaustive analysis. In the example above, the model checker only highlighted one of the two red transitions. Thus, the Mealy Verifier is designed to report an exhaustive list of counter-examples.

For instance, we implement the same LTL formula using the syntax of the Mealy verifier in Listing 2. The rule is named *auth* and it is in a CT block. CT blocks are used to encode a *conditional property* i.e. an event is reached after some prerequisites are respected. The two first line of the block correspond to premise while the last line $Fin/-$ is an observable action. Every premise is made of an event and a counter-event separated by |

Listing 2: Mealy verifier code for CVE-2022-25640

```
CT:auth
    Cert/- | I/I
    CV/- | I/I
    Fin/-
:CT
```

Contrarily to nuSMV, the mealy verifier finds all counter-examples which are represented by the two Mealy Machines represented in Figure 7. The two counter-examples correspond to the path obtained by sending the following two input sequences: $CH \cdot Fin$ and $CH \cdot Cert \cdot Fin$.

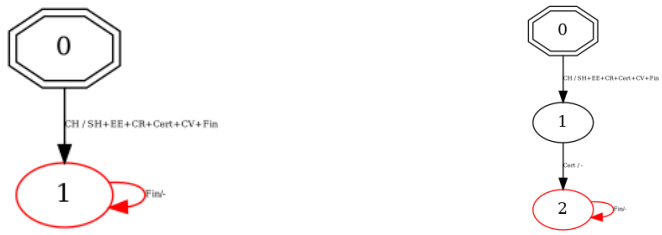

Fig. 7. Output of the Mealy verifier

5 Results

In this section, we present the results of the pipeline presented in Section 4 for the TLS 1.2, TLS 1.3 and SSH protocols. We first describe our experiments in Table 1: the used tools, the list of studied stacks and the used vocabulary for learning TLS and SSH state machines.

5.1 Analysis of TLS Stacks

We implemented a protocol mapper for TLS 1.2 and TLS 1.3, `pylstar-tls` [30]. It was developed in Python and based on scapy [3].

Applying Active Automata Learning to TLS implementations, we were able to infer the state machines of many different versions of various open source TLS implementations in different scenarios (different TLS versions, client and server inference). To rigorously and efficiently produce the state machines for around 500 different stacks, we improved the inference process using optimizations providing a 25-speedup in the best cases.

This study led to reproducing known vulnerabilities and to uncover new flaws such as CVE-2022-25638 and CVE-2022-25640 on wolfSSL. The results can be found in our paper published at ESORICS in 2022 [31].

5.2 Reproduced Vulnerabilities on SSH Stacks

In 2018, it was discovered that `libssh` before versions 0.7.6 and 0.8.4 suffered from authentication bypasses in state machines. Using AAL, we reproduce the vulnerability, CVE-2018-10933. To this aim, we infer the state machines for `libssh` server using only the Authentication and Connection Layer messages.

Figure 8 shows the inferred state machine for the vulnerable server. Even if the automaton does not directly show the problem from the CVE (which needs some vocabulary refinement), we can already witness that

	TLS	**SSH**
Learner	`pylstar` [9]	`rlstar`* [28]
Mapper	`pylstar-tls`* [30] (based on scapy [3])	`sshmapper`* [23]
Vocabulary (C->S)	ClientHello Certificate EmptyCertificate CertificateVerify ChangeCipherSpec Finished ApplicationData Alert	Kexinit DHinit Newkeys Disconnect ServiceRequest UserAuth_Password UserAuth_RSA ChannelOpen ChannelExec ChannelData ChannelClose ChannelEof
Vocabulary (S->C)	ServerHello EncryptedExtension CertificateRquest Certificate EmptyCertificate CertificateVerify ChangeCipherSpec Finished ApplicationData Alert	Kexinit DHReply Newkeys Disconnect ServiceAccept UserAuthSuccess UserAuthFailure ExtensionInfo ChannelOpenConfirmation ChannelSuccess ChannelFailure
Stacks	OpenSSL GnuTLS NSS wolfSSL erlang/OTP matrixssl	OpenSSH libssh wolfSSH

Table 1. Description of our experiments. The table describes the tools used in our experiments, but also the vocabulary we selected and the stacks we studied. Tools with a * were developed by our team and are available as open source software. For several messages, it is worth noting our mappers could generate different variants, e.g. for `Certificate` and `CertificateVerify` for TLS, and for `UserAuth_Password` and `UserAuth_RSA` for SSH.

the server accepted to open a channel (which is an operation from the Connection Layer) whereas the client never *received* a `UserAuthSuccess` message. This path is triggered by *sending* the `UserAuthSuccess`, which is normally only sent by the server, instead of the authentication request.

We also reproduced CVE-2024-2873, a known wolfSSH vulnerability presented in Figure 9, which affected version 1.4.16 and earlier. An attacker can successfully open a channel without authenticating to the server. However, to benefit from the connection layer services, the attacker should present, before or after the `ChannelOpenSession` message, an arbitrary authentication request message (`UserAuth_Password_NOK` and `UserAuth_RSA_NOK`) containing a valid username.

When the attacker presents such a message (which does not contain valid credentials), the server properly rejects its authentication request message by sending `UserAuthFailure`, meaning that the server seems to correctly reject an invalid authentication request message. However, if the attacker ignores the error message, and sends an `ChannelOpen`, then the server accepts to open a session channel and considers that the client is authenticated. This corresponds to the red path via states 0, 6 and 3, where the latter can also be reached using a valid authentication following green transitions via states 0, 1 and 3.

5.3 Focus on CVE-2025-14942 on wolfSSH

During our analysis of the inferred state machines, we also found an unexpected flow in the state machine of wolfSSH client. Our work indeed shows that it is possible to send User Authentication Layer (SSH stage 2) messages before the end of the Transport Layer (SSH stage 1). This can lead to interesting behavior for a network attacker, i.e. an attacker able to intercept the connection and answer to a vulnerable client instead of the legitimate server. Such an attacker is the very reason SSH was designed to detect and block.

This deviation from the specification concretely leads to a server authentication bypass, since the messages where the server is supposed to authenticate are completely skipped. We developed different attack scenarios that an attacker can run that we describe in details.

Password, Please? (Password Leakage). One of the consequences of this shortcut in the state machine is that a network attacker could trick the vulnerable client into sending its password.

Indeed, when the attacker sends an early `UserAuthFailure` message, as shown in Figure 10 (on the left), the client directly jumps to the

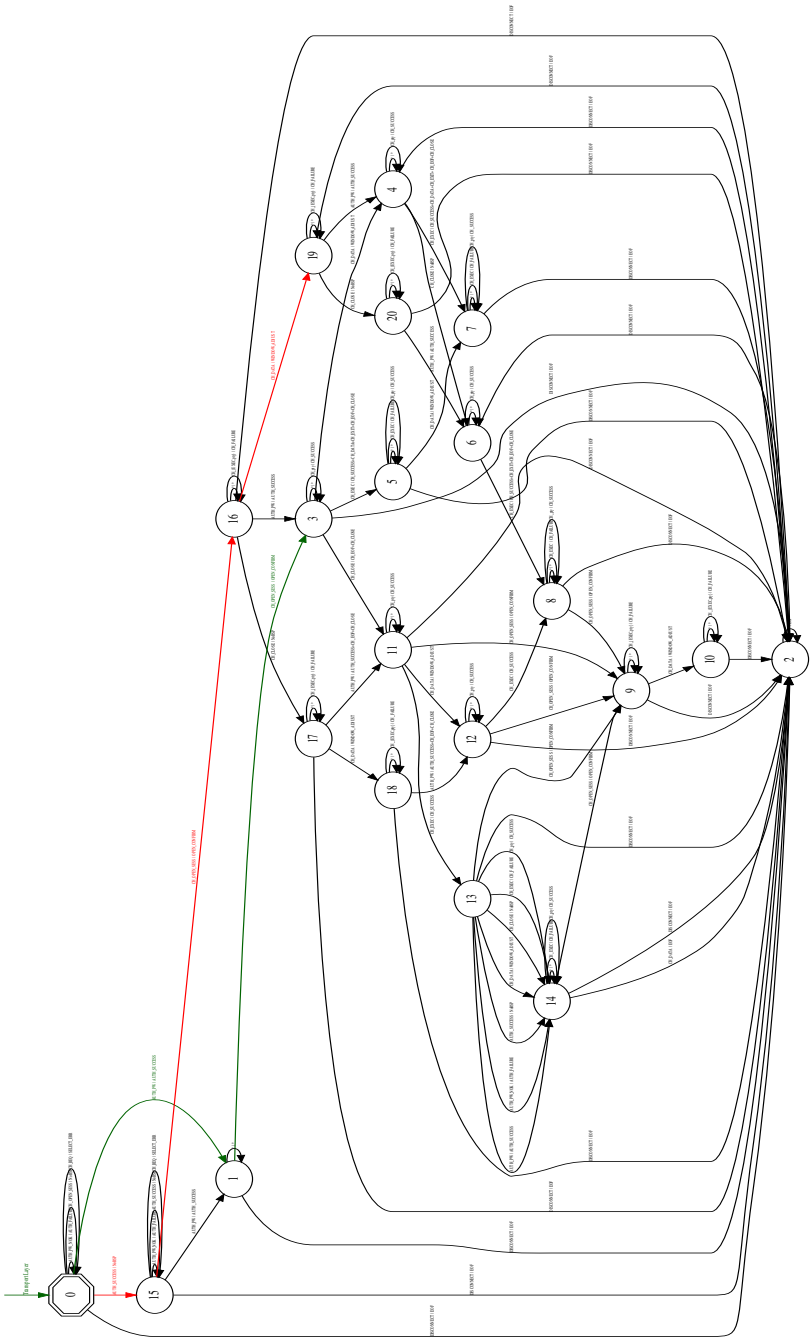

Fig. 8. CVE-2018-10933, a server authentication bypass in `libssh`.

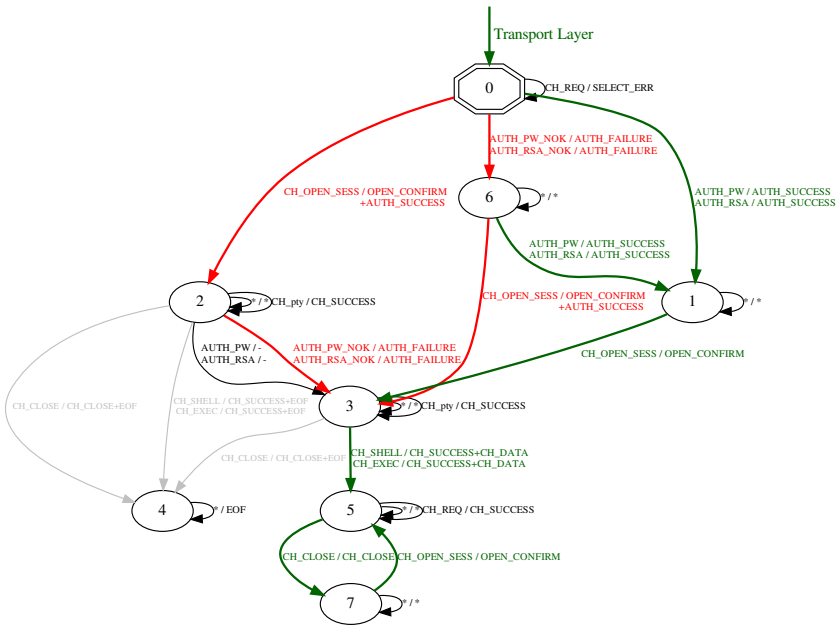

Fig. 9. CVE-2024-2873, a server authentication bypass in wolfSSH.

user authentication part of the protocol, considering it is communicating with a valid server, and happily transmits its credentials to the attacker. Specifically, the vulnerable client leaks the user's password before the server authentication, i.e. before receiving the `DHReply` message (where the server should have authenticated itself).

Obviously, this is a critical vulnerability since the password would then allow the attacker to impersonate the user everywhere the stolen password is valid.

How can I Help You Today? (Session Hijack). Instead of sending a `UserAuthFailure` message, the attacker could send a `UserAuthSuccess` message, letting the client go through and pursue the connection without any authentication. This allows the attacker to impersonate the server for the Connection Layer, which makes it possible to observe everything the client has to say to the server.

As shown on the right side of Figure 10, to lead the client to actually open a channel and send data, the attacker has to send well chosen

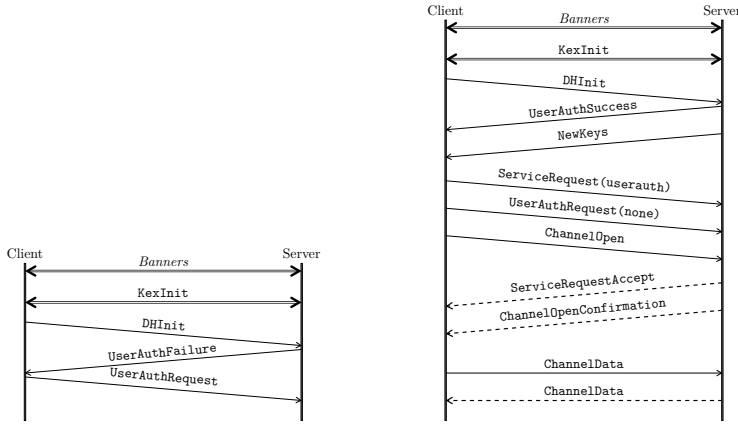

Fig. 10. Sequence diagrams for CVE-2025-14942. On the left, this is the behavior of the vulnerable client facing an early `UserAuthFailure` message with the password mechanism. On the right, the vulnerable client is sent an early `UserAuthSuccess` message, along with some well chosen messages.

messages such as `NewKeys` or `ServiceRequestAccept`, but these do not require the attacker to know any secrets the server would normally hold.

Combining this attack with the previous one, it would allow the attacker to become a Man-in-the-Middle for the SSH communication between a vulnerable client and a legitimate server, and to get access in cleartext to the whole communication.

Again, this is a critical issue which completely breaks the security guarantees we normally expect from the SSH protocol.

Can I have an Autograph? (Signature Static Forgery). A variant of the first attack consists in using a different authentication method. By forcing the client to use the public key method (using the dedicated field in the `UserAuthFailure` message), we can trick the client to send us message including a valid signature using their private key. This signature should normally cover data included in the Transport Layer, but it is replaced by an empty context here, because of the shortcut.

Getting such a signature forgery could be considered only a theoretical issue, especially since the context used for the signature is empty, and could normally not be reused in a real connection to impersonate the client. However, we found additional flaws in older versions of SSH server implementations where such a signature could be further reused to authenticate as the legitimate client. This line of attack has been closed by vulnerable implementations some time ago, but it is still worth mentioning

a client using only the public key authentication method (and not the password one), could still be at risk in some situations.

This vulnerability (and the corresponding scenarios) can be found in all versions of wolfSSH before v1.4.22, which was released on January 6th 2026. It has been reported to the editor following the principle of responsible disclosure, and we have been working with wolfSSL Inc. to check the proposed fix. The vulnerability is now identified as CVE-2025-14942 and has been rated 9.4 (out of 10) on the CVSS scale.

5.4 Fingerprinting

In 2011, Shu and Lee [36] introduced an active fingerprinting technique for network protocols based on finite state machines. Their approach aims to compute distinguishing sequences, which is a set of sequences allowing to distinguish all candidate state machines for the fingerprinting.

Given a collection of candidate state machines, the observed differences typically arise from implementation-specific error handling behaviors. For example, implementations may emit different alert messages in response to the same invalid input, or may accept unexpected messages and silently ignore them.

Using the method proposed by Shu and Lee [36], we compute a set of input message sequences that separates the inferred protocol stacks. The fingerprint of each stack is then defined as the sequence of responses it produces in response to these distinguishing sequences.

To minimize the number of distinguishing sequences, we eliminate any sequence that is a prefix of another. For instance, if $\mathcal{A}, \mathcal{B}$ and $\mathcal{A}, \mathcal{B}, \mathcal{C}$ are both in the distinguishing sequences, then we only consider $\mathcal{A}, \mathcal{B}, \mathcal{C}$ (i.e. we remove $\mathcal{A}, \mathcal{B}$ from the distinguishing sequences).

Beyond revealing subtle differences in the internal behavior of TLS and SSH implementations, protocol fingerprinting can help an attacker in identifying, with only a few selected message sequences, the specific versions of a protocol stack running on a target. This information can then be correlated with publicly known vulnerabilities (CVEs). An attacker can selectively target a system with exploits that are known to be effective against that exact implementation, significantly increasing the likelihood of successfully compromising the network.

Application to TLS 1.3 Servers It is worth noting that the following result have been already published in an academic conference [31]. To illustrate our state-machine-based fingerprinting approach, Table 2

summarizes the equivalence classes we identify for a simplified TLS 1.3 scenario without client authentication. These classes group together implementations that exhibit identical state machine behavior in response to the distinguishing input sequences. This classification highlights meaningful differences in protocol handling across TLS stacks.

When we run our experiments, separating these 13 classes only requires sending 7 distinguishing sequences:

— `ClientHello`
— `ClientHello`, `Certificate`
— `ClientHello`, `ApplicationData`
— `ClientHello`, `Finished`, `Alert(NoRenegotiation)`
— `ClientHello`, `Finished`, `Alert(CloseNotify)`
— `ClientHello`, `EmptyCertificate`, `CertificateVerify`
— `ClientHello`, `EmptyCertificate`, `InvalidCertificateVerify`

Stack	Versions	N	High-severity CVEs affecting the servers
erlang	24.0.3 - 24.2.1	9	*No high-severity CVE referenced*
GnuTLS	3.6.16 - 3.7.2	4	*2021-20231 2021-20232*
matrixssl	4.0.0 - 4.1.0	4	*2019-10914* 2019-13470
	4.2.1 - 4.3.0	6	*No high-severity CVE referenced*
NSS	3.39 - 3.40	4	2019-17006 2019-17007 2020-12403 2020-25648 2021-43527
	3.41 - 3.78	4	*2019-17006 2019-17007 2020-12403 2020-25648 2021-43527*
OpenSSL	1.1.1a - 1.1.1p	4	*2020-1967 2020-1971 2021-3449 2021-3711 2022-0778*
	3.0.0 - 3.0.4	4	*2022-0778 2022-1473* 2022-1292
wolfSSL	3.15.5 - 4.0.0	7	2019-11873 and all the ones in the next row
	4.1.0 - 4.6.0	7	*2019-15651 2019-16748 2019-18840* 2021-38597 2022-25640
	4.7.0 - 4.8.1	7	*2021-38597* 2022-25640
	5.0.0 - 5.1.1	7	2022-23408 2022-25640
	5.2.0	6	*No high-severity CVE referenced*

Table 2. TLS 1.3 server stacks grouped by state machine. N is the number of states. CVEs in italic only affect part of the equivalence class.

State-machine-based fingerprinting is inherently robust, since it relies on how protocol stacks, such as TLS and SSH, fundamentally process protocol messages rather than on easily configurable options such as supported ciphersuites.

Nevertheless, in TLS, some configuration parameters can alter the state machine. While we already taken into account features such as server-requested client authentication and TLS 1.3 middlebox compatibility, other

mechanisms, including renegotiation, may still affect accuracy and are left for future work.

6 Related Work

Active Automata Learning. AAL was first introduced to verify implementations of electronic devices (bank cards, handled readers, passports). In 2015, De Ruiter and Poll [14] inferred TLS state machines for several TLS server implementations using AAL, uncovering a range of vulnerabilities. Our work extends their findings, as their analysis focused exclusively on server-side state machines and was conducted prior to the introduction of TLS 1.3.

Active learning techniques have also been applied to a variety of other protocols and contexts. In his thesis, Bossert developed pylstar and used it to reverse-engineer communication protocols between malware and its command-and-control servers [10]. He further analyzed the behavior of HTTP/2 clients to enable robust fingerprinting [8].

Model learning was applied to SSH implementations by Fiterau-Brostean et al. in 2017 [16] Their work exhibited non-conformance issues in three SSH implementations but no vulnerabilities. Since their mapper is not available and no there exists no details about vocabulary choices, it is not possible to reuse the model checking rules which they proposed.

Additionally, in 2019, de Rasool et al. [32] used learnlib to study Google's QUIC protocol and Fiterau-Brostean et al. [15] applied AAL to analyze DTLS implementations in 2020.

State-machine-based Protocol Stack Fingerprinting. AAL has been recently applied to stack fingerprinting by inferring the SUL's state machine and then deriving protocol fingerprints using formal methods such as the approach proposed by Shu et al. [36], which computes distinguishing sequences from FSM or PEFSM models to differentiate implementations.

This combination of AAL and fingerprinting was successfully applied by Jansen Erwin to multiple TLS server implementations [22]. A similar approach was later used by Pferscher et al. [27] to fingerprint Bluetooth Low Energy devices via active learning, although their study covered fewer devices and relied on manually derived fingerprints.

7 Future Work

As discussed in Section 2.6, the duration of the inference can be very cumbersome for some implementations which have many states. Moreover,

it is sometimes desirable to extend the vocabulary to cover more scenarios or to gain more details (e.g. fingerprinting), but this makes inference longer. As a result, we are currently working on three different approaches to reduce the duration of the inference. In the **adaptive learning approach**, we rely on existing protocol knowledge through traces or user knowledge to guide the inference. In the **greybox oracle approach**, we assume that the oracle has access to the SUL binary in order to speed up counter-example lookup. In the **system optimization approach**, we instrument SUL system calls to skip learner timeouts.

8 Conclusion

This paper gives an overview of active automata learning to analyze the security of the finite state machine of stateful network protocols. We reviewed how AAL can be used, even on closed-source implementations, to extract an automaton model of an implementation. By maintaining observable differences between states, AAL offers deterministic guarantees and makes it easy to verify the soundness of the extracted model. We presented some vulnerabilities on TLS and SSH network protocols to illustrate that logical vulnerabilities on the state machine still exist in network stacks. We introduced our ongoing work and proposal to build automated pipelines for each network protocol to verify the soundness of implementation and to help developers to avoid regression.

Novelty. This paper includes results from past work published at ESORICS [31] and ARES [37]. It also introduces new unpublished results on SSH networking stacks and proposes a consistent framework for end-to-end automated testing of stateful protocols.

9 Acknowledgements

This project was sponsored by the ANR GASP project (ANR-19-CE39-0001), the CERES project funded by the CIEDS (Centre Interdisciplinaire des Études de Défense et de Sécurité d'IP Paris) and the GINS project, funded by the IMT Carnot Institute.

We would like to thank all the members of our team for their work on AAL. In particular, we want to thank Arthur Tran Van, who contributed the OPC-UA mapper and the Mealy Verifier during his PHD, Clément Parssegny who contributed to various project during his study and PHD, and all the past and current interns in the team: Pedro Bartolomei Pandozi,

Martin Horth, Mathieu Michel, Mohamed Mziou, Lorenzo Nadal Santa, Sébastien Naud, Van Nam Pham, Quentin Rabouin and Alexander Trifa.

References

1. Opc unified architecture. https://reference.opcfoundation.org/.
2. OpenSSH. `https://www.openssh.org/`.
3. Scapy. `https://scapy.net/`.
4. The SSH library. `https://www.libssh.org/`.
5. WolfSSH. `https://www.wolfssl.com/products/wolfssh/`.
6. Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, November 1987.
7. 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, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 535–552. IEEE Computer Society, 2015.
8. Geoges Bossert. Comparaisons et attaques sur le protocole http2 — georges bossert, 06 2016.
9. Georges Bossert. pylstar. `https://github.com/gbossert/pylstar`.
10. Georges Bossert, Frédéric Guihéry, and Guillaume Hiet. Netzob : un outil pour la rétro-conception de protocoles de communication. `https://github.com/netzob/netzob`, 06 2012.
11. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 1020 states and beyond. *Information and Computation*, 98(2):142–170, 1992.
12. T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, 1978.
13. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.
14. Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, Washington, D.C., August 2015. USENIX Association.
15. Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS implementations using protocol state fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2523–2540. USENIX Association, August 2020.
16. 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, page 142–151, New York, NY, USA, 2017. Association for Computing Machinery.

17. S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.
18. Angelo Gargantini. *4 Conformance Testing*, pages 87–111. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
19. G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
20. Falk Howar and Bernhard Steffen. Active automata learning as black-box search and lazy partition refinement. In Nils Jansen, Mariëlle Stoelinga, and Petra van den Bos, editors, *A Journey from Process Algebra via Timed Automata to Model Learning - Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*, volume 13560 of *Lecture Notes in Computer Science*, pages 321–338. Springer, 2022.
21. Malte Isberner, Falk Howar, and Bernhard Steffen. The ttt algorithm: A redundancy-free approach to active automata learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 307–322, Cham, 2014. Springer International Publishing.
22. Erwin Janssen, Frits Vaandrager, Joeri de Ruiter, and Erik Poll. Fingerprinting tls implementations using model learning. *Master's thesis*, 2021.
23. Olivier Levillain. `sshmapper`, an SSH mapper in Rust. `https://gitlab.com/gaspians/ssh-mapper`.
24. Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Authentication Protocol. RFC 4252, January 2006.
25. Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Connection Protocol. RFC 4254, January 2006.
26. Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, January 2006.
27. Andrea Pferscher and Bernhard K Aichernig. Fingerprinting and analysis of bluetooth devices with automata learning. *Formal Methods in System Design*, 61(1):35–62, 2022.
28. Yohan Pipereau. `rlstar`, an $L^\star$ implementation in Rust. `https://gitlab.com/gaspians/rlstar`.
29. Arjun Radhakrishna, Nicholas V Lewchenko, Shawn Meier, Sergio Mover, Krishna Chaitanya Sripada, Damien Zufferey, Bor-Yuh Evan Chang, and Pavol Černỳ. Droidstar: callback typestates for android classes. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1160–1170, 2018.
30. Aina Toky Rasoamanana and Olivier Levillain. pylstar-tls. `https://gitlab.com/gaspians/pylstar-tls`.
31. 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 Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *Computer Security - ESORICS 2022 - 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26-30, 2022, Proceedings, Part III*, volume 13556 of *Lecture Notes in Computer Science*, pages 637–657. Springer, 2022.
32. Abdullah Rasool, Greg Alpár, and Joeri De Ruiter. State machine inference of quic. *arXiv preprint arXiv:1903.04384*, 2019.

33. Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

34. Eric Rescorla and Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, August 2008.

35. R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, STOC '89, page 411–420, New York, NY, USA, 1989. Association for Computing Machinery.

36. Guoqiang Shu and David Lee. A formal methodology for network protocol fingerprinting. *IEEE Transactions on Parallel and Distributed Systems*, 22(11):1813–1825, 2011.

37. Arthur Tran Van, Olivier Levillain, and Herve Debar. Mealy verifier: An automated, exhaustive, and explainable methodology for analyzing state machines in protocol implementations. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, ARES '24, New York, NY, USA, 2024. Association for Computing Machinery.

38. Frits Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A New Approach for Active Automata Learning Based on Apartness. In *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part I*, page 223–243, Berlin, Heidelberg, 2022. Springer-Verlag.

39. Arthur Tran Van. Mealy verifier. `https://github.com/artfire52/Mealy-Verifier`.

A A Complete Inference Example

This section describes how to run an SSH inference with our tools.

A.1 SUL Preparation

The first step is obviously to choose and set up the target of our inference. For this appendix, we choose wolfSSH v1.4.21. We package the SSH server using `docker`:

Listing 3: `Dockerfile` to package wolfSSH 1.4.21

```
FROM debian:bookworm

RUN apt update && \
    apt install -y --no-install-recommends git perl gcc make libc6-dev dh-autoreconf
    ↪  ca-certificates && \
    apt clean

RUN git clone --depth 1 --branch v5.4.0-stable https://github.com/wolfSSL/wolfssl.git
RUN cd wolfssl && \
    ./autogen.sh && \
    ./configure --enable-ssh --enable-keygen --enable-opensslall && \
    make -j && \
    make install && \
    ldconfig

RUN git clone --depth 1 --branch v1.4.21-stable https://github.com/wolfSSL/wolfssh.git

RUN cd wolfssh && \
    ./autogen.sh && \
    ./configure --enable-all && \
    sed -i 's/^\(CFLAGS = *\)-Werror\(.*\)/\1\2/' Makefile && \
    make -j && \
    make install && \
    ldconfig

RUN mkdir /etc/ssh && touch /etc/ssh/sshd_config && openssl genrsa > /etc/ssh/sshd_rsa
RUN useradd -m -U sshd
RUN useradd -m -U user
RUN printf "very-secret\nvery-secret\n" | passwd user
```

You can then build this image and run the corresponding container. We assume for the next sections you expose the SSH server on the local 2222 TCP port.

A.2 Fetching and Running the Inference Tool

To run the inference, you now need to get our tools: the mapper and the learner. You can do this by cloning the ‘ssh-mapper‘ repository, which will include ‘rlstar‘ (the learning algorithm) in its dependencies. For this demonstration, we use the ‘v0.1‘ tag, which was pushed in April 2026, during the writing of this paper.

Finally, we run the inference on the SUL we prepared earlier. By default, the vocabulary will contain all the 19 messages handled by our tool (this is a little more than those described in Table 1, since we added

some invalid and debug messages). We direct our tool on the local port we exposed (`-e 127.0.0.1:2222`), to run a server inference (`-s server`). We set a tiny timeout (`-t .1`) since everything happens locally, and we use the BDist oracle with a parameter set to 3 (`bdist -bdist 3`).

Listing 4: Cloning 'ssh-mapper' and running the inference

```
% git clone --branch v0.1 https://gitlab.com/gaspians/ssh-mapper
% cd ssh-mapper
% cargo run --bin ssh-mapper -- -e 127.0.0.1:2222 -s server -t .1 bdist --bdist 3
[...]
["Disconnect", "Ignore", "Unimplemented", "Debug", "KexInit", "KexECDHInit", "NewKeys",
↪  "ServiceRequestInvalid", "ServiceRequestUserAuth", "ServiceRequestConnection", "ServiceAccept",
↪  "AuthRequestNone", "AuthRequestPassword", "AuthSuccess", "AuthFailurePassword", "ChannelOpen",
↪  "ChannelEOF", "ChannelClose", "ChannelSuccess"]
Selected BDist Equivalence Method with bdist=3
Start Lstar inference
[...]
```

A.3 Results

This particular inference produces two hypotheses before concluding, as shown in the following table. For each hypothesis, we describe the number of states, the BDist parameter for this hypothesis, and the time required to produce it.

Hypothesis	N States	BDist	Time
#1	12	1	295s
#2	24	2	1,550s

To actually finalize the result, the inference tools has to go through a complete oracle, which ensures that, if the real model actually has a BDist which is less than 3, we get the right result. The overall process took around 2 days and 3 hours. Overall, the time spent in the tool can be split as follows.

Step	Time spent
Build hypothesis #1	295s
Find counterexample #1	35s
Build hypothesis #2	1,220s
Validate hypothesis #2	180,912s
Total	182,462s