Mealy Verifier: An Automated, Exhaustive, and Explainable Methodology for Analyzing State Machines in Protocol Implementations

Arthur TRAN VAN
atran-van@telecom-sudparis.eu
Samovar, Télécom SudParis, Institut Polytechnique de Paris
Palaiseau, France

Olivier LEVILLAIN
olivier.levillain@telecom-sudparis.eu
Samovar, Télécom SudParis, Institut Polytechnique de Paris
Palaiseau, France

Hervé DEBAR
herve.debar@telecom-sudparis.eu
Samovar, Télécom SudParis, Institut Polytechnique de Paris
Palaiseau, France

ABSTRACT

Many network protocol specifications are long and lack clarity, which paves the way to implementation errors. Such errors have led to vulnerabilities for secure protocols such as SSH and TLS. Active automata learning, a black-box method, is an efficient method to discover discrepancies between a specification and its implementation. It consists in extracting state machines by interacting with a network stack. It can be (and has been) combined with model checking to analyze the obtained state machines. Model checking is designed for exhibiting a single model violation instead of all model violations and thus leads to a limited understanding of implementation errors. As far as we are aware, there is only one specialized exhaustive method available for analyzing the outcomes of active automata learning applied to network protocols,Fiterau-Brostean's method. We propose an alternative method, to improve the discovery of new bugs and vulnerabilities and enhance the exhaustiveness of model verification processes. In this article, we apply our method to two use cases: SSH, where we focus on the analysis of existing state machines and OPC UA, for which we present a full workflow from state machine inference to state machine analysis.

CCS CONCEPTS

• **Security and privacy** → *Logic and verification*; **Network security**; **Software and application security**.

KEYWORDS

Security Protocols, Active Automata Learning, Formal Verification, OPC UA, SSH

ACM Reference Format:
Arthur TRAN VAN, Olivier LEVILLAIN, and Hervé DEBAR. 2024. Mealy Verifier: An Automated, Exhaustive, and Explainable Methodology for Analyzing State Machines in Protocol Implementations. In *The 19th International Conference on Availability, Reliability and Security (ARES 2024), July 30–August 02, 2024, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3664476.3664506

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ARES 2024, July 30–August 02, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1718-5/24/07
https://doi.org/10.1145/3664476.3664506

1 INTRODUCTION

Network protocols are pervasive today and their security is essential for our systems to work properly. However, the need for more features and the pursuit of higher security expectations often lead to complex protocols, paving the way for vulnerabilities.

This complexity manifests itself in lengthy, intricate, and occasionally ambiguous specifications. Thus, creating or maintaining an implementation of a protocol is quite challenging. In particular, security protocols have to specify the management of cryptographic elements and the order of exchanged messages. This requires a state machine to keep track of the exchange. However, specifications mostly use natural language and lack formal state machines that could ease their reading. Thus, implementations are likely to deviate from the intended specification and to differ from one another. Moreover, even seemingly simple deviations can lead to security issues. Such vulnerabilities have been observed in secure protocols such as TLS (e.g. EarlyCSS [1]).

Analyzing discrepancies between implementations and specifications can leverage active automata learning.

This black box technique extracts a state machine from a system, which models how implementations handle the message flow.

Active automata learning has been used by many researchers. Some of them only manually inspect the outcomes state machines to analyze security issues [11, 12, 20]. However, the complexity of the state machines may render manual inspection incomplete.

Therefore, formal methods were introduced to analyze them. In particular, model checking, which is used to study the compliance between a state model and its specifications, was applied in conjunction with model learning to SSH [15] or TCP [14]. In addition, tools dedicated to security analysis such as Tamarin were used with automata learning [24, 25]. However, all of those methods are not fully compatible to work in conjunction with model learning to provide a security analysis. Thus, some wrong behaviors may not be detected.

To the best of our knowledge, there exists only one prior method dedicated to exhaustive analysis of model learning outputs related to network protocols [13]. This method primarily targets the identification of undesired behavior. The tool can verify if a given state machine exhibits a vulnerability described beforehand. However, vulnerabilities or bugs that are undiscovered or not well-documented may result in oversight. Expressing wrong behaviors complicates the discovery of new issues or vulnerabilities.

To address this, we develop a method that focuses on expected behaviors. Our method aims to uncover any deviations from said

expected behaviors. Hence, our approach eases the discovery of novel bugs.

In this paper, our contributions are the following:

- The design and implementation of the Mealy Verifier, a tool performing a complete analysis of network protocol implementation from the output of automata learning.
- The reproduction of existing results on SSH. We leverage existing active automata learning results [15].
- A complete workflow from active automata learning to analysis for OPC UA. To the best of our knowledge, this is the first automata learning performed on OPC UA. This workflow was performed on 150 different OPC UA servers (different versions of several implementations). It highlights the flexibility of our approach, establishing its applicability to different network protocols.
- The discovery of new bugs in OPC UA implementations. Three of them are new vulnerabilities (two acknowledged with a CVE number). It shows Mealy Verifier's ability to discover new bug patterns.

2 BACKGROUND

2.1 Mealy Machines

Networking protocol implementations send and accept a finite set of messages. A protocol specification describes some message sequences. Different responses to the same message are expected depending on messages already sent. For these reasons, it is common to represent networking protocol implementation as a finite state machine. Mealy machines best capture the difference between input messages (accepted messages) and output messages (send messages).

Definition 2.1. A **Mealy machine**, $\mathcal{M}$ is defined by a tuple $(I, O, Q, q_0, \delta, \lambda)$ where:

- I is a finite set called **input alphabet** and O is a finite set called **output alphabet**.
- Q is a finite set (of **states**) and $q_0 \in Q$ is the **initial state**.
- $\lambda : Q \times I \rightarrow O$ is the **output function**.
- $\delta : Q \times I \rightarrow Q$ is the **transition function**.

In the state diagram of a Mealy machine, an edge from a state q with input $i \in I$ is labeled with $i/\lambda(q, i)$.

λ is extended to sequences of inputs by defining $\lambda(q, \epsilon) = \epsilon$ for $q \in Q$, and for $w \in I^*$, $i \in I$, and $q \in Q$, $\lambda(q, iw) = \lambda(q, i)\lambda(\delta(q, i), w)$. I^* is the set of words based on I, and ϵ represents the empty letter. The behavior of a Mealy machine $\mathcal{M}$ can be defined with the function $A_{\mathcal{M}} : I^* \rightarrow O^*$ where $\forall w \in I^*, A_{\mathcal{M}} = \lambda(q_0, w)$. Two Mealy machines $\mathcal{M}$ and $\mathcal{N}$ are said to be equivalent if they cannot be distinguished by a sequence of I^* i.e $\forall w \in I^*, A_{\mathcal{M}}(w) = A_{\mathcal{N}}(w)$.

2.2 SSH in a nutshell

The Secure Shell Protocol (SSH) is a cryptographic network protocol to securely operate network services over an insecure network. It is mostly used for remote login and command-line execution. SSH runs over TCP and a typical transcript thereof is described in Fig 1.

An SSH exchange starts with `KEXINIT` messages, to initiate the key exchange process. It is used to agree on the cryptographic

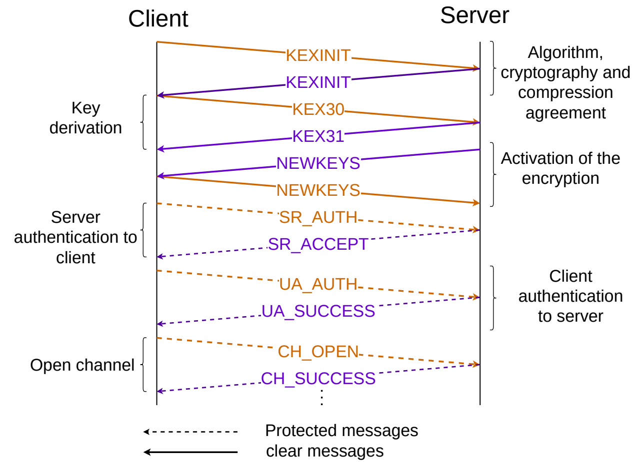

Figure 1: A Typical SSH session beginning

and compression algorithms. Keys derivation is performed through `KEX30` and `KEX31` messages. Afterward, a successful key derivation is notified with a `NEWKEYS` message to the other party. The following messages are protected using the derived keys and the agreed cryptographic algorithms.

After a successful key derivation, authentication takes place. Initially, the server proves its identity to the client using `SR_AUTH / SR_ACCEPT` messages. Then, the client authenticates itself to the server with a password or a public key through `UA_AUTH/UA_SUCCESS` messages.

Following successful authentication, the client expects to use the available services such as a remote terminal for example. To proceed, the client must establish a channel that provides access to services. Channels are opened using `CH_OPEN` and `CH_SUCCESS` messages.

2.3 OPC UA in a nutshell

OPC UA stands for Open Platform Communication Unified Architecture. It serves as an interoperability and security norm for data exchange, particularly in industrial control systems. It is used in millions of application and standardised as IEC 62541. Our focus is on the widely employed Client/Server communication model deployed over TCP. Fig. 2 illustrates a practical example of OPC UA exchange.

Each exchange starts with a Hello message, to share connection information such as the buffer size. Then, a so-called secure channel is established. This mandatory mechanism is meant to authenticate the OPC UA client application to the server and to provide the confidentiality and integrity of messages. User authentication and authorization are provided by sessions. It is worth mentioning that OPC-UA has two security layers: the protocol distinguishes application-level authentication, where the *client application* authenticates within secure channels from the *user*-level authentication, handled with sessions. However, during a discovery phase, it is also possible to create an *unprotected* secure channel without authentication, confidentiality or integrity.

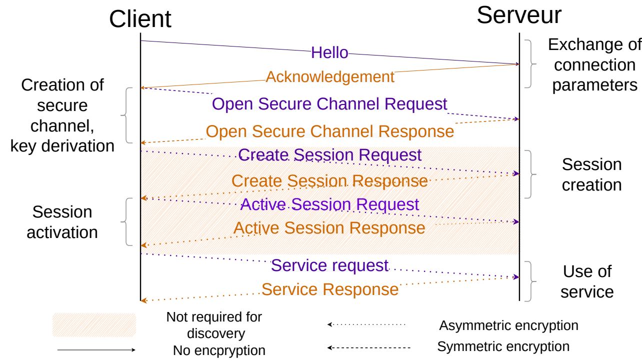

Figure 2: OPC UA exchange

After establishing the secure channel, the client aims to use the server's services. A Service is similar to a method call in a programming language. Services usually focus on the address space, which is the data that the server exposes to the clients; access to said address space requires a session. When using OPC UA over TCP, only discovery services are allowed to be sessionless (OPC UA session). Sessions are indeed used to ensure user authentication and authorization. Discovery services are used to discover information about the server. In particular, it is used to discover the server's set of cryptographic primitives and authentication methods. Naturally, discovery services are allowed to be used with an *unprotected* secure channel (i.e. without security).

If a session is required then the client needs to create one and activate it. No authentication is required for the first operation, the server only creates a session but does not bind it to a user. When the client activates the session, it authenticates itself and the server binds the created session to the client. This two-step mechanism allows a client to re-use the same session if the underlying connection crashes. Finally, the client can request services from the server.

In a typical OPC UA exchange, a client establishes the connection, opens an unprotected secure channel and then discovers the authorized security policy. Next, the client can close the unprotected secure channel, to establish a new one (with confidentiality and integrity), and creates and activates a session to access the address space.

2.4 Active Automata learning

Active automata learning is a part of model learning. Model learning consists in obtaining the model of a system, called *System Under Learning* (SUL). The model is a Mealy machine[1] or possibly another kind of state machine. We choose Mealy machines because they are an efficient representation of how message flows are handled which is common when applying model learning to analyze network protocol behavior [11, 12, 15, 22].

We can distinguish two flavors of model learning, active and passive learning. Passive learning uses traces of exchange with the SUL to learn. Consequently, it is not as efficient as active learning that interacts with the SUL [6]. Active learning is based on the *Minimal Adequate Teacher* (MAT) framework proposed by Angluin [7] and later adapted for Mealy machines [23]. A minimal adequate teacher is an entity that can answer two kinds of queries made by a learner:

- Output query: The learner asks the response to an input $w \in I^*$. The teacher replies with $A_{\mathcal{M}}(w)$. In the context of network protocols, each output query is made using a new connection, to ensure independence.
- Equivalence query: The learner has enough knowledge to build a hypothetical Mealy machine $\mathcal{H}$. It submits $\mathcal{H}$ to the teacher. If $A_{\mathcal{M}} = A_{\mathcal{H}}$, the teacher responds that the hypothesis is correct. Otherwise, it provides a counterexample. Those queries ensure the correctness of the obtained Mealy machine.

In practice, equivalence queries are approximated with different strategies using output queries (e.g. [21]). Hence, the obtained Mealy machine is an approximation of the real behavior. Nevertheless, it is considered to be sufficiently close to the real one to highlight message flow issues in all previous works. Moreover, we reproduced every message flow to ensure their existence.

Algorithms based on the MAT framework rely on a fixed input alphabet. Network protocols inherently lack such static characteristics. Operations or message fields that rely on contextual factors (encryption, sequence number…) highlight the inapplicability of using network protocol's messages as input alphabet.

To address this challenge, the learner uses a fixed alphabet, where each letter corresponds to one message of the network protocol. An intermediate component, the *mapper*, is responsible for translating these abstract letters into actual network protocol messages (as shown in Fig. 3). For example, the letter could be the string "Hello", which the mapper would translate to an actual OPC UA Hello message.

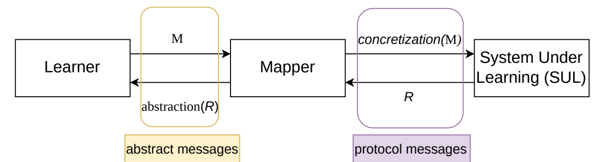

Figure 3: Active learning in practice

During the learning phase, the learner has no knowledge of the correct sequence of messages. Hence, the mapper have to deal with absurd sequences of messages regarding the specification. Furthermore, the mapper have to fill some field with default value when they have not be determined with previous messages.

3 STATE OF THE ART OF MEALY MACHINE ANALYSIS

As previously explained, model learning is an effective method to study the security of network protocol implementations. This method has been used for protocols such as TLS [22], SSH [15], TCP [14], or 802.11 handshake [20]. To analyze the obtained Mealy machines, some articles rely on manual inspection [12, 20]. This

[1]We exclusively focus on deterministic Mealy machines, which is consistent with the studied specifications. The word "deterministic" is thus omitted in the rest of the paper.

method is not suitable for complex Mealy machines. In particular, model learning implies having complete Mealy machines, meaning that from every node there is an outgoing edge for each letter of the input alphabet. Hence, even Mealy machines with a limited number of nodes may be challenging to read. Thus, manual inspection cannot guarantee systematic analysis.

Formal methods were introduced to enable a systematic analysis. Model checking is a method used to verify the compliance of a system. It aims to prove properties on a state machine modeling the system. Therefore, it seems quite adapted to the problem and several protocol implementations were analyzed with model checking [14, 15]. However, since a single counterexample is sufficient to prove the falseness of a property, model checking focuses on finding one counterexample to verify the falseness of a property. To uncover additional counterexamples related to a property, developers must iterate the process of model learning and model checking. Between each iteration of this process, the previous counterexample should be fixed in the Mealy machine. This iterative process is time-consuming.

Previous works [24, 25] use Tamarin, a tool dedicated to security protocols' formal verification. This tool is efficient however it is not designed for this purpose. To use Tamarin, the solution provided is to extract every path from the initial state of the state machine and use Tamarin on them. However, Mealy machines obtained from active automata learning contain a lot of cycles because they are complete. With cycles, the number of paths explodes. The reason is straightforward: when cycles exist on a path, we need to examine each path augmented with every combination of the cycles. Hence, many paths cannot be treated and strategies to reduce the number of paths to a restricted number of suspicious paths are used. In this case, we cannot be sure that the Mealy machine is compliant with the properties because there is still a possibility that an unexamined path proves the opposite.

Recently a specific method has been developed to address this issue [13]. It is inspired by the SPIN model checker method [19] and uses Deterministic finite automaton (DFA) intersection. We can easily translate a Mealy machine into a DFA by splitting edges into two, one containing the input and the other the output. To use this method, we have to describe the undesired behaviors as a DFA. The intersection of the undesired behavior and the state machine results in a comprehensive set of wrong behaviors. CVEs and specifications can help to build a large database of undesired behaviors. Furthermore, if there exists a perfect state machine, we may add a spurious state and find every wrong behavior. The strength of this method is its exhaustiveness. The exhaustiveness comes with being able to find every counterexample regarding a property. With a perfect state machine and spurious state, this method can find every single counterexample. However, there is no such thing as the perfect state machine for most network protocols. Hence, we are limited to identifying the specific incorrect behaviors that are in our database of undesired behaviors. Nevertheless, to ensure the security of network protocol implementations, we are required to be able to discover new bugs. Furthermore, a bug could manifest in various ways and may not be covered by the description used to analyze the Mealy machine.

To our knowledge, the SPIN-based method is the only one dedicated to the analysis of secure protocols' Mealy machines.

4 MEALY VERIFIER

Our objective is to offer a tool to study discrepancies between a specification and its implementation through the utilization of expected behaviors. Using expected behavior facilitates the formulation of properties that an implementation must verify. Importantly, we do not make assumptions about how a property is violated. Consequently, this tool can discover new bugs. Furthermore,We provide exhaustivity. It means that all counterexamples are provided.

The Mealy Verifier is a tool written in Rust, which aims to provide exhaustive and expressive Mealy machine analyses. This tool is available on GitHub https://github.com/artfire52/Mealy-Verifier. It is designed to detect bugs and vulnerabilities. To provide a complete analysis of network protocol implementations, it must be combined with model learning.

4.1 Design Rationale

The Mealy Verifier is built upon the concept of specifying expected behavior. Rather than offering a comprehensive syntax to cover all potential expected behaviors, we adopt an ad-hoc approach by presenting various types of properties. The motivation is that expected behaviors are often similar, thus we can limit the expressiveness of our properties to simplify their complete verification. Mealy verifier's properties refer to a kind of expected behavior. Expected behaviors are expressed using those properties. Furthermore, we develop an algorithm for each kind of property rather than a global one. It facilitates providing exhaustiveness for a large set of behaviors rather than having a very narrow expressiveness. Those algorithms aim to find all counterexamples for the specified property. However, we can highlight the common part of those algorithms. Those algorithms are based on the Depth First Search (DFS) algorithm or an exhaustive comparison with edges.

Moreover, there is no false positive in the output of our algorithms. All proof and algorithms are given in the Mealy Verifier repository.

The Mealy Verifier outputs graphs. There are two types of outputs. The first one is a graph that highlights the reason why a property is violated (see in Fig 4). For example, if a state and a transition are not expected to be there, they are present in the output graph. Consequently, the graph contains every counterexample. The second one is a graph where every path on the graph is a counterexample (an example is shown later in Fig 7).

Before explaining every type of property considered by the Mealy Verifier, we have to define some vocabulary that we use to make it clearer. A Mealy machine contains states and events. An event is a pair of one letter, an input and one output. Events are written `i/o`. Inputs and outputs are written using input and output letters with an extended syntax. Wildcards can be used to designate several letters at once. For example, "auth_*" may correspond to any authentication method (password, public key). We also introduce the negation. For example "!A" designates every letter other than A. Finally, the syntax allows an or operation with letters. For example, "A+B" means the letter "A" or the letter "B". This syntax facilitates the writing of properties.

Properties are written using events. The algorithms compare properties' events to the ones contained in the rules. It is done by

comparing inputs to inputs and outputs to outputs. Every algorithm only requires the Mealy machine and the rule itself.

4.2 Properties

We detail every property available with the Mealy Verifier. This set of properties is sufficient for large use cases and a high number of properties. We do not detail the syntax of the rules, all syntax information is available on the GitHub repository.

Sink as termination. A sink state is a state where no outgoing transition leads to another state. Thus, a sink state is entirely defined by the events labeling its outgoing edges. We usually expect a sink state to correspond to an end-of-connection state.

This type of property aims to verify if the sink states present in the Mealy machine match the given descriptions. The description is made by giving the authorized events labeling the outgoing edges of the sink state. A sink state that does not match the description is a counterexample.

For example, we could be interested in finding only sink states corresponding to the end of the connection. We could specify this sink state by the event `*/Eof`. It means that we are only expecting a sink state corresponding to the end of the connection. If there is another sink state, where the server remains silent and keeps the connection indefinitely open; this could potentially enable a denial-of-service attack.

Sink as target. This type of property aims to verify if a precise event leads to one sink state matching a description. This description is specified with a list of accepted outgoing events and the event leading to one of the specific sink states.

For instance, we can ensure that some error message leads to the sink state corresponding to the end of the connection. The difference with the previous one is that we focus on the event triggering a transition to a specific sink state.

Several sink states may exist on the Mealy machine due to the abstraction related to the mapper. For example, the mapper may respond directly without asking the SUL, if some input letters are supposed to change the behavior of the mapper. In the SSH inference paper [15], only one channel could be used. Thus, the mapper will answer directly to the learner that it cannot create a new channel when one is already open. The behavior is similar when the learner asks to close a channel when none is opened. Hence, there are several sink states, one with an open channel and one without an open channel.

Output. For certain messages received by the server, we want it to respond with only a limited choice of replies. This is particularly the case for malformed messages, for which we hope to receive an error message or a termination of the connection in response. It is not restricted to malicious messages. For example, the SSH inference paper [15] mentions that the SSH server must reply to the `SR_AUTH` message with `SR_ACCEPT` or shut the connection down.

A counterexample is an event with the corresponding input that has an unauthorized output. An example is given in Fig 4. `SR_AUTH` is ignored by the server which violates the property mentioned previously.

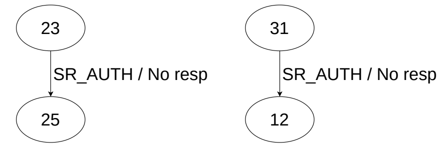

Figure 4: Example of Unexpected output for the input `SR_AUTH`.

Expected events Index. We may expect some event to appear at a precise moment in every sequence of messages. For example, the OPC UA Hello message is expected to be the first message accepted by the server. A counterexample is a transition that occurs on a path at the given position and that does not match the given event. Events leading to sink states are ignored because it means that the server ignores the message. That is the expected behavior in case of an unexpected message at the given position.

Expected events Sequence. We may expect to observe sequences of events.

Specifically, when an operation of the protocol implies a list of events to be observed in a precise order as an uninterrupted sequence. When an event on a path matches the first event in the list, then the Mealy Verifier will verify that the following ones also match the sequence. However, there are two exceptions. The first one is if a transition leads to a sink state. It means that the server refuses the unexpected message. This a correct behavior. The second exception is a list of events that are authorized. Those events are authorized to interrupt the sequence and specified in the property. The main use is to authorize a message rejection that does not lead to a sink state. For example, we can authorize every event matching with the server ignoring the messages or responding with an error. Those authorized events are defined when writing the rule.

For example, in OPC-UA, if we consider the sequence of events `hello/ack` then `open_secure_channel_request` / `OpnRepOk`. The state reached after the Hello OPC UA message may contain a cycle to itself with one event such as `read_req/No resp`. It does not indicate that the cycle does not comply with the specified property but that the server ignores other messages different from the expected ones. This event could be written as `*/No  resp` in this example.

The output of this rule will be the states and transitions that are responsible for the violation of the rule. If the sequence is violated for an event, we do not continue to verify if the sequence is respected.

Conditional event. This type of property is the most adapted for security properties.

Some actions related to network protocols require prerequisites. With Mealy machines, the action and the prerequisites are events. The action is expected only if prerequisites are satisfied. For example, reading data on an OPC UA server requires user authentication. This corresponds to the action specified with the event `read_req/ReadRepOK` while the user authentication prerequisite corresponds to the event `activate_session/AcSesResOK`.

To specify this type of property, we need to specify the event corresponding to the action that requires prerequisites.

And we also require to specify prerequisites with events i_1/o_1, i_2/o_2... Nevertheless, it is not sufficient. A prerequisite can be canceled by another event. If we still consider the example of OPC UA client reading on the server, the event that corresponds to closing the session cancels the prerequisite of user authentication. Hence, a prerequisite is a pair of events. The first event is used to change the state of the prerequisite from `False` to `True`, in our example, it is `activate_session/AcSesResOK`. The second event is used to change the internal state from `True` to `False` and in our example, it is `close_session/CloSesResOK`. We note the prerequisites as:

`activate_session/AcSesResOK|close_session/CloSesResOK`.

If more than one prerequisite is required to specify a property, the order is important. A prerequisite can become true only if previous ones are true. Furthermore, if a prerequisite is canceled by an event, all following prerequisites become false. In the case of an authentication with several events, it means that the whole authentication has to be done again to satisfy the prerequisites. If we continue with our example, by considering the creation of a session as an additional prerequisite, it means that a session can only be activated if it has been created before. A simplified counterexample is given in Fig. 5. We remind the property has the following prerequisites:

- `create_sessions/CreSesResOk |close_session/CloSesResOK`
- `activate_session/AcSesResOK |close_session/ CloSesResOK`

And the action is `Read_req/ReadRepOK`. In Fig. 5, a user can read in the address space without activating a session. However, activating the session is required to prove the user's identity.

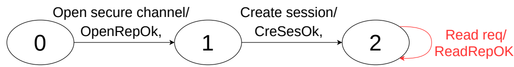

Figure 5: An actual violation of the Conditional Rule about Data Reading for OPC UA. The user can run a `Read_req` request even if the session has not been activated

Importantly, events can separate prerequisites. As long as they are satisfied when the action is observed, the behavior is correct. There is no need to verify them uninterruptedly. This type of property verifies that prerequisites are satisfied when the action is observed.

To find counterexamples, the algorithm behind this property leverages DFS starting from the state where the action is observed and using the transposed Mealy machine (the direction of the edges is reversed). It facilitates the detection of events that can cancel a prerequisite.

It is worth noting that this rule has a very specific output. It may have several graphs as output, one per state that has an outgoing edge with the label corresponding to the action. In those graphs, we ensure that every path, from a state without ingoing edges (excluding cycles) to a state where the action is observed, is a counterexample. To guarantee this feature, a state can be split into several states with the same name. The reason thereof is illustrated in Fig. 6 and in Fig. 7. The applied rule is the same as before except that we simplified the notation. The action is i_a/o_a and prerequisites are as the action event and the prerequisites are i_1/o_1 | $\bar{i}_1/\bar{o}_1$ and i_2/o_2 | $\bar{i}_2/\bar{o}_2$. In Fig. 6, We consider a subgraph of a Mealy machine manually created. The path following transitions at the top serves as a counterexample, as do the path following transitions at the bottom Thus,the output cannot omit any edge without lacking a counterexample. Nonetheless, a path respecting the property is present when taking the transition `other/other` and i_2/o_2. Thus, the graph in the figure does not only contain counterexample paths.

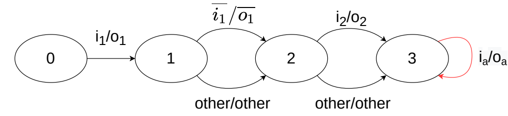

Figure 6: Example of an output without splitting the states. Other designate messages other than the one present in the specification of the property

To tackle this issue, states are split in Fig. 7. It means that the state with the same name in this output graph refers to the same state in the Mealy machine. Each node labeled 1 is corresponding to the node 1 in the Mealy machine. We can observe that with the splitting operation, every path is a counterexample. Each state has different information regarding the property. Those information are the prerequisites and event able to cancel them between the state and the state where the action is observed.

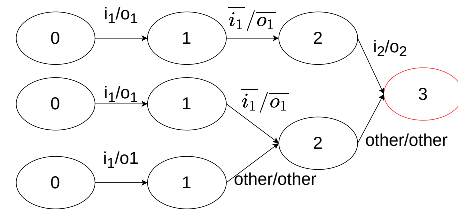

Figure 7: Example of an output with splitting the states

Restricted events. In some circumstances, the authorized events are limited. For example, we may not accept a second successful user authentication after a successful one. This is the case for SSH. Hence, after a successful authentication only events `*/!UA_SUCCESS` are authorized. The property verifies that only the given events are present. This restriction is verified after a starting event has been observed (or after the initial state if the starting event is not provided). In the previous example, the starting event is the first authentication. The restriction is not applied when a release event is reached (or in a sink state if the release event is not provided). The aim is to apply a restriction on authorized events to subgraphs. We also added an event that cancels the rule. When this event is reached the rule will not apply even if the starting event is met. It is different from the release because it aims to avoid applying the property and not only to stop its verification. Eventually, the output is simply the states and transitions that do not respect the rule.

5 SSH

This section focuses on the test cases used to evaluate the Mealy Verifier. Studying several protocols is proof that the tool is not specific to a given protocol. For SSH, we use existing Mealy machines provided in previous work [15].

5.1 Experiments

For SSH, we leverage existing Mealy machines extracted from Fiterau-Brostean et al. [15]. In this article, SSH is studied with the model checker NuSMV [9]. We aim to replicate their study with our method, to demonstrate the applicability of our method to protocols beyond OPC UA. Fiterau-Brostean's mentions 12 properties. We explain them and precise which type of property is used to model them in the Mealy Verifier:

(1) When the output of an edge indicates that the TCP connection is closed (`NO_CONN`), the edge leads to a sink state (Sink target).
(2) The SSH mapper can only handle one channel at a time. Due to this limitation:
(2.1) A channel cannot be opened while another is already open (Restricted event).
(2.2) A channel cannot be closed unless one is opened (Restricted event).
(3) Server authentication can only happen after observing key derivation messages (Conditional event).
(4) To open a channel, a client must be authenticated (Conditional event).
(5) It must be possible to rekey at any time after server authentication (pre-auth) or client authentication (auth) (Expected event sequence).
(6) When the output of an edge indicates disconnection (`DISCONNECT`), the edge leads to a sink state (Sink as target).
(7) The server refrains from sending another `KEXINIT` until it has transmitted `SR_ACCEPT` (Restricted event).
(8) Server responds to `SR_AUTH` with `SR_ACCEPT` or end of connection messages (`NO_CONN` or `DISCONNECT`) (Output).
(9) After the server authentication and before a successful client authentication, the server responds with a failure message for an incorrect user authentication attempt (Restricted event).
(10) There is only one successful user authentication per connection (Restricted event).
(11) After successful client authentication, `SR_AUTH` message must be ignored (Restricted event).
(12) When a channel is opened successfully, the server responds to the first `CH_CLOSE` with `CH_CLOSE` (Restricted event).

Those properties were checked on three implementations: DropBear, OpenSSH and Bitvise with one Mealy machine for each one. Our aim is first to reproduce the results with the model checker NuSMV and then to reproduce them with our method. However, Fiterau-Brostean et al. highlight that Bitvise implementation buffered outputs and the output vocabulary is different. Consequently, the rules need to be written differently. However, the repository does not provide those new properties. Thus, we could not reproduce the results for Bitvise implementation. We focus on OpenSSH and DropBear which have common properties provided in the repository given with SSH previous work.

5.2 Results

As previously said, we focus on OpenSSH and DropBear implementation. We reproduce the results previously obtained by Fiterau-Brostean [15]. We first replicate the results obtained with the model check NuSMV. Then, we apply our method to SSH Mealy machines.

We obtained the same results regarding violated and respected properties. However, the exhaustivity provided by our tool permits us to discover a new reason for one of those violations compared to previous work.

We diverge on the interpretation of property 8 for OpenSSH. This property mentions that only `SR_ACCEPT`, `NO_CONN` and `DISCONNECT` are acceptable replies to `SR_AUTH`. In the original study, property 8 is said to be violated because of an `UNIMPL` response. Our tool provides all the counterexamples related to these violations. Hence, we can verify that property 8 also presents an issue with unexpected `NO RESP` answers. Whether it is a simple issue or more is up for debate but it highlights the importance of having exhaustiveness. We can easily verify it with all counterexamples. With model checking, the solutions are to manually verify on the Mealy machine or to fix the Mealy machine until no counterexamples are found. Fixing the Mealy machine might be a time-consuming and challenging task, as it involves either manually analyzing the Mealy machine or iteratively applying rounds of corrections of counterexample from NuSMV, followed by rerunning the model checker. This is a use case where our tool is better than the previous approach for network protocol stack Mealy machine analysis.

6 OPC UA

Our work introduces a complete workflow for OPC UA, from active automata learning to analysis. The inference and analysis of OPC UA Mealy machines represent novel contributions. We chose OPC UA because, to our knowledge, no previous model learning work has been done on it. Additionally, OPC UA is widely used in industrial contexts.

6.1 Experiments

To perform model learning, we create our mapper. The mapper must be flexible as mentioned previously. Modifying an existing OPC UA implementation requires a deep understanding of it and to rewrite every internal mechanism. Thus, creating a dedicated implementation to function as a mapper is easier but still challenging. Finally, we created our mapper from scratch.

The inference tools is available on GitHub at https://github.com/artfire52/opc-ua-inferer.

We select the input alphabet to study the handshake, secure channels, sessions and reading/writing operations in the address space. In other words, we focus on the security of address space access, leading to shorter Mealy machine inference (from 1 hour to 10 hours depending on the implementation). The vocabulary used is described in the Mealy Verifier repository. We were interested in four implementations:

- UANET: the official implementation [3] written by the OPC Foundation responsible for OPC UA standard.

Properties	1	2.1	2.2	3	4	5.auth	5.pre-auth	6	7	8	9	10	11	12
DropBear	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
OpenSSH	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗*	✓	✓	✗	✓

Table 1: Results of SSH's Mealy machine analysis.
*** interpretation slightly differs from previous work.**

- Open62541: reference implementation written in C with an OPC UA certification for an example server [5].
- S2OPC: a stack with an OPC UA certification for an example server and an ANSSI certification [2];
- opcua-asyncio: most famous python stack for OPC UA [4].

We use model learning on several versions of each implementation. The SUL were the example servers given by developers in their repositories. We use two scenarios for our inferences. The first scenario, called *protected scenario* corresponds to secure exchanges with encryption and signature. The second one, called *unprotected scenario* corresponds to exchanges without these security measures. Overall, combining all versions of the stacks and the two scenarios, we inferred 150 Mealy machines. We use the learning algorithm L* [7] with the pylstar implementation [8]. We adapt pylstar to infer faster the SUL, using the optimization presented in previous work [22].

The OPC UA specifications ([17], section 4.3.2.3) mention that the number of sessions and the number of secure channels are limited. Thus, when inferring an OPC UA implementation, we cannot be sure that a session or a secure channel will be available on the server. Therefore, two successive output queries with the same prefix could lead to different results. For example, one succeeding in opening a secure channel, and the second one failing due to the limitation of secure channels. From the point of view of the learning algorithm, it is seen as non-deterministic behavior. Since we only consider deterministic Mealy machines, the inference will thus fail.

To address this challenge, we propose the learning architecture presented in Fig. 8. The learner is sending one output query at a time but the target server changes for each query. All targets are running the same SUL. We also assume that output queries do not interfere with each other when the maximum number of sessions or secure channels are not reached. By choosing the appropriate number of targets, we can ensure that session's and secure channel's timeouts are reached before making an output query to the same server. Those optimizations help accelerate the inference process.

After inferring OPC UA's Mealy machines, we aim to verify the following properties with the Mealy Verifier:

(1) Valid OPC UA communication starts with a Hello message ([16] section 7.1.3) (Expected Event Index).
(2) The second step in communication is to establish a secure channel ([16] section 7.1.3) (Expected Event Index).
(3) The secure channel request comes immediately after the Hello message (This property allows gathering more information than (1) and (2) since the Hello message can appear later in certain OPC UA stacks) ([16] section 7.1.3) (Expected Sequence Event).
(4) Session creation requires a secure channel ([18] section 5.6.2.1) (Conditional Event).

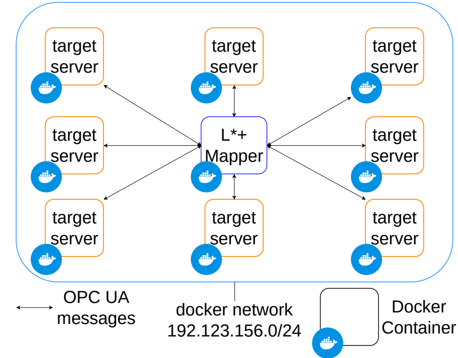

Figure 8: Learning architecture for OPC UA

(5) A session is activated after its creation ([18] section 5.6.3.1) (Conditional Event).
(6) Only an authenticated user (i.e. a user with an active session) has access to the address space ([18] part 6.3.1) (Conditional Event).
(7) Malformed messages used for inference must be rejected by the server (Output).
(8) Address space access must not be allowed on unprotected secure channels ([17] section 5.1.3) (Output).
(9) Only one sink state, corresponding to the end of the connection, is present (Sink as termination).

6.2 Results

Close versions of the same implementation are often similar. From 150 inferred Mealy machines, it remains 19 distinct Mealy machines. Unless stated otherwise, every bug or vulnerability mentioned below is new and was discovered with the Mealy Verifier. The only exception is UANET in which we did not discover new bugs or vulnerabilities. Those results are condensed in Table 2. As a reminder, in the context of OPC UA, two scenarios are used: the protected one involving confidentiality and integrity and the unprotected one without these security features. All the issues presented in this section have been reproduced.

6.2.1 Connection establishment. The issues mentioned here are highlighted by the three first properties used to verify the behaviors of OPC UA servers.

Initialization. All OPC UA exchanges that are initiated by the client must begin with a Hello message. However, Open62541 and opcua-asyncio accept the secure channel establishment to be the first message. However, `Hello` message is mandatory to be the first one sent by the client according to specification. It is required for the negotiation of the buffers' size. However, it does not seem

Implementation	Mode	Version	Initialization	Auth	After close	Session bypass	Anonymous Session	Sink State	DOS
Open62541	P	v1.1.*	✗						
		v1.2*							
		v1.3-v1.3.3		✗					
	U	v1.1.*	✗						
		v1.2*-v1.3.3							
S2OPC	P&U	1.1.0,1.2.0,1.3.0					✗		
opcua-asyncio	P	v0.9.0-v0.9.92	✗					✗	✗
		v0.9.3-0.9.95	✗		✗			✗	✗
		v0.9.97-v1.0.1	✗					✗	
	U	v0.9.0-v0.9.95	✗					✗	✗
		v0.9.97-v1.0.1	✗			✗		✗	
UANET	P	1.03.350-1.4.371.50		✗			✗		
UANET	U	1.03.350-1.4.371.50							

Table 2: Results of OPC UA Mealy machines analysis. P: protected scenario. U: unprotected scenario. Marks indicate the violation of the property.

to have much consequences on the normal behavior of the OPC UA exchange because servers use default size for buffers. It is an anomaly but not a vulnerability.

6.2.2 Sessions. We focus on rules concerning session use. A session must be used above a secure channel and ensure user authentication.

Authentication. Two tested implementations are vulnerable to authentication with the wrong token. Open62541 target server was vulnerable to wrong certificate authentication. The problem is that an optimization dedicated to Linux introduces the wrong management of accepted certificates. Hence, every certificate is accepted. UANET is vulnerable to wrong credentials authentication. Any credentials are accepted. In both cases, using a secure channel is required. Secure channels can only be established with authorized client applications. Thus, those issues remain limited.

Session bypass. Using the unprotected mode, opcua-asyncio does not verify if the session is activated. It only checks for a created session. Thus, no user authentication is performed. Using unprotected mode and bypassing the user authentication, no security remains. Any malicious user can freely access the address space, no security remains. This new vulnerability has received the number CVE-2023-26150.

Closed session. The address space is reachable after closing the session for opcua-asyncio. The consequences are limited because only an authenticated user using an authenticated application is allowed to access the address space. Even if it is not a security threat, it remains an unexpected behavior.

Anonymous session. Anonymous sessions are not forbidden in the specification. Nevertheless, they must be disabled by default. Moreover, example servers should not encourage bad habits. It affects S2OPC and UANET where the problem lies in the configuration of example servers. It is a matter of concern when we know that OPC UA security configuration is an issue in most cases [10]. However, the security offered by secure channels remains. Thus, using an authenticated application is still required.

6.2.3 Multiple sink states. We expect to have only one sink state that corresponds to the end of the connection. Nevertheless, we can observe two sink states in several opcua-asyncio Mealy machines. The second sink state corresponds to a state where the connection is kept alive. This could lead to a denial of service attack. To reach the second sink state establishing a secure channel is required. However, secure channels have a timeout and their number is limited. An attacker can not create enough secure channels to threaten the server. Thus, this has only a weak impact on the implementation. Furthermore, the potential attack against the limited number of secure channel is out of the OPC UA scope. Nevertheless, we highlight that it is still an undesired behavior.

6.2.4 Denial of service. We can obtain results directly from the inference phase. When inferring opcua-asyncio, we encountered some availability issues which can lead to denial of service attacks. The server is stuck in an infinite loop. The server is asynchronous and not multi-threaded. Hence, no other client can access the server. Furthermore, in this loop, the server allocates memory and may have impact on the host. Moreover, this vulnerability is very easy to exploit and makes it a real threat. It can be performed with only one message sent. This message can be the first one of the OPC UA exchange. This new vulnerability has received the CVE-2023-26151.

7 PERFORMANCE INSIGHT

The performance evaluation was conducted on a personal laptop equipped with an Intel® Core™ i5-1145G7 CPU and 16 GB of RAM. We focus on SSH Mealy machines because of their size and complexity compared to OPC UA ones. Using scenarios for OPC UA inference implies having simpler and more precise Mealy machines.

Our tool required a total runtime of approximately a half second for the analysis of the two SSH Mealy machines. The analysis of individual performances reveals a runtime of around 0.3s for the OpenSSH Mealy machine (30 states and 630 edges). Comparably, around 0.2s were required for the DropBear Mealy Machine (28 states and 364 edges).

Because our algorithms are based on DFS, they have a linear complexity in the number of states and edges. Thus, our tool scales well with a larger Mealy machine.

8 CONCLUSION

We present a complete method to analyze the behavior of network protocol implementations. In particular, we provide a complete analysis workflow of OPC UA implementations leading to the discovery of new bugs and vulnerabilities.

The Mealy Verifier is more precise than classic model checking since it provides a complete set of counterexamples related to a property. Naturally, it is far better than manual inspection because results can easily be reproduced and provide exhaustiveness. The exhaustiveness eases the understanding of wrong behaviors and aims to avoid missing one wrong behavior that could be related to the same property. Also, compared to the previous existing solution [13], our tool provides a way to specify expected behavior. It changes the perspective and may facilitate the formulation of properties rather than gathering unexpected behavior or having to specify the expected state machine. Moreover, our properties encourage the use of specific properties and to split properties into atomic ones. It helps with the analysis of undesired behaviors.

Furthermore, exploring the integration of verification in the active automata learning phase can enhance the performance of the all process. The workflow is time-consuming and the learning phase is the bottleneck. Melding the two processes may lead to a time-saving solution.

ACKNOWLEDGMENTS

This work is funded by the French Defense Innovation Agency (AID) under contract n° 2021650010 (CERES).

REFERENCES

[1] [n. d.]. CVE - CVE-2014-0224. Available from MITRE, CVE-2014-0160.. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0224

[2] [n. d.]. Systerel / S2OPC · GitLab. https://gitlab.com/systerel/S2OPC

[3] 2023. Official OPC UA .Net Standard Samples from the OPC Foundation. https://github.com/OPCFoundation/UA-.NETStandard-Samples original-date: 2020-05-28T13:53:11Z.

[4] 2023. opcua-asyncio. https://github.com/FreeOpcUa/opcua-asyncio original-date: 2018-08-02T07:45:42Z.

[5] 2023. open62541. https://github.com/open62541/open62541 original-date: 2013-12-20T08:45:05Z.

[6] Bernhard K. Aichernig, Edi Muškardin, and Andrea Pferscher. 2022. Active vs. Passive: A Comparison of Automata Learning Paradigms for Network Protocols. *Electron. Proc. Theor. Comput. Sci.* 371 (Sept. 2022), 1–19. https://doi.org/10.4204/EPTCS.371.1

[7] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75, 2 (Nov. 1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6

[8] Georges Bossert. 2023. pylstar : An implementation of the LSTAR Grammatical Inference Algorithm. https://github.com/gbossert/pylstar original-date: 2015-11-17T12:37:55Z.

[9] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Computer Aided Verification*, Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Ed Brinksma, and Kim Guldstrand Larsen (Eds.). Vol. 2404. Springer Berlin Heidelberg, Berlin, Heidelberg, 359–364. https://doi.org/10.1007/3-540-45657-0_29 Series Title: Lecture Notes in Computer Science.

[10] Markus Dahlmanns, Johannes Lohmöller, Ina Berenice Fink, Jan Pennekamp, Klaus Wehrle, and Martin Henze. 2020. Easing the Conscience with OPC UA: An Internet-Wide Study on Insecure Deployments. In *Proceedings of the ACM Internet Measurement Conference*. ACM, Virtual Event USA, 101–110. https://doi.org/10.1145/3419394.3423666

[11] Lesly-Ann Daniel, Erik Poll, and Joeri De Ruiter. 2018. Inferring OpenVPN State Machines Using Protocol State Fuzzing. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, London, United Kingdom, 11–19. https://doi.org/10.1109/EuroSPW.2018.00009

[12] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. 2020. Analysis of DTLS Implementations Using Protocol State Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2523–2540. https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean

[13] Paul Fiterau-Brostean, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. 2023. Automata-Based Automated Detection of State Machine Bugs in Protocol Implementations. In *Proceedings 2023 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA, USA. https://doi.org/10.14722/ndss.2023.23068

[14] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. 2016. Combining Model Learning and Model Checking to Analyze TCP Implementations. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Vol. 9780. Springer International Publishing, Cham, 454–471. https://doi.org/10.1007/978-3-319-41540-6_25 Series Title: Lecture Notes in Computer Science.

[15] Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. 2017. Model learning and model checking of SSH implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. ACM, Santa Barbara CA USA, 142–151. https://doi.org/10.1145/3092282.3092289

[16] OPC Foundation. 2022. OPC Unified Architecture Mappings. Release 1.05.01.

[17] OPC Foundation. 2022. OPC Unified Architecture Part 2 :Security Model. Release 1.04.

[18] OPC Foundation. 2022. OPC Unified Architecture Part 4 :Services. Release 1.05.00.

[19] G.J. Holzmann. 1997. The model checker SPIN. *IIEEE Trans. Software Eng.* 23, 5 (May 1997), 279–295. https://doi.org/10.1109/32.588521

[20] Chris McMahon Stone, Tom Chothia, and Joeri De Ruiter. 2018. Extending Automated Protocol State Learning for the 802.11 4-Way Handshake. In *Computer Security*, Javier Lopez, Jianying Zhou, and Miguel Soriano (Eds.). Vol. 11098. Springer International Publishing, Cham, 325–345. https://doi.org/10.1007/978-3-319-99073-6_16 Series Title: Lecture Notes in Computer Science.

[21] Arjun Radhakrishna, Nicholas V. Lewchenko, Shawn Meier, Sergio Mover, Krishna Chaitanya Sripada, Damien Zufferey, Bor-Yuh Evan Chang, and Pavol Černý. 2018. DroidStar: callback typestates for Android classes. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg Sweden, 1160–1170. https://doi.org/10.1145/3180155.3180232

[22] Aina Toky Rasoamanana, Olivier Levillain, and Hervé Debar. 2022. Towards a Systematic and Automatic Use of State Machine Inference to Uncover Security Flaws and Fingerprint TLS Stacks. In *Computer Security ESORICS 2022*, Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng (Eds.). Vol. 13556. Springer Nature Switzerland, Cham, 637–657. https://doi.org/10.1007/978-3-031-17143-7_31 Series Title: Lecture Notes in Computer Science.

[23] Muzammil Shahbaz and Roland Groz. 2009. Inferring Mealy Machines. 5850 (Nov. 2009), 207–222. https://doi.org/10.1007/978-3-642-05089-3_14 MAG ID: 1529010373.

[24] Qinying Wang, Shouling Ji, Yuan Tian, Xuhong Zhang, Binbin Zhao, Yuhong Kan, Zhaowei Lin, Changting Lin, Shuiguang Deng, Alex X. Liu, and Raheem Beyah. 2021. {MPInspector}: A Systematic and Automatic Approach for Evaluating the Security of {IoT} Messaging Protocols. 4205–4222. https://www.usenix.org/conference/usenixsecurity21/presentation/wang-qinying

[25] Xieli Zhang, Yuefei Zhu, Chunxiang Gu, and Xuyang Miao. 2021. A Formal Verification Method for Security Protocol Implementations Based on Model Learning and Tamarin. *J. Phys.: Conf. Ser.* 1871, 1 (April 2021), 012102. https://doi.org/10.1088/1742-6596/1871/1/012102