Automata-Based Automated Detection of State Machine Bugs in Protocol Implementations

Paul Fiterău-Broştean*, Bengt Jonsson,* Konstantinos Sagonas*† and Fredrik Tåquist*
*Department of Information Technology, Uppsala University, Uppsala, Sweden
†School of Electrical and Computer Engineering, National Technical Univesity of Athens, Athens, Greece

***Abstract*—Implementations of stateful security protocols must carefully manage the type and order of exchanged messages and cryptographic material, by maintaining a state machine which keeps track of protocol progress. Corresponding implementation flaws, called *state machine bugs*, can constitute serious security vulnerabilities. We present an automated black-box technique for detecting state machine bugs in implementations of stateful network protocols. It takes as input a catalogue of state machine bugs for the protocol, each specified as a finite automaton which accepts sequences of messages that exhibit the bug, and a (possibly inaccurate) model of the implementation under test, typically obtained by model learning. Our technique constructs the set of sequences that (according to the model) can be performed by the implementation and that (according to the automaton) expose the bug. These sequences are then transformed to test cases on the actual implementation to find a witness for the bug or filter out false alarms. We have applied our technique on three widely-used implementations of SSH servers and nine different DTLS server and client implementations, including their most recent versions. Our technique easily reproduced all bugs identified by security researchers before, and produced witnesses for them. More importantly, it revealed several previously unknown bugs in the same implementations, two new vulnerabilities, and a variety of new bugs and non-conformance issues in newer versions of the same SSH and DTLS implementations.**

I. Introduction

Implementations of network protocols must conform to their specifications in order to avoid security vulnerabilities and interoperability issues. Even seemingly innocent deviations from the standard specification may expose implementations to security attacks. Protocols that establish secure connections (e.g., SSH, TLS, DTLS, QUIC, etc.) must carefully manage the type and order of exchanged messages and cryptographic material, by maintaining a state machine which keeps track of how far the protocol has progressed. Any deviation from the order prescribed in the protocol's specification may constitute anything between an inconsequential error to a serious vulnerability. Corresponding implementation flaws, called *state machine bugs*, may be exploitable, e.g., to bypass authentication steps or establish insecure connections [5], [13], [18].

An approach that has proven effective for finding state machine bugs is *state fuzzing* [5], [13], [18]. It automatically infers state machine descriptions of protocol implementations using *model learning* [36], [44]. Model learning is an automated black-box technique which produces state machine models describing how an implementation handles message flows, by observing how the implementation responds to sequences of test inputs. The learned model is then analyzed to spot state machine bugs that result from flaws in the implementation's control logic. In previous works on state fuzzing, the detection of flaws has been performed (i) by ocular inspection of the learned model using expertise about the protocol's rules for ordering of messages [13], [12], [34], [18], or (ii) by pairwise comparisons between models of different implementations, whereby *deviant behaviors* (input sequences for which the compared models produce different outputs) are first extracted automatically, and then manually analyzed against the protocol's standard [29, Sect. 3.3]. Such manual inspections of models or analyses of deviant behaviors require effort and expertise. Since learned models can be both approximate and big, containing tens or even hundreds of states (as e.g., we report in Table IV), and since state machines of different implementations of the same protocol often differ significantly, ocular inspection of models can miss errors (as we show in Section VIII). Moreover, after the errors that were detected have been fixed, such manual steps must be repeated on a model (or a trace from the model) of the updated implementation, which can be rather different from the original one (and now might contain other errors). Given the importance of detecting state machine bugs, it would be desirable to have an automated technique for identifying them (e.g., one that does not require any ocular inspection of learned models).

In this paper, we present a black-box technique for detecting state machine bugs in implementations of stateful network protocols, which is fully automated if supplied with a (possibly inaccurate) model of the implementation and a catalogue of state machine bugs for the protocol in the form of DFAs. One way to obtain a model of the implementation automatically is by model learning, as we do for the protocols on which we have applied our technique. The construction of DFAs is not automated, but in Section V we present a systematic approach to assemble such a bug pattern catalogue for a protocol. Advances over previous work include the following.

1) Bug Specification: Instead of ocularly inspecting models, we automatically verify the model against specifications of allowed message orderings, encoded by finite automata. Each automaton "observes" the sequence of messages exchanged during a protocol interaction and reports whenever that sequence violates the requirement that it encodes. By their flexibility, automata can encode ordering requirements in several styles. In the typical case, a specification is provided as a catalogue

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.23068
www.ndss-symposium.org

of so-called *bug patterns*. A bug pattern is a (typically small) automaton specifying a particular error in an implementation's control logic, for instance sending two specific messages in the wrong order, or omitting an authentication step. Dually, bug patterns can also be seen as encoding specific requirements, for instance specifying the order of sending two messages. A rather complete set of bug patterns can be assembled by extracting requirements from the RFC, by considering bugs that have been previously reported for other implementations of the protocol, or by using knowledge about the protocol and exercising common sense. An alternative approach is to provide a (typically not so small) general automaton which encodes the set of allowed sequences of exchanged messages during a protocol session, and reports whenever the observed sequence deviates from this set. This approach in principle allows detection of any state machine bug for a protocol, but violations can be difficult to diagnose, and subtle bugs may be shadowed by other bugs. In the paper, we describe how we have assembled automata using both of these approaches.

2) Bug Detection and 3) Bug Validation: In black-box testing, a main problem is to search for input sequences that induce violations of requirements. As in previous works, our technique needs a state machine model of protocol behavior to constrain the search for requirement-violating inputs; such a model can be obtained automatically using model learning, or be available from a different version of the implementation. By adapting existing techniques from model checking, we combine such a model with a bug pattern to obtain a set of candidate requirement-violating inputs. If the model M accurately represents the implementation's behavior, then any input in this set will expose a state machine bug. However, often M is only approximate, e.g., because learning was not given enough time or because M was obtained from an older version of the implementation. Our technique therefore applies the candidate inputs until an actual requirement violation is observed (or a threshold is reached). The overall approach is then an effective search for requirement violations that typically converges after a modest number of tests.

To evaluate our technique, we have applied it to the detection of state machine bugs in three SSH (Secure SHell) server implementations as well as in nine DTLS server and nine DTLS client implementations. The DTLS (Datagram Transport Layer Security) protocol is a variation of TLS over UDP. It is widely used in wireless networks, and is one of the primary protocols for securing IoT applications. Implementing a state machine for DTLS is complicated by the fact that DTLS's main challenge is to support the stateless and unreliable transport of UDP, which allows for messages to arrive out of order and/or fragmented. Unsurprisingly, this can result in various subtle implementation bugs.

Our evaluation shows that our technique, using the same models obtained by model learning and used in prior works of state fuzzing SSH [21] and DTLS [18] server implementations, is able to detect, in most cases within seconds, *all* bugs and security vulnerabilities that these two works have reported. Moreover, our technique automatically produces bug witnesses for all these bugs (something that these works did not do). Finally, our technique is able to detect a significant number of *previously unknown* state machine bugs in the same implementations, both using bug patterns that we have created following the systematic approach we describe in Section V, but also using bug patterns for state machine bugs that prior work has verbally described as a bug in some server implementation but which ocular inspection has failed to spot as a bug in some other server implementation. We hold that these results demonstrate the effectiveness of encoding protocol ordering requirements using finite automata and the additional power that automated bug detection and validation offer to state fuzzing techniques.

In summary, this paper makes the following contributions:

- Proposes a new *fully black-box* technique for detecting vulnerabilities and bugs in implementations of stateful network protocols, whose starting point is a model and a set of protocol state machine bugs encoded as automata. From this point onward, the technique is *fully automated*.
- Provides evidence for our technique's (i) *generality* by applying it to widely-used implementations of two different network security protocols (SSH and DTLS), and (ii) *effectiveness* in automatically detecting vulnerabilities and bugs which were previously unknown and/or missed by ocular inspection of the learned models.
- Describes various ways that a reasonably complete catalogue of state machine bug patterns can be assembled.

Responsible Disclosure: We have reported the bugs to the respective projects complying with their security procedures.

Outline: In the next section we review related work, and then provide background on model learning and the SSH and DTLS protocols (Section III). In Sections IV and VI we give a high-level overview of our general technique, and provide formal definitions of its key concepts and algorithms. We describe our implementation (Section VII) and evaluate its effectiveness (Section VIII). The paper ends with some final remarks.

II. Related Work

Stateful security protocols and their implementations have been thoroughly analyzed for different kinds of vulnerabilities and bugs. In the case of TLS, previously discovered security vulnerabilities include cryptographic attacks (e.g., Bleichenbacher's attack [7] and CBC padding oracle attacks [45]), and the Heartbleed [40] bug in OpenSSL caused by a buffer over-read. State machine bugs include the Early CCS injection vulnerability [46]. An effective framework for testing TLS implementations is TLS-Attacker [41], which lead to the discovery of numerous security vulnerabilities. Detection of state machine bugs in TLS implementations was done using a combination of automated testing and manual source code inspection [6], uncovering previously unknown vulnerabilities.

State Fuzzing: Systematic state fuzzing through analysis of protocol state machines obtained by model learning was pioneered by de Ruiter and Poll [13], uncovering new vulnerabilities. For DTLS, corresponding work was performed by Fiterău-Broştean et al. [18], who developed a state fuzzing framework, DTLS-Fuzzer, based on TLS-Attacker, and applied it to DTLS server implementations discovering several vulnerabilities. Both these works rely on manual analysis of the state machine model in order to find bugs. Not only is this process time-consuming, owing to the (typically large) size of the models, but it may easily miss bugs (as we show in our evaluation). Moreover, it requires deep knowledge of the protocol, and has to be repeated for every new implementation tested. Last but not least, it is very difficult to produce bug witnesses by a visual examination of a model of the implementation. Our work addresses these shortcomings by automating the analysis of models.

State fuzzing has also been applied to other stateful protocols including TCP, SSH, OpenVPN, the 802.11 4-Way Handshake, and IPsec. In the works on OpenVPN [12] and the 802.11 4-way Handshake [34], the detection of state machine bugs is through ocular inspection of models. In the works on TCP [17], SSH [21] and IPsec [24], protocol requirements were encoded in linear temporal logic (LTL) and model checked against the learned models. In contrast, in our work, protocol requirements are supplied as finite automata, which are strictly more expressive than LTL formulas for safety properties [48]. Moreover, our technique validates witnesses of model bugs automatically, whereas in these prior works violating sequences were either not validated, or were applied manually against the implementation.

Another black-box approach for detecting state machine bugs uses model learning to generate models of several different implementations of a protocol, and then compares these models to find discrepancies in them. This approach, which has been applied to TCP [4], MQTT [42], QUIC [15] and 4G LTE [29] protocol implementations, can be regarded as *differential testing*, but with differences detected in models rather than in output to individual test input. DIKEUE [29] is notable among these works because it can *automatically* produce a set of input sequences that induce *deviant behaviours* in the learned models (i.e., input sequences for which different models result in different outputs). However, differential testing approaches require an oracle to determine whether the differences detected in the models are actual bugs, are benign due to underspecification in the standard, or are simply false positives caused by e.g., inaccuracies in the learned models. Existing differential testing approaches, including DIKEUE, perform this classification of the differences manually (e.g., by comparison with the protocol standard). Also, unlike our approach, some of these works [15], [29] do not come with any automated step to check that the differences also exist in the tested implementations and not only in their learned models. This can be a burden, as the number of model differences or deviant-behaviour-inducing input sequences to investigate can be quite large, especially if they are not diverse enough. The DIKEUE paper [29] notes this as an issue to tackle, and proposes a scheme for grouping deviations into diversity classes and limiting the number of elements produced for each class. In general, compared to our approach, the main advantage of an approach based on differential testing of models is that it can be performed without requiring to be supplied with automata that encode state machine bugs. On the other hand, differential testing requires the existence of at least two protocol implementations and, fundamentally, cannot detect bugs that exist in all (pairs of) models which are diffed.

Model Checking: Our approach combines techniques from model learning and model checking [11]. This combination was first proposed as *black box checking* [35] and *adaptive model checking* [23], an adaptation of model checking to a black-box setting, and applied to simple finite-state systems. In our approach, protocol requirements or their violations are represented by automata. (In model checking works, requirements are often represented as a temporal logic formula, which are then converted to automata, as e.g., done in the model checker SPIN [25].) Most model checkers also require a model of the analyzed system, which is typically provided manually, e.g., from some design specification. The intersection of this model with an automaton representing a requirement is then explored for bugs. This approach is often used for analyzing protocols, e.g., to find attacks on the 4G LTE protocol [28], on TCP [32], [31], or on vehicle protocols [27]. Models can also be obtained using whitebox techniques (recall that our approach is black-box) from source code, e.g., using symbolic execution of manually instrumented source code [26], [47] or static analysis [9].

Model Based Testing: Model-based testing [8], [43] is a related black-box approach, which differs from the above works in that the model (typically manually supplied) reflects the *desired* (as opposed to actual) behavior of the system under test (SUT); a goal of testing is to check that the SUT conforms to the model, and/or to use the model to guide test generation. Combinations of model based testing and model learning are surveyed by Aichernig et al. [2]. In contrast to model-based testing works, our approach regards models as reflecting *actual* behavior of the implementation; by using the implementation to validate the requirement-violating input, we can compensate for inaccuracies in the (learned) model of the SUT.

III. Background

A. Model Learning

Model learning is an automated black-box technique which needs to know the input and output alphabets of the SUT. In most cases of learning models of protocol implementations, some form of abstraction is also needed, i.e., a way to map concrete protocol packets to abstract alphabet symbols, and vice versa, in order to interact with the SUT [1]. Most model learning algorithms (e.g., the classic L^* algorithm [3] or the more recent TTT algorithm [30]) produce a deterministic Mealy machine as a model by operating in two alternating phases: *hypothesis construction* and *hypothesis validation*.

During hypothesis construction, sequences of input symbols (Is) are sent to the SUT, observing the sequences of output symbols (Os) that are generated in response. When certain convergence criteria are met, the learning algorithm constructs a *hypothesis*, which is a minimal deterministic Mealy machine that is consistent with the Os. Thus, at this point, for all Is that have been sent to the SUT, the constructed hypothesis H produces the same output as the output sequences Os observed from the SUT. For the remaining inputs, H predicts corresponding outputs by extrapolating from the observed Os. To validate that these predictions agree with the behavior of the SUT, learning then moves to the validation phase, in which the SUT is subject to a conformance testing algorithm which aims to validate that the behavior of the SUT agrees with H. If conformance testing does not find any counterexample, learning terminates and returns the current hypothesis as the inferred model M of the SUT. If a counterexample (i.e., an I on which the SUT and H disagree) is found, the hypothesis construction phase is re-entered to build a more refined hypothesis H' which also takes that I (and its observed output) into account. If the loop of hypothesis construction and validation does not terminate, this indicates that the behavior of the SUT cannot be captured by a deterministic Mealy machine whose size and complexity is within reach of the employed learning algorithm. Still, even in these cases, the last constructed hypothesis can be used as an *approximate model* of the SUT.

B. *Secure Shell (SSH)*

The SSH protocol is a client-server protocol which enables accessing network services securely over an unsecured network. Our work is concerned with version 2 of SSH, also known as SSH-2. The protocol is structured in three layers. The Transport layer is responsible for securing communication between the client and the server using symmetric keys. The Authentication layer operates over the Transport layer and enables the client to authenticate with the server. On top of the Authentication layer operates the Connection layer, mediating access to network services. These layers correspond to the three phases of a typical SSH session, for which an illustration is shown in Fig. 1.

Key Establishment: The client and server begin their interaction using the SSH transport layer protocol [51] by establishing the keys that will be used to secure the higher layers. The two sides begin by sharing their preferences for cryptographic algorithms, which they communicate by exchanging *KEXINIT* (*KI*). Using the negotiated algorithm, they then perform the actual key exchange in order to establish fresh session keys. A Diffie-Hellman key exchange, which all implementations are required to support, involves the client sending *KEX30* (*K30*) to which the server responds by *KEX31* (*K31*). Key exchange ends with the two sides exchanging *NEWKEYS* (*NK*). With this message, a side informs the other that it will deploy the generated session keys and use them to encrypt follow-up messages. At this point, the client requests the user authentication service by sending *SERVICE_REQUEST_USERAUTH* (*SReqAuth*). If the server grants the request, it responds with *SERVICE_ACCEPT* (*SAcc*), allowing the authentication phase to begin. In SSH, key exchange can be performed later in order to renew session keys, in a process we will refer to as *rekey*.

Authentication: To authenticate, the client sends a user authentication request message with the desired authentication method, and method-specific arguments [49]. Implementations *must* support authentication using public keys (*UAReqPK*). The server checks the request and responds accordingly with *USERAUTH_SUCCESS* (*UASucc*) or *USERAUTH_FAILURE* (*UAFail*). Upon successful authentication, the protocol can enter its last phase.

Connection: The authenticated client is now able to open a channel with the server, over which it can run various services such as a remote terminal. The client first sends *CHANNEL_OPEN* (*COpen*) to request opening a channel with the server. On receiving a positive response (*COpenSucc*), the client can request a remote terminal to be run on this channel by sending *CHANNEL_REQUEST_PTY* (*CReqPty*) and receiving *CHANNEL_SUCCESS* (*CSucc*) if the request is granted. Once operation on the channel is over, the client can close it by issuing *CHANNEL_CLOSE* ($CClose_c$), to which the server responds with its own *CHANNEL_CLOSE* ($CClose_s$). SSH allows for multiple channels to be opened simultaneously [50].

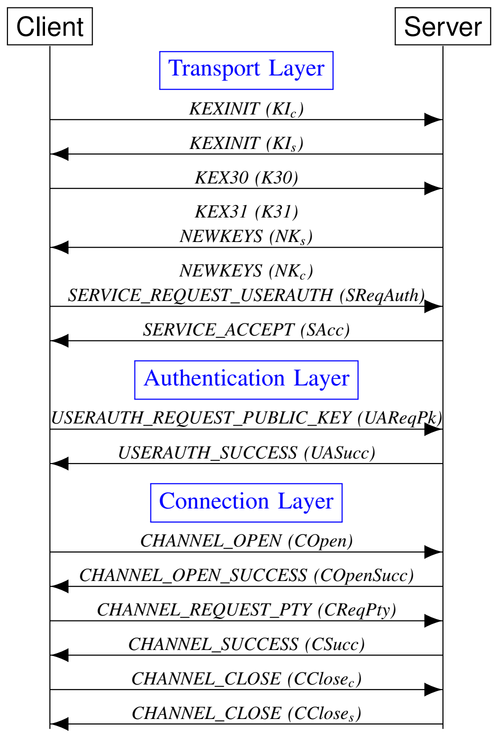


Fig. 1: Sequence diagram of an SSH session using Diffie Hellman key exchange and public key authentication. The Transport and Authentication layer exchanges are mandatory.

C. *Datagram Transport Layer Security (DTLS)*

DTLS is an adaptation of the cryptographic Transport Layer Security (TLS) [37] protocol for use with Datagram-based transport protocols, such as UDP. Our work is concerned with DTLS version 1.2 [38], noting that at the time of writing, version 1.3 [39] has only recently been standardized. The DTLS protocol is divided into two parts, a Record Protocol and a Handshake Protocol. The Record Protocol encrypts messages to be transmitted using symmetric key encryption, based on a pre-negotiated secret between the peers. This secret is negotiated using the Handshake Protocol when the connection is first established. DTLS extends the TLS protocol to allow for messages to arrive out of order or fragmented.

Figure 2 shows the flow of a DTLS handshake. The first flight of the handshake begins with the client sending a *ClientHello (CH)* message to the server. Note that we will introduce all messages with their full names, but we will subsequently refer to them with their abbreviations, which are shown in parentheses. This message contains, among other things, a random nonce and a list of the cipher suites supported by the client. In DTLS, a server may respond to this with a *HelloVerifyRequest (HVR)* message, which contains a stateless cookie. The client must respond to this message with a second *CH* message, containing the same cookie. This cookie exchange is there to help prevent Denial of Service attacks from spoofed IP addresses.

After the cookie exchange, the server starts flight 4 by sending a *ServerHello (SH)* message. This message contains the cipher suite chosen by the server from those supported by the client. If the chosen cipher suite requires a certificate to be used for the key exchange, the server sends a *Certificate* ($Cert_s$)

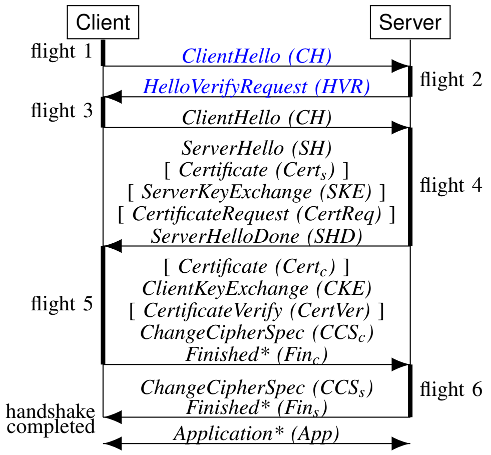


Fig. 2: TLS/DTLS handshake. Messages unique to DTLS are colored blue, optional messages are in [square brackets] and encrypted messages are marked by an asterisk*. Inside parentheses, we show the abbreviations we will use.

message carrying the certificate and public key. Some cipher suites, such as Diffie-Hellman, require additional information which the server then provides in a *ServerKeyExchange (SKE)* message. The server may also request client authentication by sending a *CertificateRequest (CertReq)* message. This message contains a list of supported certificate types. Finally, the server marks the end of this flight by sending the *ServerHelloDone (SHD)* message.

The next flight contains the client's responses to the server. If the server requested client authentication, the client sends a *Cert$_c$* message. This message must conform to one of the certificate types requested by the server. If it does not, the client must send an empty *Cert$_c$* message. After this, the client creates a premaster secret, which it communicates to the server with a *ClientKeyExchange (CKE)* message. If the client sent a *Cert$_c$* containing a certificate, the client now sends a signature in the *CertificateVerify (CertVer)* message.

After sending the *CKE* message (and the optional *CertVer*), the client deploys the cipher suite. At this point the client sends a *ChangeCipherSpec (CCS$_c$)* to inform the server that all future messages will be encrypted. Finally, the client marks the end of the flight by sending its *Finished (Fin$_c$)* message. This message contains a *digest*: a hash of all handshake messages sent to and received by the client, excluding those involved in the cookie exchange. If this digest does not match with the messages the server has sent and received, the handshake is terminated by the server. Otherwise, the server also deploys the cipher suite and sends a *CCS$_s$*, followed by a *Fin$_s$*. This last message contains a digest of the messages sent to and received by the server. At this point, the handshake is established. *Application (App)* data can now be exchanged by the two peers.

It is possible for a client and server to renew cryptographic key material, using a process known as *renegotiation*. A client can request a renegotiation at any point by sending a *CH* message. If the server accepts the request, it will respond with a *SH* (or a *HVR*, if it wishes to do another cookie exchange). A server may only request renegotiation after successful completion of a handshake, by sending a *HelloRequest (HR)*. If the client accepts the request, it will respond with a *CH*, starting a new handshake.

IV. Bug Detection Framework

In this section, we describe a general technique to detect vulnerabilities and bugs in implementations of protocols which are explicitly or implicitly defined via state transitions. We refer to the implementation to analyze as the *system under test (SUT)* and treat it as a black box. To interact with the SUT, support from a test harness is required. Sometimes, this protocol-specific test harness is available in a model learning tool for the protocol.

A. Obtaining the SUT's State Machine Model

Our technique takes as input the model of the SUT's state machine. Sometimes, such a model is already available and can be supplied directly. In these cases, the model learning step we describe here is skipped. However, in most cases, the model is constructed automatically using model learning. We will illustrate model learning and the supporting infrastructure that it requires using an example from DTLS.

DTLS-Fuzzer [16], the tool we use for learning models of DTLS implementations and will describe in Section VII, is often effective in constructing exact and relatively small models. For example, for OpenSSL 1.1.1b DTLS servers, learning converges in five to six hours, and the constructed models contain only between 14 and 22 states (depending on whether authentication is disabled, optional, or required). But there also exist DTLS implementations for which learning does not converge. An example of such an implementation is JSSE. After running DTLS-Fuzzer for two days on a JSSE 12.0.2 server using ECDH, the hypothesis that has been constructed by DTLS-Fuzzer consists of 124 states, and learning has still not converged. We stress that the technique we describe in this section will perform its steps on *this big and approximate model* of a JSSE 12.0.2 server. However, because a Mealy machine with 124 states is unreadable, Fig. 3 shows a reduced version of it, consisting of only twelve states. Let us explain how to read this Mealy machine. Its edges are labeled with elements of the alphabet(s) on two sides of a slash ('/'), with inputs on the left and outputs on the right. At its initial state (0), the server either accepts application data (*App*) and does not output any response, or accepts a *ClientHello (CH)* message and responds with a *HelloVerifyRequest*. At state 1, besides *App*, the server also accepts a *CH* and then replies with a sequence of five messages (*SH*, *Cert$_s$*, *SKE*, *CertReq*, and *SHD*, in this order). The remaining states are similar, but we also use a shorthand notation (on the self-edges of states 6, 7, and 11) to denote a *union* of inputs and corresponding outputs. We can also notice the path that correctly completes the handshake, colored in blue, starting from the initial state and ending in state 7. The remaining paths, colored red, will be explained in Section IV-C.

B. Encoding Vulnerabilities and Bugs

Given a model of the SUT, M, the idea is to search M for paths that violate the security and correctness requirements of the protocol or look suspiciously like bugs. There are three issues that need to be addressed to realize this simple idea and make it effective in practice: (1) How do we capture what a requirement violation or bug looks like in M? (2) How do we efficiently search for buggy paths in M? (3) How do

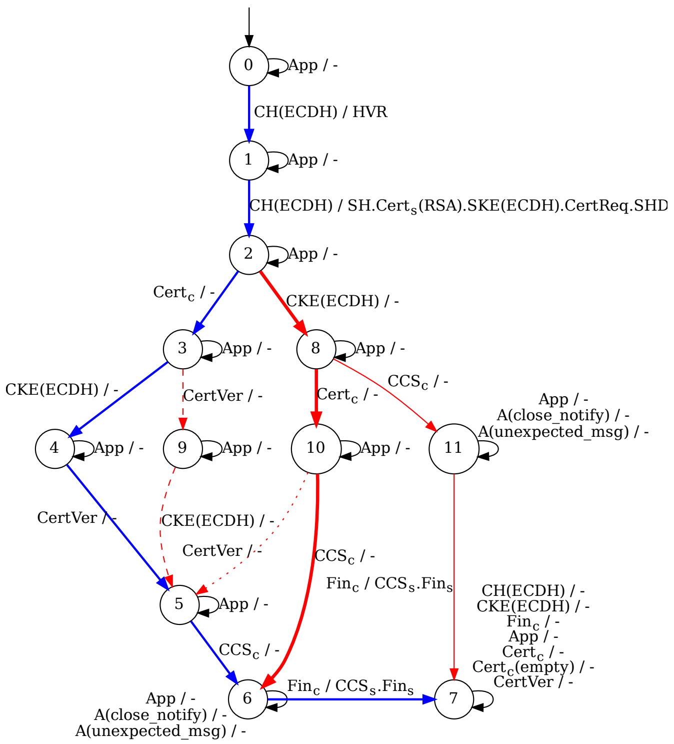


Fig. 3: Reduced model of a JSSE 12.0.2 DTLS server.

we demonstrate that the actual SUT contains bug(s) that are triggered when executing the code that corresponds to a buggy path in M? Let us first answer the first of these questions.

Every network protocol specifies security and correctness requirements in its specification (e.g., in its RFC). Naturally, we can go over this specification and extract (a complete set of) these requirements. The second input that our technique gets supplied is a catalogue of bug patterns, each of them encoded as an automaton that accept sequences of messages that expose the state machine bug. Let us illustrate this encoding by an example from DTLS. Assume that we are testing a DTLS server which is configured to require a valid certificate from the client. It is obviously an error —in fact, a vulnerability— for the server to request a certificate (with a *CertReq* message) and then enter the phase that completes the handshake without receiving a certificate message from the client ($Cert_c$). To make this more concrete, recall from Fig. 2 that the server sends a *CertReq* in flight 4 and enters handshake completion by sending a CCS_s message (flight 6). We can therefore capture this vulnerability with an automaton that accepts sequences of messages that expose it. Such an automaton is shown in Fig. 4.

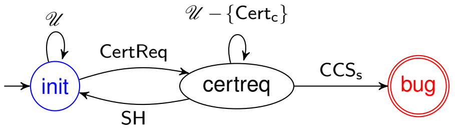


Fig. 4: DFA capturing the vulnerability of completing a handshake without a Certificate message from the client.

Let us explain this Deterministic Finite Automaton (DFA) and also the notation that we use. The possible paths from the initial to the final "bug" state express the vulnerability we explained above: there is a *CertReq* followed by a CCS_s message from the server, without any $Cert_c$ message from the client in between. We use the symbol $\mathscr{U}$ for what is often known as the set of all "other" symbols of the alphabet Σ of a DFA (i.e., all symbols except those that are shown as labels in the other outgoing transitions from a state). So, for the initial state $\mathscr{U}$ denotes $\Sigma - \{\mathsf{CertReq}\}$, and for the middle state $\mathscr{U}$ denotes $\Sigma - \{\mathsf{SH}, \mathsf{CCS_s}\}$. By including a SH transition back to the initial state, we allow our DFA to avoid false alarms (due to renegotiation). In DTLS, a client can restart the handshake process at any point by sending a *ClientHello* message to the server. Some DTLS servers will respond with a *HVR* at this point and the handshake will be restarted. However, there also exist DTLS implementations that skip repeating the cookie exchange step and restart the handshake from flights 3 and 4 instead (cf. Fig. 2). It is therefore safer and more uniform to use the *second* message that a server sends (SH) to denote that the handshake is restarting, and may be completed correctly with another *CertReq* from the server.

Related to the fact that all our automata are *deterministic*, we note that for all symbols in Σ for which there is no outgoing transition from a state, there are implicit transitions for these symbols to a "sink" state, which we do not show in order not to clutter the pictures. For example, for the automaton in Fig. 4 there are implicit transitions out of the final "bug" state for all symbols in Σ, and there is also a transition for $Cert_c$ out of the "certreq" state to the sink state, signifying that if a certificate is sent from the client, then we do not reach the accepting state of our DFA (i.e., there is no bug).

The second vulnerability we capture with a DFA (Fig. 5) is similar. In a server configured to require a certificate from the client, a client is authenticated (i.e., handshake is completed) with the client sending a certificate ($Cert_c$) but *without* sending a *CertificateVerify* to the server. Note that this is a serious security hole. It may indicate that an attacker can authenticate using someone else's certificate without proving ownership of the certificate with a *CertVer*.

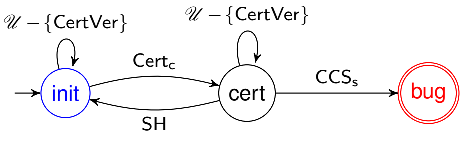

$\mathscr{U} - \{\mathsf{CertVer}\}$
$\mathscr{U} - \{\mathsf{CertVer}\}$
Cert_c
init
cert
SH
CCS_s
bug

Fig. 5: DFA capturing the vulnerability of authenticating a client without it sending a CertificateVerify message.

Besides vulnerabilities, note we can also employ DFAs to capture other kinds of protocol state machine errors. For example, the DFA of Fig. 6 captures the bug that, in a valid handshake with a server that uses ECDH, the server cannot consume a *CKE* before the $Cert_c$ message from the client. In fact, we can generalize the CKE(ECDH) edge in the DFA of Fig. 6 to work for all key exchange algorithms that a server supports and which require the client to provide a

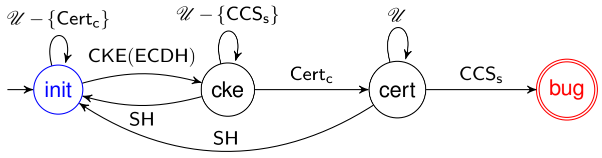
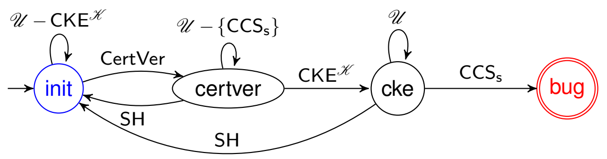

$\mathscr{U} - \{\mathsf{Cert_c}\}$
$\mathscr{U} - \{\mathsf{CCS_s}\}$
$\mathscr{U}$
CKE(ECDH)
init
cke
Cert_c
cert
CCS_s
bug
SH
SH

Fig. 6: DFA capturing a server completing handshake, but consuming a ClientKeyExchange before a client Certificate.



$\mathscr{U} - \mathsf{CKE}^{\mathscr{K}}$
$\mathscr{U} - \{\mathsf{CCS_s}\}$
$\mathscr{U}$
init
CertVer
certver
$\mathsf{CKE}^{\mathscr{K}}$
cke
$\mathsf{CCS_s}$
bug
SH
SH

Fig. 7: DFA capturing a server completing handshake, but consuming a CertificateVerify before ClientKeyExchange.

certificate. For example, we can specify as label of that edge the set $\{\mathsf{CKE(DH)}, \mathsf{CKE(ECDH)}, \mathsf{CKE(RSA)}\}$. We will use the shorthand $\mathsf{CKE}^{\mathscr{K}}$ to refer to such a set.

Our last DFA example (Fig. 7) is similar. It captures another case of invalid sequencing of messages: a server completing the handshake while consuming a *CertVer* before the *CKE* message from the client.

Having explained how to encode requirements into automata, let us consider how to obtain such requirements in the first place. There are several possibilities: DFAs can be directly constructed and refined from i) specific requirements in a protocol's specification (e.g., "The $Cert_c$ message is sent by the client only if the server requests a certificate." [37, p. 55]), ii) from any previously reported state machine bug for protocol implementations, or iii) by using our knowledge about the protocol and exercising common sense. We note that the set of DFAs capturing requirements, vulnerabilities, and common bugs of a protocol implementation can be specified offline and incrementally. In addition to this, in Section V we present a complementary approach, which systematically constructs bug patterns from a description of the allowed sequences of exchanged messages in a protocol session.

C. Detecting Bugs in the Model of the SUT

We are at the point where we have a model M of a SUT in the form of a Mealy machine, and we also have a set of bug patterns expressed as DFAs. How do we find whether M contains state machine bugs or not? First, note that the two formalisms (Mealy machines and DFAs) look similar but technically they are different. Mealy machines specify input-output relations of the symbols in their transitions, while DFAs define a language: the set of words that the DFA accepts. Also, note that searching or enumerating all paths in M will not work. Most models, besides having many paths due to many states, may even have an infinite number of paths. For example, notice the self edges in all states of Fig. 3. We need to do something more effective than a search.

In a nutshell, what our technique does in this step is convert the Mealy machine model of the SUT to a DFA and intersect it with each of the DFAs describing the bug patterns. We will describe the algorithms of this step formally in Section VI. By applying those algorithms on our running example, the JSSE 12.0.2 server, our technique will detect that: i) the vulnerability described by the DFA of Fig. 4 (completing a handshake without a *Cert* message) is present in the path $0{\to}1{\to}2{\to}8{\to}11{\to}7$; ii) the vulnerability described by the DFA of Fig. 5 (authenticating a client without a *CertVer*) appears in the path $0{\to}1{\to}2{\to}8{\to}10{\to}6{\to}7$; iii) the bug captured by the DFA of Fig. 6 (*CKE* before $Cert_c$) appears in the path $0{\to}1{\to}2{\to}8{\to}10{\to}5{\to}6{\to}7$; and iv) the bug captured by the DFA of Fig. 7 (*CertVer* before *CKE*) appears in the path $0{\to}1{\to}2{\to}3{\to}9{\to}5{\to}6{\to}7$. Also, our technique will discover that the *CertVer* before *CKE* bug also exists on the 0 to 7 path which passes via the edges $2{\to}8$ and $8{\to}10$ (thick red lines).

D. Validating the Model Bugs in the SUT

Alas, automatically detecting vulnerabilities and bugs in the model of the SUT does not suffice. As explained in Section IV-A, learning may not have converged, and the inferred model may be approximate. So, the previous step may report bugs that do not exist in the SUT, and of course we want to filter them out automatically. But even in the cases where the model is precise, reporting a bug without providing a test case that exhibits it makes reproducing and fixing the bug very hard. Thus, the last step of our technique uses the DFA produced by the DFA intersection step sketched in the previous section and the test harness to construct sequences of protocol packets that it then runs against the SUT in order to validate the presence of the bug in the actual system and return them as *bug witnesses* in the reports to the developers. We will present this step as Algorithm 1 in Section VI. But we note that the existence of this step explains why our technique will not report any false positives even if the SUT's model is approximate. Our technique will automatically validate all bugs on the SUT.

V. Systematic Assembly of Bug Patterns

The effectiveness of our technique depends on the set of bug patterns. In Section IV-B, we described how to construct bug patterns to detect violations of ordering requirements. Such bug patterns can be extracted from specific requirements in a protocol's specification, from any previously reported state machine bug for protocol implementations, or from any other (violation of a) property of interest. In practice, these sources suffice for producing a rather complete set of bug patterns. In some cases, it is desirable to have a complementary recipe for systematically deriving bug patterns from a general description of allowed protocol behavior, e.g., when there is no source of previously reported bugs, or to make the set of bug patterns more complete. In this section, we will describe one such general recipe. We used it for creating bug patterns for DTLS clients, since there we started from an empty set of bug patterns, and also for the DTLS and SSH servers in order to add additional bug patterns that we might have missed.

Most network protocols constrain the kinds of messages that may be sent and received at each state, and also their order for establishing a connection. For example, the RFC for the DTLS handshake protocol specifies that messages must be exchanged in the sequence shown in Fig. 2. Furthermore, it specifies the following two requirements: (i) a client must *never* send a message which deviates from the given sequence, and (ii) upon receiving an unexpected message from the server, the client should end the session, either by ignoring subsequent server messages or by responding to them by $Alert_c$. Combining these two requirements with the sequence diagram of Fig. 2, we can construct the *general* bug pattern shown in Fig. 8.

In this DFA, the allowed sequence of messages corresponds to the horizontal path from the initial state to the final state. The states in this path are all non-accepting, since no requirement is violated by remaining in this sequence of states. In fact, there are typically several allowed sequences, depending on the key

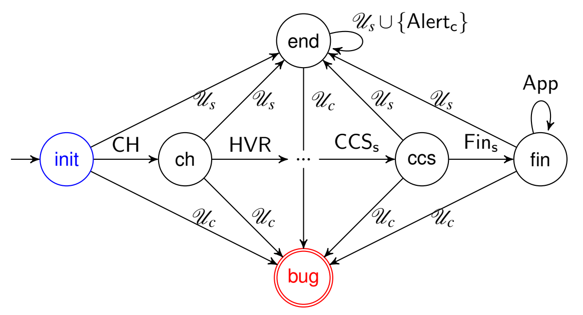


Fig. 8: (Simplified view of) DFA capturing violations of DTLS handshake sequencing. $\mathscr{U}_c$ and $\mathscr{U}_s$ denote all "other" client, respectively, server messages.

exchange algorithm, client authentication setting, etc., turning the "middle path" into a graph with several outgoing transitions from some states. For the sake of readability, Fig. 8 shows only one allowed sequence of transitions. The DFA also contains two additional states, and transitions that capture violations of the above RFC requirements (i) and (ii).

- Violations of requirement (i) are captured by the transitions from a state in the "middle path" to the bottom accepting bug state. They are labeled by $\mathscr{U}_c$, which denotes the set of disallowed client messages for which there are no other outgoing transitions. Such an $\mathscr{U}_c$-transition will be taken whenever the client sends a deviating message, and the bug pattern will report a requirement violation.
- Violations of requirement (ii) are captured by the transitions to the top end state, together with the transition from that state to the accepting bug state. The transitions to the end state are labeled by $\mathscr{U}_s$, denoting the set of deviating server messages. Such a transition will be taken whenever the server sends a deviating message; if the client thereafter responds with anything else than an *Alert*$_c$ message, a requirement violation occurs, which is captured by the transition labeled $\mathscr{U}_c$ to the bug state.

General bug patterns, like the one in Fig. 8, capture most protocol state machine bugs that implementations contain. Thus, in principle they suffice. However, they have two drawbacks: 1) their size is big and bug detection using them can be slow; 2) whenever a violation is found, besides a bug witness they provide no other indication of what specific protocol requirement is violated. For these reasons, when starting from such a general bug pattern, it is a good idea to add *specific* bug patterns to the bug pattern catalogue whenever the general one(s) detect some bug in an implementation. These specific bug patterns are tried first (and detect bugs very fast, as we will show in our evaluation) and only then the more general bug patterns are used. For example, we have identified four general classes of such violations for DTLS, corresponding to four types of specific bug patterns: an expected message is not received (e.g., the missing *Cert*$_c$ vulnerability), an expected message is received in the wrong order, before another required message (e.g., *CertReq* before *Cert*), an expected message is received multiple times (e.g., multiple *CertReq*), and an unexpected message is generated (e.g., unexpected *CH*). Thus, even without any knowledge about prior bugs reported for a protocol, a quite extensive bug pattern catalogue can be assembled starting from a general bug pattern, like the one in Fig. 8, and creating specific bug patterns for classes of common state machine bugs.

VI. Algorithms

We provide formal definitions of concepts and formalisms used in Section IV, and of the algorithms we employ.

Mealy Machines: Models of protocol implementations are assumed to be given as Mealy machines describing how the implementation generates output messages in response to input messages. Mealy machines are finite state automata with finite alphabets of input and output symbols. They are widely used to model the behavior of protocol entities [33], [10]. Starting from an initial state, they process one input symbol at a time. Each input symbol triggers the generation of a sequence of output symbols and brings the machine to a new state, from which the next input symbol can be processed. An example Mealy machine was shown in Fig. 3.

Formally, a *Mealy machine* $\mathscr{M}$ is a tuple $(I, O, Q, q_0, \delta, \lambda)$, where I and O are alphabets of *input* and *output symbols*, respectively, Q is a set of *states*, containing an initial state q_0, where $\delta : Q \times I \to Q$ is a *transition function*, which for each state $q \in Q$ and input symbol $i \in I$ gives a next state $\delta(q,i)$, and where $\lambda : Q \times I \to O^*$ is an *output function*, which for each state $q \in Q$ and input symbol $i \in I$ gives an output $\lambda(q,i)$, which is a (possibly empty) sequence of symbols.

DFAs: Bug patterns are formulated as DFAs that characterize the set of sequences of symbols that exhibit the described bug. Formally, a *deterministic finite automaton* (*DFA*) over an alphabet Σ is a tuple $\mathscr{A} = (\Sigma, Q, q_0, \Delta, F)$, where Σ is a finite set of *symbols*, Q is a set of *states*, containing an initial state q_0, where $\Delta : Q \times \Sigma \to Q$ is a *transition function*, which for each state $q \in Q$ and symbol $l \in \Sigma$ gives a next state $\Delta(q,l)$, and where $F \subseteq Q$ is a set of *accepting states*. The transition function is extended from symbols to sequences of symbols in the standard way, by defining

$$\begin{aligned} \Delta(q,\varepsilon) &= q \\ \Delta(q,wa) &= \Delta(\Delta(q,w),a) \end{aligned}$$

where ε is the empty sequence. A sequence of symbols w is *accepted* iff $\Delta(q_0,w) \in F$. The *language* accepted by $\mathscr{A}$, denoted $\mathscr{L}(\mathscr{A})$, is the set of accepted sequences of symbols.

In DFAs for our bug patterns, the alphabet Σ is the set of input and output messages received and generated. Drawing conventions for bug patterns were described in Section IV-B.

Transforming a Mealy Machine to a DFA: In order to combine a protocol model $\mathscr{M}$ of the SUT with bug patterns, we must first transform $\mathscr{M}$, which is in the form of a Mealy machine, to a DFA $\mathscr{A}_{\mathscr{M}}$. We do this in the natural way, by letting $\mathscr{A}_{\mathscr{M}}$ accept all sequences of inputs and outputs that can be exhibited by letting $\mathscr{M}$ first read input, then produce the corresponding sequence of outputs, then read input, etc. Formally, a Mealy machine $\mathscr{M} = (I, O, Q, q_0, \delta, \lambda)$ is transformed to the DFA $\mathscr{A}_{\mathscr{M}} = ((I \cup O), Q_M, q_0, \Delta_M, Q_M)$, where $Q_M = Q \cup Q_{aux}$, i.e., Q_M extends Q by a set of auxiliary states Q_{aux}, which is defined as follows: For each $q \in Q$ and $i \in I$ with $\lambda(q,i) \neq \varepsilon$, letting $\lambda(q,i) = o_1 \cdots o_n$, the set Q_{aux} contains the set of states $\{p_{q,i,o_1 \cdots o_k} : 0 \leq k < n\}$. The transition function Δ is defined as follows: for $q \in Q$ and $i \in I$, whenever $\lambda(q,i) = \varepsilon$, then

$$\Delta(q,i) = \delta(q,i)$$

and whenever $\lambda(q,i) = o_1 \cdots o_n$ with $1 \leq n$, then

$$\begin{aligned}
\Delta(q,i) &= p_{q,i} \\
\Delta(p_{q,i,o_1\cdots o_{k-1}}, o_k) &= p_{q,i,o_1\cdots o_k} \text{ for } 1 \leq k < n \\
\Delta(p_{q,i,o_1\cdots o_{n-1}}, o_n) &= \delta(q,i)
\end{aligned}$$

As an illustration, the transition from state 6 to state 7 in the model of Fig. 3 is transformed to the following transitions. Note that all the states are accepting.

$$\begin{tikzcd}
6 \arrow[r, "\mathsf{Fin_c}"] & p_{6,\mathsf{Fin_c}} \arrow[r, "\mathsf{CCS_S}"] & p_{6,\mathsf{Fin_c},\mathsf{CCS_S}} \arrow[r, "\mathsf{Fin_S}"] & 7
\end{tikzcd}$$

Intersecting the DFA of the Model with each Bug Pattern: As the next step, for each bug description $\mathscr{A}_b$, we construct the DFA $\mathscr{A}(\mathscr{A}_\mathscr{M}, \mathscr{A}_b)$ which accepts the intersection of $\mathscr{L}(\mathscr{A}_\mathscr{M})$ and $\mathscr{L}(\mathscr{A}_b)$. We will denote this DFA by $\mathscr{A}_\cap$. This DFA accepts the set of sequences that (according to the model) can be performed by the implementation, and which expose the bug specified by $\mathscr{A}_b$. Letting $\mathscr{A}_b = (\Sigma, Q, q_0, \Delta, F)$ and $\mathscr{A}_\mathscr{M} = (\Sigma, Q_M, q_{0M}, \Delta_M, Q_M)$, the DFA $\mathscr{A}_\cap$ can be constructed as the cross-product $(\Sigma, Q_M \times Q, (q_{0M}, q_0), \Delta', Q_M \times F)$, where $\Delta'((q_M, q), l)$ is defined as $(\Delta(q_M, l), \Delta(q, l))$ for each $q_M \in Q_M$, $q \in Q$ and $l \in \Sigma$. This cross-product can then be polished by removing unreachable states and states from which no accepting state is reachable. If no states remain, i.e., $\mathscr{L}(\mathscr{A}_\cap) = \emptyset$, then (if the model $\mathscr{M}$ is accurate) the implementation does not exhibit the bug represented by $\mathscr{A}_b$. Otherwise, i.e., if $\mathscr{L}(\mathscr{A}_\cap)$ is non-empty, we extract test sequences from $\mathscr{A}_\cap$ as follows.

Extracting and Applying Bug-Exposing Test Sequences: In the final step, sequences that, according to $\mathscr{A}_\cap$ will expose bugs, are extracted and used to construct test cases, which are applied to the SUT. Recall that $\mathscr{A}_\cap$ accepts the set of sequences that (according to the model) can be performed by the SUT, and which expose the bug specified by $\mathscr{A}_b$. For a sequence w of input and output symbols, let $inputs(w)$ denote its subsequence of input symbols. We hope that sequences w in $\mathscr{L}(\mathscr{A}_\cap)$ will expose the bug as follows: Apply the sequence $inputs(w)$ to the SUT. Let w_{obs} be the sequence of observed input and output messages, where the outputs appear after their triggering input symbol and before the next input. If $w_{obs} \in \mathscr{L}(\mathscr{A}_b)$, then we have a witness for the bug, consisting of the input $inputs(w)$ and the observation w_{obs}. If the model faithfully represents the possible behaviors of the SUT, then any sequence w in $\mathscr{L}(\mathscr{A}_\cap)$ will expose the bug. If not, then only some (and possibly none) of them will expose the bug. Therefore, we iteratively generate sequences in $\mathscr{L}(\mathscr{A}_\cap)$ and apply them to the SUT until a bug is found, or until some threshold is reached. In our implementation, we generate accepting sequences of increasing length, using breadth-first search (BFS) from accepting states. We restrict the generation to sequences that cause the DFA to visit each state at most K times for some suitable K. One advantage of using BFS as the generation strategy is that the SUT's response to short input sequences, which visit each state only a few times, is more likely to be accurate in a model obtained from model learning. Another advantage is that shorter sequences are easier to understand when used as bug witnesses.

Algorithm 1 shows a realization of this testing strategy. It maintains a work queue WQ of pairs (q,w) of states and sequences, such that $\Delta(q,w)$ is accepting, initialized by the pairs (q,ε) where q is accepting. Each iteration of the loop

Algorithm 1: Extraction and application of test sequences.

```
Inputs: A DFA $\mathscr{A}_\cap = (\Sigma, Q, q_0, \Delta, F)$;   // intersection between
        A bug pattern $\mathscr{A}_b$;                              // SUT's model and $\mathscr{A}_b$
        A threshold $K$ for $maxStateVisits$.
Result: An exposed bug in the SUT with a witness
        $(inputs(w), w)$, if one exists within the threshold $K$.
$WQ := \{(q,\varepsilon) : q \in F\}$;
while $WQ$ is nonempty do
    $(q,w) :=$ dequeue($WQ$);
    if $q = q_0$ then
        apply $inputs(w)$ as test input to SUT;
        let $w_{obs}$ be the sequence of observed messages;
        if $w_{obs} \in \mathscr{L}(\mathscr{A}_b)$ then return $(inputs(w), w_{obs})$;
    foreach $(q' \in Q, l \in \Sigma)$ such that $\Delta(q', l) = q$ do
        if $maxStateVisits(q', lw, \mathscr{A}_\cap) \leq K$ then
            insert $(q', lw)$ into $WQ$
```

extracts a pair (q,w) from WQ. If q is the initial state, then w is a sequence accepted by $\mathscr{A}_\cap$, which should be tested by supplying the sequence of inputs of w to the SUT. If the observed sequence of messages w_{obs} is accepted by $\mathscr{A}_b$, a witness for the bug has been found and the algorithm returns. Otherwise, the current loop iteration continues by constructing the pairs of form (q', lw) such that q is reached on symbol l from q', implying that $\Delta(q', lw)$ is accepting, and inserting into WQ those pairs for which the corresponding accepting run visits each state at most K times. We use $maxStateVisits(q, w, \mathscr{A})$ to denote the maximal number of times that some state is visited on a run of $\mathscr{A}$ on input w, starting from state q.

VII. Implementation

Let us now present how the framework and the algorithms of Sections IV and VI are realized. To analyze a SUT for vulnerabilities and bugs, our implementation needs a Mealy machine model of the SUT's behavior, which can either be provided by the user or can be learned. For model learning we need an abstraction mechanism that maps concrete protocol packets to abstract alphabet symbols, and vice versa. The implementation of this mechanism also serves as our test infrastructure, leveraging its ability to translate sequences of input symbols into concrete packets, send them to the SUT, and generate corresponding output symbols from the response.

A. Learning and Testing SSH Servers

To learn models for SSH server implementations we used the publicly available artifact [20] of prior work on model learning and model checking SSH implementations [21]. That work checked the learned models against properties that were extracted from the protocol's RFC and formulated in LTL. We could reuse the artifact's infrastructure with only minor changes that improve its conformance testing algorithm. Its infrastructure comes equipped with an implemented abstraction, or MAPPER, responsible for mapping from symbols to packets and vice versa. The MAPPER operates as a test harness. It receives (e.g., from the learner) input symbols over a socket connection, sends corresponding packets to the SUT and returns output symbols generated by abstracting the responses. The MAPPER supports input and output symbols for common types of SSH messages; refer to Fig. 1. In particular, since user authentication requests are parameterized by the method and validity of credentials, separate symbols are defined for each parameter combination. The MAPPER stores in variables the sender-side and receiver-side session keys generated after every Diffie-Hellman exchange. It respectively deploys the sender-side keys after sending NK_c to the SUT, and the receiver-side keys after receiving NK_s from the SUT. The algorithms used are those negotiated after the last *KEXINIT* exchange. The MAPPER also keeps track of the number of open channels. To make learning easier, it allows at most one channel to be open at a time. Upon receiving a *COpen* input symbol when a channel is already open, it responds by the special output symbol *CMax*, without relaying the corresponding message to the SUT. Similarly, it responds with the symbol *CNone* upon receiving a channel input symbol other than *COpen*, when there is no open channel.

We defined bug patterns for each of the eleven correctness properties for SSH servers that prior work [21] encoded in LTL. We also extended this set with five new bug patterns. The first three of them check that state does not change in the Connection and Authentication layers following a rekey (i.e., the rekey procedure should be transparent). For example, Auth Fail After Rekey is used to check that authentication is still possible after rekey if valid credentials are provided. The fourth bug pattern (Missing *NEWKEYS*) captures key exchanges and rekeys completed without an NK_s. The bug was identified in the witnesses generated for other bug patterns, based on which a corresponding pattern was constructed. Finally, the last bug pattern (Missing *SERVICE_REQUEST_USERAUTH*) captures the server performing authentication without requesting the authentication service (*SReqAuth*). In addition, we constructed a general bug pattern for the SSH transport layer, similar to the general bug pattern of Fig. 8 for DTLS clients. Applying this general bug pattern enabled us to derive the Missing *SERVICE_REQUEST_USERAUTH* bug pattern.

B. Learning and Testing DLTS Servers and Clients

For learning and testing DTLS implementations, we used DTLS-Fuzzer [16]. DTLS-Fuzzer supports input and output symbols for common types of DTLS messages, such as the messages in Fig. 2. Some of these symbols (e.g., *CH*, *SH*, *SKE*, *CKE*) are parameterized by the key exchange algorithm used, *Cert* by the type of certificate contained, etc. In order to translate between symbols and concrete packets, DTLS-Fuzzer maintains the state of its interaction with the SUT. For each input symbol, DTLS-Fuzzer creates the corresponding DTLS message and uses variables in its state to configure message fields. The message is then packaged and sent to the SUT. From the SUT's reply, DTLS-Fuzzer assembles DTLS messages, generates corresponding output and updates its state again. A key state variable is *cipherState*, which stores the symmetric keys used for encryption/decryption. This variable is initially set to null meaning there is no encryption/decryption, and updated on sending/receiving *CKE* based on the key material exchanged in the last *CH–SH* exchange. The keys are deployed whenever *CCS* is sent. Other variables regard the maintenance of the digest, cookie, etc. We extended DTLS-Fuzzer to also handle DTLS clients by reusing existing functionality for the input and output symbols that clients and servers have in common.

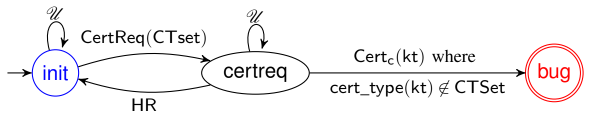


Fig. 9: DFA capturing the Wrong Certificate Type bug.

To check whether a SUT exhibits a bug, our implementation takes a formulation of the bug as a DFA $\mathscr{A}_b$, from a file in DOT format. In the DOT file, we can compactly encode finite sets of alphabet symbols using parameterization, appropriate set notation, and a fixed set of pre-defined functions over small domains. Loading this file involves expanding notation representing sets of symbols (e.g., $\mathscr{U}$, $\mathsf{CKE}^{\mathscr{K}}$, etc.) and handling these functions. We illustrate the use of a pre-defined function using a bug pattern for DTLS clients. According to the TLS 1.2 specification [14, p. 53], the "certificate provided by the client MUST contain a key that is compatible with certificate_types" in a *CertReq* message by the server. Alternatively, the client may send a $Cert_c$ message containing no certificates. The DFA of Fig. 9 captures (the violation of) this requirement using the parameter kt to denote the key type of an included certificate, CTset to denote the set of certificate types in a *CertReq* message, and function cert_type, which, for a key type, returns the compatible certificate type. Note that the domains for available key types and certificate types are small (less than five elements). If the message $Cert_c$ from the client contains a certificate key of type kt whose compatible certificate type is not included in the certificate types requested by the server, the client bug is detected. Thus, on loading this bug pattern from a file, the loader has to map cert_type to the appropriate function in our implementation which returns the compatible certificate type for a given key type.

Our implementation realizes the algorithms in Section VI by a general-purpose function for converting a Mealy machine to a DFA. For DFA intersection, we use functionality available in AutomataLib [22]. The extraction and application of test sequences from $\mathscr{A}_\cap$ is performed according to Algorithm 1.

VIII. Evaluation

In this section, we evaluate our technique in terms of its effectiveness in detecting vulnerabilities and bugs in the SSH and DTLS implementations we use as SUTs. The main questions we want to address are: 1) Is our technique capable of detecting and validating the same set of bugs that ocular inspections of SSH and DTLS server models by security researchers have previously identified? 2) Does automation pay off in terms of being able to detect additional bugs in the same protocol implementations? 3) Has manual inspection of the models missed any bugs and, if so, how many? 4) Is our technique effective also in newer versions of SSH and DTLS servers? 5) How does it perform on DTLS clients, on which state fuzzing has not been applied before? We provide answers to these questions in the subsections below.

SUTs: Our evaluation considered the most recent versions of three widely-used SSH servers (BitVise, Dropbear and OpenSSH) and a total of nine different DTLS implementations (GnuTLS, JSSE, MbedTLS, OpenSSL, PionDTLS, Scandium, WolfSSL, and two variants of TinyDTLS). In addition, we experimented with several different versions of all these systems. However, in the tables, we report results for only two (or three) of them: the DTLS server versions used by Fiterău-Broştean et al. [18] in their paper, and the most recent version of these implementations at the time of this writing. GnuTLS, MbedTLS, OpenSSL, and WolfSSL should be well-known. JSSE is the Java Secure Socket Extension. PionDTLS is a Go implementation of DTLS 1.2 for WebRTC. Scandium is the DTLS implementation which is part of Eclipse's Java CoAP implementation. The two TinyDTLS variants are lightweight implementations specifically designed for IoT devices. We refer to Eclipse's variant as TinyDTLS^E, and to Contiki-NG's as TinyDTLS^C.

To test DTLS implementations, we used the server/client programs executable via utilities that come packaged with the implementations (e.g., `gnutls-cli` for GnuTLS). Exceptions were JSSE, PionDTLS and Scandium, for which we wrote our own programs. To test DTLS clients, we needed to ensure that client programs produced observable behavior when completing the handshake. This was done by making them echo any *App* messages they received via the standard DTLS send/receive method calls.

A. Effectiveness in Detecting Known Bugs in Servers

As mentioned, we defined bug patterns for all eleven LTL properties that previous work for SSH [21] checked. Our technique quickly confirmed that all seven properties that were violated in older SSH server implementations are still violated.

Fiterău-Broştean et al. [18] used model learning and protocol state fuzzing to analyze DTLS *server* implementations for state machine bugs. Their approach consisted in using DTLS-Fuzzer to learn (sometimes approximate) models of DTLS implementations, *manually* scrutinizing these models for state machine bugs, and then confirming them by studying the relevant source code. Can our black-box technique automatically detect the same set of bugs, and if not, why? Our experiment started from the publicly available artifact of their paper. What we did was the following: We studied the vulnerabilities and bugs that they reported [18], classified them into categories, and defined fifteen bug patterns for them. Refer to all white columns/cells of Table II which shows tick symbols for the four vulnerabilities (✔) and the eighteen bugs (✔) in the ten versions of DTLS server implementations for which problems were reported. We then took the models that DTLS-Fuzzer generated for these versions of DTLS servers (focusing on models whose alphabets only use PSK, or use a certificate-enabled key exchange algorithm and were generated for a setting where the certificate was required), and applied the technique we have described in this paper. Our implementation detected, automatically, all 22 vulnerabilities and bugs. Moreover, it generated and validated witnesses for all of them.

In short, we can answer the first question of our list positively. On a wide variety of two different network security protocol implementations, our technique is able to automatically detect and validate the same set of state machine bugs that security researchers were also able to ocularly identify.

A reasonable concern about what we have done in this first set of experiments for SSH and DTLS is that perhaps we have tweaked the bug patterns we defined to be effective just for these experiments. In the next two subsections (Sections VIII-B and VIII-C), we will show that this is not the case.

TABLE I: Bugs detected in various SSH server implementations. Everything colored in cyan background and blue color is new w.r.t. prior work [21].

	BitVise	Dropbear	OpenSSH
Bug Pattern	8.49	v2020.81	8.8p1
Early Service Accept	-	-	-
Unauthenticated Client	-	-	-
Rekey Fail Before Auth	-	-	✔
Rekey Fail After Auth	✔	-	-
Continue After Disconnect	-	-	-
Invalid Response Before Newkeys	-	-	-
Invalid *SERVICE_REQUEST_USERAUTH* Response	✔	-	✔
Invalid Auth Rejection Response	-	-	-
Multiple *USERAUTH_SUCCESS*	-	-	-
Unignored Auth Request After *USERAUTH_SUCCESS*	✔	-	✔
Invalid *CHANNEL_CLOSE* Response	-	✔	✔
Auth Fail After Rekey	-	-	-
Channel Open Fail After Rekey	-	-	-
Channel Request Terminal Fail After Rekey	-	-	-
Missing *NEWKEYS*	✔	-	✔
Missing *SERVICE_REQUEST_USERAUTH*	-	✔	-

B. Detecting New Bugs in SSH Servers

As mentioned in Section VII-A, for SSH servers, we extended the set of eleven bug patterns of prior work (whose names are shown in black in Table I) with five additional ones (whose names are shown in blue). We then applied this extended set of bug patterns to the most recent versions of SSH server implementations. As models we used the hypotheses generated after two days of learning. We note that learning converged to a final hypothesis only for Dropbear v2020.81.

Our technique was able to find eleven bugs in total. Of these, seven confirm violations also identified in older versions of these SSH servers [21]. In contrast, the remaining four (the Invalid *CHANNEL_CLOSE* Response bug in OpenSSH 8.8p1 and those in the last two rows of Table I) were *previously unknown*. We analyze these bugs below.

Invalid CHANNEL_CLOSE Response: Upon receiving a request to close an open channel via a *CClose* message, implementations are required to respond with a *CClose* of their own [50, p. 9]. We found both Dropbear and OpenSSH to violate this requirement. Dropbear appears to always respond with *CHANNEL_EOF* (*CEOF*), which expresses a party's intention to no longer send data to a channel, instead of $CClose_s$. In contrast, OpenSSH defies the requirement in a very specific scenario, right after it has generated NK_s to complete its side of the rekey. At this point, OpenSSH will readily process messages encrypted using old keys, which is problematic in itself. It will, for example, process *COpen* to open a channel, responding with *COpenSucc*. However, it stays silent if it then receives $CClose_c$, and only produces $CClose_s$ upon receiving NK_c from the client. We reported the bugs for both implementations. Dropbear's main developer acknowledged the bug, and fixed it in Dropbear's next release (v2022.82).

Incomplete Rekey: During the rekey procedure, both OpenSSH and BitVise process upper-layer messages received after sending NK_s, but before receiving NK_c. The processed messages are encrypted using old session keys. Strictly speaking, the RFC does not explicitly disallow this behavior. It only restricts parties from *sending* upper layer messages during the rekey procedure [51, p. 19]. That said, intuitively, a rekey is meant to renew keys in both directions. By not waiting for NK_c before processing upper layer messages, OpenSSH and BitVise essentially allow communication to proceed with old keys used in one direction. This partially defeats the purpose of the rekey procedure.

TABLE II: Vulnerabilities (✔), known bugs (✔), and new bugs (✔) detected in various DTLS server implementations. Crosses (✖) denote bugs that our technique found, but could not validate. Everything in cyan background and with blue text are bug patterns (rows) and DTLS versions (columns) that were not considered by Fiterău-Broştean et al. in their paper [18].

	GnuTLS			JSSE		MbedTLS		OpenSSL		PionDTLS		Scandium		TinyDTLS^C	TinyDTLS^E	WolfSSL	
Bug Pattern	3.5.19	3.6.7	3.7.1	12.0.2	16.0.1	2.16.1	2.26.0	1.1.1b	1.1.1k	e4481fc	2.0.9	2.0.0-M16	2.6.2	53a0d97	8414f8a	4.0.0	4.7.1r
Certificate-less Client Authentication	-	-	-	✔	-	-	-	-	-	-	-	-	-	-	-	-	-
CertificateVerify-less Client Authentication	-	-	-	✔	-	-	-	-	-	-	-	-	-	-	-	-	-
ChangeCipherSpec before CertificateVerify	-	-	-	✔	-	-	-	-	-	✔	-	-	-	-	-	-	-
ClientKeyExchange before Certificate	-	-	-	✔	-	-	-	-	-	-	-	-	-	-	-	-	-
Continue After Closure Alert	-	-	-	✔	-	-	-	✔	✔	-	-	-	-	-	-	-	-
Continue After Fatal Error Alert	-	-	-	✔	-	-	-	✔	✔	-	-	-	-	-	-	-	-
Early Finished	-	-	-	-	-	-	-	-	-	✔	-	✔	-	-	-	-	-
Finished before ChangeCipherSpec	-	-	-	-	-	-	-	-	-	✔	-	-	-	-	-	-	-
HelloVerifyRequest Retransmission	-	-	-	-	-	-	-	-	-	✔	-	-	-	-	-	-	-
Insecure Renegotiation	-	-	-	-	-	-	-	-	-	-	-	-	-	✔	✔	-	-
Internal Error on Finished	-	-	-	-	-	-	-	✔	✔	-	-	✔	✔	-	-	-	-
Invalid DecryptError Alert	-	-	-	-	-	-	-	-	-	-	-	-	-	✔	✔	-	-
Invalid Finished as Retransmission	-	-	-	-	-	-	-	✔	✔	-	-	-	-	-	-	-	-
Invalid HelloVerifyRequest	✔	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Multiple Certificate	-	-	-	✔	-	-	-	-	-	-	-	✖	-	-	-	-	-
Multiple ChangeCipherSpec	-	-	-	✔	-	-	-	-	-	-	-	✔	-	-	-	-	-
Non-conforming Cookie	✔	✔	✔	-	-	✔	✔	✔	✔	-	-	-	-	-	-	-	-
Unauthenticated ClientKeyExchange	-	-	-	✔	-	-	-	-	-	-	-	-	-	-	-	-	-

Service Request Not Needed: Dropbear allows authentication to proceed immediately after it receives NK_c from the client. It does not require for the authentication service to be first requested (via the *SReqAuth* – *SAcc* exchange). We inquired Dropbear's lead developer about this behavior and were told that it is intentional. Dropbear is designed to disregard *SReqAuth* messages, as they are deemed to not make a difference to the overall flow of a session. This design choice also removes the need of keeping track of these messages.

C. Detecting New Bugs in DTLS Servers

The DTLS server results are shown in Table II. Cells with cyan background are new. As can be seen, even in the ten versions of DTLS servers that were considered in the previous section (white columns), our technique was able to detect seven additional bugs that were missed by visual inspection of their models. Two of these bugs (the Multiple ChangeCipherSpec bug in JSSE 12.0.2 and the Invalid DecryptError Alert bug in Scandium) concern bug patterns for bugs that had been detected in some other implementation. This proves that manual inspection of (complicated) models can easily miss bugs.

Regarding the newer versions of these DTLS server implementations (cyan columns), our technique was able to automatically detect eight bugs in them. In addition, on Scandium 2.0.0-M16, whose learned model is approximate, our technique detected one bug that it was not able to validate, which we touch on later in this section. We remark, in passing, that Table II also shows which implementations have fixed vulnerabilities and bugs in their latest versions. For example, we can see that JSSE, PionDTLS and Scandium have improved.

In a nutshell, we can answer the next three questions in our list also positively: 2) automation is effective in detecting additional bugs, 3) the number of bugs which have been missed by ocular inspection is significant, and 4) our technique remains effective in recent SSH and DTLS implementations. It should be clear that automation pays off.

Below, we briefly analyze some of the more notable previously unknown bugs found in DTLS servers.

Multiple Invalid Handshakes: On JSSE 12.0.2, we detected two bugs (Multiple Certificate and Multiple ChangeCipherSpec) that were missed by visual inspection of its model. These bugs involve the server completing a handshake during which it receives multiple instances of these messages. The bug witnesses that our framework generates indicate variations of the same non-conforming behavior. Sending the input sequence *CH, CH, CKE, CCS_c, $Cert_c$* and Fin_c, in this order, will not complete the handshake. It will, instead, bring the JSSE server to a state that is close to handshake completion. Our two witnesses continue this sequence in two ways, both of which prompt the server to complete the handshake and await application data. The continuations are: CCS_c, $Cert_c$ for the Multiple Certificate bug, and CCS_c, *CH* for the Multiple ChangeCipherSpec bug. The CCS_c message appears to be required for triggering this bug; replacing it by a different message prevents handshake completion.

InternalError on Finished: Prior work found this bug to affect OpenSSL. We found it also affects Scandium 2.0.0-M16. Implementations should reject messages such as *Finished* when these messages are received in unexpected orders, and either respond with *UnexpectedMessage* (the correct *Alert* message for these cases [14, p. 30]), or not at all. Executing the bug witness reveals that upon receiving a Fin_c, which comes after two *CH(PSK)* messages, a Scandium 2.0.0-M16 server implementation tries to process it and fails in doing so, generating a `NullPointerException`. The new bug was reported and quickly addressed by Scandium's developers.

Non-conforming Cookie Computation: According to the DTLS RFC [38, p. 17], the cookie included in *HVR* should be computed over fields in the *initial CH*, including the supported cipher suites field. Hence, the server should not accept a second *CH* which uses different supported cipher suites from the first (and respond to it with *SH*). Yet, this is exactly the case for DTLS servers such as GnuTLS, MbedTLS, and OpenSSL, which ignore the RFC requirement. Both older and newer versions of these servers are affected from this bug.

Buffered Alert before Finished: Alerts that are fatal or are of type close_notify should cause termination of the session, irrespective of whether they are received or sent. Non-compliance to this requirement is captured in the Continue After Closure Alert and Fatal Error Alert bug patterns. We found these bugs to affect both versions of OpenSSL, which buffer alerts (along with App_c) sent by the client between the CCS_c and Fin_c messages. The OpenSSL server does eventually process these alerts upon receiving Fin_c.

TABLE III: Previously unknown vulnerabilities (✓) and bugs (✔) detected in various DTLS client implementations.

	GnuTLS		JSSE		MbedTLS		OpenSSL		PionDTLS		Scandium		TinyDTLS^C	TinyDTLS^E	WolfSSL	
Bug Pattern	3.6.7	3.7.1	12.0.2	16.0.1	2.16.1	2.26.0	1.1.1b	1.1.1k	e4481fc	2.0.9	2.0.0-M16	2.6.2	53a0d97	8414f8a	4.0.0	4.7.1r
CertificateRequest before Certificate	-	-	✔	✔	-	-	-	-	✔	-	✔	-	-	-	-	-
Continue After Closure Alert	✔	✔	✔	✔	-	-	✔	✔	-	-	-	-	-	-	-	-
Continue After Fatal Error Alert	✔	✔	✔	✔	-	-	✔	✔	✔	-	-	-	-	-	-	-
Early Finished	-	-	-	-	-	-	-	-	✓	-	✓	-	-	-	-	-
Finished Before ChangeCipherSpec	-	-	-	-	-	-	-	-	✔	-	-	-	-	-	-	-
Invalid DecryptError Alert	-	-	-	-	-	-	-	-	-	-	-	-	✔	✔	-	-
Multiple Certificate	-	-	✔	✔	-	-	✔	✔	✔	-	✔	-	-	-	-	-
Multiple CertificateRequest	-	-	✔	✔	-	-	✔	✔	✔	-	✔	-	✔	✔	-	-
Multiple ChangeCipherSpec	-	-	-	-	-	-	-	-	-	-	✔	-	-	-	-	-
Multiple ServerKeyExchange	-	-	✔	✔	-	-	✔	✔	✔	-	✔	-	-	-	-	-
Premature HelloRequest	-	-	-	-	✔	✔	-	-	-	-	-	-	-	-	-	-
ServerHello Flight Restart	-	-	-	-	-	-	-	-	✔	-	-	-	-	-	-	-
Switching Cipher Suite	-	-	-	-	-	-	✔	✔	-	-	-	-	-	-	-	-
Unexpected ClientHello	-	-	-	-	-	-	✔	✔	✔	-	-	-	-	-	✔	✔
Unrequested Certificate	-	-	-	-	-	-	✔	✔	-	-	-	-	-	-	-	-
Wrong Certificate Type	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	-	-	-	-	-

Bugs that cannot be validated can still indicate problems: For Scandium 2.0.0-M16, there is a single case where a model bug could not be validated. In such cases, our framework still generates a failed bug witness (a witness that does not manage to expose the presence of the bug on the SUT). Investigating failed bug witnesses can yield some notable findings. For example, by executing the failed witness for the Multiple Certificate bug, we found that the Scandium 2.0.0-M16 server comes close to completing a handshake when it receives multiple $Cert_c$ messages, one after the other. What stops it from doing so is the fact that the Fin_c we sent it at the end does not have the digest the server expects. Tweaking manually what is included in the digest allowed us to complete this invalid handshake and demonstrate the existance of the bug.

D. Detecting New Bugs in DTLS Clients

We extended DTLS-Fuzzer with the ability to learn models of DTLS client implementations. We also defined a set of 16 bug patterns for DTLS clients. The automation of our approach allowed us to also analyze DTLS client libraries for vulnerabilities and bugs, something which was done for the first time as far as we know. Refer to Table III showing results from analyzing 16 different DTLS client versions. In total, we were able to detect *two new vulnerabilities and* 64 *previously unknown bugs*. Below, we analyze some of the more interesting client bugs we found.

Multiple Certificate, CertificateRequest and ServerKeyExchange: The TLS 1.2 specification [14] does not allow for multiple messages of the same type to be sent. The proper behavior by a client when it receives more than one message of the same type is to terminate the handshake. However, for specific messages, JSSE, both versions of TinyDTLS and Scandium 2.0.0-M16 may instead continue to process messages, and eventually generate the messages from flight 5. We reported the bugs, prompting fixes in JSSE and TinyDTLS^E.

These bugs also exists in OpenSSL, but only in renegotiated handshakes. Upon receiving a second message of the same type in the first handshake, OpenSSL will terminate the handshake with a fatal alert (usually Unexpected Message). However, in some cases, OpenSSL will continue a renegotiated handshake upon receiving multiple $Cert_s$ or SKE messages. For several of these cases, it is also required that the cookie exchange be omitted from the initial handshake, while for others the cookie exchange must not be omitted. The Multiple Certificate, Multiple CertificateRequest and Multiple ServerKeyExchange bugs in OpenSSL have been reported and confirmed. Given the size of the model (over 350 states) and how specific these message sequences needed to be for this buggy behavior to appear, finding these bugs without an automated approach would have been very difficult.

Early Finished: The PionDTLS e4481fc and Scandium 2.0.0-M16 clients both exhibit the Early Finished vulnerability, which allows a handshake to complete without a *ChangeCipherSpec* from the server (CCS_s). This causes the client to process future messages sent to it in plaintext. However, the client still requires a valid Fin_s message to complete the handshake, with a valid digest. This means that an attacker cannot simply block a CCS_s, as that would lead to an invalid *verify_data* segment in the Fin_s message. But if a bug in a server would cause it to not send a CCS_s message, and thus not deploy the cipher suite, this bug in the client would enable an attacker to observe an established plaintext connection. (Introducing such a bug to a server could be done by injecting a backdoor into a DTLS library.) In both PionDTLS and Scandium, this client vulnerability was fixed in later versions.

Wrong CertificateRequest Type: Interestingly enough, this bug was detected in all implementations except Scandium 2.0.0-M16 and WolfSSL. As explained in Section III-C, if the client does not have a certificate which conforms to the type requested by a *CertReq* message from the server, the client must send an empty *Cert* message. The DFA describing this bug was shown in Fig. 9. Our analysis for Scandium 2.6.2 client found it to exhibit conforming behavior. The same analysis also discovered that the Scandium 2.6.2 server expects conforming behavior from clients. This leads to a potential interoperability problem between a Scandium 2.6.2 server and non-conforming clients. This bug has been reported, and has been confirmed by MbedTLS and OpenSSL developers.

Unexpected Certificate: Another issue with renegotiated handshakes in OpenSSL is that the client can perform client authentication without it being requested by the server. This requires a *CertReq* to be sent by the server prior to the *HR* which starts the new handshake. This, and the bugs described in the previous paragraph, indicate that, in handshake renegotiation, OpenSSL uses messages received before the renegotiation has started, which it should not do. We have reported this bug to OpenSSL developers who confirmed it.

TABLE IV: Quantitative measurements for experiments on DTLS client implementations.

	GnuTLS		JSSE		MbedTLS		OpenSSL		PionDTLS		Scandium		TinyDTLS^C	TinyDTLS^E	WolfSSL	
	3.6.7	3.7.1	12.0.2	16.0.1	2.16.1	2.26.0	1.1.1b	1.1.1k	e4481fc	2.0.9	2.0.0-M16	2.6.2	53a0d97	8414f8a	4.0.0	4.7.1r
Mealy Machine Nodes	27	39	133	113	34	40	387	387	49	26	38	12	22	17	13	17
DFA Nodes (Minimized)	52	77	166	146	59	71	621	621	82	45	94	25	44	32	27	29
Validated Bugs	3	3	7	7	2	2	9	9	10	1	7	0	2	2	1	1
Input Sequence Messages	8	8	48	48	6	6	89	89	54	5	42	0	8	8	1	2
Test Sequences Tried	3	3	7	7	2	2	9	9	10	1	7	0	2	2	1	1
Time (ms)	2611	2637	35595	35534	1994	1983	10880	11113	11020	1543	28405	61	2267	2315	1211	1641

Premature HelloRequest: This is a conformance bug present in MbedTLS. According to the TLS 1.2 specification [14, p. 38], a client should ignore any *HR* message sent by the server before the handshake has been concluded. However, the MbedTLS clients will always respond to a *HR* with a *CH*, and then continue with the ongoing handshake. This bug has been reported and confirmed.

Switching Cipher Suite: If a DTLS server sends to an OpenSSL client a *SH* followed by a *HVR*, the client will respond with a *CH*, allowing a new *SH* to be sent. With this second *SH*, it is possible for the server to switch to a different cipher suite, even a different key-exchange algorithm. This is, in effect, a renegotiation not supported by the DTLS RFC.

Continue After Fatal Error Alert: Some server messages will cause the client to send a Handshake Failure alert. Upon sending this fatal alert, the client should terminate the handshake. However, PionDTLS e4481fc will instead continue to consume server messages of flight 4, and even generate the messages of flight 5. This is a bug which was fixed in later versions. The Continue After Fatal Error Alert bug, and the similar Continue After Closure Alert, were also captured in JSSE and OpenSSL where this was caused by alerts being buffered. GnuTLS also exhibits behavior captured by these two bug patterns: the client will ignore an alert sent by the server, if it is sent before the *SH*.

E. Responsible Disclosure and Impact

We responsibly disclosed all the new bugs we found using appropriate channels. The developers' feedback is summarized below, with more details provided in the Appendix. With five exceptions, all the bugs found in DTLS libraries have already been confirmed. All the new bugs reported for JSSE, Scandium and TinyDTLS^E have been fixed in later versions of these libraries, as is the case for the bug involving CHANNEL_CLOSE we reported for SSH Dropbear. Regarding the JSSE bugs, which are for properties not considered in prior works, they were marked as "security in depth", meaning they are security vulnerabilities resulting in significant modifications to the library's source code. The bugs are fixed in JSSE 18.0.1. The fact that most bugs have been confirmed and four libraries have been fixed speaks for the relevance of our findings.

F. Quantitative Measurements

We also provide some quantitative measurements. For model learning DTLS implementations, we used the same learning and test algorithms as, and parameters suggested by, the artifact of paper [18]. Learning models of DTLS clients and servers took roughly the same amount of time as that reported in [18], with most experiments finishing in a day or less. We stopped model learning after two days if it did not appear to converge to a final model, in which case we used the last generated hypothesis as the approximate model. With the learned models, bug detection using *specific* bug patterns was performed relatively quickly, with the majority of the time spent in validating the bugs on the SUT. For that purpose, we used a *maxStateVisits* value of one in Algorithm 1.

Table IV shows that checking all 16 client bug patterns took less than one minute to finish in all SUTs. The most time-consuming experiments were for PionDTLS e4481fc, Scandium 2.0.0-M16 and the two tested versions of OpenSSL and JSSE. These are the implementations for which most bugs were found. With the exception of PionDTLS, these are also implementations for which model learning did not converge. As a result, their models captured coarser approximations of their behaviors. Even with such models, only one test sequence was required to validate each bug.

Effectiveness on Inaccurate Models: To assess how model accuracy affects the effectiveness of our technique, we conducted an experiment using the implementations for which model learning did not converge after two days of learning: these are the JSSE 12.0.2, OpenSSL 1.1.1b and Scandium 2.0.0-M16 client implementations. We used models with increasing accuracy, generated for these implementations after each hour of model learning up to one day. We then collected the number of bugs found, bugs validated, and the number of test sequences validation required. Results show that our technique can find and validate all the bugs reported in Table III using models generated early in the learning process, after one (Scandium 2.0.0-M16), four (JSSE 12.0.2) and twelve (OpenSSL 1.1.1b) hours of model learning. When using a less accurate model, bug validation may require the generation and execution of more test sequences. For example, the model generated after twelve hours for OpenSSL 1.1.1b required 52 test sequences to validate the nine bugs found in that implementation. By contrast, the model generated after two days of learning, only needed nine test sequences (cf. Table IV). Another trade-off is that a less accurate model is more likely to validate false positives. For example, the models generated for OpenSSL 1.1.1b using between two and 24 hours of model learning all contained the Early Finished and Multiple ChangeCipherSpec bugs. These bugs could not be validated, even after spending an entire day of testing. In contrast, these bugs do not appear in the model generated after two days, used in our evaluation, indicating that they are a result of inaccuracies in model learning.

Effectiveness of Using Models of Old Versions: In two separate experiments, we applied our technique to test server implementations of JSSE versions 11.0.1 through 11.0.9. In the first experiment, to test a version we used a model learned for it after two days. In the second experiment, the model for JSSE 11.0.1 was used to test all subsequent versions. From our first experiment we noticed a gradual decline, from nine to two, in the number of bugs validated for each subsequent version, with no new bugs introduced. Comparing the results of our two experiments, we remark that for each release, the number of bugs validated was the same, irrespective of the model used. However, using the model of an old version to test a newer version sometimes required more test sequences (e.g., 22 vs six to validate six bugs in 11.0.6). This suggests that, if we want to test a new version quickly (e.g., to check if some bug is fixed), we may use the model of the old version instead of learning a model anew, which may take several hours. Doing so may, however, miss bugs introduced in the new version.

Effectiveness of Specific vs. General Bug Patterns: We also performed an experiment where we used just one general bug pattern similar to that of Fig. 8 for bug detection, instead of using the 16 specific bug patterns for DTLS clients. For Scandium 2.0.0-M16, our technique managed to generate and validate witnesses for *all* seven client bugs (cf. Table III) but only after trying 26,965 test sequences and spending half a day in this process. In contrast, bug detection and validation using specific bug patterns requires only seven test sequences and finishes in less than 30 seconds (cf. Table IV). There were also SUTs (e.g., JSSE 12.0.2 and OpenSSL 1.1.1b) for which using only the general bug pattern for DTLS clients managed to detect some of the bugs quickly but also failed to produce witnesses for all bugs after running for a day. In short, these experiments show that bug detection with a general bug pattern is inferior to using specific (and small) bug patterns.

Scalability: Our implementation is able to handle non-trivial models. The client model for JSSE 12.0.2 has 133 states, while the OpenSSL models have 387 states each. Such models would have been very difficult to effectively analyze ocularly.

The overall conclusion that can be drawn is that, in the presence of a learned model, bug detection using specific bug patterns can be done quickly, in a matter of minutes if not seconds. Still, our framework benefits from accurate models. A more accurate model reduces the time bug detection takes, as fewer tests have to be run for bug validation. Finally, our framework can be applied on models with hundreds of states that would be challenging to analyze manually.

IX. Discussion

The effectiveness of our automated technique depends on its two inputs: the model of the protocol implementation and the DFAs encoding protocol bugs. In this section, we briefly discuss how the quality of these inputs may affect our technique.

Dependence on the Model: In most cases, the model of the protocol implementation is obtained by model learning. When this model is inaccurate, for example because learning did not handle the implementation's non-determinism properly or because it did not converge within the given time limit, witnesses for bugs that are identified in the model may fail to validate on the SUT. We note that our experience with such cases is limited: we have encountered only a single model bug that failed to validate (for Scandium 2.0.0-M16 servers). In general, however, when witnesses of a model bug fail to validate, it is difficult to determine whether the bug exists in the SUT and a witness for it could have been produced by a better test harness component or with more witness generation attempts, or whether the model bug is instead caused by model inaccuracy, in which case it would not be present in a more accurate model. Another reason why bug witnesses may fail to validate in the SUT is when the abstraction employed to obtain the model is different from the one implemented by the test harness used for validation. In our work, we avoid such cases by using the same test harness for model learning and witness validation.

Dependence on the DFAs: The DFAs that encode protocol state machine bugs form the specification of our technique. Naturally, if the specification of what protocol interactions to consider as bugs is erroneous, our technique will flag as bugs situations which are not. For this reason, the bug patterns must be accurate. As a concrete example, bug patterns for DTLS servers should allow the possibility to restart the handshake when receiving a *CH* message, which is why the DFAs of Figs. 4 to 7 contain *SH* edges back to the init state. Our experience is that although manually constructed specific bug patterns are often slightly inaccurate at first, it is also quite easy to correct them with small local changes and turn them into accurate ones.

X. Conclusions

We have presented an automated black-box technique for detecting state machine bugs that can be encoded using finite automata in implementations of stateful network protocols. Our technique takes as input a catalogue of bug patterns, each encoded as a finite automaton which accepts sequences of messages that exhibit a bug or violate a requirement, and a (not necessarily accurate) model of the implementation, obtained by model learning. We have applied this technique to the detection of state machine bugs in nine different DTLS server and client implementations, including their most recent versions, and to three SSH server implementations. Our evaluation shows that our technique can automatically detect all bugs and security vulnerabilities that previous researchers have reported, as well as detect a significant number of previously unknown state machine bugs in the same implementations, including bugs that ocular inspection missed in spite of having detected an analogous bug in some other implementation. We hold that these results demonstrate the effectiveness of encoding protocol ordering requirements using finite automata and the additional power that automated bug detection and validation offer to state fuzzing techniques.

We provide a virtual machine [19] with source code, the results of our experiments, and instructions on how to reproduce them.

Acknowledgments

This research was partially funded by the Swedish Foundation for Strategic Research through project aSSIsT, the Swedish Research Council (Vetenskapsrådet), and the Knut and Alice Wallenberg Foundation through project UPDATE. We also thank our shepherd and the other three NDSS 2023 reviewers for comments and questions that have improved the presentation aspects of our work.

References

[1] F. Aarts, B. Jonsson, J. Uijen, and F. W. Vaandrager, "Generating models of infinite-state communication protocols using regular inference with abstraction," *Formal Methods in System Design*, vol. 46, no. 1, pp. 1–41, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10703-014-0216-x

[2] B. K. Aichernig, W. Mostowski, M. R. Mousavi, M. Tappler, and M. Taromirad, "Model learning and model-based testing," in *Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172. Revised Papers*, ser. LNCS, vol. 11026. Springer, 2018, pp. 74–100. [Online]. Available: https://doi.org/10.1007/978-3-319-96562-8_3

[3] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987. [Online]. Available: https://doi.org/10.1016/0890-5401(87)90052-6

[4] G. Argyros, I. Stais, S. Jana, A. D. Keromytis, and A. Kiayias, "SFADiff: Automated evasion attacks and fingerprinting using black-box differential automata learning," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Oct. 2016, pp. 1690–1701. [Online]. Available: https://doi.org/10.1145/2976749.2978383

[5] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of TLS," *Commun. ACM*, vol. 60, no. 2, pp. 99–107, Feb. 2017. [Online]. Available: https://doi.org/10.1145/3023357

[6] B. Beurdouche, A. Delignat-Lavaud, N. Kobeissi, A. Pironti, and K. Bhargavan, "FlexTLS: A tool for testing TLS implementations," in *9th USENIX Workshop on Offensive Technologies*, ser. WOOT 15. USENIX Association, Aug. 2015. [Online]. Available: https://www.usenix.org/system/files/conference/woot15/woot15-paper-beurdouche.pdf

[7] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1," in *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference*, ser. LNCS, vol. 1462. Berlin / Heidelberg: Springer, Aug. 1998, pp. 1–12. [Online]. Available: https://doi.org/10.1007/BFb0055715

[8] M. Broy, B. Jonsson, J. Katoen, M. Leucker, and A. Pretschner, Eds., *Model-Based Testing of Reactive Systems, Advanced Lectures*, ser. LNCS. Springer, 2005, vol. 3472. [Online]. Available: https://doi.org/10.1007/b137241

[9] Y. Cao, Z. Wang, Z. Qian, C. Song, S. V. Krishnamurthy, and P. Yu, "Principled unearthing of TCP side channel vulnerabilities," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS 2019. ACM, Nov. 2019, pp. 211–224. [Online]. Available: https://doi.org/10.1145/3319535.3354250

[10] C. Y. Cho, D. Babic, E. C. R. Shin, and D. Song, "Inference and analysis of formal models of botnet command and control protocols," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS 2010. ACM, Oct. 2010, pp. 426–439. [Online]. Available: https://doi.org/10.1145/1866307.1866355

[11] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., *Handbook of Model Checking*. Springer, 2018. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8

[12] L.-A. Daniel, E. Poll, and J. de Ruiter, "Inferring OpenVPN state machines using protocol state fuzzing," in *IEEE European Symposium on Security and Privacy (EuroS&P) Workshops*. IEEE, Apr. 2018, pp. 11–19. [Online]. Available: https://doi.org/10.1109/EuroSPW.2018.00009

[13] J. de Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *24th USENIX Security Symposium*. USENIX Association, Aug. 2015, pp. 193–206. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter

[14] T. Dierks and E. Rescorla, "The transport layer security TLS protocol version 1.2," RFC 5246, Aug. 2008. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5246.txt

[15] T. Ferreira, H. Brewton, L. D'Antoni, and A. Silva, "Prognosis: Closed-box analysis of network protocol implementations," in *ACM SIGCOMM 2021 Conference*. ACM, Aug. 2021, pp. 762–774. [Online]. Available: https://doi.org/10.1145/3452296.3472938

[16] P. Fiterau-Brostean, B. Jonsson, K. Sagonas, and F. Tåquist, "Dtls-fuzzer: A DTLS protocol state fuzzer," in *15th IEEE Conference on Software Testing, Verification and Validation*, ser. ICST 2022. IEEE, Apr. 2022, pp. 456–458. [Online]. Available: https://doi.org/10.1109/ICST53961.2022.00051

[17] P. Fiterău-Broştean, R. Janssen, and F. W. Vaandrager, "Combining model learning and model checking to analyze TCP implementations," in *Computer Aided Verification - 28th International Conference, CAV 2016, Proceedings, Part II*, ser. LNCS, vol. 9780. Springer, 2016, pp. 454–471. [Online]. Available: https://doi.org/10.1007/978-3-319-41540-6_25

[18] P. Fiterău-Broştean, B. Jonsson, R. Merget, J. de Ruiter, K. Sagonas, and J. Somorovsky, "Analysis of DTLS implementations using protocol state fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2523–2540. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean

[19] P. Fiterău-Broştean, B. Jonsson, K. Sagonas, and F. Tåquist, "Artifact for the Paper 'Automata-Based Automated Detection of State Machine Bugs in Protocol Implementations'," Sep. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.7129240

[20] P. Fiterău-Broştean, T. Lenaerts, J. de Ruiter, E. Poll, F. W. Vaandrager, and P. Verleg, "Source code and data relevant for the paper "Model Learning and Model Checking of SSH Implementations"," https://hdl.handle.net/2066/184275.

[21] ——, "Model learning and model checking of SSH implementations," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, ser. SPIN 2017. ACM, 2017, pp. 142–151. [Online]. Available: https://doi.org/10.1145/3092282.3092289

[22] M. Frohme and M. Isberner, "AutomataLib," 2020. [Online]. Available: https://learnlib.de/projects/automatalib/

[23] A. Groce, D. A. Peled, and M. Yannakakis, "Adaptive model checking," *Log. J. IGPL*, vol. 14, no. 5, pp. 729–744, 2006. [Online]. Available: https://doi.org/10.1093/jigpal/jzl007

[24] J. Guo, C. Gu, X. Chen, and F. Wei, "Model learning and model checking of IPSec implementations for internet of things," *IEEE Access*, vol. 7, pp. 171 322–171 332, Nov. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8913552

[25] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279–295, 1997. [Online]. Available: https://doi.org/10.1109/32.588521

[26] M. E. Hoque, O. Chowdhury, S. Y. Chau, C. Nita-Rotaru, and N. Li, "Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs," in *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN 2017. IEEE Computer Society, Jun. 2017, pp. 627–638. [Online]. Available: https://doi.org/10.1109/DSN.2017.36

[27] S. Hu, Q. A. Chen, J. Sun, Y. Feng, Z. M. Mao, and H. X. Liu, "Automated discovery of denial-of-service vulnerabilities in connected vehicle protocols," in *30th USENIX Security Symposium (USENIX Security 2021)*. USENIX Association, Aug. 2021, pp. 3219–3236. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/hu-shengtuo

[28] S. R. Hussain, O. Chowdhury, S. Mehnaz, and E. Bertino, "LTEInspector: A systematic approach for adversarial testing of 4G LTE," in *25th Annual Network and Distributed System Security Symposium*, ser. NDSS 2018. The Internet Society, Feb. 2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_02A-3_Hussain_paper.pdf

[29] S. R. Hussain, I. Karim, A. A. Ishtiaq, O. Chowdhury, and E. Bertino, "Noncompliance as deviant behavior: An automated black-box noncompliance checker for 4G LTE cellular devices," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. ACM, Nov. 2021, pp. 1082–1099. [Online]. Available: https://doi.org/10.1145/3460120.3485388

[30] M. Isberner, F. Howar, and B. Steffen, "The TTT algorithm: A redundancy-free approach to active automata learning," in *Runtime Verification: 5th International Conference, RV 2014, Proceedings*, ser. LNCS. Springer, Sep. 2014, vol. 8734, pp. 307–322. [Online]. Available: https://doi.org/10.1007/978-3-319-11164-3_26

[31] S. Jero, M. E. Hoque, D. R. Choffnes, A. Mislove, and C. Nita-Rotaru, "Automated attack discovery in TCP congestion control using a model-guided approach," in *Proceedings of the Applied Networking Research Workshop*, ser. ANRW 2018. ACM, Jul. 2018, p. 95. [Online]. Available: https://doi.org/10.1145/3232755.3232769

[32] S. Jero, H. Lee, and C. Nita-Rotaru, "Leveraging state information for automated attack discovery in transport protocol implementations," in *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN 2015. IEEE Computer Society, Jun. 2015, pp. 1–12. [Online]. Available: https://doi.org/10.1109/DSN.2015.22

[33] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines—a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996. [Online]. Available: https://ieeexplore.ieee.org/document/533956

[34] C. McMahon Stone, T. Chothia, and J. de Ruiter, "Extending automated protocol state learning for the 802.11 4-way handshake," in *Computer Security*, ser. LNCS, vol. 11098. Cham: Springer International Publishing, Aug. 2018, pp. 325–345.

[35] D. A. Peled, M. Y. Vardi, and M. Yannakakis, "Black box checking," *J. Autom. Lang. Comb.*, vol. 7, no. 2, pp. 225–246, 2002. [Online]. Available: https://doi.org/10.25596/jalc-2002-225

[36] H. Raffelt, M. Merten, B. Steffen, and T. Margaria, "Dynamic testing via automata learning," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, no. 4, pp. 307–324, 2009. [Online]. Available: https://doi.org/10.1007/s10009-009-0120-7

[37] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov, "Transport layer security (TLS) renegotiation indication extension," RFC 5746, Feb. 2010. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5746.txt

[38] E. Rescorla and N. Modadugu, "Datagram transport layer security version 1.2," RFC 6347, Jan. 2012. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6347.txt

[39] E. Rescorla, H. Tschofenig, and N. Modadugu, "The datagram transport layer security (DTLS) protocol version 1.3," RFC 9147, Apr. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9147

[40] Riku, Antti, Matti, and N. Mehta, "Heartbleed, CVE-2014-0160," 2015. [Online]. Available: http://heartbleed.com/

[41] J. Somorovsky, "Systematic fuzzing and testing of TLS libraries," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 1492–1504. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978411

[42] M. Tappler, B. K. Aichernig, and R. Bloem, "Model-based testing IoT communication via active automata learning," in *Software Testing, Verification and Validation, (ICST) 2017 IEEE International Conference on*. IEEE Computer Society, Mar. 2017, pp. 276–287. [Online]. Available: https://doi.org/10.1109/ICST.2017.32

[43] M. Utting and B. Legeard, *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann, 2007.

[44] F. W. Vaandrager, "Model learning," *Commun. ACM*, vol. 60, no. 2, pp. 86–95, 2017. [Online]. Available: https://doi.org/10.1145/2967606

[45] S. Vaudenay, "Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ..." in *Advances in Cryptology - EUROCRYPT 2002*, ser. LNCS. Berlin / Heidelberg: Springer, Apr. 2002, vol. 2332, pp. 534–545. [Online]. Available: https://doi.org/10.1007/3-540-46035-7_35

[46] D. Wagner and B. Schneier, "Analysis of the SSL 3.0 protocol," in *Proceedings of the 2nd USENIX Workshop on Electronic Commerce*. Berkeley, CA, USA: USENIX Association, 1996, pp. 29–40. [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/ec96/full_papers/wagner/wagner.pdf

[47] Z. Wang, S. Zhu, K. Man, P. Zhu, Y. Hao, Z. Qian, S. V. Krishnamurthy, T. L. Porta, and M. J. D. Lucia, "Themis: Ambiguity-aware network intrusion detection based on symbolic model comparison," in *Proceedings of the 8th ACM Workshop on Moving Target Defense*, ser. MTD@CCS 2021. ACM, Nov. 2021, pp. 31–32. [Online]. Available: https://doi.org/10.1145/3474370.3485669

[48] P. Wolper, "Temporal logic can be more expressive," *Inf. Control.*, vol. 56, no. 1/2, pp. 72–99, 1983. [Online]. Available: https://doi.org/10.1016/S0019-9958(83)80051-5

[49] T. Ylönen and C. Lonvic, "The secure shell (SSH) authentication protocol," RFC 4252, Jan. 2006. [Online]. Available: https://www.rfc-editor.org/info/rfc4252

[50] ——, "The secure shell (SSH) connection protocol," RFC 4254, Jan. 2006. [Online]. Available: https://www.rfc-editor.org/info/rfc4254

[51] ——, "The secure shell (SSH) transport layer protocol," RFC 4253, Jan. 2006. [Online]. Available: https://www.rfc-editor.org/info/rfc4253

TABLE V: Status of bugs reported for DTLS implementations. For each bug, the status is a clickable link to the curresponding issue unless the bug was privately disclosed, in which case its status is marked with '*'.

Implementation	Bug	Status
	Server Bugs	
GnuTLS 3.7.1	Non-Conforming Cookie	**confirmed**
MbedTLS 2.26.0	Non-Conforming Cookie	**wontfix**
OpenSSL 1.1.1k	Continue After Closure Alert	**confirmed**
	Continue After Fatal Alert	**confirmed**
	Internal Error on Finished	**confirmed**
	Invalid Finished as Retransmission	**confirmed**
	Non-Conforming Cookie	**confirmed**
Scandium 2.6.2	Internal Error on Finished	**fixed**
	Client Bugs	
GnuTLS 3.7.1	Continue After Closure Alert	**reported**
	Continue After Fatal Alert	**reported**
	Wrong Certificate Type	**confirmed**
JSSE 16.0.1	CertificateRequest before Certificate	**fixed***
	Continue After Closure Alert	**fixed***
	Continue After Fatal Error Alert	**fixed***
	Multiple Certificate	**fixed***
	Multiple CertificateRequest	**fixed***
	Multiple ServerKeyExchange	**fixed***
	Wrong Certificate Type	**fixed***
MbedTLS 2.26.0	Premature HelloRequest	**confirmed**
	Wrong Certificate Type	**confirmed**
OpenSSL 1.1.1k	Continue After Closure Alert	**confirmed**
	Continue After Fatal Error Alert	**confirmed**
	Multiple Certificate	**confirmed**
	Multiple ServerKeyExchange	**confirmed**
	Multiple CertificateRequest	**confirmed**
	Switching Cipher Suite	**confirmed**
	Unexpected ClientHello	**confirmed**
	Unrequested Certificate	**confirmed**
	Wrong Certificate Type	**confirmed**
PionDTLS 2.0.9	Wrong Certificate Type	**confirmed**
TinyDTLS^C 53a0d97	Invalid DecryptError Alert	**reported**
	Multiple CertificateRequest	**reported**
TinyDTLS^E 8414f8a	Invalid DecryptError Alert	**fixed**
	Multiple CertificateRequest	**fixed**
WolfSSL 4.7.1r	Unexpected ClientHello	**reported**

Section A details how bugs were reported to developers and the bugs' status, while Section B reports results from an experiment with inaccurate models.

Appendix A Reporting Bugs

In this appendix, we explain how bugs were reported, and what has been the developers' reaction to them at the time of this writing. All new bugs mentioned in our paper were reported to developers using standard channels, which vary depending on the implementation. For most implementations, we communicated the bugs by creating issues on the implementation's public repositories. Before doing so, we made sure that the bug did not present a security risk, meaning it could be safely disclosed to the public. For BitVise and Dropbear, we contacted the developers by their private email addresses, while for OpenSSH we used the developer mailing list. In the case of JSSE, bugs were reported by submitting a bug report following the procedure specified in the Java website.

For **reported** bugs, we closely monitored their status. A reported bug is **confirmed** if we get an acknowledgement from developers. Some confirmed bugs have been subsequently **fixed**. Others were judged to be a result of an intentional design choice. We mark these bugs as **wontfix**. In what follows, we detail on the status of the bugs reported for DTLS and SSH implementations.

A. DTLS Implementations

Table V shows the status of reported bugs for DTLS implementations. With a few exceptions, all the bugs have been confirmed by developers. In general, this was done not only by changing the status of the issue (e.g. from 'bug report' to 'bug'), but also by developers providing a confirmatory message. This message occassionally included an explanation on the potential cause of the problem.

The bugs reported for JSSE, Scandium and TinyDTLS^E have lead to fixes in the respective implementations. We tested these fixes by re-running the witnesses our technique generated, and ensuring they were no longer valid. Bugs found in JSSE bugs were assigned the "Security in Depth" marker. According to Oracle, this makes them:

> *Security vulnerability issues that result in significant modifications of Oracle code or documentation in future releases, but are not of such a critical nature that the modifications would be distributed in Critical Patch Updates.*

The bugs have been addressed in JSSE 18.0.1. Bugs confirmed for MbedTLS clients were added to the implementation's backlog. The Non-Conforming Cookie bug found in MbedTLS servers was also confirmed, however, developers told us they would not fix it, since this would lead to an undesirable increase in MbedTLS's code size.

B. SSH Server Implementations

We also reported all the new bugs found in SSH servers. The Invalid *CHANNEL_CLOSE* Response and Missing *SERVICE_REQUEST_USERAUTH* bugs found in Dropbear were both confirmed. Dropbear's developer said they planned to issue a fix for the former, while the latter is a result of a design choice (i.e., it is a **wontfix** bug). BitVise's head developer also confirmed our bugs, however, they said no fix would be planned, as the bugs do not represent security vulnerabilities. We have not received feedback from OpenSSH developers at the time of this writing.

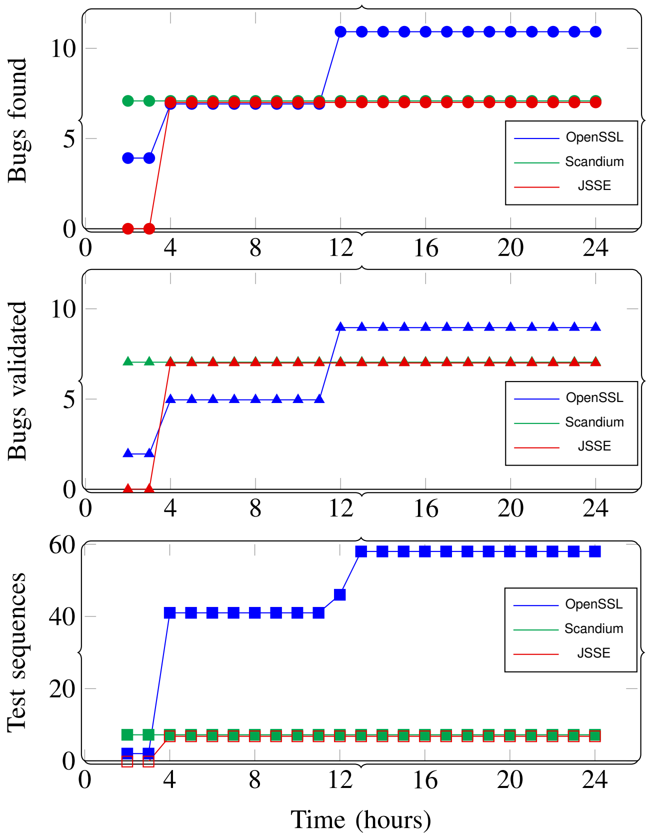


Fig. 10: Number of bugs found/validated and number of input sequences required to validate bugs for OpenSSL, Scandium and JSSE clients, with increasingly accurate models.

Appendix B
Effectiveness on Inaccurate Models

To assess how the accuracy of the models affects the effectiveness of our technique, we applied it to three implementations, JSSE 12.0.2, OpenSSL 1.1.1b and Scandium 2.0.0-M16. We extracted from the learning process a hypothesis each hour, and applied our technique using the hypotheses as models. Figure 10 shows the number of bugs found during, number of bugs validated, and number of input sequences for the validated bugs, as time (and therefore accuracy of the models) increases.