Model Learning and Model Checking of SSH Implementations

Paul Fiterău-Broştean[*]
Radboud University Nijmegen
p.fiterau-brostean@science.ru.nl

Toon Lenaerts
Radboud University Nijmegen
toon.lenaerts@student.ru.nl

Erik Poll
Radboud University Nijmegen
erikpoll@cs.ru.nl

Joeri de Ruiter
Radboud University Nijmegen
joeri@cs.ru.nl

Frits Vaandrager
Radboud University Nijmegen
F.Vaandrager@cs.ru.nl

Patrick Verleg
Radboud University Nijmegen
patrickverleg@gmail.com

ABSTRACT

We apply model learning on three SSH implementations to infer state machine models, and then use model checking to verify that these models satisfy basic security properties and conform to the RFCs. Our analysis showed that all tested SSH server models satisfy the stated security properties. However, our analysis uncovered several violations of the standard.

CCS CONCEPTS

•Networks →Protocol correctness; •Theory of computation →Active learning; •Software and its engineering →Model checking; •Security and privacy →*Logic and verification; Network security;*

KEYWORDS

Model learning, model checking, SSH protocol

ACM Reference format:
Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. 2017. Model Learning and Model Checking of SSH Implementations. In *Proceedings of SPIN Symposium, California, USA, July 2017 (SPIN'17),* 10 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

SSH is a security protocol that is widely used to interact securely with remote machines. The Transport layer of SSH has been subjected to security analysis [31], incl. analyses that revealed cryptographic shortcomings [5, 7, 20].

Whereas these analyses consider the abstract cryptographic protocol, this paper looks at actual implementations of SSH, and investigates flaws in the program logic of these implementations, rather than cryptographic flaws. Such logical flaws have occurred in implementations of other security protocols, notably TLS, with Apple's 'goto fail' bug and the FREAK attack [8]. For this we use model learning (a.k.a. active automata learning) [6, 21, 28] to infer state machines of three SSH implementations, which we then analyze by model checking for conformance to both functional and security properties.

The properties we verify for the inferred state machines are based on the RFCs that specify SSH [32–35]. These properties are formalized in LTL and verified using NuSMV [13]. We use a model checker since the models are too complex for manual inspection (they are trivial for NuSMV). Moreover, by formalizing the properties we can better assess and overcome vagueness or under-specification in the RFC standards.

This paper is born out of two recent theses [19, 29], and is to our knowledge the first combined application of model learning and model checking in verifying SSH implementations, or more generally, implementations of any network security protocol.

Related work. Chen et al.[12] use the MOPS software model checking tool to detect security vulnerabilities in the OpenSSH C implementation due to violation of folk rules for the construction of secure programs such as "Do not open a file in writing mode to stdout or stderr". Udrea et al.[27] also investigated SSH implementations for logical flaws. They used a static analysis tool to check two C implementations of SSH against an extensive set of rules. These rules not only express properties of the SSH protocol logic, but also of message formats and support for earlier versions and various options. Our analysis only considers the protocol logic. However, their rules were tied to routines in the code, so had to be slightly adapted to fit the different implementations. In contrast, our properties are defined at an abstract level so do not need such tailoring. Moreover, our black box approach means we can analyze any implementation of SSH, not just open source C implementations.

Formal models of SSH in the form of state machines have been used before, namely for a manual code review of OpenSSH [23], formal program verification of a Java implementation of SSH [22], and for model based testing of SSH implementations [9]. All this research only considered the SSH Transport layer, and not the other SSH protocol layers.

Model learning has previously been used to infer state machines of EMV bank cards [3], electronic passports [4], hand-held readers for online banking [11], and implementations of TCP [14] and TLS [25]. Some of these studies relied on manual analysis of learned models [3, 4, 25], but some also used model checkers [11, 14].

Instead of using active learning as we do, it is also possible to use passive learning to obtain protocol state machines [30]. Here network traffic is observed, and not actively generated. This can then also provide a probabilistic characterization of normal network

[*]Supported by NWO project 612.001.216, Active Learning of Security Protocols (ALSEP).

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SPIN'17, California, USA
© 2017 Copyright held by the owner/author(s). 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

traffic, but it cannot uncover implementation flaws that occur in strange message flows, which is our goal.

2 MODEL LEARNING

2.1 Mealy machines

A *Mealy machine* is a tuple $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$, where I is a finite set of inputs, O is a finite set of outputs, Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times I \to Q$ is a transition function, and $\lambda : Q \times I \to O$ is an output function. Output function λ is extended to sequences of inputs by defining, for all $q \in Q$, $i \in I$ and $\sigma \in I^*$, $\lambda(q, \epsilon) = \epsilon$ and $\lambda(q, i\sigma) = \lambda(q, i)\lambda(\delta(q, i), \sigma)$. The behavior of Mealy machine $\mathcal{M}$ is defined by function $A_{\mathcal{M}} : I^* \to O^*$ with $A_{\mathcal{M}}(\sigma) = \lambda(q^0, \sigma)$, for $\sigma \in I^*$. Mealy machines $\mathcal{M}$ and $\mathcal{N}$ are *equivalent*, denoted $\mathcal{M} \approx \mathcal{N}$, iff $A_{\mathcal{M}} = A_{\mathcal{N}}$. Sequence $\sigma \in I^*$ *distinguishes* $\mathcal{M}$ and $\mathcal{N}$ if and only if $A_{\mathcal{M}}(\sigma) \neq A_{\mathcal{N}}(\sigma)$.

2.2 MAT Framework

The most efficient algorithms for model learning all follow the pattern of a *minimally adequate teacher (MAT)* as proposed by Angluin [6]. Here learning is viewed as a game in which a *learner* has to infer an unknown automaton by asking queries to a teacher. The teacher knows the automaton, which in our setting is a Mealy machine $\mathcal{M}$, also called the System Under Learning (SUL). Initially, the LEARNER only knows the input alphabet I and output alphabet O of $\mathcal{M}$. The task of the LEARNER is to learn $\mathcal{M}$ via two types of queries:

- With a *membership query*, the LEARNER asks what the response is to an input sequence $\sigma \in I^*$. The teacher answers with the output sequence in $A_{\mathcal{M}}(\sigma)$.
- With an *equivalence query*, the LEARNER asks whether a hypothesized Mealy machine $\mathcal{H}$ is correct, that is, whether $\mathcal{H} \approx \mathcal{M}$. The teacher answers *yes* if this is the case. Otherwise it answers *no* and supplies a *counterexample*, which is a sequence $\sigma \in I^*$ that triggers a different output sequence for both Mealy machines, that is, $A_{\mathcal{H}}(\sigma) \neq A_{\mathcal{M}}(\sigma)$.

The MAT framework can be used to learn black box models of software. If the behavior of a software system, or System Under Learning (SUL), can be described by some unknown Mealy machine $\mathcal{M}$, then a membership query can be implemented by sending inputs to the SUL and observing resulting outputs. An equivalence query can be approximated using a conformance testing tool [18] via a finite number of *test queries.* A test query consists of asking the SUL for the response to an input sequence $\sigma \in I^*$, similar to a membership query. Note that this cannot rule out that there is more behavior that has not been discovered. For a recent overview of model learning algorithms for this setting see [17].

2.3 Abstraction

Most current learning algorithms are only applicable to Mealy machines with small alphabets comprising abstract messages. Practical systems typically have parameterized input/output alphabets, whose application triggers updates on the system's state variables. To learn these systems we place a *mapper* between the LEARNER and the SUL. The MAPPER is a transducer which translates concrete inputs to abstract inputs and concrete outputs to abstract outputs. For a thorough discussion of mappers, we refer to [2].

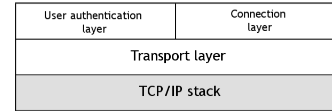

Figure 1: SSH protocol layers

3 THE SECURE SHELL PROTOCOL

The Secure Shell Protocol (or SSH) is a protocol used for secure remote login and other secure network services over an insecure network. It runs as an application layer protocol on top of TCP, which provides reliable data transfer, but does not provide any form of connection security. The initial version of SSH was superseded by a second version (SSHv2), after the former was found to contain design flaws which could not be fixed without losing backwards compatibility [15]. This work focuses on SSHv2.

SSHv2 follows a client-server paradigm. The protocol consists of three layers (Figure 1):

(1) The *transport layer protocol* (RFC 4253 [35]) forms the basis for any communication between a client and a server. It provides confidentiality, integrity and server authentication as well as optional compression.
(2) The *user authentication protocol* (RFC 4252 [32]) is used to authenticate the client to the server.
(3) The *connection protocol* (RFC 4254 [33]) allows the encrypted channel to be multiplexed in different channels. These channels enable a user to run multiple applications, such as terminal emulation or file transfer, over a single SSH connection.

Each layer has its own specific messages. The SSH protocol is interesting in that outer layers do not encapsulate inner layers. This means that different layers can interact. Hence, it makes sense to analyze SSH as a whole, instead of analyzing its constituent layers independently. Below we discuss each layer, outlining the relevant messages which are later used in learning, and characterising the so-called *happy flow* that a normal protocol run follows.

At a high level, a typical SSH protocol run uses the three constituent protocols in the order given above: after the client establishes a TCP connection with the server, (1) the two sides use the Transport layer protocol to negotiate key exchange and encryption algorithms, and use these to establish session keys, which are then used to secure further communication; (2) the client uses the user authentication protocol to authenticate to the server; (3) the client uses the connection protocol to access services on the server, for example the terminal service.

3.1 Transport layer

SSH runs over TCP, and provides end-to-end confidentiality and integrity using session keys. Once a TCP connection has been established with the server, these session keys are securely negotiated using a *key exchange* algorithm, the first step of the protocol. The key exchange begins by the two sides exchanging their preferences for the key exchange algorithm to be used, as well as encryption,

compression and hashing algorithms. Preferences are sent with a KEXINIT message. Subsequently, key exchange using the negotiated algorithm takes place. Following this algorithm, one-time session keys for encryption and hashing are generated by each side, together with an identifier for the session. The main key exchange algorithm is Diffie-Hellman, which is also the only one required by the RFC. For the Diffie-Hellman scheme, KEX30 and KEX31 are exchanged to establish fresh session keys. These keys are used from the moment the NEWKEYS command has been issued by both parties. A subsequent SR_AUTH requests the authentication service. The happy flow thus consists of the succession of the three steps comprising key exchange, followed up by a successful authentication service request. The sequence is shown in Figure 2.

Figure 2: The happy flow for the Transport layer.

Key re-exchange [35, p. 23], or *rekeying*, is an almost identical process, the difference being that instead of taking place at the beginning, it takes place once session keys are already in place. The purpose is to renew session keys so as to foil potential replay attacks [34, p. 17]. It follows the same steps as key exchange. A fundamental property of rekeying is that it should preserve the state; that is, after the rekeying procedure is completed, the protocol should be in the same state as it was before the rekeying started, with as only difference that new keys are now in use.

3.2 Authentication layer

Once a secure tunnel has been established, the client can authenticate. For this, four authentication methods are defined in RFC 4252 [32]: password, public-key, host-based and none. The authentication request includes a user name, service name and authentication data, which consists of both the authentication method as well as the data needed to perform the actual authentication, such as the password or public key. The happy flow for this layer, as shown in Figure 3, is simply a single protocol step that results in a successful authentication. The messages UA_PW_OK and UA_PK_OK achieve this for respectively password and public key authentication.

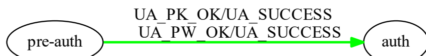

Figure 3: The happy flow for the user Authentication layer.

3.3 Connection layer

Successful authentication makes services of the Connection layer available. The Connection layer enables the user to open and close channels of various types, with each type providing access to specific services. Of the various services available, we focus on the remote terminal over a session channel, a classical use of SSH. The happy flow consists of opening a session channel, CH_OPEN, requesting a "pseudo terminal" CH_REQUEST_PTY, optionally sending and managing data via the messages CH_SEND_DATA, CH_WINDOW_ADJUST, CH_SEND_EOF, and eventually closing the channel via CH_CLOSE, as depicted in Figure 4.

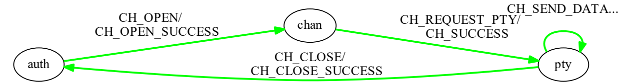

Figure 4: The happy flow for the Connection layer.

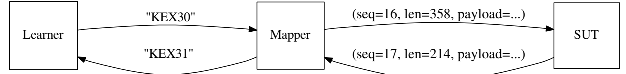

Figure 5: The SSH learning setup.

4 THE LEARNING SETUP

The learning setup consists of three components: the LEARNER, the MAPPER and the SUL. The LEARNER generates abstract inputs, representing SSH messages. The MAPPER transforms these messages into well-formed SSH packets and sends them to the SUL. The SUL sends response packets back to the MAPPER, which in turn, translates these packets to abstract outputs. The MAPPER then sends the abstract outputs back to the LEARNER.

The LEARNER uses LearnLib [24], a Java library implementing L^* based algorithms for learning Mealy machines. The MAPPER is based on Paramiko, an open source SSH implementation written in Python[1]. We opted for Paramiko because its code is relatively well structured and documented. The SUL can be any existing implementation of an SSH server. The three components communicate over sockets, as shown in Figure 5.

SSH is a complex client-server protocol. In our work so far we concentrated on learning models of the implementation of the server, and not of the client. We further restrict learning to only exploring the terminal service of the Connection layer, as we consider it to be the most interesting from a security perspective. Algorithms for encryption, compression and hashing are left to default settings and are not purposefully explored. Also, the starting state of the SUL is one where a TCP connection has already been established and where SSH versions have been exchanged, which are prerequisites for starting the Transport layer protocol.

4.1 The learning alphabet

The alphabet we use consists of inputs, which correspond to messages sent to the server, and outputs, which correspond to messages received from the server. We split the input alphabet into three parts, one for each of the protocol layers.

Learning does not scale with a growing alphabet, and since we are only learning models of servers, we remove those inputs that are not intended to ever be sent to the server[2]. Furthermore, from the Connection layer we only use messages for channel management and the terminal functionality. Finally, because we will only explore protocol behavior after SSH versions have been exchanged, we exclude the messages for exchanging version numbers.

The resulting lists of inputs for the three protocol layers are given in tables 1-3. In some experiments, we used only a subset of the most essential inputs, to further speed up experiments. This

[1] Paramiko is available at http://www.paramiko.org/

[2] This means we exclude the messages SERVICE_ACCEPT, UA_ACCEPT, UA_FAILURE, UA_BANNER, UA_PK_OK, UA_PW_CHANGEREQ, CH_SUCCESS and CH_FAILURE from our alphabet.

restricted alphabet significantly decreases the number of queries needed for learning models while only marginally limiting explored behavior. We discuss this again in Section 5. Inputs included in the restricted alphabet are marked by '*' in the tables below.

Table 1 lists the Transport layer inputs. We include a version of the KEXINIT message with `first_kex_packet_follows` disabled. This means no guess [35, p. 17] is attempted on the SUL's parameter preferences. Consequently, the SUL will have to send its own KEXINIT in order to convey its own parameter preferences before key exchange can proceed. Also included are inputs for establishing new keys (KEX30, NEWKEYS), disconnecting (DISCONNECT), as well as the special inputs IGNORE, UNIMPL and DEBUG. The latter are not interesting, as they are normally ignored by implementations. Hence they are excluded from our restricted alphabet. DISCONNECT proved costly time wise, so was also excluded.

Message	Description
DISCONNECT	Terminates the current connection [35, p. 23]
IGNORE	Has no intended effect [35, p. 24]
UNIMPL	Intended response to unrecognized messages [35, p. 25]
DEBUG	Provides other party with debug information [35, p. 25]
KEXINIT*	Sends parameter preferences [35, p. 17]
KEX30*	Initializes the Diffie-Hellman key exchange [35, p. 21]
NEWKEYS*	Requests to take new keys into use [35, p. 21]
SR_AUTH*	Requests the authentication protocol [35, p. 23]
SR_CONN*	Requests the connection protocol [35, p. 23]

Table 1: Transport layer inputs

The Authentication layer defines one single client message type in the form of the authentication request [32, p. 4]. Its parameters contain all information needed for authentication. Four authentication methods exist: none, password, public key and host-based. Our mapper supports all methods except the host-based authentication because some SUTs don't support this feature. Both the public key and password methods have OK and NOK variants, which provide respectively correct and incorrect credentials. Our restricted alphabet supports only public key authentication, as the implementations processed this faster than the other authentication methods.

Message	Description
UA_NONE	Authenticates with the "none" method [32, p. 7]
UA_PK_OK*	Provides a valid name/key pair [32, p. 8]
UA_PK_NOK*	Provides an invalid name/key pair [32, p. 8]
UA_PW_OK	Provides a valid name/password pair [32, p. 10]
UA_PW_NOK	Provides an invalid name/password pair [32, p. 10]

Table 2: Authentication layer inputs

The Connection layer allows the client to manage channels and to request/run services over them. In accordance with our learning goal, our mapper only supports inputs for requesting terminal emulation, plus inputs for channel management as shown in Table 3. The restricted alphabet only supports the most general channel management inputs. Those excluded are not expected to produce state change.

4.2 The mapper

The MAPPER must provide a translation between abstract messages and well-formed SSH messages: it has to translate abstract inputs listed in Tables 1-3 to actual SSH packets, and translate the SSH packets received in answer to our abstract outputs.

Message	Description
CH_OPEN*	Opens a new channel [33, p. 5]
CH_CLOSE*	Closes a channel [33, p. 9]
CH_EOF*	Indicates that no more data will be sent [33, p. 9]
CH_DATA*	Sends data over the channel [33, p. 7]
CH_EDATA	Sends typed data over the channel [33, p. 8]
CH_WINDOW_ADJUST	Adjusts the window size [33, p. 7]
CH_REQUEST_PTY*	Requests terminal emulation [33, p. 11]

Table 3: Connection layer inputs

A special case here occurs when no output is received from the SUL; in that case the MAPPER gives back to the learner a NO_RESP message, to indicate that a time-out occurred.

The sheer complexity of the MAPPER meant that it was easier to adapt an existing SSH implementation, rather than construct the MAPPER from scratch. Paramiko already provides mechanisms for encryption/decryption, as well as routines for constructing and sending the different types of packets, and for receiving them. These routines are called by control logic dictated by Paramiko's own state machine. The MAPPER was constructed by replacing this control logic with one dictated by messages received from the LEARNER.

The MAPPER maintains a set of state variables to record parameters of the ongoing session, including for example the server's preferences for key exchange and encryption algorithm, parameters of these protocols, and – once it has been established – the session key. These parameters are updated when receiving messages from the server, and are used to concretize inputs to actual SSH messages to the server.

For example, upon receiving a KEXINIT, the MAPPER saves the SUL's preferences for key exchange, hashing and encryption algorithms. Initially these parameters are all set to the defaults that any server should support, as required by the RFC. The MAPPER supports Diffie-Hellman key exchange, which it will initiate if it gets a KEX30 input from the learner. After this, the SUL responds with a KEX31 message (assuming the protocol run so far is correct), and from this message, the MAPPER saves the hash, as well as the new keys. Receipt of the NEWKEYS response from the SUL will make the MAPPER use the new keys earlier negotiated in place of the older ones, if such existed.

The MAPPER contains a buffer for storing channels opened, which is initially empty. On a CH_OPEN from the learner, the MAPPER adds a channel to the buffer with a randomly generated channel identifier, on a CH_CLOSE, it removes the channel (if there was any). The buffer size, or the maximum number of opened channels, is limited to one. Initially the buffer is empty. Lastly, the MAPPER also stores the sequence number of the last received message from the SUL. This number is then used when constructing UNIMPL inputs.

In the following cases, inputs are answered by the MAPPER directly instead of being sent to the SUL to find out its response: (1) on receiving a CH_OPEN input if the buffer has reached the size limit, the MAPPER directly responds with CH_MAX; (2) on receiving any input operating on a channel (all Connection layer inputs other than CH_OPEN) when the buffer is empty, the MAPPER directly responds with CH_NONE; (3) if connection with the SUL was terminated, the MAPPER responds with a NO_CONN message, as sending further messages to the SUL is pointless in that case.

In many ways, the MAPPER acts similar to an SSH client, hence the decision to built it by adapting an existing client implementation.

4.3 Practical complications

SSH implementations can exhibit non-determistic behavior. The learning algorithm cannot cope with non-determinism – learning will not terminate – so this has to be detected, which our MAPPER does. There are a few sources of non-determinism in SSH:

1. Underspecification in the SSH specification (for example, by not specifying the order of certain messages) allows some non-deterministic behavior. Even if client and server do implement a fixed order for messages they sent, the asynchronous nature of communication means that the interleaving of sent and received messages may vary. Moreover, client and server are free to intersperse DEBUG and IGNORE messages at any given time[3]
2. Timing is another source of non-deterministic behavior. For example, the MAPPER might time-out before the SUL had sent its response. Some SULS also behave unexpectedly when a new query is received too shortly after the previous one. Hence in our experiments we adjusted time-out periods accordingly so that neither of these events occur, and the SUL behaves deterministically all the time.

To detect non-determinism, the MAPPER caches all observations in an SQLite database and verifies if a new observation is different to one cached from a previous protocol run. If so, it raises a warning, which then needs to be manually investigated.

An added benefit of this cache is that it allows the MAPPER to supply answer to some inputs without actually sending them to the SUL. This sped up learning a lot when we had to restart experiments: any new experiment on the same SUL could start where the previous experiment left of, without re-running all inputs. This was an important benefit, as experiments could take several days.

Another practical problem besides non-determinism is that an SSH server may produce a sequence of outputs in response to a single input. This means it is not behaving as a Mealy machines, which allows for only one output. Dealing with this is simple: the MAPPER concatenates all outputs into one, and it produces this sequence as the single output to the LEARNER.

A final challenge is presented by forms of 'buffering', which we encountered in two situations. Firstly, some implementations buffer incoming requests during rekey; only once the rekeying is complete are all these messages processed. This leads to a NEWKEYS response (indicating rekeying has completed), directly followed by all the responses to the buffered requests. This would lead to non-termination of the learning algorithm, as for every sequence of buffered messages the response is different. To prevent this, we treat the sequence of queued responses as a single output BUFFERED.

Secondly, buffering happens when opening and closing channels, since a SUL can close only as many channels as have previously been opened. Learning this behavior would lead to an infinite state machine, as we would need a state 'there are *n* channels open' for every number *n*. For this reason, we restrict the number of simultaneously open channels to one. The MAPPER returns a custom response CH_MAX to a CH_OPEN message whenever this limit is reached.

[3] The IGNORE messages are aimed to thwart traffic analysis.

5 LEARNING RESULTS

We use the setup described in Section 4 to learn models for OpenSSH, BitVise and DropBear SSH server implementations. OpenSSH represents the focal point, as it is the most popular implementation of SSH (with over 80 percent of market share in 2008 [5]) and the default server for many UNIX-based systems. DropBear is an alternative to OpenSSH designed for low resource systems. BitVise is a well-known proprietary Windows-only SSH implementation.

In our experimental setup, LEARNER and MAPPER ran inside a Linux Virtual Machine. OpenSSH and DropBear were learned over a localhost connection, whereas BitVise was learned over a virtual connection with the Windows host machine. We have adapted the setting of timing parameters to each implementation.

OpenSSH was learned using a full alphabet, whereas DropBear and BitVise were learned using a restricted alphabet (as defined in Subsection 4.1). The primary reason for using a restricted alphabet was to speed up learning. Most inputs excluded were inputs that either didn't change behavior (like DEBUG or UNIMPL), or that proved costly time-wise, and were not critical to penetrating all layers. A concrete example is the user/password based authentication inputs (UA_PW_OK and UA_PW_NOK). It would take the system 2-3 seconds to respond to an invalid password, a typical countermeasure to slow down brute force attacks. By contrast, public key authentication resulted in quick responses. The DISCONNECT input presented similar challenges, as it would take a varying amount of time until the system responded. This was particularly problematic for BitVise.

For the test queries we used random and exhaustive variants of the testing algorithm described in [26], which generate efficient test suites. Tests generated comprise an access sequence, a middle section of length k and a distinguishing sequence. The exhaustive variant for a set k generates tests for all possible middle sections and all states. Passing all tests then provides some notion of confidence, namely, that the learned model is correct unless the (unknown) model of the implementation has at least k more states than the learned hypothesis. The random variant produces tests with randomly generated middle sections. No formal confidence is provided, but past experience shows this to be more effective at finding counterexamples since k can be set to higher values. We executed a random test suite with k of 4 comprising 40000 tests for OpenSSH, and 20000 tests for BitVise and DropBear. We then ran an exhaustive test suite with k of 2 for all implementations.

Table 4 describes the exact versions of the systems analyzed together with statistics on learning and testing, namely: (1) the number of states in the learned model, (2) the number of hypotheses built during the learning process and (3) the total number of learning and test queries run. For test queries, we only consider those run on the last hypothesis. All learned models along with the specifications checked can be found at https://gitlab.science.ru.nl/pfiteraubrostean/Learning-SSH-Paper/tree/master/models.

SUT	States	Hypotheses	Mem. Q.	Test Q.
OpenSSH 6.9p1-2	31	4	19836	76418
BitVise 7.23	65	15	24996	58423
DropBear v2014.65	29	8	8357	64478

Table 4: Statistics for learning experiments

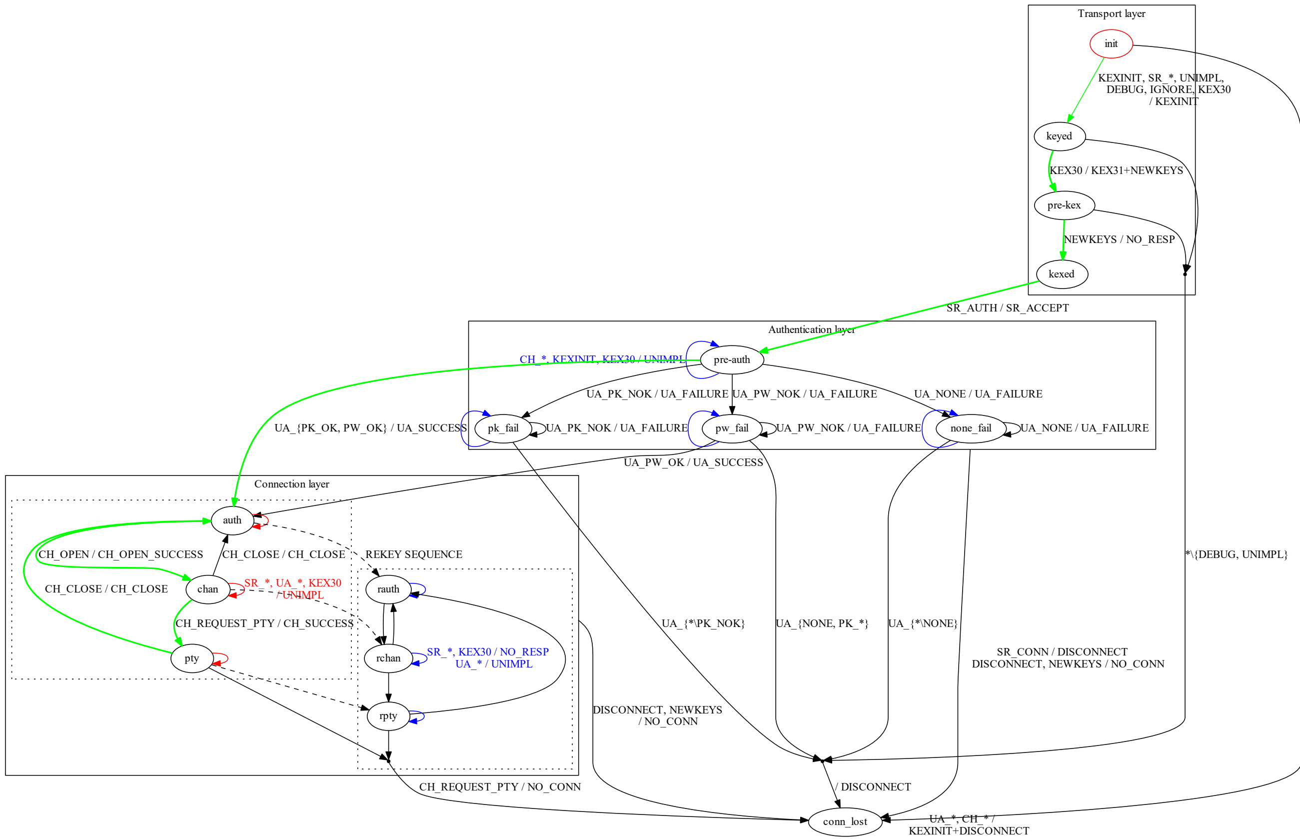

Figure 6: Model of the OpenSSH server. States are collected in 3 clusters, indicated by the rectangles, where each cluster corresponds to one of the protocol layers. We eliminate redundant states and information induced by the MAPPER, as well as states present in successful rekeying sequences. Wherever rekeying was permitted, we replaced the rekeying states and transitions by a single *REKEY SEQUENCE* transition. We also factor out edges common to states within a cluster. We replace common disconnecting edges, by one edge from the cluster to the disconnect state. Common self loop edges are colored, and the actual i/o information only appears on one edge. Transitions with similar start and end states are joined together on the same edge. Transition labels are kept short by regular expressions(UA_* stands for all inputs starting with UA_) or by factoring out common start strings. Green edges highlight the happy flow.

The large number of states is down to several reasons. First of all, some systems exhibited buffering behavior. In particular, BitVise would queue responses for higher layer inputs sent during key re-exchange, and would deliver them all at once after rekeying was done. Rekeying was also a major contributor to the number of states. For each state where rekeying is possible, the sequence of transitions constituting the complete rekeying process should lead back to that state. This leads to two additional rekeying states for every state allowing rekey. Many states were also added due to MAPPER generated outputs such as CH_NONE or CH_MAX, outputs which signal that no channel is open or that the maximum number of channels have been opened.

Figure 6 shows the model learned for OpenSSH, with some edits to improve readability. The happy flow, in green, is fully explored in the model and mostly matches our earlier description of it[4]. Also explored is what happens when a rekeying sequence is attempted. We notice that rekeying is only allowed in states of the Connection layer. Strangely, for these states, rekeying is not state preserving, as the generated output on receiving a SR_AUTH, SR_CONN or KEX30 changes from UNIMPL to NO_RESP. This leads to two sub-clusters of states, one before the first rekey, the other afterward. In all other states, the first step of a rekeying (KEXINIT) yields (UNIMPL), while the last step (NEWKEYS) causes the system to disconnect.

We were puzzled by how systems reacted to SR_CONN, the request for services of the Connection layer. These services can be accessed once the user had authenticated, without the need of a prior service request. That in itself was not strange, as authentication messages already mention that connection services should start after authentication [5]. Unexpected was that an explicit request either lead to UNIMPL/NO_RESP with no state change, as in the case of OpenSSH, or termination of the connection, as in the case of BitVise. The latter was particularly strange, as in theory, once authenticated, the user

[4]The only exception is in the Transport layer, where unlike in our happy flow definition, the server is the first to send the NEWKEYS message. This is also accepted behavior, as the protocol does not specify which side should send NEWKEYS first.

[5]This is a technical detail, the message format of authentication messages requires a field which says the service started after authentication. The only option is to start Connection layer services.

should always have access to the service, and not be disconnected when requesting this service. Only DropBear seems to respond positively (SR_ACCEPT) to SR_CONN after authentication.

We also notice the intricate authentication behavior: after an unsuccessful authentication attempt the only authentication method still allowed is password authentication. Finally, only BitVise allowed multiple terminals to be requested over the same channel. As depicted in the model, OpenSSH abruptly terminates on requesting a second terminal. DropBear exhibits a similar behavior.

6 SECURITY SPECIFICATIONS

A NuSMV model is specified via a set of finite variables together with a transition-function that describes changes on these variables. Specifications in temporal logic, such as CTL and LTL, can be checked for truth on specified models. NuSMV provides a counterexample if a given specification is not true. We generate NuSMV models automatically from the learned models. Generation proceeds by first defining a NuSMV file with three variables, corresponding to inputs, outputs and states. The transition-function is then extracted from the learned model and appended to this file. This function updates the output and state variables for a given valuation of the input variable and the current state. Figure 7 gives an example of a Mealy machine and its associated NuSMV model.

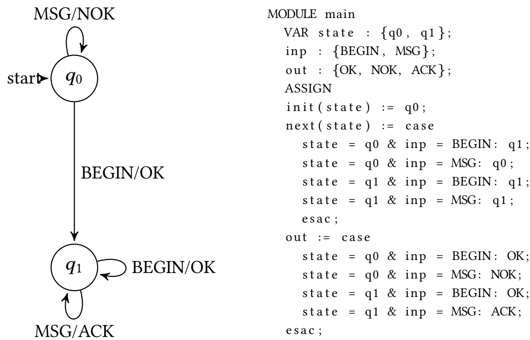

Figure 7: Mealy machine + associated NuSMV code

The remainder of this section defines the properties we formalized and verified. We group these properties into four categories:

(1) *basic characterizing properties*, properties which characterize the MAPPER and SUL assembly at a basic level. These hold for all implementations.
(2) *security properties*, these are properties fundamental to achieving the main security goal of the respective layer.
(3) *key re-exchange properties*, or properties regarding the rekey operation (after the initial key exchange was done).
(4) *functional properties*, which are extracted from the SHOULD's and the MUST's of the RFC specifications. They may have a security impact.

A key note is that properties are checked not on the actual concrete model of the SUL, but on an abstract model, which represents an over-approximation of the SUL induced by the MAPPER. This is unlike in [14], where properties where checked on a concretization of the learned model, concretization obtained by application of a reverse mapping. Building a reverse mapper is far from trivial given the MAPPER's complexity. Performing model checking on an abstract model means we cannot fully translate model checking results to the concrete (unknown) model of the implementation. In particular, properties which hold for the abstract model do not necessarily hold for the implementation. Properties that don't hold however, also don't hold for the SUL.

Before introducing the properties, we mention some basic predicates and conventions we use in their definition. The happy flow in SSH consists in a series of steps, the user first exchanges keys, then requests for the authentication service, followed up by supplying valid credentials to authenticate, concluded by opening of a channel. Whereas the first step is complex, the subsequent steps can be captured by the simple predicates *hasReqAuth*, *validAuthReq* and *hasOpenedChannel* respectively. The predicates are defined in terms of the output generated at a given moment, with certain values of this output indicating that the step was performed successfully. For example, CH_OPEN_SUCCESS indicates that a channel has been opened successfully. Sometimes we also need the input that generated the output, so as to distinguish this step from a different step. In particular, requesting the authentication service is distinguished from requesting the connection service by SR_AUTH. To these predicates, we add predicates for valid, invalid and all authentication methods, a predicate for the receipt of NEWKEYS from the server, and receipt of KEXINIT, which can also be seen as initiation of key (re-) exchange. These latter predicates have to be tweaked in accordance with the input alphabet used and with the output the SUL generated (KEXINIT could be sent in different packaging, either alone, or joined by a different message). Their formulations correspond to the OpenSSH setting. Finally, by *connLost* we define a predicate suggesting that connection was lost, and by *endCondition*, the end condition for most higher layer properties.

```
hasReqAuth := inp=SR_AUTH & out=SR_ACCEPT;
validAuthReq := out=UA_PK_OK | out=UA_PW_OK;
hasOpenedChannel := out=CH_OPEN_SUCCESS;
validAuthReq := inp=UA_PK_OK | inp=UA_PW_OK;
invAuthReq := inp=UA_PK_NOK | inp=UA_PW_NOK | inp=UA_NONE;
authReq := validAuthReq | invalidAuthReq;
receivedNewKeys := out=NEWKEYS | out=KEX31_NEWKEYS;
kexStarted := out=KEXINIT;
connLost := out=NO_CONN | out=DISCONNECT;
endCondition := kexStarted | connLost;
```

Our formulation uses NuSMV syntax. We also use the weak until operator W, which is not supported by NuSMV, but can be easily defined in terms of the until operator U and globally operator G that are supported: $p\,W\,q \;=\; p\,U\,q \,|\, G\,p$. Many of the higher layer properties we formulate should hold only until a disconnect or a key (re-)exchange happens, hence the definition of the *endCondition* predicate. This is because the RFC's don't specify what should happen when no connection exists. Moreover, higher layer properties in the RFC's only apply outside of rekey sequences, as inside a rekey sequence, the RFC's suggest implementations to reject all higher layer inputs, regardless of the state before the rekey.

6.1 Basic characterizing properties

In our setting, a single TCP connection with the system is made and once this connection is lost (e.g. because the system disconnects),

it can not be re-established. The moment a connection is lost is marked by generation of the NO_CONN output. From this moment onwards, the only outputs encountered are the NO_CONN output (the MAPPER tried but failed to communicate with the SUL), or outputs generated by the MAPPER directly, without querying the system. The latter are CH_MAX (channel buffer is full) and CH_NONE (channel buffer is empty). With these outputs we define Property 1 which describes the "one connection" property of our setup.

Property 1.
```
G ( out=NO_CONN ->
  G ( out=NO_CONN | out=CH_MAX | out=CH_NONE ) )
```

Outputs CH_MAX and CH_NONE are still generated because of a characteristic we touched on in Subsection 4.2. The MAPPER maintains a buffer of open channels and limits its size to 1. From the perspective of the MAPPER, a channel is open, and thus added to the buffer, whenever CH_OPEN is received from the learner, regardless if a channel was actually opened on the SUL. In particular, if after opening a channel via CH_OPEN an additional attempt to open a channel is made, the MAPPER itself responds by CH_MAX without querying the SUL. This continues until the LEARNER closes the channel by CH_CLOSE, prompting removal of the channel and the sending on an actual CLOSE message to the SUL (hence out!=CH_NONE). A converse property can be formulated in a similar way for when the buffer is empty after a CH_CLOSE, in which case subsequent CH_CLOSE messages prompt the MAPPER generated CH_NONE, until a channel is opened via CH_OPEN and an actual OPEN message is sent to the SUL. Conjunction of these two behaviors forms Property 2.

Property 2.
```
(G ( inp=CH_OPEN) ->
    X ( ( inp=CH_OPEN -> out=CH_MAX)
      W ( inp=CH_CLOSE & out!=CH_NONE) ) ) &
(G ( inp=CH_CLOSE) ->
    X ( ( inp=CH_CLOSE -> out=CH_NONE)
      W ( inp=CH_OPEN & out!=CH_MAX) ) )
```

6.2 Security properties

In SSH, upper layer services assume some notions of security, notions which are ensured by mechanisms in the lower layers. These mechanisms should have to be first engaged for the respective upper layer services to become available. As an example, the authentication service should only become available after exchanging and employing of kryptographic keys (key exchange) was done in the Transport layer, otherwise the service would be running over an unencrypted channel. Requests for this service should therefore not succeed unless key exchange was performed successfully.

Key exchange implies three steps which have to be performed in order but may be interleaved by other actions. Successful authentication necessarily implies successful execution of the key exchange steps. We can tell each key exchange step were successful from the values of the input and output variables. Successful authentication request is indicated by the predicate defined earlier, *hasReqAuth*. Following these principles, we define the LTL specification in Property 3, where O is the once operator. Formula *Op* is true at time t if p held in at least one of the previous time steps $t' \leq t$.

Property 3.
```
G ( hasReqAuth ->
    O ( ( inp=NEWKEYS & out=NO_RESP) &
      O ( ( inp=KEX30 & out=KEX31_NEWKEYS) &
        O ( out=KEXINIT) ) ) )
```

Apart from a secure connection, Connection layer services also assume that the client behind the connection was authenticated. This is ensured by the Authentication layer by means of an authentication mechanism, which only succeeds, and thus authenticates the client, if valid credentials are provided. For the implementation to be secure, there should be no path from an unauthenticated to an authenticated state without the provision of valid credentials. We consider an authenticated state as a state where a channel has been opened successfully, captured by the predicate *hasOpenedChannel*. Provision of valid/invalid credentials is indicated by the outputs UA_SUCCESS and UA_FAILURE respectively. Along these lines, we formulate this specification by Property 4, where S stands for the since operator. Formula *pSq* is true at time t if q held at some time $t' \leq t$ and p held in all times t'' such that $t' < t'' \leq t$.

Property 4.
```
G ( hasOpenedChannel ->
  out!=UA_FAILURE S out=UA_SUCCESS )
```

6.3 Key re-exchange properties

According to the RFC [33, p. 24], re-exchanging keys (or rekeying) (1) is preferably allowed in all states of the protocol, and (2) its successful execution does not affect operation of the higher layers. We consider two general protocol states, pre-authenticated (after a successful authentication request, before authentication) and authenticated. These may map to multiple states in the learned models. We formalized requirement (1) by two properties, one for each general state. In the case of the pre-authenticated state, we know we have reached this state following a successful authentication service request, indicated by the predicate *hasReqAuth*. Once here, performing the inputs for rekey in succession should imply success until one of two things happen, the connection is lost(*connLost*) or we have authenticated. This is asserted in Property 5. A similar property is defined for the authenticated state.

Property 5.
```
G ( hasReqAuth ->
  X ( inp=KEXINIT -> out=KEXINIT &
    X ( inp=KEX30 -> out=KEX31_NEWKEYS &
      X ( inp=NEWKEYS -> out=NO_RESP) ) ) W
        ( connLost | hasAuth ) )
```

Requirement (2) cannot be expressed in LTL, since in LTL we cannot specify that two states are equivalent. We therefore checked this requirement directly, by writing a simple script which, for each state q that allows rekeying, checks if the state q' reached after a successful rekey is equivalent to q in the subautomaton that only contains the higher layer inputs.

6.4 Functional properties

We have formalized and checked several other properties drawn from the RFCs. We found parts of the specification unclear, which sometimes meant that we had to give our own interpretation before we could formalize. A first general property can be defined for the DISCONNECT output. The RFC specifies that after sending this message, a party MUST not send or receive any data [35, p. 24]. While we cannot tell what the server actually receives, we can check that the server does not generate any output after sending DISCONNECT. After a DISCONNECT message, subsequent outputs should be solely derived by the MAPPER. Knowing the MAPPER

induced outputs are NO_CONN, CH_MAX and CH_NONE, we formulate by Property 6 to describe expected outputs after a DISCONNECT.

PROPERTY 6. G (out=DISCONNECT ->
X G (out=CH_NONE | out=CH_MAX | out=NO_CONN))

The RFC states in [33, p. 24] that after sending a KEXINIT message, a party MUST not send another KEXINIT, or a SR_ACCEPT message, until it has sent a NEWKEYS message(*receivedNewKeys*). This is translated to Property 7.

PROPERTY 7. G (out=KEXINIT ->
X ((out!=SR_ACCEPT & out!=KEXINIT) W receivedNewKeys))

The RFC also states [33, p. 24] that if the server rejects the service request, "it SHOULD send an appropriate SSH_MSG_DISCONNECT message and MUST disconnect". Moreover, in case it supports the service request, it MUST send a SR_ACCEPT message. Unfortunately, it is not evident from the specification if rejection and support are the only allowed outcomes. We assume that is the case, and formalize an LTL formula accordingly by Property 8. For a service request (SR_AUTH), in case we are not in the initial state, the response will be either an accept (SR_ACCEPT), disconnect (DISCONNECT), or NO_CONN, output generated by the MAPPER after the connection is lost. We adjusted the property for the initial state since all implementations responded with KEXINIT which would easily break the property. We cannot yet explain this behavior.

PROPERTY 8. G ((inp=SR_AUTH & state!=s0) ->
(out=SR_ACCEPT | out=DISCONNECT | out=NO_CONN))

The RFC for the Authentication layer states in [32, p. 6] that if the server rejects the authentication request, it MUST respond with a UA_FAILURE message. Rejected requests are suggested by the predicate *invAuthReq*. In case of requests with valid credentials (*validAuthReq*), a UA_SUCCESS MUST be sent only once. While not explicitly stated, we assume this to be in a context where the authentication service had been successfully requested, hence we use the *hasReqAuth* predicate. We define two properties, Property 9 for behavior before an UA_SUCCESS, Property 10 for behavior afterward. For the first property, note that (*hasReqAuth*) may hold even after successful authentication, but we are only interested in behavior between the first time (*hasReqAuth*) holds and the first time authentication is successful (out=UA_SUCCESS), hence the use of the O operator. As is the case with most higher layer properties, the first property only has to hold until the end condition holds(*endCondition*), that is the connection is lost(*connLost*) or rekey was started by the SUL (*kexStarted*).

PROPERTY 9. G ((hasReqAuth & !O out=UA_SUCCESS) ->
(invalidAuthReq -> out=UA_FAILURE)
W (out=UA_SUCCESS | endCondition))

PROPERTY 10. G (out=UA_SUCCESS -> X G out!=UA_SUCCESS)

In the same paragraph, it is stated that authentication requests received after a UA_SUCCESS SHOULD be ignored. This is a weaker statement, and it requires that all authentication messages (suggested by *authReq*) after a UA_SUCCESS output should prompt no response from the system(NO_RESP) until the end condition is true. The formulation of this statement shown in Property 11.

PROPERTY 11. G (out=UA_SUCCESS ->
X ((authReq-> out=NO_RESP) W endCondition))

The Connection layer RFC states in [33, p. 9] that on receiving a CH_CLOSE message, a side MUST send back a CH_CLOSE, unless it had already sent this message for the channel. The channel must have been opened beforehand (*hasOpenedChannel*) and the property only has to hold until the end condition holds or the channel was closed(*out=CH_CLOSE*). We formulate Property 12 accordingly.

PROPERTY 12. G (hasOpenedChannel ->
((inp=CH_CLOSE) -> (out=CH_CLOSE))
W (endCondition | out=CH_CLOSE))

6.5 Model checking results

Table 5 presents model checking results. Crucially, the security properties hold for all three implementations. We had to slightly adapt our properties for BitVise as it buffered all responses during rekey (incl. UA_SUCCESS). In particular, we used *validAuthReq* instead of *out=UA_SUCCESS* to suggest successful authentication.

	Property	Key word	OpenSSH	Bitvise	DropBear
Security	Trans.		✓	✓	✓
	Auth.		✓	✓	✓
Rekey	Pre-auth.		X	✓	✓
	Auth.		✓	X	✓
Funct.	Prop. 6	MUST	✓	✓	✓
	Prop. 7	MUST	✓	✓	✓
	Prop. 8	MUST	X*	X	✓
	Prop. 9	MUST	✓	✓	✓
	Prop. 10	MUST	✓	✓	✓
	Prop. 11	SHOULD	X*	X*	✓
	Prop. 12	MUST	✓	✓	X

Table 5: Model checking results

Properties marked with '*' did not hold because implementations chose to send UNIMPL, instead of the output suggested by the RFC. As an example, after successful authentication, both BitVise and OpenSSH respond with UNIMPL to further authentication requests, instead of being silent, violating Property 11. Whether the alternative behavior adapted is acceptable, is up for debate. Definitely the RFC does not suggest it, though the RFC does leave room for interpretation of the UNIMPL message.

DropBear is the only implementation that allows rekey in both general states of the protocol. DropBear also satisfies all Transport and Authentication layer specifications, however, problematically, it violates the property of the Connection layer. Upon receiving CH_CLOSE, it responds by CH_EOF instead of CH_CLOSE, not respecting Property 12.

7 CONCLUSIONS

We have combined model learning with abstraction techniques to infer models of the OpenSSH, Bitvise and DropBear SSH server implementations. We have also formalized several security and functional properties drawn from the SSH RFC specifications. We have verified these properties on the learned models using model checking and have uncovered several minor standard violations. The security-critical properties were met by all implementations.

Abstraction was provided by a MAPPER component placed between the LEARNER and the SUL. The MAPPER was constructed from an existing SSH implementation. The input alphabet of the MAPPER explored key exchange, setting up a secure connection, several authentication methods and opening and closing channels over which the terminal service could be requested. We used two input alphabets, a full version for OpenSSH, and a restricted version for the other implementations. The restricted alphabet was still sufficient to explore most aforementioned behavior.

We encountered several challenges. Firstly, building a MAPPER presented a considerable technical challenge, as it required re-structuring of an actual SSH implementation. Secondly, because we used classical learning algorithms, we had to ensure that the abstracted implementation behaved like a (deterministic) Mealy Machine. Here time-induced non-determinism was difficult to eliminate. Buffering also presented problems, leading to a considerable increase in the number of states. Moreover, the systems analyzed were relatively slow, which meant learning took several days. This was compounded by the size of the learning alphabet, and it forced us into using a reduced alphabet for two of the implementations.

Limitations of the work, hence possibilities for future work, are several. First of all, the MAPPER was not formalized, unlike in [14], thus we did not produce a concretization of the abstract models. Consequently, model checking results cannot be fully transferred to the actual implementations. Formal definition of the mapper and concretization of the learned models (as defined in [2]) would tackle this. The MAPPER also caused considerable redundancy in the learned models; tweaking the abstractions used, in particular those for managing channels, could alleviate this problem while also improving learning times. This in turn would facilitate learning using expanded alphabets instead of resorting to restricted alphabets. Furthermore, the MAPPER abstraction could be refined, to give more insight into the implementations. In particular, parameters such as the session identifier could be extracted from the MAPPER and potentially handled by existing Register Automata learners[1, 10]. These learners can infer systems with parameterized alphabets, state variables and simple operations on data. Finally, we suppressed all timing-related behavior, as it could not be handled by the classical learners used; there is preliminary work on learning timed automata[16] which could use timing behavior.

Despite these limitations, our work provides a compelling application of learning and model checking in a security setting, on a widely used protocol. We hope this lays some more groundwork for further case studies, as well as advances in learning techniques.

REFERENCES

[1] F. Aarts, P. Fiterău-Broştean, H. Kuppens, and F.W. Vaandrager. 2015. Learning Register Automata with Fresh Value Generation. In *ICTAC 2015 (LNCS)*, Vol. 9399. Springer, 165–183.

[2] F. Aarts, B. Jonsson, J. Uijen, and F.W. Vaandrager. 2015. Generating Models of Infinite-State Communication Protocols using Regular Inference with Abstraction. *FMSD* 46, 1 (2015), 1–41.

[3] F. Aarts, J. de Ruiter, and E. Poll. 2013. Formal Models of Bank Cards for Free. In *Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE CS, 461–468.

[4] F. Aarts, J. Schmaltz, and F. Vaandrager. 2010. Inference and Abstraction of the Biometric Passport. In *Leveraging Applications of Formal Methods, Verification, and Validation*. LNCS, Vol. 6415. Springer, 673–686.

[5] M.R. Albrecht, K.G. Paterson, and G.J. Watson. 2009. Plaintext Recovery Attacks against SSH. In *SP'09*. IEEE, Washington, DC, USA, 16–26.

[6] D. Angluin. 1987. Learning regular sets from queries and counterexamples. *Inf. and Comp.* 75, 2 (Nov. 1987), 87–106.

[7] M. Bellare, T. Kohno, and C. Namprempre. 2004. Breaking and Provably Repairing the SSH Authenticated Encryption Scheme: A Case Study of the Encode-then-Encrypt-and-MAC Paradigm. *ACM Trans. Inf. Syst. Secur.* 7, 2 (May 2004), 206–241.

[8] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. 2017. A Messy State of the Union: Taming the Composite State Machines of TLS. *CACM* 60, 2 (2017), 99–107.

[9] E. Boss. 2012. *Evaluating implementations of SSH by means of model-based testing.* Bachelor's Thesis. Radboud University.

[10] S. Cassel, F. Howar, and B. Jonsson. 2015. RALib: A LearnLib extension for inferring EFSMs. In *DIFTS*. http://www.faculty.ece.vt.edu/chaowang/difts2015/papers/paper_5.pdf

[11] G. Chalupar, S. Peherstorfer, E. Poll, and J. de Ruiter. 2014. Automated Reverse Engineering using LEGO. In *Proc. USENIX workshop on Offensive Technologies (WOOT'14)*. 1–10.

[12] H. Chen, D. Dean, and D. Wagner. 2004. Model Checking One Million Lines of C Code. In *NDSS*. The Internet Society. http://www.isoc.org/isoc/conferences/ndss/04/proceedings/Papers/Chen.pdf

[13] A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. 2002. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *CAV (LNCS)*, Vol. 2404. Springer, 359–364.

[14] P. Fiterău-Broştean, R. Janssen, and F.W. Vaandrager. 2016. Combining Model Learning and Model Checking to Analyze TCP Implementations. In *CAV'16 (LNCS)*, Vol. 9780. Springer, 454–471.

[15] A. Futoransky and E. Kargieman. 1998. An attack on CRC-32 integrity checks of encrypted channels using CBC and CFB modes. Online. https://www.coresecurity.com/system/files/publications/2016/05/KargiemanPacettiRicharte_1998-CRC32.pdf, (1998).

[16] O. Grinchtein, B. Jonsson, and M. Leucker. 2010. Learning of event-recording automata. *TCS* 411, 47 (2010), 4029–4054.

[17] M. Isberner. 2015. *Foundations of Active Automata Learning: An Algorithmic Perspective.* Ph.D. Dissertation. TU Dortmund.

[18] D. Lee and M. Yannakakis. 1996. Principles and methods of testing finite state machines — a survey. *Proc. IEEE* 84, 8 (1996), 1090–1123.

[19] T. Lenaerts. 2016. *Improving Protocol State Fuzzing of SSH.* Bachelor's Thesis. Radboud University.

[20] K.G. Paterson and G.J. Watson. 2010. Plaintext-Dependent Decryption: A Formal Security Treatment of SSH-CTR. In *EUROCRYPT 2010*. LNCS, Vol. 6110. Springer, 345–361.

[21] D. Peled, M.Y. Vardi, and M. Yannakakis. 2002. Black Box Checking. *Journal of Automata, Languages, and Combinatorics* 7, 2 (2002), 225–246.

[22] E. Poll and A. Schubert. 2007. Verifying an implementation of SSH. In *WITS'07*. 164–177.

[23] E. Poll and A. Schubert. 2011. *Rigorous specifications of the SSH Transport Layer.* Technical Report. Radboud University.

[24] H. Raffelt, B. Steffen, T. Berg, and T. Margaria. 2009. LearnLib: a framework for extrapolating behavioral models. *STTT* 11, 5 (2009), 393–407.

[25] J. de Ruiter and E. Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *USENIX Security*. USENIX Association, Washington, D.C., 193–206.

[26] W. Smeenk, J. Moerman, D.N. Jansen, and F.W. Vaandrager. 2015. Applying Automata Learning to Embedded Control Software. In *ICFEM 2015 (LNCS)*, Vol. 9407. Springer, 1–17.

[27] O. Udrea, C. Lumezanu, and J.S. Foster. 2008. Rule-based static analysis of network protocol implementations. *Inf. and Comp.* 206, 2-4 (2008), 130–157.

[28] F.W. Vaandrager. 2017. Model Learning. *CACM* 60, 2 (2017), 86–95.

[29] P. Verleg. 2016. *Inferring SSH state machines using protocol state fuzzing.* Master's thesis. Radboud University.

[30] Y. Wang, Z. Zhang, D. Yao, B. Qu, and L. Guo. 2011. Inferring Protocol State Machine from Network Traces: A Probabilistic Approach. In *Applied Cryptography and Network Security*. LNCS, Vol. 6715. Springer, 1–18.

[31] S.C. Williams. 2011. Analysis of the SSH Key Exchange Protocol. In *Cryptography and Coding*. LNCS, Vol. 7089. Springer, 356–374.

[32] T. Ylonen and C. Lonvick. 2006. The Secure Shell (SSH) Authentication Protocol. RFC 4252, IETF, Network Working Group. (2006).

[33] T. Ylonen and C. Lonvick. 2006. The Secure Shell (SSH) Connection Protocol. RFC 4254, IETF, Network Working Group. (2006).

[34] T. Ylonen and C. Lonvick. 2006. The Secure Shell (SSH) Protocol Architecture. RFC 4251, IETF, Network Working Group. (2006).

[35] T. Ylonen and C. Lonvick. 2006. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, IETF, Network Working Group. (2006).