BACHELOR THESIS
COMPUTER SCIENCE

RADBOUD UNIVERSITY

Improving Protocol State Fuzzing of SSH

Author:
Toon Lenaerts
S4321219

First supervisor/assessor:
Prof. dr. Frits Vaandrager
`f.vaandrager@cs.ru.nl`

Second supervisor:
MSc. Paul Fiterau-Brostean
`p.fiterau-brostean@science.ru.nl`

Second assessor:
Dr. ir. Erik Poll
`e.poll@cs.ru.nl`

January 23, 2017

Abstract

With the scale and use of the Internet nowadays, it is crucial that we can effectively test the correctness and security of systems that handle our personal data. In this thesis, we improve upon a previous work by Verleg. Verleg used protocol state fuzzing to test several implementations of the SSH protocol. By adapting a more formal methodology, we achieve higher confidence in our results.

We improve upon Verleg's methodology in three ways: First, we expand the scope of our learned model, by learning a single model for the three layers of SSH. Then, we achieve a degree of confidence in our model by using a more advanced testing algorithm. Finally, we increase confidence in the analysis of model by performing formal model checking instead of manual inspection of results.

Contents

Chapter 1

Introduction

Over the past few decades, the internet has grown to become an essential part of western society. These days, the everyday person uses the internet for banking and government matters. It is therefore crucial that communication over the internet can be done in a secure way. To verify if a system is secure, we use a variety of testing techniques. These techniques have to be well defined and performed in order to produce correct results. In this thesis, we improve upon a work by Verleg [18]. We apply a more systematic and formal methodology in order to produce more convincing results.

Fuzz testing or fuzzing is a testing technique where specific kinds of input data are sent to a software system. The behavior of the system is then monitored for anomalies. Different kinds of input data can be sent to the system under learning (SUL) in order to test for different properties. For example, invalid input can be sent to provoke an exception in the SUT. Another example is sending a longer input than expected to provoke buffer overflow errors.

In this work, the fuzzing technique we focus on is protocol state fuzzing. In protocol state fuzzing a state machine is inferred from a protocol implementation. This state machine is then inspected for unwanted behavior. Protocol state fuzzing is done as follows: First, we choose an implementation of a certain protocol. Then, we send messages formatted correctly according to the protocol. Note that these messages can contain wrong information and can be sent during any phase of the protocol. All of the received output is recorded. Based on the order of inputs and received outputs we generate a state machine which should be representative of the SUL. This state machine is tested for compliance with the SUL. If the machine is incorrect, we continue sending messages until a new machine has been learned. Otherwise, learning is done. Finally, we can verify whether the machine is complaint with the protocol the SUL is based on.

Secure Shell (SSH) is a network protocol used to connect and interact with remote machines in a secure way. It is originally created as a secure replacement for telnet and unsecure remote login protocols like rlogin. The second version of SSH, SSHv2 has been standardized by the IETF [22, 20, 23, 21]. It is one of the most frequently used security suites on the Internet [18].

Recently, Verleg [18] has performed protocol state fuzzing on implementations of the Secure Shell protocol. Verleg learns three models for each of the tested implementations; one for each of the layers defined in SSH. These models are then evaluated by hand to find security related issues. For this evaluation, Verleg has defined several security properties from the RFCs of SSH. With this approach, Verleg has found a bug in the OpenSSH implementation and security related issues in several other implementations.

Nevertheless, the work of Verleg can be improved upon. There are three aspects where the work can be improved. First, it would be interesting to learn a model for the entirety of the SSH protocol, which could show additional security issues. Secondly, Verleg uses a method of testing that can be improved upon. Finally, checking the generated models with security properties can be done automatically, by using a model checker. In this thesis, we repeat Verleg's experiment, and improve upon all three of the mentioned aspects.

Verleg split the SSH protocol into its constituent layers and learned each layer separately. This was done due to the complexity of the system, which meant that learning it as a whole would be infeasible. By utilizing caching and a different testing technique, we have overcome this problem, and have obtained a model of the entire system.

Verleg uses a relatively simple algorithm to test generated models. This algorithm simply runs random tests on the model, and assumes the model is correct after a certain number of tests are done. This does not provide a measure of confidence in the generated models. However, more advanced testing methods exist that do give confidence in tested models. One such method is developed by Smeenk et al. [14], based on the adaptive distinguishing sequence algorithm by Yannakakis and Lee [12]. We replace Verleg's testing algorithm with this algorithm. And thus obtain this degree of confidence in our generated model.

Verleg checks his generated models by hand. As shown in [8], models obtained from fuzzing can be checked with a model checker. Testing properties with a model checker is more thorough than checking a model by hand, especially for larger models. Model checking thus gives a higher confidence in generated conclusions. We formalize the properties Verleg used, and verify our learned model with these formal properties.

In this thesis we first provide an overview of protocol state fuzzing and the Secure Shell algorithm in Section 3. A brief overview of related work is given in Section 2. Then, in Section 4 we explain in detail what we altered from Verleg's work. Section 5 gives the results of our improved learning setup. We further expand on some of the decisions made for this thesis in

Section 6. Finally, we present our conclusions in Section 7.

Chapter 2

State of the Art

The researh most related to this work is, naturally, Verleg's thesis [18]. Aside from that the related work can be split into two areas: work on SSH security and work on protocol state fuzzing.

Various formal analysis has been performed on SSH. It has been proven that, under some assumptions, the Diffie-Hellman ciphersuite in the SSH-protocol is securely authenticated and confidential [5]. Formal analysis has shown that the SSH key exchange protocol is secure [19]. A different analysis has shown that plaintext recovery from SSH is possible if CBC mode is used [3].

More practical work has also been done: An internet-wide scan has shown weaknesses in duplicate SSH keys [9]. An open source implementation of SSH has been formally specified and verified [13]. Machine learning has also been applied to SSH security: A detector for brute-force attacks on SSH has shown successful results [11]. Timing analysis of keystrokes has shown to speed up exhaustive search for SSH passwords [15].

By using regular inference, it is possible to construct a model from observations of behaviour, a process known as automata learning [4]. Active automata learning, where an exploration phase and testing phase are alternated, is an efficient method of automata learning [16].

Protocol state fuzzing has successfully been performed on electronic passports[2]m, EMV bank cards [1] and printer software [14]. It has also successfully been used on internet protocols: TLS [7] and TCP [8, 10]. In the TLS case security issues were found for three implementations. Work has also been done on protocol state fuzzing of SSH, where servers have been found to not follow the RFC standard [17]. A continuation on this work has shown unwanted behaviour in an implementation of SSH [18].

Chapter 3

Preliminaries

In this section, we provide some background information for the rest of the thesis. First, we explain Mealy machines in Section 3.1. Section 3.2 expands on typical learning setups and the setup Verleg used. Finally, in Section 3.3 we provide a brief overview of the SSH-protocol.

3.1 Mealy Machines

We use a Mealy machine to represent the state machine of an SSH implementation. A Mealy machine is a DFA represented by a 6-tuple $\mathcal{M} = (Q, q_0, I, O, \delta, \lambda)$, where:

- Q is the set of states.
- $q_0 \in Q$ is the initial state.
- I is the input alphabet.
- O is the output alphabet.
- δ is the transition function $Q \times I \to Q$, which gives the resulting state from sending a certain input when in a certain state.
- λ is the transition function $Q \times I \to O$, which gives the received output when sending a certain input from a certain state.

Mealy machines can be graphically represented as a graph. States are shown as nodes and state transitions as edges. Edges are labeled as "$<$ *input* $>$ / $<$ *output* $>$" corresponding to the transition functions: If $\delta(q_0, A) = q_1$ and $\lambda(q_0, A) = X$ then the label for the edge between q_0 and q_1 will be "A/X". An example visualization for a Mealy machine is shown in Figure 3.1.

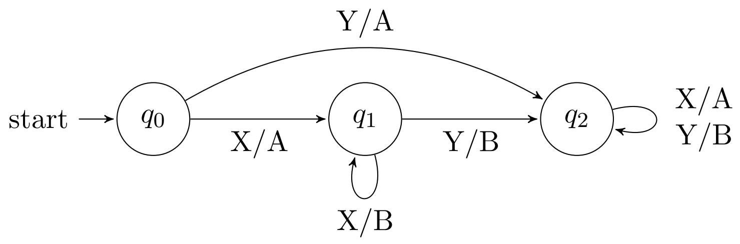

Figure 3.1: Example Mealy machine

3.2 Learning state machines

In protocol state fuzzing, a state machine of the implementation is learned by actively sending and receiving messages to and from the System Under Learning (SUL). In Verleg's work [18], a single message is called a *query*, and a sequence of messages a *trace*, we keep the same convention. The state machine is formed by sending consecutive traces to the SUL and analyzing the output. In our case, the trace to be sent is determined by an adaptation of the L* algorithm for learning Mealy machines. When a trace is completed the SUL has to be reset to its initial state.

After sending a sufficient amount of traces, a hypothesis is formed. The hypothesis is checked for correctness by an equivalence oracle. Said oracle sends traces to the SUL, then predicts an output through the hypothesis, and compares this output to the actual output received from the SUL. Since the SUL can is a black box, it is impossible to guarantee that the oracle will find any existing counterexample in finite time. However, measures can be taken such that the testing oracle provides a certain degree of confidence in the model.

In our case, sending messages to the SUL means sending packets that contain information like the packet length, a sequence number or a list of supported encryptions. This information can be represented as parameters of packets. However, parameterized messages cannot be adequately expressed by Mealy machines. Therefore we abstract this information away, leaving us with an abstract representation of messages. To convert these abstract messages to correct packets and back, a mapper is used. This mapper has to keep track of state variables, and has to perform actions such as encryption and compression. This means that the mapper itself contains a state machine, which is based on existing knowledge about the protocol used in the SUL.

Figure 3.2 shows the setup Verleg used, and serves as an example for a typical learning setup. Here "Learner" is the program that uses the L*-based algorithm to generate traces of abstract messages to send to the SUL. The learner sends these messages to the mapper, which translates them to

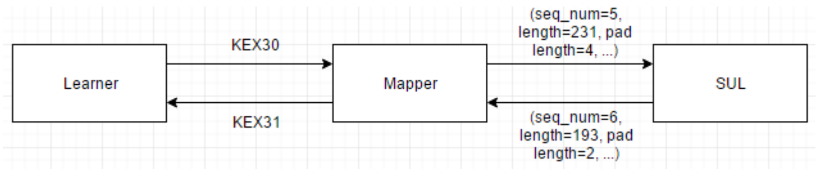
Learner
KEX30
KEX31
Mapper
(seq_num=5, length=231, pad length=4, ...)
(seq_num=6, length=193, pad length=2, ...)
SUL

Figure 3.2: A learning setup

concrete packages, these packages are then sent to the SUL. The response of the SUL is converted to an abstract message by the mapper and sent to the learner.

3.3 Secure Shell

The SSH-protocol uses a client-server structure consisting of three components. These components are referred to as layers, note however that messages generated in outer layers do not encapsulate those generated by inner layers, instead they are separate, and are distinguished by a message number. The three components are as follows:

- The transport layer protocol creates the basis for communication between server and client, providing a key exchange protocol and server authentication. The key exchange protocol is performed over three roundtrips. During the first, both client and server send a KEXINIT message. Then, the client sends a KEX30 message, the server responds with a KEX31 message. Finally, both parties send a NEWKEYS message, which indicates that the keys sent in the second step can be used.

- The user authentication protocol is used to authenticate a client to the server, for example, through a username and password combination, or through SSH-keys.

- The connection protocol provides various services to the connected client. It multiplexes the encrypted connection into different channels over which services are provided. Example services are file transfer or access to a remote terminal. Typical messages are requests for opening or closing channels, or requests for the earlier named services.

Chapter 4

Method

Since we improve the methodology of Verleg's work [18] in three ways, we will expand on the three categories of changes separately. In Section 4.1, we explain our changes to Verleg's setup to enable learning of all three SSH-Layers. Then, in Section 4.2, we provide a brief explanation of the testing algorithm we used. Finally, in Section 4.3, we explain how we have used model checking to verify the security properties Verleg defined.

4.1 Building a complete model

As said previously, Verleg split the model construction of SSH into the three layers defined in the SSH-specification. The first reason for this is practical; it takes a very long time to infer a model of the entire protocol. Combining this with the problems of non-determinism Verleg experienced, and lack of handling this non-determinism, meant that inferring the full model would often fail partway through. The second reason is the interaction between layers that SSH allows. For example, in any state of the connection layer, it is possible to perform a rekey sequence. In a rekey sequence, the transport layer messages for the key exchange procedure are transmitted again. A rekey procedure should not interfere with any higher layer; in a state model, this means that the state before and after the rekey should be the same state. Verleg experienced problems with these sequences, which caused errors in his models. We assume that more advanced testing methods will find counterexamples for these errors, and therefore we can train a model of the entire protocol.

We investigated the methods of caching Verleg used in his setup. The result of queries are saved during the learning process, however, the saved queries are only used to detect non-determinism. Verleg had implemented a way to resume an old learning run with the saved queries, but this implementation contained errors. We have expanded on Verleg's implementation for resuming old learning processes. We have not expanded further on handling non-determinism and caching, we explain why in Section 6.1. In order to reduce the amount of non-determinism, we have to fine-tune timing parameters in the mapper, which is done mostly on a trial-and-error basis. As a result, running a long query takes a large amount of time, and therefore the entire fuzzing process is quite time consuming.

Since we want to improve the process of state fuzzing used by Verleg, rather than add upon the size of his research, we only map a single SSH implementation. The implementation we have used is OpenSSH, version 6.6p1.

4.2 Improving Testing

We improve upon the testing procedure Verleg used by using a different equivalence oracle. Verleg has used a random-walk oracle, this oracle randomly chooses a path through the model, sends the required inputs to the SUL, and checks if the output of the SUL corresponds with the expected output in the model. This is repeated for some time, until the model is assumed to be correct. We instead use an implementation of an algorithm specified by Smeenk et al. [14] based on an algorithm by Lee & Yannakakis [12] to find adaptive distinguishing sequences. This algorithm enables us to find additional counterexamples and gives a degree of confidence in the final model.

The algorithm uses distinguishing sequences for every state of the hypothesis, a distinguishing sequence is a set of in- and outputs that provides a unique result for a certain state. It allows to check the identity of a state, by sending the distinguishing sequence and checking if the output of the SUL corresponds with the predicted output. If the outputs are equal, the initial state is the state for the distinguishing sequence.

A single test of the algorithm has the following structure: First, a random walk to a certain state in the model is performed. Then, a walk of a specific path-length is done. Finally, the distinguishing sequence for the resulting state is performed. The algorithm can perform runs like these exhaustively for an incrementing path-length. If the algorithm does not find a counterexample up to a specific path-length, we can say that the tested model is correct, unless the actual model is larger than the checked hypothesis by a number of states larger than the tested path-length. The algorithm can be run either exhaustively or randomly. Running it randomly is quicker to find counter-examples, but does not provide the confidence that running it exhaustively does.

4.3 Model Checking

Verleg has analysed his models using security definitions in natural language. This analysis was done by hand. In order to increase the confidence in security analysis performed on learned models, we use model checking to check formal security properties on our model.

We use the NuSMV [6] model checker to formally state and verify the properties defined by Verleg. NuSMV is a model checker where a state of a model is specified as a set of finite variables, a transition-function can then be defined to change these variables. Specifications in temporal logic, such as CTL and LTL, can be checked for truth on specified models. NuSMV will provide a counterexample if a certain specification is not true.

The NuSMV model is generated from our learned model. The learned model is a Mealy machine, generating the NuSMV model from this is straightforward. The states in the model directly translate to states in NuSMV. The NuSMV model contains two variables, one for the input alphabet, and one for the output alphabet. The NuSMV transitions are generated by taking a combination of a state and an input-word, and defining an output and next value for state for said combination. Figure 4.2 shows a transition function for the Mealy machine in Figure 4.1. Note that, as a side-effect of using a simple way to generate the NuSMV-code, we use the input or output as a whole, rather than a collection, in the case where we send or recieve multiple packets.

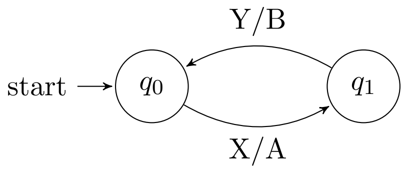
$$\begin{tikzcd} \text{start} \arrow[r] & q_0 \arrow[r, bend right, "X/A"'] & q_1 \arrow[l, bend right, "Y/B"'] \end{tikzcd}$$

Figure 4.1: Example Mealy machine

```
next(state) := case
        state = q0 & input = X: q1;
        state = q1 & input = Y: q0;
esac;
output := case
        state = q0 & input = X: A;
        state = q1 & input = Y: B;
esac;
```

Figure 4.2: NuSMV transition functions for Figure 4.1

The properties we have verified are LTL-representations of the security properties Verleg has defined for each layer of SSH. Verleg has defined three properties. The property of the connection layer, however, is very open ended. It is: “We will ... look at unexpected state machine transitions that can point towards potential implementation flaws”, we can not properly represent this in LTL. The properties for the transport- and user authentication layers, however, can be represented in LTL.

Chapter 5

Results

In this section, we present our results for the three improvements we made to Verleg's [18] work. In Section 5.1 we show the state machine we inferred from the OpenSSH implementation. Then, in Section 5.2 we give some testing statistics and results, and explain some properties we found in the generated model. Finally, in Section 5.3 we show the LTL properties we checked with the model.

5.1 Trained Model

The result of learning OpenSSH is a model with 26 states and 619 transitions. An edited version can be seen in Figure 5.1. The happy path, as defined by Verleg, is indicated in green and dashed edges. The more complete version contains nine states that can be reached by attempting to open a channel before performing user authentication. These states make make the model less readable while not containing any interesting behaviour; Although they can perform a successful authentication, no channels can be opened, and all possible paths from these states eventually lead to a disconnect-state. Therefore we have removed them from Figure 5.1.

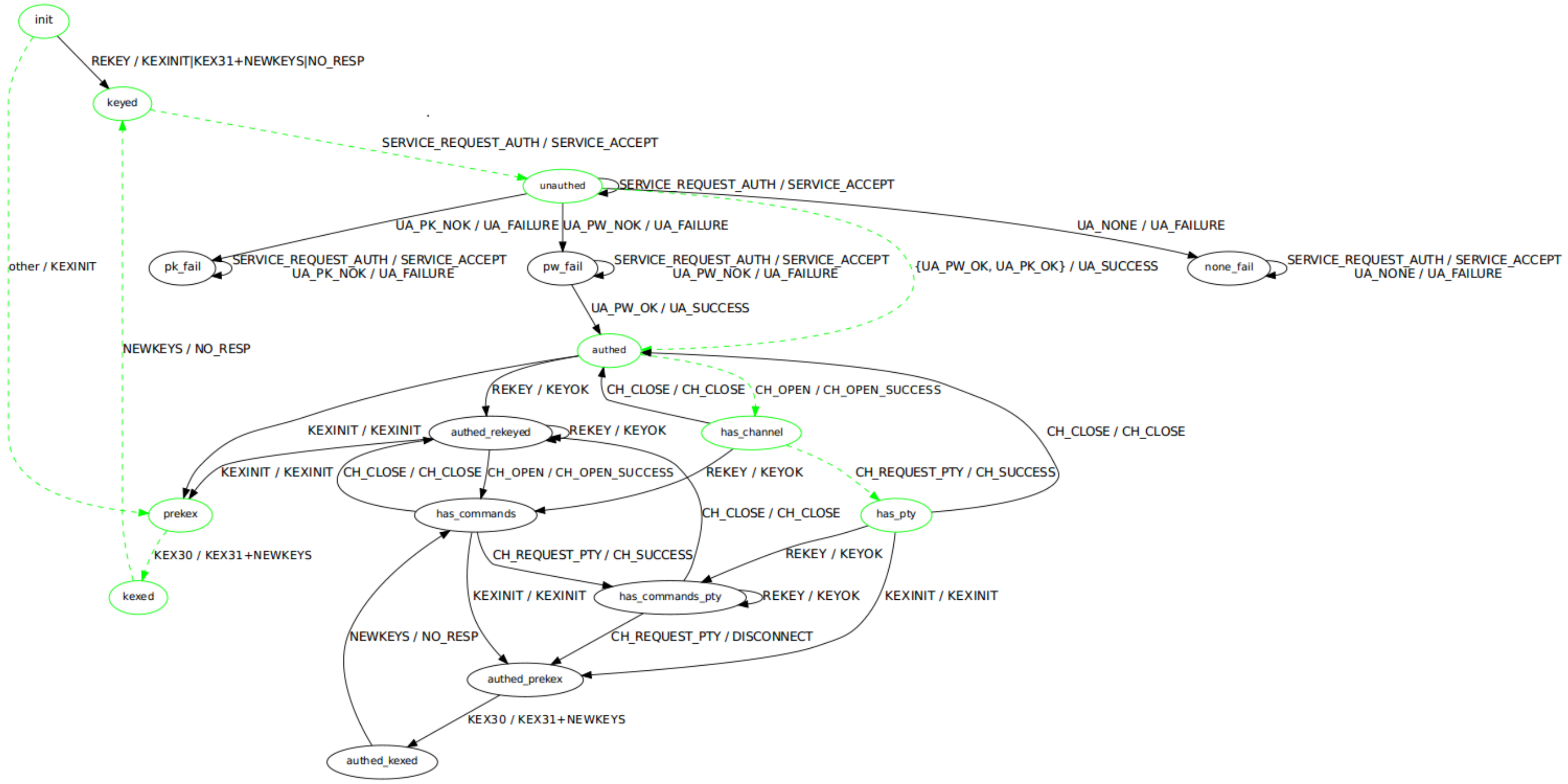

Figure 5.1: Inferred state machine, edited for readability

5.2 Testing

We used the Yannakakis-tool on the generated hypotheses. For the final model, we completed an exhaustive test up to a path length of two. This implies that if our model is faulty, a correct model must have at least 28 states, two modre than the 26 states of the model we learned. During the entire learning process, only two hypotheses were constructed, the counter example to the first hypothesis was found after 865 test sequences. Verifying the second hypothesis took 32.728 test sequences.

On comparing our model with Verleg's model, we can find some significant differences. To illustrate them, we isolated the connection layer from Figure 5.1, and emphasized the differences. The result is shown in Figure 5.2. Verleg's state machine of the connection layer of OpenSSH showed a fault, where closing a channel from the "has_commands_pty" would result in connection termination. Our model does not show this behavior, closing a channel from the "has_commands_pty" state leads to a new state, from which other actions are still allowed.

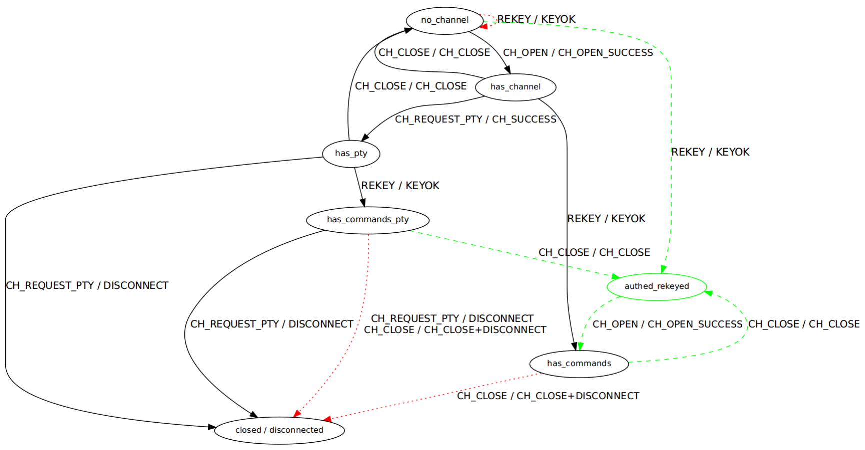

Figure 5.2: Connection layer of the inferred state machine. Dotted transitions in red are unique to Verleg's models, the state and dashed transitions in green are unique to our work.

A few days before the deadline of this thesis, a counterexample was found for the learned model. As predicted (see 4.1 and 6.2) a rekey sequence in the connection layer is not preserving the state of the model. We have supplied the counterexample to the learning setup, and a new hypothesis has been learned. However, we have not been able to finish testing the new model in time. We have been able to test about 12.000 random test sequences on the hypothesis, but exhaustive testing has not been completed. Additionally we have attempted to learn a model with a limited input alphabet, this should limit the number of necessary queries and allows us to fine-tune the timing parameters further. This setup has also produced a number of hypotheses. We have run about 16.000 random test sequences on the latest hypothesis, however, we have not been able to finish exhaustive testing. We have been able to run a significant amount of random test sequences on the new hypothesis, which gives us some confidence in the hypothesis.

5.3 Model Checking

We have converted two security properties of the three defined by Verleg [18] into LTL specifications and used the NuSMV model checker [6] to check the properties for truthfulness with the state machine we have inferred. We expand on each property individually, in the context of the layer it applies for.

5.3.1 Transport Layer

Verleg's security property for the Transport layer is defined as follows: "We consider a transport layer state machine secure if there is no path from the initial state to the point where the authentication service is invoked without exchanging and employing cryptographic keys." We consider this path to be a path where a key exchange is performed successfully, and then followed by a successful service authentication request. Note that the key exchange does not have to be performed in consecutive steps, it can temporarily be interrupted by other actions.

In natural languange, we can say the property should be something like: "If the authentication service is invoked successfully, then the final step of the key exchange must have been done, the second step must have been done earlier, and the first step even earlier." The key exchange consists of three specific steps which have to be performed in order. The LTL in Figure 5.3 is the result. The O-operator used here is the Once-operator, a past time LTL operator, which is true if its argument is true in a past time instant.

$$\begin{aligned}&G(\text{authentication request successful} \rightarrow\\&\ O(\text{NEWKEYS successful \&}\\&\quad O(\text{KEX\_30 successful \&}\\&\quad O(\text{ KEXINIT successful}))))\end{aligned}$$

Figure 5.3: LTL for Transport layer with natural language

Translating this LTL to NuSMV syntax means that the remaining natural language has to be replaced with equality checks for the variables in the NuSMV state. These variables are equal to the in- and output of a transition in the Mealy machine generated by the mapper. A successful invocation of the authentication service is translated as a request for the authentication service being done, and the request being accepted by the server. The third step of the key exchange, the client sending a newkeys-packet, is successful if the server sends no response. Conveniently, no server response is an indication that the newkeys was received correctly, in our tests, the server would return a packet if the newkeys was performed incorrectly. The second step, sending the kex30 packet, is very much like the third step. Finally, the first step of the protocol is translated a bit differently, since we found that sending any packet to the server from the initial state was recognized as the first step of the key exchange protocol. Therefore, sending the first step, a kexinit-packet, can be replaced by almost any packet. The mapper contains a shorthand which performs the complete key exchange, called rekey, this is also added to the LTL specification. The resulting LTL-specification is shown in Figure 5.4.

```
G (
    (input=SERVICE_REQUEST_AUTH & output=SERVICE_ACCEPT)
  ->
      (O ( (input=NEWKEYS & output= NO_RESP) &
         O ( (input=KEX30 & output=KEX31_NEWKEYS) &
            O (output=KEXINIT)
      ))
    |
      (O (input=REKEY & output=KEXINIT_KEX31_NEWKEYS_NO_RESP))
  )
)
```

Figure 5.4: Transport layer LTL in NuSMV syntax.

5.3.2 User Authentication Layer

The security property is as follows: “We consider a user authentication layer state machine secure if there is no path from the unauthenticated state to the authenticated state without providing the correct credentials.” Since we have no method to directly determine if the model is in an authenticated state, we consider the authenticated state to be a state where opening a channel can be done successfully. Since opening a channel should only be possible after user authentication, this should show if we are in an authenticated state. The specification then is rather simple: Always, if a channel is opened succesfully, then there has not been a failure in user authentication since a successful user authentication. Opening a channel successfully in the model is straightforward, a request to open a channel is sent to the server, and a success message is received back. Checking whether there has been a success or failure in user authentication is done by checking if the server ever sends the UA_FAILURE or UA_SUCCESS packets. Additional checks if the user actually made a user authenication request could potentially be added. The resulting LTL is shown in Figure 5.5. This LTL uses another past time operator: The S- (or Since-) operator is true if its first argument holds for every state since its second argument holds, note that the second argument has to hold somewhere in the path for the since-operator to hold.

```
G (  (input= CH_OPEN & output= CH_OPEN_SUCCESS)
-> (!(output=UA_FAILURE) S output= UA_SUCCESS) )
```

Figure 5.5: User Authentication layer LTL in NuSMV syntax.

The learned model satisfies both LTL specifications, according to NuSMV.

Chapter 6

Discussion

In this Section we discuss and motivate some of the choices we made in this thesis. In Section 6.1 we expand on the caching method we use. Then, in Section 6.2 we give our consideration for the testing parameters used. In Section 6.3 we provide explanation for some faults in the generated model. Finally, in Section 6.4 we expand on the security properties used.

6.1 Caching

In our setup, we improve upon the caching methods Verleg [18] used. This allows us to resume an old, possibly crashed, run of the setup. Because of this, we are able to build a state machine of the entire SSH protocol. However, it is possible to implement much more extensive methods of caching and detection of non-determinism. For instance, we have not implemented any way of checking a trace which results in non-determinism. This can be done by, for example, sending a trace which results in non-determinism again, and comparing the output of both traces. If the second output does not result in non-determinism, we might assume that something went wrong during the first run of the trace (a packet was lost, for example), and can continue learning with the second output. We decided the caching we used was adequate, and thus have not implemented more advanced methods

6.2 Chosen Confidence

For our equivalence oracle, we use the Yannakakis-Lee-based algorithm [14] in order to build a measure of confidence for the generated state machine. The confidence increases by running the algorithm with a longer path-length. As explained, we run up to a path length of two. We do this due to time considerations; for our generated model, testing up to a path length of two requires 32.728 test sequences, A path length of up to three would take 353.835 sequences, four would take 7.276.082. These tests are often long traces of inputs, and running them can take up to a couple of minutes per trace. Testing three hundred thousand traces with our setup is not feasible. However, one of Verleg's concerns for training all three layers of SSH simultaneously is the rekey procedure. The concern that the model will not implement the rekey procedure in higher layers correctly. The procedure contains three steps before returning to the state before the rekey. A confidence with a path length up to four should correctly implement the rekey procedure for all the states in the SSH-specification where rekey is possible. To remedy this problem, we have used the same solution Verleg has used. We use a separate command which performs an entire rekey procedure in the setup. Because of this, the model should still handle the entire procedure correctly. However, as we later have experienced, this does not provide confidence in performing only parts of the rekey procedure, or performing the procedure differently, for example. This has lead to a counterexample in our current model (see 5.2).

6.3 Faulty rekeys in State Machine

As explained in Section 5.2, our model does not show the disconnecting behavior Verleg experienced. However, it still contains some strange behavior. In SSH, performing a rekey procedure in the "higher" connection layer should not influence actions in that layer. Therefore, in the generated state model, performing a rekey from a certain state in a higher layer should always return to that state upon finishing the procedure. In our model, performing a rekey from the "authed", "has_channel" and "has_pty" states ends in the "authed_rekeyed", "has_commands" and "has_commands_pty" states respectively.

6.4 Security property coverage

As explained in Section 4.3, we have not formalized a security property for the connection layer. This however, means that the section of the state machine where some interesting behavior happens (see Section 5.2 and Section 6.3) is not covered by a security definition. From observing the problems we have found we have not found any significant security related problem, however, observing by eye for mistakes goes against the goal of this work, and therefore leaves us with a weakness in our models.

Chapter 7

Conclusion

In this thesis we improved upon Verleg's [18] thesis by using a more systematic methodology. By doing this, we have shown that the already useful protocol state fuzzing can also produce results with a measure of confidence. The improvements we have made have not substantially lengthened learning time and haven't substantially complicated the process of protocol state fuzzing. We therefore conclude that, as a whole, we have improved upon the work by Verleg, and have made an attempt to create more formal and trustworthy testing methods.

We have trained a model of the entire SSH-protocol, whereas Verleg could only learn its three contituent layers separately due to timing concerns. The model contains the desired behavior of SSH. Therefore, we have been able to successfully train all three layers of SSH simultaneously. Our model uses the entire training alphabet for all three layers, rather than a specialized alphabet per layer. Because of this, our model can contain information not present in Verleg's models. Therefore, we have improved upon Verleg's work by learning the three layers of SSH in a single model.

By using a more advanced testing method, we have gained a measure of confidence in our generated model. Either our model is representative of the SSH implementation, or the real model is at least two states larger. Using a random walk algorithm does not give this measure of confidence. We have therefore improved upon Verleg's work by using this testing algorithm.

We have checked two LTL properties with our model. These properties are formalized from the security definitions for SSH defined by Verleg. According to NuSMV, both properties hold for our generated model. We have shown an application of using model checking to verify properties of a model generated through protocol state fuzzing. Through model checking we have gained a measure of confidence for the conclusions made about our model. This confidence is not present in models, when they are purely checked by hand. Therefore, we have improved upon Verleg's work by using model checking for the generated model.

Possible future work is extending our setup with different implementations of SSH, similar to Verleg's original work. It is also possible to extend the tested properties and test different aspects than security.

Some parameter abstractions are included in our setup, these can be removed, and used in learning. This would require us to use more advanced learners, which can infer state machines with parameters, since Mealy machines cannot do so. For example, various parameters are negotiated in the key exchange protocol, these could be extracted. SSH also uses a sequence number which could be extracted.

Verleg's thesis contains additional suggestions for further work that remain relevant: Extending the alphabet, using different SSH settings or implementing additional SSH behavior. Some of the suggestions Verleg makes are related to decisions they took to keep learning times shorter, applying the suggestions would extend learning time. A different direction would be to attempt to shorten learning time, this makes it feasible to run the testing algorithm with higher parameters, resulting in more confidence in learned models. An example for this is to create two models, one with the full alphabet, and one with a limited alphabet. The model with the limited alphabet can be tested more thoroughly. It is then possible to find counterexamples for the full model by removing all transitions that contain words not present in the limited alphabet, and checking whether the result and the limited model are equal.

Bibliography

[1] Fides Aarts, Joeri De Ruiter, and Erik Poll. Formal models of bank cards for free. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 461–468. IEEE, 2013.

[2] Fides Aarts, Julien Schmaltz, and Frits Vaandrager. Inference and abstraction of the biometric passport. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 673–686. Springer, 2010.

[3] M. R. Albrecht, K. G. Paterson, and G. J. Watson. Plaintext recovery attacks against SSH. In *2009 30th IEEE Symposium on Security and Privacy*, pages 16–26, May 2009.

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

[5] Florian Bergsma, Benjamin Dowling, Florian Kohlar, Jörg Schwenk, and Douglas Stebila. Multi-ciphersuite security of the secure shell (SSH) protocol. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 369–381, New York, NY, USA, 2014. ACM.

[6] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.

[7] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, 2015.

[8] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. Combining model learning and model checking to analyze TCP implementations. In *International Conference on Computer Aided Verification*, pages 454–471. Springer, 2016.

[9] O. Gasser, R. Holz, and G. Carle. A deeper understanding of SSH: Results from internet-wide scans. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–9, May 2014.

[10] Ramon Janssen. Learning a state diagram of TCP using abstraction. 2013.

[11] Mobin Javed and Vern Paxson. Detecting stealthy, distributed SSH brute-forcing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 85–96, New York, NY, USA, 2013. ACM.

[12] D. Lee and M. Yannakakis. Testing finite-state machines: state identification and verification. *IEEE Transactions on Computers*, 43(3):306–320, Mar 1994.

[13] Erik Poll and Aleksy Schubert. Verifying an implementation of ssh.

[14] Wouter Smeenk, Joshua Moerman, Frits Vaandrager, and David N Jansen. Applying automata learning to embedded control software. In *International Conference on Formal Engineering Methods*, pages 67–83. Springer, 2015.

[15] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on SSH. In *USENIX Security Symposium*, volume 2001, 2001.

[16] Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. In *Formal Methods for Eternal Networked Software Systems*, pages 256–296. Springer, 2011.

[17] Max Tijssen. Automatic modeling of SSH implementations with state machine algorithms. 2013.

[18] P Verleg. Inferring SSH state machines using protocol state fuzzing. 2016.

[19] Stephen C. Williams. *Analysis of the SSH Key Exchange Protocol*, pages 356–374. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[20] Tatu Ylonen and Chris Lonvick. The secure shell (SSH) authentication protocol. RFC 4252. 2006.

[21] Tatu Ylonen and Chris Lonvick. The secure shell (SSH) connection protocol. RFC 4254. 2006.

[22] Tatu Ylonen and Chris Lonvick. The secure shell (SSH) protocol architecture. RFC 4251. 2006.

[23] Tatu Ylonen and Chris Lonvick. The secure shell (SSH) transport layer protocol. RFC 4253. 2006.