Analysing SSH Clients Using Protocol State Fuzzing

Robert Lasu
Abstract

Protocol state fuzzing is a black-box testing technique which, by automatically exchanging messages with a System Under Test (SUT), constructs a state machine, that captures how the SUT handles messages. This state machine is then analysed for implementation flaws, which can be specification violations, conformance issues or even vulnerabilities. This testing technique has successfully been used to test implementations of widely-used protocols including DTLS, TLS, TCP, and QUIC. Secure shell (SSH) is a popular protocol used to securely access remote machines over a non-trusted network. Previous work on state fuzzing SSH implementations has mainly focused on servers, thus the clients have received much less attention. It is equally important to test both clients and servers and this thesis aims to bridge the gap between testing SSH servers and clients.

In this thesis, protocol state fuzzing was used to infer a state machine of SSH clients and manually analyse them for implementation flaws, vulnerabilities, and conformance issues. To that end, a previously developed framework for protocol state fuzzing of SSH servers was updated and extended to support clients during this project. During the analysis the SSH implementation Dropbear was found to violate the SSH specifications. This led to incompatibility between the SSH implementation and the framework used and had to be solved manually. This is not an inherent problem of the framework and thus Dropbear could face similar compatibility problems with other SSH implementations. During the key exchange, Dropbear sends messages which were explicitly disallowed. Even though the key exchange could be completed successfully, this behavior is a violation of the specifications.

In this thesis, the first, to my knowledge, protocol state fuzzing framework has been presented. The framework has successfully been applied to several SSH client implementations and the models generated have been manually analysed. The analysis of the models discovered violations of the SSH specification. These violations are described and discussed throughout this thesis.
Table of Content

1 Introduction 2

2 Background 3
   2.1 Mealy machine ........................................... 3
   2.2 Model learning ........................................... 4
   2.3 Secure Shell Protocol ................................. 4

3 State Fuzzing Framework 7
   3.1 Input alphabet .......................................... 8
   3.2 Output Alphabet ........................................ 9
   3.3 Implementations details ......................... 10
      3.3.1 Updating Paramiko ............................... 10
      3.3.2 Client fuzzing .................................. 11

4 Experiments 12
   4.1 Alphabet ............................................... 12
   4.2 Experimental Setup .................................. 13
      4.2.1 Dropbear ........................................ 13
      4.2.2 OpenSSH .......................................... 13
   4.3 Quantitative Measurements .................. 14
   4.4 Challenges ............................................. 15

5 Analysis of State Machines 16
   5.1 Dropbear ............................................... 16
   5.2 OpenSSH ............................................... 19
   5.3 Comparing Dropbear & OpenSSH ........... 20

6 Conclusion and Future work 21

A Dropbear server test 25

B Configure files 26
1 Introduction

The Secure Shell protocol (SSH) is a widespread protocol used to access machines remotely in a secure way. SSH has been around since the middle of the 1990s when it was first developed by Tatu Ylönen [1]. Due to security issues [1] it was succeeded by the second version, SSHv2, which we will refer to as SSH throughout the thesis. Due to its widespread use and the security sensitivity, it is important to test both the implementations of SSH and the specifications which infer the implementations. The specifications of SSH implementation are defined by the Internet Engineering Task Force, in four Request for Comments, RFCs, [2][3][4][5], which have expanded with multiple extensions over the years. These define how an SSH implementation should behave and how not to.

Protocol state fuzzing, or state fuzzing for short, is a black-box testing technique that uses model learning [6] to probe the behaviour of a System Under Test (SUT) to possibly unexpected sequences of a well-formatted message. The behaviour is captured in a model which can be analyzed to verify if the SUT is responding as expected and conforming to the specifications. In the last two decades, state fuzzing has proved effective in finding security vulnerabilities which otherwise could be missed using other techniques such as code review or statistical analysis [7]. It has been successfully applied to several protocols including TLS [8], DTLS [9], TCP [10] and SSH [11], [12]. Although protocol state fuzzing has been used to test SSH implementations, the focus has mainly concentrated on SSH servers while SSH clients have received much less attention. It is, however, equally important to test the client implementations as it is to test the servers, still, this has not, to the author’s knowledge, been done to any extent. This project aims at addressing this issue by extending an existing state fuzzing framework to support both SSH servers and clients and applying it to popular libraries such as OpenSSH and Dropbear.

Thesis outline. First I will review the model learning process and go through the SSH protocol layers (Section 2). Section 3 starts with an overview of the framework used during the thesis, before giving details on the actual implementation. Experiment setups are described in Section 4 together with quantitative measurements, the section concludes with challenges which arise during the experiments. After the experiments, I compare the models generated to each other and the specifications, (Section 5). Section 6 ends this thesis with conclusions and directions for further work.
2 Background

In this section, important topics relevant to this thesis will be outlined. This includes a brief description of model learning and an overview of the SSH protocol.

2.1 Mealy machine

Mealy machines are state machines which can be used to model reactive systems such as network protocol implementations. A Mealy machine is a finite-state machine where the output is dependent on both the input it receives and the current state. This implies there exists only one output for every transition for a given input and state. A Mealy machine is describable as a 6-tuple, \((I, O, Q, q_0, \delta, \lambda)\), where \(I\) is a finite set of input symbols, \(O\) is a finite set of output symbols, \(Q\) is a finite set of states, \(q_0\) is the initial state, \(\delta\) is a transition function; \(\delta(Q, I) \rightarrow Q\), and \(\lambda\) is a output function; \(\lambda(Q, I) \rightarrow O\). To better analyse a Mealy machine, a graphic representation can be used where nodes represent the state and where every transition between states needs an input and generates an output.

Figure 1 is an example of a graphic representation of a Mealy machine with three states, the input and output alphabet consisting of the symbols 0 and 1. From the initial state \(s_0\), an input 0 generates an output of 0 and transitions back to itself, while an input of 1 from state \(s_0\) is generating the same output, 0, but transition to state \(s_1\). From \(s_1\) an input of 0 would transition back to state \(s_0\) while 1 would transition to \(s_2\) both with the same output 0. In state \(s_2\) a transition back to \(s_0\) is done by the input of 0, this is generating the output of 1, while the input 1 will transition back to state \(s_2\) with 0 as output.
2.2 Model learning

The framework used in the thesis uses model learning [6] to infer a Mealy machine representing an implementation. Model learning builds the Mealy machine by sending symbols from an input alphabet and registering the output received. This is done in two alternating phases, hypothesis construction and validation. When the model learning is in the hypothesis construction phase, symbols from the input alphabet are sent and the response is registered. This continues until certain convergence criteria have been fulfilled, after which a hypothesis is constructed based on the messages exchanged [9]. When a hypothesis has been created the model learning enters the phase of hypothesis validation to test the hypothesis against the implementation under scrutiny. During this phase, conformance tests are run to find a counterexample to the hypothesis. If a counterexample is found, model learning will start a new iteration of hypothesis construction, taking into consideration the previous hypothesis and the counterexample found. This alternating hypothesis construction and validation continues until the hypothesis validation phase is unable to find a counterexample, in which case the hypothesis is returned. As described in [9], there is no absolute guarantee that the hypothesis returned by the model learning is correct, even though several conformance testing algorithms do provide this under certain assumptions.

2.3 Secure Shell Protocol

SSH is a client-server protocol, running on top of TCP, used to securely access remote machines and execute services on the machines. There are two versions of SSH. In this thesis, we are referring to the second version of SSH.
The SSH specifications are defined by the RFC [2][3][4][5] and SSH implementation should conform to these specifications. The protocol is built on top of the TCP/IP stack and consists of three layers, dependent on the one before as visualised in Figure 2.

![Figure 2: The layer architecture of SSH.](image)

The Transport layer handles the key exchange between the server and the client while also providing data confidentiality, server authentication, data integrity, and compression [3].

The Authentication layer runs on top of the Transport layer, providing client authentication over one single encrypted tunnel [4].

The Connection layer provides services such as terminal emulation or file transfer. Several services can be run on a single encrypted tunnel by multiplexing channels [5].

Figure 3 illustrates the expected messages exchange in SSH during a typical interaction. We call this exchange the happy flow.
Figure 3: The message exchanged between the client and the server for the three layers of SSH.

The Transport Layer: Both the client and the server are sending their own **KEXINIT** as soon as the TCP connection has been established but is illustrated as consecutive messages in Figure 3. After the initial key exchange parameters have been exchanged with the **KEXINIT** message, the actual key exchange is initialised by the client sending **KEX30**. The server responds with the corresponding **KEX31**, and the key exchange is done. Before any new keys are in use, both the client and the server have to deploy the keys by sending **NEWKEYS**. When both participants have sent and received the **NEWKEYS**, the keys are deployed, and further communication is now encrypted. After key deployment, the client is requesting an authentication service to be authenticated to the server with **SR_REQUEST_USERAUTH**. If the server accepts the request it is responding with an **SR_ACCEPT** message.

The Authentication Layer is used to authenticate the client to the server, the client sends a user authentication request, **UA_REQUEST**. If the credentials are accepted by the server, it will respond with a **UA_SUCCESS** message, informing the client that the authentication was successful.
**The Connection Layer:** After the client is authenticated, it can request to open a channel by sending `CH_OPEN`. If the request can be satisfied the server responds with an acknowledgement message `CH_OPEN_CONF` after which the client can request which service to open up on that channel. In Figure 3 the client requests a terminal emulation with `CH_REQUEST_PTY` and the server accepts the request with `CH_SUCCESS`. Any participants can close the channel by sending `CH_CLOSE` to which the other party responds with its own `CH_CLOSE`.

### 3 State Fuzzing Framework

The framework used in this thesis consists of three components, the Learner, the Mapper, and the SUT which are independent modules communicating over network sockets. The framework was originally developed for state fuzzing of SSH servers previously done by Verleg [11] and Fiterau-Brostean, et. al [12] and will be extended for state fuzzing of clients.

![Figure 4: Overview of message exchange within the framework.](image)

The Learner is using model learning to infer a Mealy machine of an SSH implementation. It does so by communicating with the Mapper and transmitting high-level abstract messages, called symbols, and does not know how an SSH message is built, sent, or received. The Mapper will receive symbols from the Learner and translate these into actual SSH messages and send them to the SUT. The SUT is an implementation of SSH and is responding to the Mapper with SSH messages. The Mapper translates these messages into a symbol which it sends back to the Learner. The information flow is illustrated in Figure 4. The Learner is based on LearnLib [13], which implements several model learning algorithms. The Mapper consists of two main parts, the Paramiko framework [14] and a wrapper, the control, which communicates with the Learner and uses the Paramiko framework to send SSH messages. The Paramiko framework is a Python implementation of the SSH protocol which provides client functionality, it also has a server mode which provides server functionality. Two implementations of SSH will be under scrutiny, OpenSSH [15] and Dropbear [16]. OpenSSH is one of the most widespread...
and popular SSH implementations and Dropbear is a small implementation of SSH, built to have a small memory footprint.

3.1 Input alphabet

The input alphabet used is defined within the Learner module. In this section, only the subset of the alphabet which will be used during experiments will be presented. Table 1 and Table 3 present the symbols used, a '*' indicate which messages had to be implemented.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEXINIT</td>
<td>Exchange all supported parameters with the SUT</td>
</tr>
<tr>
<td>KEX31*</td>
<td>Initiate the key exchange</td>
</tr>
<tr>
<td>NEWKEYS</td>
<td>Deploying the new keys and communicate this to the SUT</td>
</tr>
<tr>
<td>SR_ACCEPT*</td>
<td>Accepts the service requested by the SUT</td>
</tr>
</tbody>
</table>

Table 1: Transportation layer alphabet used.

With KEXINIT the information on supported algorithms and parameters for the key exchange is exchanged between the Mapper and the SUT as well as saving the KEXINIT information of both the SUT and the Mapper itself. KEXINIT is also used as an initialisation for re-keying during the execution of the protocol.

To complete the key exchange, the Mapper has to send the KEX31 message. This initialises the internal derivation of the keys which will be based on the information gathered from both the sent and the received KEXINIT.

When the new keys are to be deployed, a NEWKEYS message has to be both sent and received by the Mapper. By sending a NEWKEYS message, the Mapper is also activating the encryption of the outgoing messages using the encryption key derived during the KEX31 message routine. Upon the Mapper receiving a NEWKEYS the outgoing messages from that point on will be encrypted. If the NEWKEYS is sent before the key exchange has been completed no keys are deployed.

For the SUT to authenticate, it first has to ask for the authentication service. A service is accepted by the Mapper by sending the message SR_ACCEPT, this does not change the internal state of the Mapper.
After the initial key exchange has finished the SUT is requesting the authentication service. There are two valid responses to an authentication service request; UA_SUCCESS or UA_FAILURE. Both messages are responses to the SUT whether the Mapper is accepting the credentials or not, this does however, not change the internal state of the Mapper itself.

### 3.2 Output Alphabet

The symbols of the output alphabet are defined within the Mapper. The Learner is constructing a Mealy machine and thus needs exactly one response for every symbol sent. However, the SUT does not always respond with a single message but sometimes responds with multiple messages or none at all. In the case the SUT does not respond at all, the Mapper has to generate a response for the Learner, these include NO_RESP and NO_CONN. The NO_RESP symbol indicates that the SUT has not responded with any message but the connection is still alive. In contrast, the NO_CONN symbol indicates that the SUT has silently terminated the connection. If multiple responses are received from the SUT, the Mapper has to build a combined message. A combined message consists of multiple symbols from the output alphabet concatenated together in the order they were received, e.g. KEXINIT+KEX30. For a more in-depth description, see [11]. Various implementations of SSH, such as BitVise, buffer messages when re-keying[12], thus leading to a burst of messages directly after the NEWKEYS. Every sequence of buffered messages could be different leading to an infinite number of transitions and states. The solution was to add a ‘*’ at the end of the first symbol if equal and consecutive symbols were received [12], thus treating all symbols received at least two consecutive times as the same output, e.g. KEXINIT+KEX30*

In Table 3, only the subset of the responses which is received during experiments is presented. The reason for this is to easier understand the visualised Mealy machines presented in Section 4. The response messages which are the same messages already presented in section 3.1 will be omitted from the table. A ‘*’ is indicating which output symbols had to be implemented.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UA_SUCCESS*</td>
<td>Communicate a successfully authentication of the SUT</td>
</tr>
<tr>
<td>UA_FAILURE*</td>
<td>Communicate a failed authentication of the SUT</td>
</tr>
</tbody>
</table>

**Table 2**: Authentication alphabet used.
### Table 3: The response alphabet of the Mapper.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIMPL</td>
<td>The SUT ignores the message, no changes to state</td>
</tr>
<tr>
<td>NO_CONN</td>
<td>The connection between the Mapper and the SUT has terminated</td>
</tr>
<tr>
<td>NO_RESP</td>
<td>The connection has not terminated but the SUT has not responded within the timeout parameter</td>
</tr>
<tr>
<td>KEX30*</td>
<td>The SUT signals the initialization of the key exchange</td>
</tr>
<tr>
<td>SR_REQUEST</td>
<td>The SUT asks the server for a service</td>
</tr>
<tr>
<td>UA_REQUEST</td>
<td>A authentication service to authenticate the SUT</td>
</tr>
<tr>
<td>CH_OPEN</td>
<td>A request to open up a channel to transfer data through</td>
</tr>
</tbody>
</table>

### 3.3 Implementations details

The Mapper can be divided into two parts, one which Verleg has implemented [11], called the control, and one which consists of the Paramiko framework. The control is communicating with the Learner directly through network sockets and with the SUT through the Paramiko framework. The Paramiko framework maintains the state of the SSH implementation, by storing critical parameters.

#### 3.3.1 Updating Paramiko

Two reasons arise to update Paramiko, with newer guidelines, key exchanges have become deprecated and thus SSH implementations are moving away from less secure key exchange algorithms supported by the older version of Paramiko, such as the Diffie-Hellman SHA1 [17]. With the latest version of Paramiko, support for more secure key exchange algorithms is included. As a result, migrating from Python 2.7 to Python 3.10 was required.

Due to changes within the Paramiko framework previously done by Verleg [11], a reference version of the old Paramiko had to be downloaded to compare the changed version to an unchanged version. The changes made to the old version of Paramiko were observed and could afterwards be carried over to the newest version of Paramiko and replace the old implementation. There were some issues in the integration part, including syntactic changes which were due to the migration from Python 2.7 to Python 3.10, as well as how the developers of Paramiko changed the way they represent some of the data structures such as the Message. Another problem was due to a part of a function which unset critical variables thus making it impossible to generate any key for the key exchange. This had been removed in Verlegs version, but
during the update process, it had been added again. This was overlooked and caused extensive problems until it was discovered and could be solved.

Before continuing with the implementation of the client fuzzer, validation tests had to be run on the Mapper. The tests included sending symbols to the Mapper manually, trying to execute the happy flow, when symbols corresponding to the connection layer messages were sent, the Mapper became frozen. The tests revealed that the updated Paramiko framework had not been fully successful, the transport and the authentication layer were working properly, but the connection layer symbols could not be parsed correctly by the Mapper. This was due to a write-lock which becomes inaccessible after the authentication sequence and thus makes it impossible to open up a data channel. As a result, the connection layer inputs currently have to be excluded from the input alphabet when fuzzing servers.

### 3.3.2 Client fuzzing

In order for the framework to support client fuzzing, the Mapper and the Learner had to be extended. The input and output alphabet both need to be increased, and the Mapper also had to be able to parse client-side messages and translate between the alphabets and SSH messages. The Learner’s adaptation is to extend the alphabet used, no other changes are necessary. This is done by extending the fullAlphabet defined within the Learner module, with the new messages the Learner is able to build a Mealy machine for an SSH client. Several messages are exactly the same for both the client and the server and could thus be reused for the client experiments. Other messages such as KEX31, SR_ACCEPT, UA_SUCCESS and UA_FAILURE had to be implemented. Paramikos server mode was the foundation for the client fuzzer, providing a function for building and parsing several messages. Some parts had to be adapted to work as was intended, for example, the building of KEX31 and SR_ACCEPT, and some messages could be used as is, messages such as NEWKEYS and KEXINIT.

As well as being able to send new types of messages the Mapper also had to be able to parse new types of messages received from the SUT. A client is expected to send a different set of messages than a server would, so for the Mapper to accept new messages these had to be added to the accepted message range. Now the Mapper would accept these without any errors, but the Mapper would only relay these messages back to the Learner. This is accepted in most cases because the information received is not important for the state fuzzing experiment, except for the KEX30 message. This contains
information critical for the key derivation and must be stored for later use, thus a parser was implemented for the \( \text{KEX30} \) message.

The client fuzzing extension was limited to the transport and authentication layer only. Due to the inaccessibility of the connection layer described in Section 3.3.1, the connection layer of the Mealy machine generated could not be validated and thus the connection layer was omitted.

Due to the nature of SSH clients, when the connection is terminated the client is as well. This lead to the need for a client driver within the Mapper, which terminates the current instance of the SUT and launches a new instance each time a new test is executed.

4 Experiments

In this section, the state fuzzing setup used during the experiments is described. All experiments were run in a virtual machine using VirtualBox, simulating a Linux Debian 64-bit environment, with an allocation of two Intel Skylake i5 cores running at 4500 MHz and 2048 MB of DDR4 running at 1500 MHz. Experiments were run for 12 hours which which proved to be sufficient, see Section 4.4.

Before running any new experiments, a reference version of state fuzzing Dropbear servers were run. This was done to compare the model generated with previous work to confirm that the framework was working as intended and that the model learning had not been impaired by the changes made. The Mealy machine was visually comparable to the work of Verleg [11] and deemed to represent the Dropbear server, see Appendix A.

4.1 Alphabet

The alphabet can be divided into three categories; the Transport Layer, the Authentication Layer, and the Connection layer. As discussed in Section 3.3.1 the Connection layer is never experimented on since it is not yet supported by the Mapper. There are a large number of symbols in the alphabet and hence some restrictions had to be done due to poor scalability of model learning. First; only messages which a server is supposed to send are used during experiments. Second; experiments will only be done for the transport and authentication layers.
4.2 Experimental Setup

Experiments are done by supplying a configure file to the Learner, see Appendix B, containing information relevant to the specific experiment. Such information includes, the input alphabet to be used for learning, the algorithms for conformance testing and various settings for testing, the time limit set for the experiment etc. The Learner supports several learning algorithms libraries and the one used during the experiments is TTT [18], a refined version of the L* algorithm [19]. Conformance testing is using a randomised version of the Wp-Method. This combination of learning algorithm and conformance testing has been successfully used in prior work, [9].

4.2.1 Dropbear

Experiments run on the Dropbear implementation were done with the full range of server-side messages of the transport layer and the authentication layer except for the UA_FAILURE. When UA_FAILURE was included, Dropbear would respond within the timeout limit and sometimes not. This might be an inherent characteristic behaviour of the client to make it less time-efficient for a malicious client to provide guessed credentials to an SSH server using Dropbear a client. This results in non-deterministic behaviour of the SUT which the Learner cannot handle, causing the Learner to terminate prematurely. Experiments run for both layers were successful and generated the model shown in Figure 5. The alphabet used for Dropbear experiments is presented in Table 4.

| KEXINIT   |
| KEX31     |
| NEWKEYS   |
| SR_ACCEPT |
| UA_SUCCESS|

Table 4: The alphabet used during Dropbear experiments.

4.2.2 OpenSSH

OpenSSH could be tested using the full range of transport and the authentication layer messages. Although the transport and authentication layers are separate layers they are not fully independent and have some coupling between each other. When running experiments on both layers in combination, OpenSSH started to exhibit non-determinism. The non-determinism appeared to occur when the Learner tried to re-key which OpenSSH did not allow, thus
leading to a response which sometimes was a `NO_CONN` and sometimes a `DISCONNECT`. Even though the outcome of these two seems to be similar and actually may be, the implementation of the Learner is sensitive to differences and handles them as two different state transitions which lead to non-determinism and premature termination. Because of this problem, the layers were tested individually. The alphabet used during experiments on the transport layer is listed in Table 5 and the alphabet used on the authentication layer is listed in Table 6.

| KEXINIT   |
| KEX31     |
| NEWKEYS   |
| SR_ACCEPT |

**Table 5:** The alphabet used during OpenSSH experiments on the transport layer.

| UA_SUCCESS |
| UA_FAILURE |

**Table 6:** The alphabet used during OpenSSH experiments on the authentication layer.

## 4.3 Quantitative Measurements

In Table 7 the statistics gathered from the experiments are presented. There are two interesting observations in the statistics, first: the number of learning tests is small in comparison to the number of conformance tests, between 4.8% and 0.7% of the tests were run during the hypothesis construction. Second: the relatively short time it took to generate the last hypothesis in comparison to the total length of the experiment, after the first 0.5% to 5.0% of the total time the last hypothesis was generated. All tests ran for 12 hours, the statistics gathered showed that it took the model learning no more than 37 minutes to generate a representation of the SSH implementation. The remaining time, over 11 hours, the Learner tried to find a counterexample but was unable to find one. With at least 95% of the time spent on hypothesis validation, but unable to find a counterexample, the length of the experiments is deemed sufficient and the results valid. The models of the transport layer generated is large in comparison to previous work [11]. Due to interactions which lead to different keys being derived between the Mapper and the SUT, several states were created to model this behaviour.
### Table 7: The statistics for each experiment.

<table>
<thead>
<tr>
<th>Implementation and version</th>
<th>Alphabet Used</th>
<th>Number of Hypothesis</th>
<th>Number of states</th>
<th>Time to last Hypothesis [MM:SS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dropbear v2022.83</td>
<td>KEXINIT, KEX31, NEWKEYS, SR_ACCPET, UA_SUCCESS</td>
<td>6</td>
<td>10</td>
<td>6.730</td>
</tr>
<tr>
<td>OpenSSH 8.4p1</td>
<td>KEXINIT, KEX31, NEWKEYS, SR_ACCPET</td>
<td>5</td>
<td>11</td>
<td>3.504</td>
</tr>
<tr>
<td>OpenSSH 8.4p1</td>
<td>UA_SUCCESS, UA_FAILURE</td>
<td>2</td>
<td>2</td>
<td>1.307</td>
</tr>
</tbody>
</table>

#### 4.4 Challenges

During the first experiments run with Dropbear, the Mapper would consistently terminate prematurely due to the exception: "Client 'e' is out of range". This occurred when receiving a KEX30 from the SUT. The 'e' is part of KEX30 and is used to derive the keys using the Diffie-Hellman key exchange method [3]. Taking a closer look at the implementation of the Mapper revealed that the Dropbear client is providing a negative number and thus the Mapper is raising the exceptions. Consulting the RFC specifications showed that providing $e$ as a negative number should not be allowed due to the calculations necessary for key derivation, thus Dropbear does violate the specifications [3]. Without changes to the Mapper to accept $e$ as a negative number, we were unable to consistently negotiate keys. This could possibly impair the interoperability of Dropbear with other SSH implementations in the same way as experienced during this experiment. Dropbear state that it is compatible with OpenSSH [16]. As seen, the Dropbear client and the Paramiko server are incompatible with each other, and the incompatibility could cause problems with the interoperability between Dropbear and OpenSSH if an OpenSSH server would terminate the connection in the same way Paramiko does. Even though the key exchange can be achieved with a negative number, this is still a violation and should be changed to conform to the specifications, to secure compatibility with other SSH implementations.
5 Analysis of State Machines

When analyzing the models we are looking for deviations from the specifications defined by the RFC. In all experiments when the SUT is started it will immediately send a message to the Mapper and is illustrated as a response to nothing on the incoming edge of the initial state s0. If there exists a path which is not the happy flow but still produces the same state this could be a possible violation or weakness of the SSH implementation and should be investigated. There can be several issues which could produce these deviations, including implementation flaws or external factors such as how the Mapper is implemented or timing errors. In this section, each model generated will be analysed and at the end, they will be compared to each other.

5.1 Dropbear

In Figure 5 the Mealy machine of Dropbear is visualised. Going through the happy flow, first the KEXINIT+KEX30 is received from the SUT. As the KEXINIT is sent, the SUT is responding with KEX30 and we transition from state s0 to s4. Proceed to send the KEX31 message, receiving NEWKEYS+SR_REQUEST+UA_REQUEST transitioning to s6. From here the transition to s8 is done when deploying the new keys with NEWKEYS, this time the SUT produces NO_RESP. From state s8 we can send SR_ACCEPT and afterwards, the UA_SUCCESS, both of which are transitioning back to state s8. From state s8, Dropbear supports re-key, this is initialised by the KEXINIT, transitioning to s9 while receiving KEXINIT+KEX30*. To complete the key exchange the KEX31 is sent and the transition back to state s6 where the deployment of the new keys can be done with NEWKEYS, and thus back to s8.

When analysing the Mealy machine of Dropbear there are two responses from the SUT which stand out. First; during the re-key phase from state s8 the SUT is responding with KEXINIT+KEX30*. It is not an expected behaviour and seems like Dropbear is parsing and responding to the received KEXINIT twice. Although this behaviour does not seem to impair security, it is unnecessary. The expected behaviour would have been to send only KEXINIT+KEX30. Second; in state s4 when sending a KEX31 to the SUT, red square in Figure 5, the response is rather interesting. The SUT is responding with NEWKEYS+SR_REQUEST+UA_REQUEST, this is not allowed by the RFC [3]. Before any participants are allowed to send authentication layer messages, keys have to be deployed. By sending SR_REQUEST and
UA_REQUEST without waiting for the Learner to deploy the new keys, the SUT violates this requirement [3]. The only messages which are allowed during key negotiation are generic transport layer, algorithm negotiation, and key exchange messages. UA_REQUEST is not included in the accepted message range while the SR_REQUEST message is explicitly disallowed by the specifications. This is a clear violation and should be changed to conform to the specifications.

A side-effect of Dropbear sending both the service request of authentication SR_REQUEST and the actual service UA_REQUEST is that the Learner could accept the authentication credentials without accepting the service. The RFC does not explicitly disallow this behaviour, but the expected message exchange described in Section 1 in [3] implies the acceptance of a service is required before sending the actual service itself. This would imply that a SR_ACCEPT should precede the UA_SUCCESS. Although, the behaviour of Dropbear is acceptable if the server authentication is using explicit authentication, which is used during the experiments. If, on the other hand, a server would authenticate using an implicit method, the RFC states that a client must wait for the server to accept the service before sending any other message [3].

There is one case when different keys are derived by the SUT and the Mapper, which happens if a KEX31 is sent before a KEXINIT, framed by the yellow square in Figure 5. The variables on which KEX31 is dependent are different between the Mapper and the SUT when the second KEX31 is sent, resulting in different derivations of the decryption key of the Mapper. This could be due to how a non-valid KEX31 message is parsed and built by the Mapper. Using debug log of the SUT shows that the Mappers decryption key is derived differently from the encryption key of the SUT, while the SUT can decrypt messages sent from the Mapper. In conclusion, only one key has been corrupted resulting in a NO_RESP response to the Learner even though the SUT can decrypt the messages from the Mapper.
Figure 5: Model of Dropbear transport and authentication layer, where the Happy Flow is illustrated in green and the re-key within is illustrated in blue. The red square is framing the messages Dropbear is sending before the keys has been deployed and the yellow square is framing the key derivation error.
5.2 OpenSSH

Within OpenSSH, there are two paths which result in different keys being derived by the SUT and the Mapper. One which is more or less identical to the one occurring in Dropbear, where the decryption key of the Mapper is derived differently from the SUTs encryption key leading to the inability to decrypt SUT messages, yellow frame in the bottom right of Figure 7. The second occurrence has the encryption key of the Mapper derived differently from the decryption key of the SUT. This results in a successful decryption of the SUTs response but the Mapper is unable to send encrypted messages to the SUT, yellow frame in the bottom left of Figure 7. This is confirmed using the debug log of the SUT, which states that messages received are encrypted but have a corrupted MAC and is a probable response if the encryption or decryption keys differ from each other. The reason behind this behaviour is that when the preceding KEXINIT is sent it changes the local key exchange object and thus the internal state. The KEX31 is dependent on this information to derive the key and the fact that the SUT is ignoring the second KEXINIT the information is miss-matched and the encryption key of the Mapper is derived using different data than what the SUT is using for deriving its decryption key.

The OpenSSH implementation does support re-key, however, it does so only after a successful authentication of the client. If a re-key is executed in a pre-authenticated state, OpenSSH does not allow this and will silently close the connection, resulting in a NO_CONN response from the Mapper.

![Figure 6: Model of OpenSSH authentication layer, where the Happy Flow is illustrated in green.](image-url)
Figure 7: Model of OpenSSH transport layer, where the Happy Flow is illustrated in green. The yellow squares is framing key derivation errors.

5.3 Comparing Dropbear & OpenSSH

As was seen in section Section 5.1 Dropbear violates the RFC specifications during the key exchange, in contrast, the OpenSSH client is waiting for the NEWKEYS before continuing to send authentication layer messages. Another
difference is how Dropbear and OpenSSH handles out-of-order SR_ACCEPT messages. Dropbear terminates the connection for all of the out-of-order SR_ACCEPT, while OpenSSH has a more lenient way of handling it, ignoring the message and keep the connection alive. It is unclear how a client should react to out-of-order SR_ACCEPT, it is stated that if a service is declined by the server it should send a DISCONNECT message and must disconnect [3]. With this formulation, one can argue that a client implementation should respond accordingly, with a DISCONNECT message and terminate the connection. As of now, there is no clear formulation of how a client should behave, and thus the only conclusion is that Dropbear and OpenSSH handle this in different ways, neither of which has any side effects found to impair security.

Both implementations support re-key but do so during different states. Dropbear supports re-key as soon as the first keys are deployed while OpenSSH allows re-key after the client has been authenticated. The RFC only requires that a re-key is not allowed during an existing key exchange, no other restrictions are defined, leading to different interpretations and ideas of implementation [3].

6 Conclusion and Future work

In this thesis, I present, to the best of my knowledge, the first protocol state fuzzing framework of SSH servers and clients. The focus of the thesis has been on updating the existing framework, previously developed by [11] and [12], extending it with the ability to conduct protocol state fuzzing of SSH client implementations, and experimenting on different SSH client implementations, Dropbear and OpenSSH, analysing and comparing them to the RFC specifications [2]–[5]. All of which have been successful.

During the analysis, violations were found in the Dropbear client implementation. A non-valid parameter, originating from the initial key exchange messages, sent by the Dropbear client caused the Mapper to terminate prematurely. Messages which are explicitly disallowed were also sent by the Dropbear client, resulting in the possibility for a server to accept client credentials prematurely.

In both the Dropbear and the OpenSSH clients, there were interactions which lead to key derivation errors, most likely caused by how the Mapper is building invalid messages. Because OpenSSH is generally ignoring out-of-order messages, this results in an additional key exchange error where keys were
derived using miss-matched variables.

There are several directions for future work, first; fully integrate Paramiko into the Mapper. This includes the full support of all previously functional symbols in the input alphabet. Second; extend the output alphabet of the Mapper to support receiving all possible client messages. Third; extend the support for other key exchange methods than Diffie-Hellman. Fourth; investigate why the Mapper and the SUT are deriving different keys when a KEX31 message is sent before a KEXINIT message.
References


A Dropbear server test

This appendix includes the visualised Mealy machines generated by the framework, before and after the new version of Paramiko had been upgraded. Figure 8 was generated after the Paramiko framework had been updated while, Figure 9 is produced by Verleg [11] in prior work.

Figure 8: A Mealy machine of Dropbear server, generated after the updated Paramiko.

Figure 9: A Mealy machine which was produced before the Paramiko was updated. Including transport layer alphabet only.
B Configure files

Figure 10, Figure 11, and Figure 12 are the complete configure files which were provided to the Learner for each experiment run.

```bash
# the name of the set you want to learn
name=DROPBEAR_CLIENT_WHOLE_PROTOCOL_
# the eq oracles used (RANDOM, EXHAUSTIVE, WORDS, CONFORMANCE), sep. by `
#eqOracle=RANDOM
# the maximum number of tests (for the RANDOM eq oracle)
maxNumTests=5000
# the specification file selected (relative to within the input dir), used by the CONFORMANCE oracle
specification=bv_spec.smv
# the time limit set for learning experiments
#PT <num> H/M/S PT10M
# P <num> D P1D P1DT6H
timeLimit=PT12H
# a maximum number of learning rounds (= hypotheses generated)
maxRounds=2
# path from ssh-learner root dir
cache=./cache/which-test/cache.db
# the maximum number of retries attempted in case of non-determinism before concluding that it
cannot be resolved
maxNonDeterminationRetries=3
# the address at which the mapper is listening
mapperAddress=localhost:8000
# the input alphabet, sep by `;`, start with `!` inputs that you want commented out
alphabet=!DISCONNECT;
!IGNORE;
!UNIMPL;
!DEBUG;
KEXINIT;
!KEXINIT_PROCEED;
!KEX30;
KEX31;
NEWKEYS;
SR_AUTH;
SR_ACCEPT;
SR_CONN;
!UA_PK_NOK;
!UA_PK_OK;
!UA_PW_NOK;
!UA_PW_OK;
UA_NONE;
UA_SUCCESS;
UA_FAILURE;
ICH_OPEN;
ICH_CLOSE;
ICH_EOP;
ICH_DATA;
ICH_EXTENDED_DATA;
ICH_WINDOW_ADJUST;
ICH_REQUEST_PTY
```

Figure 10: The configuration file supplied to the Learner, for experiments run on Dropbear client.
Figure 11: The configuration file supplied to the Learner, for experiments run on OpenSSH clients on transport layer.
Figure 12: The configuration file supplied to the Learner, for experiments run on OpenSSH clients on authentication layer.