Master Thesis
SSH Implementations: State Machine Learning and Analysis
Yuzhu Yan
Master Thesis

SSH Implementations: State Machine Learning and Analysis

by

Yuzhu Yan

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on 28 September 2017 at 13:30

Project duration: February 2017 - September 2017
Thesis committee: Dr. Ir. S. E. Verwer, TU Delft, supervisor
Dr. Ir. Stjepan Picek, TU Delft
Dr. Ir. Alessandro Bozzon, TU Delft
Ir. Dragos Mihai Amzucu, Brightsight

An electronic version of this thesis is available at http://repository.tudelft.nl/.
Abstract

Analyzing large cryptographic protocol implementations can be challenging since their implementations do not perfectly match the standard [6]. The popular, highly configurable remote login method, Secure Shell (SSH) is such an example. In this thesis, we researched the fuzzing methodologies for SSH implementations. Three tools (Backfuzz, Paramiko-sshfuzz and Protocol state fuzzing) were implemented to explore their capabilities and to determine the most effective one. The protocol state fuzzing technique resulted to be the most promising approach since it is well-developed and has recently revealed a few abnormal behaviors of SSH [6], moreover it is also actively used in several cryptographic protocol implementations (i.e. TLS). Consequently, we applied this method on a real SSH implementation, the OpenSSH library (OpenSSH6.7-p1). The results are analyzed against the source code and RFC standards. To solve the readability problem of the results caused by the complex architecture of the SSH protocol, we combined the obtained SSH state machine with D3.js data visualization technique. As a result, we developed a tool for debugging SSH implementations based on the protocol state fuzzing, code review and D3.js. Lastly, the utility tool is evaluated in a survey and future works are presented.
This Master of Science thesis is accomplished under the supervision of Brightsight and the Cyber Security group at the Faculty of EEMCS at Delft University of Technology. At this point, I want to express my gratitude for all the people who gave me any forms of help during this nine-months work. First of all, I would like to thank my supervisors Dragos Amzucu from Brightsight and Sicco Verwer from Cyber Security group for their academic guidance and feedback, as well as all the thesis committee members for the comments and remarks. Secondly, I want to thank Paul Fiterău-Broștean for sharing his knowledge and experience of the state machine learning for SSH implementations. Moreover, I would like to thank my colleagues at Brightsight for their advice and Angel de Castro for his supports and encouragement during the whole thesis. Last but not least, I want to thank my parents and friends for their unconditional support.

Yuzhu Yan
Delft, September 2017
# Contents

Preface .................................................. iii
List of Figures ........................................... vii
List of Tables ........................................... viii
List of Listings ........................................... ix

1 Introduction ........................................... 1
   1.1 Research questions ................................. 4
   1.2 Thesis structure ................................... 5

2 Background and Related Work ...................... 7
   2.1 Overview ........................................... 7
   2.2 SSH Protocol ...................................... 7
       2.2.1 Usage ......................................... 8
       2.2.2 Architecture .................................. 9
   2.3 SSH Server Implementation ....................... 11
   2.4 SSH Fuzzing Tool ................................ 12
       2.4.1 Backfuzz ..................................... 14
       2.4.2 Paramiko-sshfuzz ............................. 14
       2.4.3 Other tools .................................... 14
   2.5 Protocol State Fuzzing ............................. 15
   2.6 State machine .................................... 15
       2.6.1 Deterministic finite state machine .......... 15
       2.6.2 Mealy Machine ............................... 16
   2.7 Active Learning ................................... 17
       2.7.1 L* Algorithm ................................ 17
       2.7.2 Learnlib ...................................... 19
   2.8 Passive Learning .................................. 19
   2.9 D3.js .............................................. 21

3 Fuzzing SSH server implementation ............... 23
   3.1 Backfuzz .......................................... 23
   3.2 Paramiko .......................................... 25
   3.3 Protocol State Fuzzing ............................. 27
       3.3.1 Experimental Setup .......................... 27
       3.3.2 Result ........................................ 29
   3.4 Discussion ........................................ 30
# Code Review & Visualization

4.1 Code review .................................................. 33
  4.1.1 Static analysis ......................................... 34
  4.1.2 Dynamic Analysis ...................................... 35
4.2 State Machine Visualization by D3.js ....................... 36
  4.2.1 Static Interface ......................................... 36
  4.2.2 Discussion: Dynamic Interface ....................... 37

# Result and Analysis

5.1 Anomaly Detection ........................................ 41
  5.1.1 Test Case ................................................ 41
  5.1.2 Challenges .............................................. 44
5.2 Passive Learning using DFASAT .............................. 46
5.3 Limitation of the Mapper .................................. 47
5.4 Security evaluators’ Perceptions on the tool .............. 48
  5.4.1 Survey conducted at Brightsight ..................... 48
  5.4.2 Threats to Validity ...................................... 50
  5.4.3 Conclusion ............................................... 51

# Conclusion and Future Work

6.1 Conclusion .................................................. 53
6.2 Future work .................................................. 55

References ........................................................ 56
## List of Figures

1.1 Inferred state machine for JSSE 1.8.0_25. ........................................ 3

2.1 A simplified overview of SSH port forwarding ............................... 8
2.2 SSH Protocol Stack ............................................................... 9
2.3 The happy flow in the Transport Layer ..................................... 10
2.4 The happy flow in the Authentication Layer ................................ 11
2.5 The happy flow in the Connection Layer .................................... 11
2.6 Real-time interface of AFL fuzzer. .............................................. 13
2.7 Transition diagram of a deterministic finite state machine. ............. 16
2.8 Simple Mealy machine .......................................................... 17
2.9 Performance comparison of Learnlib, JLearn and LibAlf ................. 20
2.10 DFASAT solver work flow ....................................................... 20
2.11 State-merging example. ......................................................... 21
2.12 A force-directed graph visualized by D3.js .................................. 22

3.1 Backfuzz input pattern example. ................................................ 23
3.2 Backfuzz log under fuzzing ....................................................... 24
3.3 OpenSSH server log under fuzzing ............................................ 25
3.4 Code snapshot of the Paramiko-shshfuzz fuzzing algorithm ............. 25
3.5 Paramiko-shshfuzz fuzzing server log ....................................... 26
3.6 Paramiko server under attack ................................................... 26
3.7 The work flow of the message in protocol state fuzzing. ................. 28
3.8 Refined model learned from OpenSSH 6.9p1-2. .............................. 30

4.1 Butterfly diagram of SSH exchange identification function ............ 35
4.2 Static interface designed to visualize the learned SSH model .......... 37
4.3 Integration of Protocol State Fuzzing framework and D3.js .......... 38

5.1 Test case ................................................................. 43
5.2 Challenge ................................................................. 45
5.3 Examples of functions working logic ....................................... 46
5.4 Transitions for state-merging analysis ..................................... 47
5.5 Result of the ranking question in the survey ............................... 50

6.1 Butterfly diagram of SSH exchange identification function in sshd.c file. 61
6.2 Obtained SSH state machine using DFASAT ............................... 62
# List of Tables

2.1 State transition table example .............................................. 16  
2.2 Initial observation table ...................................................... 18  
5.1 Comparison of data visualization methods .............................. 49  
6.1 Input alphabet for the transport layer .................................. 59  
6.2 Input alphabet for the authentication layer ......................... 59  
6.3 Input alphabet for the connection layer ............................... 59
Listings

5.1 do_authentication2 function .............................................. 42
5.2 input_service_request function ........................................... 42
5.3 Updated do_authentication2 function .................................... 44
Secure Shell (SSH) is one of the most widely used secure protocol suites [1], it was proposed as a replacement for the non-protected login protocol (telnet) and insecure file transfer methods (e.g. FTP). It protects the communication's security and integrity by providing an encrypted connection over the client-server network architecture. SSH is widely used by network administrators to manage systems and applications remotely. The most known service is the secure remote login from one computer to another.

SSH is a complex application-level protocol [2], and its specifications are standardized by RFC (Request for Comments) [4252-4254]. The SSH implementation quickly gained popularity after the first release in 1995 [3]. At present, many SSH implementations are available for different environments. Dropbear\(^1\) is a lightweight open source SSH implementation particularly used for memory-constrained environments (e.g. Embedded Linux). Bitvise\(^2\) only provides SSH services to devices using Windows systems. In this research, our investigation centers around OpenSSH since it is one of the most popular open source SSH implementations. It had over 80 percent of market share in 2008 [4], being used as the default server for many unix-based systems.

Although the OpenBSD development group claims that OpenSSH is developed with a rigorous security process, a number of security flaws have still been discovered over the years [5]. Not only that, a recent paper claims that the RFC standards tend to be improperly implemented. OpenSSH and many other popular ones are reported in paper [6] to have implementation issue due to the underspecification in the RFCs and programming errors. These security weaknesses threaten the security of information through SSH communications and make them become the targets of

---

1https://matt.ucc.asn.au/dropbear/dropbear.html
2https://www.bitvise.com/
attackers.

A way (method) to effectively test and evaluate the security of an SSH implementation is needed. Fuzzing [7] can be an ideal approach in this case since it is heavily used [8] by developers or quality assurance (QA) teams before releasing a product to the market. This highly automated testing technique works by providing invalid, unexpected or random inputs which are expected to trigger crashes, failing built-in code assertions or potential memory leaks due to the bugs in programs. It has proven to be a successful technique at discovering critical vulnerabilities in security protocol implementations. For example, this year 4 serious security vulnerabilities ([CVE-2017-7520]-[CVE-2017-7522] and CVE-2017-7508) in OpenVPN were found by a fuzzing technique. In the report [9] of them, the author mentioned “Most of the issues were found through fuzzing. I hate admitting it.” In addition, several important security flaws (i.e. CVE-2015-0291 and CVE-2015-1788) in OpenSSL (The widely used SSL protocol implementation) and OpenSSH [6] were detected by the fuzzer AFL using the mutation-based fuzzing technique.

Protocol state fuzzing is a useful state-of-the-art fuzzing technique to systematically analyze security protocol implementations [10]. As a black-box testing technique, protocol state fuzzing can automatically infer the state machine of a program. It uses the approach called active learning which is a semi-supervised state machine learning process. It infers the state machine by sending inputs to the target program under the supervision of a feedback loop. The obtained model gives insight into the logic of the program. Spurious behaviors on the state machine might be indications of flaws in the program design. De Ruiter and Poll [10] applied it on TLS protocol implementations and discovered a security flaw (Figure 1.1) on JSSE 1.8.0_25. This bug was fixed soon after they reported it to the development team.

The aim of this thesis is to obtain a methodology which can effectively detect the potential security vulnerabilities on the SSH implementations. To achieve this, we start with an investigation on the available SSH fuzzing tools. Three SSH fuzzing tools (Back-fuzz, Paramiko-sshfuzz and Protocol state fuzzing) are deployed to fuzz the SSH servers. Our research reveals that Back-fuzz has stopped being maintained in an early stage and Paramiko-sshfuzz only provides an interface which supports basic brute-force fuzzing algorithms. This indicates that more work needs to be done to effectively fuzz an SSH server. Neither of these two approaches is well-documented as well. Compared to them, protocol state fuzzing is thought to be the most promising technique for SSH implementations since it is currently in development and has recently discovered several unintended behaviours of the OpenSSH server [6].

To have a deep understanding of the protocol state fuzzing technique, we research the papers relevant to this topic and dove into the internal design of the SSH protocol state fuzzing prototype mentioned in paper[4]. We obtain the state machine of

Figure 1.1: The inferred state machine model for JSSE 1.8.0_25. The circles represent states. The line with arrow is the transition between states. The transitions in green are the correct path leading to an exchange of application data. The red dashed line indicates the security flaw.

OpenSSH6.7-p1 by applying this approach on the OpenSSH server. Subsequently, we investigate the learned state machine. Firstly, we examine the state machine by referencing the RFCs. There is no obvious violation observed. Secondly, we identify the interesting behaviours on the state machine as the test cases and explain them by performing code review on the source code of OpenSSH6.7-p1. However, the source code inspection of the SSH implementations is not easy due to the complex design of the protocol. Thus we employ a systematical analysis by using two common code review methods, static analysis and dynamic analysis. The tool-based static analysis gives us an overview of the structure of the source code. We learn the logic of the core source code from the static analysis and use it to assist the dynamic analysis. The dynamic analysis is conducted by testing the test case after setting the server in the debugging mode. By combining the observed server log with the static analysis results, we explain the programming logic behind the abnormal behaviors on the state machine.

The aforementioned research gives us the insights of the SSH protocol, the internal design of OpenSSH6.7-p1 and the automated fuzzing technique protocol state fuzzing. We believe that the state machine of an SSH implementation reflects the logic of its design and it can be a helpful approach to detect logic errors in the implementations. Our investigation of obtaining an effective SSH implementation testing tool does not stop here. We also propose a design of an SSH state machine
based debugging by implementing the data visualization technique D3js. On the interface of this implementation, we visualize the obtained results from the protocol state fuzzing and the code review. Users can access the source code of any parts they are interested in on the state machine by simply interacting with the interface. Later we explain the usage of this tool (development) by showing example test case and discuss the limitations and challenges of it. We hope this extensive development can not only help the testers easily spot the spurious behavior on the SSH state machine but also help locate errors faster, in the complex source code of the SSH implementations.

To assess the usefulness and get a professional opinions on our work, we presented our research and development to the engineers at Brightsight, a company which is dedicated to security evaluation. The positive feedback from the participants proves that our SSH state machine based debugging development can indeed bring benefits for the development of the SSH implementations. Furthermore, the professional suggestions from them should also be valued in the future development.

1.1. Research questions
To guide the project five questions were identified and analyzed. In the end, they are expected to be answered by the results of this project.

RQ1: How good are the existing fuzzing techniques for SSH implementations?
To answer this question, we perform an investigation on the available SSH implementations fuzzing tools. We select three black-box testing tools (Backfuzz, Paramiko-sshfuzz and Protocol state fuzzing) and apply them on the SSH servers. In chapter three, we explain why protocol state fuzzing is a valuable tool for this research. It focuses on 5 aspects (Code quantity, fuzzing method, performance, extensibility and development status)

RQ2: To what extent can the state machine learning methodologies be used to improve the understanding and testing of SSH implementations?
To answer this question, we first try to obtain the state machine of an SSH implementation. OpenSSH is selected as our experiment subject due to its significant user base. Through the theoretical study, we learn that both protocol state fuzzing (active automata learning methodology) and DFASAT (passive automata learning methodology) can extract the state machine from SSH implementations. Hence we decide to implement both. For the first method, we contacted a researcher who has expertise in the field of protocol state fuzzing. Under his supervision, we perform this technique on the actual SSH server and obtain the state machine of it. Subsequently, we perform passive learning by using a tool called DFASAT on the same SSH server. The experimental setup of it is explained in chapter 5. The obtained result consists of 23 states, it is identical to the result obtained using protocol state
1.2. Thesis structure

This research consists of 6 chapters. In the second chapter, we give an overview of the background knowledge of SSH protocol and all the involved technologies. In chapter 3, we introduce different SSH fuzzing tools and deploy three fuzzing techniques on the SSH servers. Then, a comparison between the fuzzing tools is carried out based on the fuzzing results. In the following chapter, we investigate code review approaches on the SSH source code. Subsequently, they are applied to analyze the learned state machine. At the end of this section, we give a description of the design of the SSH state machine based debugging tool. In chapter 5, we discuss the performance, limits and challenges of the tool we developed by showing test cases. The results of passive learning and the survey are also presented here. Finally, the thesis is concluded in chapter 6.

fuzzing.

Later we perform code review on the OpenSSH source code and use the result to analyze the interesting behaviours observed in the learned state machine. To achieve this, we consult the code review methods which are suitable for OpenSSH from a security protocol testing expert. In chapter 4 we describe the exact approaches we used. After that we identify the interesting behaviors by examining the learned state machine. By using code review methods, we analyze the corresponding code of the behaviors and explain the program logic behind them.

RQ3: How can we take advantage of the current protocol state fuzzing technique to obtain a useful (easy) debugging tool for SSH implementations?

The protocol state fuzzing results reveal the logic of the SSH implementation. Hence we hope to obtain a debugging tool based on this result. During our development, we integrate protocol state fuzzing with other techniques such as D3.js and the code review results. The design of it is described in chapter 4 and its usage is explained by showing an example in chapter 5.

To test the usefulness and get the professional advice about our development, we conduct a survey on the security evaluators from Brightsight. The survey consists of three parts: preliminaries of the tool, demonstration and questionnaires. In the end, we received positive feedback from the evaluators and suggestions which can be used to optimize the tool in the future.
2

Background and Related Work

2.1. Overview

In this chapter, we give an insight on all the background knowledge derived from the literature review. In the first section the Request for Comments (RFC) standards are introduced, these standards define SSH. Subsequently, we carry out an extensive exploration on existing SSH fuzzing methodologies, such as the automated and systematic fuzzing approach called protocol state fuzzing. The research has shown it is superior over other approaches. As protocol state fuzzing is a state-of-the-art fuzzing technique which is completely new to us, we use two sections to illustrate all the related knowledge: State machine and active learning. Furthermore, we also investigated a passive learning approach to analyze SSH implementations, and intend on using the results to compare active and passive learning (Shown in chapter 5). At the end of this chapter D3.js framework is introduced, D3.js will be used to represent the results of protocol state fuzzing..

Multiple fields are involved in our research. To give a direct, clear and simple literature review, this chapter is divided into nine sections. Each section specifies one of the earlier mentioned technologies.

2.2. SSH Protocol

In this section we present all the required SSH knowledge, summarized from the literature review. Apart from the general introduction of the protocol itself, we also refer to RFC standards to familiarize the reader with the specifications of different messages and layers in the protocol.
2.2.1. Usage

Secure Shell (SSH) is a cryptographic network protocol running on top of TCP/IP, to enable remote network service operations over an untrusted network. SSH provides a series of services including well-known ones like remote login and the execution of commands, and a number of other secure network services, such as SCP (Secure Copy), SFTP (Secure FTP), SSHFS (Secure Shell File System), ISMS (Integrated Security Model for SNMP) and Port Forwarding (tunneling). SSH deploys a client-server architecture. When a connection is established, all the messages between client and server will pass through a encrypted channel. TCP port 22 is assigned as a default port for SSH service connection. Figure 2.1 shows an example of port forwarding between client and server [11].

![Figure 2.1: A simplified overview of SSH port forwarding. The encrypted connection guarantees the confidentiality and integrity of the messages between the SSH client and the SSH server. When the application requests a service on the remote server, the SSH client sends the request over the secure channel to the server. The server then communicates with the actual application server to execute the service requested by client side.](image)

Two entities participate in the SSH communication showed in Figure 2.1, the SSH client and the SSH server. The SSH server is a software program that uses the secure shell protocol for accepting connections from remote computers. The SSH client is a software program that uses secure shell to connect to a remote server. Both ends of the client/server connection are encrypted. Later on, we will explain how to implement protocol state fuzzing on an SSH implementation. The protocol state fuzzing tool will act as a SSH client and communicate with the SSH server. The tool collects and analyzes all the messages during the communication.

There are two generations of SSH, v1.x, v2.x. The initial version SSHv1.0 was developed by Tatu Ylönen as an alternative for insecure login protocols (e.g. TELNET). SSHv2.0, which was launched at 2006, provided a standardized definition of SSH, fixed a number of security flaws and added many new features\(^1\). Although SSH v2.x evolves from v1.x, they are incompatible with each other. For backward compatibility reasons, some servers includes both SSH versions. This is advertised as SSH v1.99, as defined in RFC 4253 [12].

---

\(^1\) SSH2 was designed to avoid patent issues regarding RSA, to fix the CRC data integrity problem that SSH1 has, and for a number of other technical reasons. [source](https://www.openssh.com/goals.html)
2.2.2. **Architecture**

There are multiple ways to classify the structure of SSH protocol. For example, according to the function of its components it can be identified as a server, client, session or key. Also, it can be classified into different layers. In accordance with RFC standard, SSH protocol is standardized into three layers: Transport Layer Protocol (RFC 4253), Authentication Protocol (RFC 4252) and Connection Protocol (RFC 4254). Figure 2.2 shows the relationship between SSH protocol and TCP/IP protocol, as well as the internal architecture of SSH [13]. Next, we will introduce the protocol specifications and the happy flow of each layer\(^2\). We follow the name rule defined by the RFC standard. However, in some figures the name of the transitions will be different from the RFC standards, since they originate from different scientific papers [4]. We have attached a mapping table to give a specific explanation in the Appendix A. To check the source of Figure 2.3, 2.4, 2.5, we refer to paper [4].

![SSH Protocol Stack](image)

**Figure 2.2: SSH Protocol Stack.** The diagram is composed by five blocks: TCP block, IP block and three other blocks which constitute SSH protocol. Neither the SSH protocol itself nor each layer of SSH protocol is independent. SSH protocol depends on TCP/IP protocol. The SSH session can only start after a TCP/IP connection is established. The three layers of SSH protocol, rely on each other, too. The Transport Layer Protocol is a basis for the secure network services. It guarantees a strong encryption, server authentication and integrity protection. The Authentication Protocol runs on the top of the Transport Layer and provides a single authenticated tunnel for the connection layer, as well as authentication methods. The Connection Protocol runs on the top of the Transport Layer and the Authentication Protocol. It provides channels for networks services and converges all multiplexed channels into a single encrypted tunnel.

**Transport layer** is a secure, low level transport protocol which provides strong encryption, server authentication, integrity protection and optional compression. It supplies a basis for all the following communication during an SSH session.

When the client tries to connect with the server, the events in the Transport Layer happen as a sequence: the client first initiates a SSH connection by sending a TCP connection request to the server. This execution follows the TCP protocol. All the

---

\(^2\)A happy flow is when the protocol follows a normal execution sequence for each layer
following messages run on top of the established TCP connection. Once the connection is established, both the client and the server send an identification string to exchange the version information. Key exchange uses the algorithm required by the RFC standard and is preceded by an exchange of their identification strings. RFC stipulates that Diffie-Hellman algorithm is the only key exchange algorithm that can be used. During the key exchange, the client and the server will negotiate the authentication algorithm and the session key to be used. The preferences of both sides are sent with a SSH_MSG_KEXINIT message standardized by RFC 4253. SSH_MSG_NEWKEYS sent by both sides marks the end of the key exchange. Any subsequent messages between the client and the server must use the newly created key and algorithms. Soon after, the client sends a SSH_MSG_SERVICE_REQUEST message to the server, and the server is obliged to respond SSH_MSG_SERVICE_ACCEPT if the requested service is supported. Otherwise, a message SSH_MSG_DISCONNECT should be sent to terminate the connection. A happy flow of the transport layer proposed by paper [4] is showed in Figure 2.3.

Authentication protocol is a middle layer between the SSH transport layer protocol and the connection protocol. It provides a single authenticated tunnel for an SSH connection protocol. Authentication is a client-driven process. The server initiates the authentication by sending a message containing its supported authentication methods to the client. The authentication methods include: public key, password or host-based client authentication methods. The authentication method can also be set as “none”. However, it is not recommended for “none” to be listed in the supported authentication methods sent out by the server. If the client retries many times exceeding the threshold, or the requested service is unavailable, the server should disconnect the connection due to timeout or failed authentication. If the authentication request is accepted by the server, it should reply SSH_MSG_USERAUTH_SUCCESS, otherwise it will send SSH_MSG_USERAUTH_FAILURE. As soon as the client is successfully authenticated, all messages will run on the top of the authentication protocol. Any other authentication requests received by the server will be ignored. The happy flow for the authentication layer is showed in Figure 2.4.
2.3. SSH Server Implementation

Connection protocol runs on the top of transport layer protocol and authentication protocol. The connection layer services are only available when the authentication is successful. A service is named as a channel. Services supported by the SSH protocol includes forwarded TCP/IP connections, forwarded X11 connections and remote execution of commands. Either the client or the server can send a channel request at any time. Multiple channels are multiplexed into a single connection. When a channel request is received, the recipient should reply either SSH_MSG_REQUEST_SUCCESS or SSH_MSG_REQUEST_FAILURE to accept or deny the request. When the SSH_MSG_CHANNEL_OPEN message is dispatched to open a new channel, the recipient is obliged to reply SSH_MSG_CHANNEL_CONFIRMATION or SSH_MSG_CHANNEL_OPEN_FAILURE. SSH_MSG_CHANNEL_CLOSE message should be sent when one side wants to terminate a channel. A channel is only considered as closed when both sides send and receive SSH_MSG_CHANNEL_CLOSE message. Figure 2.5 is an example of a client requesting to open a channel for a “pseudo terminal” service.

2.3. SSH Server Implementation

As earlier mentioned, RFCs are official documents for internet standard which include specifications, communications protocols, procedures and events. It is published to standardize the principal technical development and standards-setting bodies for the internet protocol. In this case, a question emerges: since the development of application of the internet protocol has been regulated by RFC standard, the published internet protocol should always follow the regulations proposed by the standard. Any violation of protocol logic should not be seen. Unfortunately, earlier research has shown that this is not always the case. Filăță-Broștean, et al.[4] uncovered that minor violations of the standard exist in several SSH imple-
mentations, i.e. Dropbear, OpenSSH, etc. Fiterău-Broştean, et al. [4] is not the first one who to notice this, other similar logical flaws have been pointed out earlier. Chen et al. [14] and Udrea et al. [15] conduct investigation on OpenSSH C implementation and two C implementation of SSH to prove the security vulnerabilities. It has actually reached a consensus that protocol violations are common in SSH implementations. These experiments outcome conducted by predecessors’ are the starting point for our study of SSH implementation logic. Next, we would like to present an introduction about a few available SSH implementations as a preliminary of our study.

A SSH server is a software program configured by the secure shell protocol aiming at bridging connections to a client. It secures network communication through encrypting network traffic over multiple authentication methods by building up secure tunneling between the client and the server. In general, SSH implementations can be either open source software or proprietary software. We only investigate two open source SSH implementations in this research due to the convenience of investigating and modifying the code. The first implementation is the most widely used open source SSH implementation, OpenSSH. Another one is Dropbear, which is designed as a lightweight SSH implementation for environments with low memory and processor resources (e.g. embedded devices).

**OpenSSH** also known as OpenBSD Secure Shell is a suite of programs based on the Secure Shell (SSH) protocol. It relies on LibreSSL library for some of its cryptographic routines. In 1999, the initial version was launched as part of OpenBSD operating system. OpenSSH is developed with a rigorous security process [16]. It assures the encryption for all traffic to eliminate eavesdropping, connection hijacking, and other attacks. Furthermore, it also provides a large suite of secure tunneling capabilities, several authentication methods, including password, public-key authentication, per-user keys, host-based authentication, and other sophisticated configuration options. OpenSSH supports two major sub-variant protocols SSHv1.3 and SSHv1.5 and offers some compatibility with the SSHv2.0 protocol.

**Dropbear** Dropbear is a relatively small SSH software program available for both SSH server and client. It is served as a replacement for standard OpenSSH in the environment with low memory and processor resources. Dropbear implements SSHv2.0. It is compatible with OpenSSH key authentication. To save storage, some features can be disabled during the compiling process.

### 2.4. SSH Fuzzing Tool

Many software security flaws can only be uncovered under very specific conditions [17], i.e. a specific configuration of inputs together with a certain runtime. One approach to detect software vulnerabilities is fuzzing. Fuzzing is an automated testing technique to capture software vulnerabilities by injecting invalid, unexpected, or random data inputs to a computer program. Fuzzing programs can be classified as generation-based or mutation based depending if the input is generated from
scratch or from other inputs. It can also be categorized into white, grey or black-box fuzzing according to the awareness of the internal structure of the software. Furthermore, it can be a dumb or smart fuzzing, depending on the awareness of the input structure [18].

Nowadays, a simple clustering of fuzzing into several categories is not enough to cover the features of a fuzzing tool because of the booming software development. More and more sophisticated, specialized software development makes the traditional fuzzing technique become less powerful than it used to be. Therefore, the variation of traditional fuzzing targeting at specific type of software has flourished in recent years. For instance, configuration fuzzing, proposed by Dai, H., Murphy, C., & Kaiser, G [17], can mutate the configuration of running applications to check for vulnerabilities that can only arise in certain conditions. Mutation-based fuzzing tool American Fuzzy Lop (AFL) (see Figure 2.6) employs genetic algorithms to detect significant software flaws and increase code coverage of test cases at the same time [19].

![American Fuzzy Lop (AFL) Interface](image)

Figure 2.6: Real-time interface of AFL fuzzer. AFL provides a interface where users can monitor the fuzzing status. AFL fuzz programs through a genetic algorithms, it mutates inputs according to the genetics inspired rules which ranks the inputs by a fitness function based on the unique code coverage.

The trade-off between the cost of the software designing and the fuzzing efficiency should be considered by the developers of fuzzing tools. For example, the traditional dumb fuzzing only use completely random data with no knowledge of what the expected input should look like. It is an inexpensive method but with low efficiency since sometimes the inputs are only accepted by the programs under conditions. Smart Fuzzing can solve this problem since it can create valid input with alternations. However, smart fuzzing tool requires higher cost on designing since it needs to be programmed with the knowledge of the input format (A protocol definition or rules for a file format).
One question from our research is to find out the most useful and efficient methodologies to fuzz SSH implementations. Hence, we conducted a survey about existing SSH fuzzing tools. We found from our survey that the development in this field is quite inactive compared to other protocols, i.e. SSL. The most promising tools are described below:

### 2.4.1. Backfuzz

**Backfuzz** is a fuzzing framework for multiple protocols (FTP, HTTP, IMAP, SSH, etc) written in python [20]. The framework is relatively simple and it can send a specific message pattern as input to the testing server. The script has several predefined functions. The user is required to write their own plugin if they want to implement Backfuzz for other protocols. We tried the tool and it performed a basic dumb fuzzing process (See next chapter). This project is in beta stage and not in a very active development. The author mentions “Backfuzz requires a lot of work to get better” in his GitHub repository. ³.

### 2.4.2. Paramiko-sshfuzz

Paramiko is a Python library of the SSHv2 protocol, providing both client and server functionality [21]. Paramiko-sshfuzz [22] is a generic decorator based SSH protocol message fuzzer based on Paramiko library. Compare to backfuzz, Paramiko-sshfuzz is more mature but still not an off-the-shelf fuzzing tool for SSH protocol. The fuzzing_corpus file provided by Paramiko-sshfuzz allows users to write custom fuzzing algorithms or use other testing tool (i.e. Learnlib) as extension to fuzz SSH implementations.

### 2.4.3. Other tools

Apart from applying the two frameworks stated above, we also investigated as much state-of-the-art research findings as possible to sharpen our understanding about SSH fuzzing techniques. Similar to the already mentioned technique, configuration fuzzing, researchers also applied it to OpenSSH to collect fuzzable configuration variables, and create a fuzzing function to fuzz OpenSSH by mutating the configurations [17]. Moreover, a commercial fuzzer, beSTORM also claims that their tool can perform completely automated dynamic testing on the SSH protocol. To our knowledge, the most suitable methodology we have found so far is a systematic and automated technique called protocol state fuzzing which all our following work is based on. Due to its importance to our work, we use an individual section to introduce it.

³ [https://github.com/localh0t/backfuzz](https://github.com/localh0t/backfuzz)
2.5. Protocol State Fuzzing

Protocol State Fuzzing is an automated and systematic analysis approach that uses state machine learning to infer state machines from protocol implementations. The state machine reflects the logic of systems, by inspecting them we expect to discover suspicious behavior that might be an indication of flaws in the program logic \cite{10}. One significant feature of this black box fuzzing technique is it requires no awareness of internal structure of the program. This implies that it can fuzz closed source SSH implementations written in any language. The fuzzed implementation is called System Under Test (SUT).

Similar to a traditional fuzzer sending special inputs to SUT, protocol state fuzzing is essentially a program that fuzzes implementations with different sequences of messages. It automatically extracts state machines from protocol implementations according to the messages, by means of automated state machine learning techniques. Hence, to design a testing tool using protocol state fuzzing, the tester should have a deep understanding of protocol communication logic.

State machine learning has been successfully applied to different fields. Aarts, et al.\cite{23} obtained state machine models for banking smartcards. The model helps anyone familiar with EMV’s standard, it can easily identify logical decisions taken in an implementation. Using similar methodologies, researchers also tested biometric passports \cite{24} and smartcard readers \cite{25} for Internet banking. In the matter of protocol implementations, Fiterău-Broştean, et al.\cite{26} reveal several instances in which TCP implementations do not conform to their RFC specifications.

2.6. State machine

By sending inputs to the SUT and observing its responses, protocol state fuzzing tools infer a finite state machine. The result is shown as a Mealy machines, therefore we give an overview on state machine knowledge.

State machines represent the logic behind computer programs, they can be classified as finite state machine or infinite state machine. The former is composed by a finite number of states and transitions and their behavior can be modeled into flow graphs. When a condition is met a state can transits from one to another. In this research, two types of finite state machines are analyzed: deterministic finite state automata (DFAs) and Mealy machines.

2.6.1. Deterministic finite state machine

A deterministic finite state machine (DFA) contains 5-tuple, \( M = (Q, \Sigma, \delta, q_0, F) \), consisting of:

- A finite set of states \( Q \)
- A finite input alphabet \( \Sigma \)
• A transition function $\delta: Q \times \Sigma \rightarrow Q$, which maps input symbols with the subsequent state

• The initial state $q_0$, belonging to $Q$

• A set of accepting states $F$, belonging to $Q$

Deterministic means the uniqueness of the computation. Here we give an example\(^4\) of a state transition table, representing a DFA. Table 2.1 defines the transition diagram in Figure 2.7

<table>
<thead>
<tr>
<th>$\rightarrow$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$q_2$</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$q_1$</td>
<td>$q_1$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_1$</td>
</tr>
</tbody>
</table>

Table 2.1: A state transition table example. The $\rightarrow$ points to the initial state: $q_0$. The $*$ labels the final state $q_1$.

![Transition Diagram](http://www.cse.chalmers.se/coquand/AUTOMATA/o2.pdf)

Figure 2.7: A transition diagram of a deterministic finite state machine. Each arrow is taken as a transition and labeled by a number which is an input from the input alphabet. The start state, $q_0$, is identified by a circle with an orphaned arrow pointing towards it, it reveals the beginning of computation. The diagram ends up in the final state $q_0$ shown in a double circle.

### 2.6.2. Mealy Machine

In state machine learning, we use Mealy machines to represent protocol entities. A Mealy machine \([27]\) is a finite state machine (FSM), whose output depends on the present state as well as the present input. A Mealy machine is a deterministic finite-state transducer, which means for each input and state, there is at most one possible transition. A Mealy machine is illustrated as a 6-tuple $M = (S, S_0, \Sigma, \Lambda, T, G)$ containing the following:

• A finite set of states $S$

• Initial state $S_0$, it belongs to $S$

• A finite input alphabet $\Sigma$

\(^4\)http://www.cse.chalmers.se/ coquand/AUTOMATA/o2.pdf
• A finite output alphabet \( \Lambda \)

• A transition function \( T: S \times \Sigma \rightarrow S \), which maps input symbols to the subsequent state

• A transition function \( G: S \times \Sigma \rightarrow \Lambda \), which maps input symbols to the subsequent output

Figure 2.8 shows an example\(^5\) of Mealy machine:

![Mealy Machine Diagram](https://en.wikipedia.org/wiki/Mealy_machine)

Figure 2.8: A simple Mealy machine. A Mealy machine is composed by states (shown as circles) and transitions between states (shown as arrows). Transitions start from the current state to the successive state and are labeled with X/Y. X represents the input symbol from the alphabet (shown in red), Y represents the output symbol (shown in blue). The initial state \( S_i \) contains a transition from \( S_i \) to \( S_0 \) with an input 0 and output 0. When given an input 1, state \( S_i \) transform to state \( S_0 \) with an output 0.

### 2.7. Active Learning

As we explained, protocol state fuzzing is an automated technique to obtain state machines from protocol implementations. The fuzzing tool infers state machine models by providing inputs and observing outputs. The automated learning technique used is called active learning due to its dynamic interaction with SUT. In this section, we will first have a look at the active learning algorithm we implemented in protocol state fuzzing, called L* algorithm, as well as the state-of-the-art active learning tool, Learnlib [28].

#### 2.7.1. L* Algorithm

Deterministic Finite Automata (DFAs) are a useful way to describe a regular language. The L* algorithm [29], proposed by Angluin, is a polynomial active automata learning algorithm to extract DFA from a software, using a minimally adequate Teacher (MAT) framework. In MAT, the software to be learned (SUT in our prototype) is a Teacher who is the only one knowing the automaton. Another entity

\(^5\)https://en.wikipedia.org/wiki/Mealy_machine
in the framework is the Learner. It knows nothing about the automata except the input and output alphabet. The framework works in a semi-supervised process: The Learner infers the automata by posting questions to the Teacher. The latter only replies to known inputs when it receives a request. Angluin proves that one can efficiently learn a finite state machine if the Teacher is capable to answer all kinds of queries about the target finite state machine:

- **Membership query.** The Learner asks the Teacher about the output by sending an input sequence. Input alphabet contains multiple input sequences.

- **Equivalence query.** According to the queries and corresponding answers, the Learner conjures a hypothetical Mealy machine and asks if it is equal to the Mealy machine of the Teacher. If so, the Teacher answers “yes”, otherwise replies “no” together with a counterexample. The counterexample is an input sequence formed by input alphabet. The Learner refines the conjecture through the reply of counterexample by membership queries.

The learning process iterates until learner’s conjecture is equivalent to the target finite state machine.

To record the learning process of the L* algorithm, paper [29] uses an observation table which is denoted as \((S, E, T)\) to save the information about the queries. In the table, \(S\) is a nonempty finite prefix-closed set, \(E\) represents a nonempty finite suffix-closed set. The finite input alphabet is defined as \(A\). A function \(T\) maps \((S \cup S \cdot A) \cdot E\) to \(\{0, 1\}\). The unknown regular set (the Teacher) is \(U\). \(T(u)\) is 1 if and only if \(u\) is a member of \(U\). Initially, the observation table is \(S = E = \{ \lambda \}\).

<table>
<thead>
<tr>
<th>(T)</th>
<th>(\lambda)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\lambda)</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.2: Example of the initial observation table. \(S = E = \{ \lambda \}\)

The observation table will be used by L* algorithm to eventually construct a DFAs acceptor. Table 2.2 is an example of the observation table visualized by using two-dimensional arrays. The row is consisted by \(S \cup S \cdot A\), the column is \(E\). The element \(\lambda\) of \(S\) in the row will be used to construct the acceptor. The elements \((0, 1)\) of \(S \cdot A\) in the row are used to construct the transition function. The elements in the column belongs to \(E\). The entry for row \(s\) and column \(e\) is \(T(s \cdot e)\).

The table will be re-constructed by the L* algorithm each time after receiving a reply from the U, the re-construction is determined by if the table is consistent and closed. The consistent observation table means that for any elements \(s_1\) and \(s_2\) in \(S\) satisfy \(\text{row}(s_1) = \text{row}(s_2)\), for all \(a\) in \(A\), \(\text{row}(s_1 \cdot a) = \text{row}(s_2 \cdot a)\) holds.
closed observation table should satisfy that for any element \( t \) in \( S \cdot A \), there exists one element \( s \) in \( S \), \( row(t) = row(s) \) holds. The following algorithm 1 shows the details of the process [29].

**Algorithm 1 L* algorithm**

**Require:** Initialize \( S \) and \( E \) to \( \{ \lambda \} \), construct the initial observation table \((S, E, T)\)

repeat
  while \((S, E, T)\) is not closed or not consistent do
    if \((S, E, T)\) is not consistent then
      find \( s_1 \) and \( s_2 \) in \( S \), \( a \in A \), and \( a \in E \) such that
      \( row(s_1) = row(s_2) \) and \( T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e) \)
      add \( a \cdot e \) to \( E \)
      and extend \( T \) to \((S \cup S \cdot A) \cdot E\) using membership queries
    end if
    if \((S, E, T)\) is not closed then
      find \( s_1 \in S \) and \( a \in A \) such that
      \( row(s_1 \cdot a) \) is different from \( row(s) \) for all \( s \in S \)
      add \( s_1 \cdot a \) to \( S \)
      and extend \( T \) to \((S \cup S \cdot A) \cdot E\) using membership queries
    end if
    Once \((S, E, T)\) is closed and consistent, let \( M = M(S, E, T) \)
    Make the conjecture \( M \)
    if the teacher replies with a counter-example \( t \) then
      add \( t \) and all its prefixes to \( S \)
      and extend \( T \) to \((S \cup S \cdot A) \cdot E\) using membership queries
    end if
  end while
until the Teacher replies yes to the conjecture \( M \)
Halt and output \( M \)

\section*{2.7.2. Learnlib}

The open-source Java library Learnlib is based on the L* algorithm, it is deployed in a MAT framework to carry out the active automata learning. Learnlib evolves from two closed source learning tools, JLearn and LibAlf, which support Mealy machine learning and DFA learning, respectively. 2.9 gives a performance comparison result of Learnlib, JLearn and LibAlf [30].

\section*{2.8. Passive Learning}

In this section we will give a high-level introduction about another state machine learning methodology called passive learning. Although it is not the core knowledge of this thesis, we still implement it on the SSH server since we want to see whether the results will be the same as the one derived from active learning. The details of
Passive learning does not employ MAT framework since it is an offline approach which does not require real-time data to guide the learning process. The passive state machine learning algorithms are employed before or after the execution of the program. It copes with a given set of observations to learn the model, while active learning algorithms can ask for the additional observations if needed. The work flow in Figure 2.10 shows an example of a passive learning process. As we can see the execution traces are collected and labeled after executed the programs. Then the passive learning tool DFASAT solver is deployed to process the traces to obtain the DFA model. DFASAT solver is based on the greedy heuristic algorithm called evidence-driven state-merging (EDSM) algorithm. The purpose of using it is to reduce the size of the state machine through merging states, based on Boolean satisfiability (SAT). It will eventually produce an equivalent DFA with the minimal number of states. The design of DFASAT we refer to paper [32].

State-merging algorithms are commonly used in passive learning. It works by merging equivalent states, producing a new state with the same incoming and outgoing
transitions. This merged state will replace the old ones. This process iterates until no more consistent merges are possible. State-merging algorithms can efficiently reduce the size of a state machine. Figure 2.11 shows a state-merging example mentioned in paper [32]. There are many state-merging algorithms. One successful state-merging algorithm, Red-blue algorithm, also uses the same principle in Figure 2.11, but with adding colors to states to guide the merging process.

![State-merging example](image)

Figure 2.11: A state-merging example. The left figure shows the original DFA. The States to be merged are encircled in a dotted line. In the middle figure, state 0 and state 2 have already been merged. The merging of state 1 and 3 depends on if their incoming transitions from state 0/2 are using the same inputs. The final DFA is showed on the right with the smallest DFA size and consistent with the given DFA on the left.

### 2.9. D3.js

The result of the protocol state fuzzing technique is a state machine written in DOT, a plain text graph description language. One common way to visualize the plain text to the state machine graph is using the graph visualization software called Graphviz. However, we implemented in on the SSH protocol state fuzzing result and obtained a graph with a low human readability due to the complex logic of the SSH protocol. This readability problem makes it hard for the testers to follow their concerned states/transition and identify the details of the state machine since the irrelevant information on the state machine distract their attentions. To solve this problem, we decide to use a more advanced data visualization technique called D3.js.

D3.js (Data-driven documents) is a javascript library created to achieve web-based interactive visualization. It is an auxiliary method to solve the complexity and incomprehensibility of big data problems, increasing the value of the data. It allows users to effectively and quickly explore data and communicate their insight by interacting with the data. Its advantages are:

- Well-suited for data related web programming since it can greatly reduce the code amount.
- Highly customizable. No pre-built in visualization methods, instead relies on many available libraries.
• High compatibility. Built using common web standards such as HTML, CSS, Javascript and svg, it does not require extra learning efforts. D3.js supports any modern browser.

Figure 2.12 displays an example of a D3.js implementation of the mobile patent suits in the mobile communication industry. Compared to its original static description, this interactive D3.js graph provides a more comprehensible illustration of the relationship between the manufactures to the readers. Nevertheless this example is a basic force-directed graph, it is still valuable for our research since it shares the common features (i.e. nodes and links) with the SSH state machine. Based on this graph and a number of additional D3.js functionality, we eventually visualized the obtained SSH state machine and extended it a useful SSH debugging tool. The design of this development is explained in section 4.2.1.

Figure 2.12: An example of a force-directed graph visualized by d3js, it shows the patent-related suits between mobile device/component manufactures. The graph is composed by links and nodes. The nodes represent the mobile manufactures. The dashed links are resolved suits and the green links are licensing. The distance between nodes is maintained by the pre-configured force. Users are allowed to interact with this graph by dragging nodes. The layout will be reconstructed under the force between nodes after each time dragging.

https://blogs.thomsonreuters.com/answerson/mobile-patent-suits-graphic-day/
We investigate the available SSH fuzzing methodologies and three fuzzing tools are deployed to fuzz the SSH implementations. In this chapter, we use three sections to introduce the experimental setup and the results, respectively. At the end of this chapter, we compare and discuss these tools in five aspects: code quantity, fuzzing method, performance, extensibility and development status. The comparison result turns out that protocol state fuzzing is the most suitable approach for SSH implementations among all.

3.1. Backfuzz

The first analyzed tool is Backfuzz, an open source fuzzing tool designed for different protocols. As earlier mentioned, this fuzzing tool is a simple Python script containing predefined functions for several protocols including FTP, HTTP and SSH, etc. Testers who wants to fuzz other protocols need to write custom plugins. Moreover, it also provides a few optional input patterns written by the author (see Figure 3.1).

![Backfuzz input pattern example.](image-url)
We first configure Backfuzz then use it to fuzz the OpenSSH server. Figure 3.2 and Figure 3.3 illustrate the fuzzing results of this experiment. The red block in Figure 3.2 indicates the execution fuzzing command. The arguments in the command shows that it is going to the fuzz SSH server running at port 22 on local host with the package length from 100 to 1000 bytes, and that the SSH plugin and the CyclicExtended pattern are deployed. The yellow block is a test case log information example showing the actual package length. If we check the server side, a log can be seen as showed in Figure 3.3. The blue block presents the execution command as earlier explained. In the green block, the server (sshd) says ”Bad protocol version identification” indicating that the message can not be recognized as protocol version information due to our random message. This happens because when the client initiates a connection with SSH server, it should always first send the client version information to the server. After a few attempts by the client, the server finally is closing the session as showed in the orange block. This can be explained as the server will disconnect if the client does not try to authenticate in a certain period of time, as documented in the FreeBSD manual:

"login_grace_time  Gives the grace time for clients to authenticate themselves (default 120 seconds). If the client fails to authenticate the user within this many seconds, the server disconnects and exits. A value of zero indicates no limit." [33]

As we can see from the above experiment, when using the default setup, the Backfuzz server can only perform the dumb fuzzing without the awareness of the structure of the packets and the SUT, it fuzz the OpenSSH server by simply sending the packets containing random pattern with random length to the server. The fuzzing stops at the beginning of the Transport Layer Protocol, this only proves one thing: the server (OpenSSH in this case) will close the session if the client fails to authenticate after the timeout period.
3.2. Paramiko

Earlier we discussed that Paramiko-sshfuzz is an open source SSH fuzzing framework based on the SSHv2 protocol Python library Paramiko. There is a corpus file in its source code which contains a few basic mutation algorithms to inspire users. Users are encouraged to conduct further development based on these extensive paramiko fuzzing API. Figure 3.4 is a code snapshot showing the two examples of the provided algorithms: one is adding random integers the other one is adding random strings into the packets.

![Figure 3.3: OpenSSH server log under the fuzzing from Backfuzz server](image)

![Figure 3.4: Code snapshot of the Paramiko-sshfuzz fuzzing algorithm](image)
We use three algorithms to fuzz the SSH server, add_string, add_int and add_boolean. Figure 3.5 and Figure 3.6 presents the logs of the fuzzer and the SSH server during fuzzing. It can be seen in the last few lines in Figure 3.6 that the server shows “Connect reset by peer”. This is a fatal error raised by the server indicating that the connection was closed due to the successive unusual messages sent by the fuzzer.

Figure 3.5: Paramiko-sshfuzz fuzzer log

Figure 3.6: Paramiko server under attack

As an extensive SSH fuzzing API, Paramiko-sshfuzz has been better developed compared to Backfuzz despite both approaches only support dumb fuzzing by randomly mutating the inputs. The fuzzing that can be performed with Paramiko-sshfuzz is also limited at the beginning of the Transport Layer Protocol which proves that the SSH session will be closed by the server after receiving unusual messages in the
transport layer.

### 3.3. Protocol State Fuzzing

Compared to Backfuzz and Paramiko-sshfuzz, the prototype of protocol state fuzzing is very complicated. In paper [6], Fiterău-Broștean described how to apply this approach on three SSH implementations to infer the SSH state machine models. With the guidance from the author, we replicated the state machine learning process mentioned in the paper and applied it on our local OpenSSH server.

#### 3.3.1. Experimental Setup

As earlier mentioned, when performing an active learning process by using L* algorithm to infer a state machine, two entities are necessary: the Learner who learns the state machine from a program with the guidance of the algorithms and the Teacher (SUT). To fuzz SSH server implementations, we use a L* algorithm implemented by Learnlib, which can carry out active automata learning to extract Mealy machine from the Teacher. This is achieved by sending inputs and observing the outputs. The Teacher (SUT) is the one who knows the automata but only replies when it receives a request. The local OpenSSH server plays the role of the Teacher in our setting. This black-box testing approach can fuzz any type of SSH server implementation.

However in practice two components are not enough to fuzz SSH with protocol state fuzzing. The messages sent out by the Learner are abstract inputs to represent concrete SSH messages instead of the actual SSH messages, these cannot be recognized by the OpenSSH server. To solve this problem, a transducer, called the Mapper, is used to bidirectionally map the abstract messages and well-formed SSH messages between the Learner and the Teacher. The Mapper is developed from Python-written Paramiko, since it provides mechanisms for SSHv2 protocol functionality and is compatible with SSH implementations in general[1].

The Mapper works as follows: when the Mapper is initiated, it will first establish a connection with the SSH server via a socket. Meanwhile, a loop is created listening at port 8000, it waits for the connection and input messages from the client (the Learner). Figure 3.7 explains the internal design of the Mapper by giving an example of the work flow of the UA_PK_OK message (User authentication with the valid public key). There are three important files in the Mapper: mapper.py, messages.py and transport.py. The first file, mapper.py contains the main program and it is responsible for handling all the communications with the Learner. The earlier mentioned bidirectionally mappings are in charged by the two mapping functions in the second file messages.py. The mapping function MSG_MAPPING maps the abstract message from the Learner to the corresponding functions which are defined in the third file, transport.py. The handlers in transport.py will handle the

[1] Paramiko can work with other implementations such as Cisco, but it only fully support standard OpenSSH implementations. [34]
communications with the SUT by sending or receiving the standard SSH messages. Another mapping function MSG_NAMES will interpret the concrete reply from the SUT into the abstract message which is understandable for the Learner. However, if there is no reply from the SUT over a certain time, the Mapper will dispatch NO_RESPONSE message to the Learner indicating timeout.

Figure 3.7: The work flow of the UA_PK_OK message in the protocol state fuzzing. The content in the dashed line block is an overview of the internal structure of the Mapper. The green arrows imply that the message is transmitting in the direction of the Learner to the SUT. The blue flows is showing the opposite.

One advantage of using a Mapper is that the prototype can be used to fuzz any SSH server implementation (Written in any language), since the only entity communicating with the SUT is the Mapper. The Mapper only sends and receives RFC standardized SSH messages, acting as a SSH client communicating with a SSH server. However, as SSH is a complex client-server protocol, it would be a heavy work to achieve all the SSH client functions available in regular SSH clients. Therefore, we restricted the behavior of the Mapper to only explore the terminal service in the connection layer, as it is considered to be interesting from a security perspective [4]. The other functions, such as algorithms for encryption, compression and hashing are set as the default settings.

Moreover in this experiment we define an SSH message as a query, a trace is composed by one or multiple queries. To infer a state machine from a SUT, multiple traces are required to sent to the Mapper by the Learner, which is guided by the L* algorithm. Each trace always starts with the query “RESET SUT” to terminate current connection with SUT and initiate a new one. This execution guarantees the SUT always starts working from the initial state. The Learner possesses an input
3.3. Protocol State Fuzzing

alphabet containing a set of queries to be used. It sends all possible combinations of queries as traces to test the SSH server. The resulting output is sent back from the SUT and is used to infer states and decide which subsequent input symbols to send. It is notable that increasing the size of the input alphabet will cause rapid growth of learning time due to the increasing number of operations. This is because the number of membership queries grows with the increase in alphabet size. In such a case, two types of input alphabets are defined, full alphabet [4] and restricted alphabet (See Appendix A). The restricted alphabet preserves the most interesting states for us and excludes inputs that do not cause state transitions, or inputs that proved costly time-wise (i.e. DISCONNECT).

3.3.2. Result

We implemented the setup we described in previous section on OpenSSH-6.7 for a state machine model. All the related code, files and results, have been uploaded to GitHub repository\(^2\). The restricted input alphabet was used, its time consumption was significantly less than for the full input alphabet. The experiment took 52312759ms, 3757 membership queries had been performed to obtain the final state machine with 23 states. No obvious violations of RFC standards are spotted.

In addition, we compared our result with the result presented in paper [4]. The states and transitions involved into the happy flows are identical. There are few differences exist due to the implementations of different versions (OpenSSH 6.9p1-2 in paper [4]) and the different definitions of the input alphabet. One example of the difference is, the states in the state machine from the paper contain more self-loops. This is caused by the deployment of the full input alphabet. These extra transitions indicate that these extra inputs requested more services from the SSH server. However, this does not effect our conclusion because of the two reasons: firstly, there is no suspicious transitions observed caused by these extra inputs since their behaviors can be explained by RFCs. Secondly, the extra transitions has no influence on the normal execution of the happy flows.

Figure 3.8 shows a refined model presented in paper [4]. The result generated by the Learner does not directly show as a graph, instead it is saved as a DOT (A plain text graph description language) format. To improve the readability, a graph visualization software, Graphviz is needed to visualize the result. However, Graphviz has more advantages of visualizing simple state machine, the high complexity of SSH protocol results in a messy and overlapped graph with low readability and comprehensibility even after processing with Graphviz. This leads to the readability problem which brings inconvenience for the researchers who would like to work on SSH models. Therefore we decide to extend the current protocol state fuzzing prototype to a tool which can solve the this problem and improve the testers’ comprehension of the SSH implementations at the same time. The development of this tool will be introduced in the next chapter.

\(^2\)https://github.com/yuzhuY/sshfuzzing
Fuzzing SSH server implementation

### Figure 3.8: A refined model learned from OpenSSH 6.9p1-2. The sequences of messages expected by the server and resulting in a satisfying result for specific layer, defined as happy flow, are highlighted in green. The states are categorized into three clusters, each one represents a layer in SSH protocol. An input together with corresponding output labels a transition. Each transition is triggered by the input on the left of “/” and receives an output showing on the right. Some transition labels are shortened by regular expressions, for example, SR_* indicates all the inputs starting with SR_, multiple inputs in a transition like KEXINIT, SR.*, UNMPL, DEBUG, IGNORE, KEX30/KEXINIT stand for any of these inputs producing the same output KEXINIT.

### 3.4. Discussion
Here we give the comparison of three fuzzing tools, Backfuzz, Paramiko-sshfuzz and Protocol state fuzzing. We compare them through five aspects:

- **Code quantity** discuss about the amount of the code of each fuzzing tool.

**Backfuzz.** Backfuzz has the least quantity out of the three approaches. The source code provides the interface written in Python scripts to connect with the protocols and inject mutation-based inputs. Besides this, there are a few example plugins provided to fuzz the protocols such as ftp, http, imap and ssh.

**Paramiko-sshfuzz.** In the aforementioned experiment only a few code were executed since we only performed a basic dumb fuzzing. However, we need to realize that Paramiko-sshfuzz is not isolate, it is based on the Paramiko library which supports all the functionality of SSHv2. This means Paramiko-sshfuzz can hook into all kinds of SSH messages in the library, if necessary.
Protocol state fuzzing. This fuzzing tool has the largest code quantity compared with the other two approaches and the internal design is relatively complicated as well. It is composed by three components, Java written Learnlib, Paramiko and SUT. Among all, the protocol state fuzzing is the most difficult to understand since it is a novel approach on fuzzing.

- **Fuzzing method.** All of these three tools are using brute-force dumb fuzzing, but the methods of the input generation are different. In the previous experiments, Backfuzz and Paramiko-sshfuzz randomly mutated inputs without the awareness of the structure of SSH message. Protocol state fuzzing defined an input alphabet representing the standard SSH messages in the happy flows, it injected the sequences of messages by sending all the combinations of the elements from the input alphabet.

- **Performance.** Backfuzz and Paramiko-sshfuzz fuzzed the Transport Layer protocol and their exploration stopped at the beginning the of this layer. Users can analyze the fuzzing status by checking the log of the fuzzer (SSH client). The fuzzing result proves that the transport layer works as expected that the SSH session will be closed by the server under the attack (Receiving unusual messages). On the other hand, the protocol state fuzzing automatically fuzzed the three layers of the protocol and generated an intuitive SSH state machine as the result. Users can analyze the fuzzing result by examining the obtained SSH state machine for the spurious behaviour.

- **Extensibility.** These three approaches are all extensible. Since Backfuzz and Paramiko-sshfuzz are both at the early stage, users have more space to explore them according to their needs. The design of Protocol state fuzzing is more suitable for analyzing the SSH protocol with state machine learning approach. Paramiko-sshfuzz and Protocol state fuzzing both rely on the SSH protocol library Paramiko, therefore they do not need much efforts to construct messages with SSH features, they can simply hook into the Paramiko library and use the functions.

- **Development status.** These three tools are all not well documented. Protocol state fuzzing is slightly better than the other two since there is a paper [4] and a few theses [6] described the design of this technique. As a state-of-the-art technique, it is not only used to fuzz the SSH protocol but also actively implemented for other security protocols such as TLS. Backfuzz was brought up at 2012 and no maintenance afterwards. Paramiko-sshfuzz started in 2008 with the most recent update in 2015.

Our research of the SSH fuzzing tools indicate that the development in this area is not active. Backfuzz and Paramiko-sshfuzz both are still in an early stage, not
mature enough to be used into the to market as a fuzzing tool. The automated and systematic fuzzing technique protocol state fuzzing turned to be the most suitable approach for SSH fuzzing.
Although we obtained the SSH state machine by implementing the protocol state fuzzing technique on OpenSSH6.7-p1, there is still one thing that needs to be discussed: the readability problem of the SSH state machine caused by the complex design of the protocol. To solve this, we design an interactive interface using the data visualization library D3.js. Besides visualizing the SSH state machine on the interface, we also inspect the source code of OpenSSH6.7-p1 and display the source code information on the interface. The source code inspection is achieved through two code review approaches, static analysis and dynamic analysis. In the final design of the interface, users are only allowed to interact with the interface in general ways such as dragging, zooming and panning, etc, but also can check the executed code (functions) of the transitions by clicking them. We hope this extra development together with the protocol state fuzzing technique can be used as a debugging methodology for SSH implementations. Since state machine gets involved into the whole research, we name this methodology as the SSH state machine based debugging tool.

This chapter is divided into two parts: the first section is about the code review, where we introduce the two approaches to inspect the source code of OpenSSH6.7-p1. The second section describes the design of the interface (D3.js implementation).

4.1. Code review

In this section, we inspect the source code of OpenSSH6.7-p1 since we want to link the source code with the transitions on the state machine and display this information on the interface to users. There will bring two benefits: firstly, testers can analyze the interesting behaviors showing on the state machine in the perspective of the source code (This is performed on the state machine of OpenSSH6.7-p1 and illustrated in section 5.2). Secondly, in the testing phase especially for the large
programs, if the tester have the executed code of the transitions in mind, he can fast locate the bug in the source code which resulted in the spurious transition. We will explain this usage by showing test cases in section 5.1 and the usefulness of it has been confirmed by the professionals from the survey we conducted in section 5.4.

To link the source code with transitions, we perform the code review on the source code of OpenSSH6.7-p1. Code review is a common software engineering mechanism to find mistakes, improve overall quality and reduce the risk of bugs during the initial development phase. However, performing code review on large programs is never easy. There are two approaches, static analysis and dynamic analysis, depending on whether the program is executing during the analysis. The static analysis approach is using manual inspection, which is general but very time-consuming. Testers are not only required to understand the program but also need to know what the test cases look like before the rigorous inspection [35]. Nevertheless, testers are still prone to have a subjective conclusion of programs. In contrast, tool-based static analysis is much more efficient. The tools will look for a fixed set of patterns, or rules in the code. However human evaluation of tool’s output is still needed.

4.1.1. Static analysis

Static code analysis is a white-box testing approach for computer software program debugging, achieved by analyzing code without executing the program. It is always carried out as a part of a Code Review. In this thesis, a tool-based static analysis is performed. The tool used is Understand, and integrated development environment (IDE) to analyze the source code of OpenSSH6.7-p1. It is a proprietary software used to comprehend and maintain code with visualizations and metrics. Users can visualize the architecture of the source code to try and optimize software design. Figure 4.1 is an example of a butterfly diagram created by Understand. This diagram displays all the calls made by the function sshd_exchange_identification. This function is previously called by main() in the file sshd.c from OpenSSH6.7-p1.

The files of the source code of OpenSSH6.7-p1 can be classified into two categories: files supporting the SSH client and files supporting the SSH daemon. We only analyzed SSH daemon files since they implement the SSH server functionality. Some files can be executed on their own without the need of running the SSH daemon files, for instance ssh-keygen.c file a script for key generation and maintenance. The main program of the SSH daemon is written in sshd.c file. When the SSH daemon receives a connection request from the Client, it will first check if all the information contained in the request is valid. If so, the main program will call the server loop function. The key exchange process and all on-wards actions will be executed after the server loop is launched.
4.1. Code review

Figure 4.1: Butterfly diagram of SSH exchange identification function in sshd.c file. Created by Understand. The enlarged figure is in Appendix B. The source of the arrow is the parent function, the arrow points to the the child functions. The function sshd_exchange_identification is shown in a red rectangle. This indicates that all the components in this diagram are relevant to it, allowing us to see the relationship between sshd_exchange_identification and all the other functions. Users can click any rectangle to further investigate.

4.1.2. Dynamic Analysis

States and transitions are not independent in the learned SSH state machine. A state can be reached after executing a sequence of transitions, named an execution trace. A transition from one state to another is caused by executing one or several functions from one or multiple files. The total number of files in SSH source code is nearly 1000. This implies code review can be very complicated even for one transition, not to mention reviewing an execution trace containing several transitions. Only using static analysis and human comprehension to review the code is not enough. Hence, another code review methodology, dynamic code analysis is introduced to work together with static analysis, we hope this can make the code review process more efficient and accurate.

Dynamic code analysis is a method to analyze computer software while being executed on a real or virtual processor. One dynamic code analysis tool we tried on the SSH server was GDB (The GNU Debugger). Testers can set breakpoints in the program to specify the place where the program should stop. The SSH server itself also has a debugging mode. It is possible to view the log of all the activities happening on the server side in this mode. To perform dynamic analysis, we first identify execution traces as test inputs, based on the interesting states. Then, following the test inputs, we communicate with the OpenSSH server via Telnet. The log information of the communication is analyzed and used to confirm if the suspected
functions are executed. It is noted that the OpenSSH server we tested is not the pre-installed one on Linux. It was downloaded from OpenSSH Portable Release\(^1\). After compiled, it run as an independent OpenSSH local server.

4.2. State Machine Visualization by D3.js

In this section, we explain the design of the interface which visualized the SSH state machine by implementing D3.js. The final design is named as Static Interface to differentiate it from our initial design called Dynamic Interface. Although our first attempt did not produce satisfactory results, we present the encountered problems at the end of this section as a possible future work.

4.2.1. Static Interface

The design of the static interface based on the forced-directed graph\(^2\) from D3.js library. Additionally, the following functions are added to adapt to the complex SSH state machine:

- **Dragable nodes.** Dragable nodes enable users to freely reorganize the structure of state machine.
- **Sticky nodes.** Nodes can stick to a specific location on the canvas where the mouse is released.
- **Self-loop.** SSH state machine contains massive self-loops.
- **Zoom and pan.** Easier to see details of state machine.
- **Double click to zoom**
- **Multi-links between two nodes.**
- **Nodes and links labels.**
- **Clickable link label.** By clicking the label, the information from the JSON file is rendered onto the Source code Overview block which is on the left on the interface.
- **Properties determining transition colors and direction of the arrows.**
- **Load graph from JSON file.**
- **Embed D3.js project into HTML**
- **The source code information is clickable, redirect user to a file in source code**

\(^1\)OpenSSH Portable Release https://www.openssh.com/portable.html
\(^2\)Forced-directed Graph https://bl.ocks.org/mbostock/4062045
4.2. State Machine Visualization by D3.js

Figure 4.2: A static interface designed to visualize the learned SSH state machine. On the right, the state machine is visualized based on the forced-directed graph, composed by the interactive elements, links and nodes. Users can freely manipulate these elements, i.e. dragging, zooming, panning, which are useful to inspect details, especially for complex programs. When the transitions, represented as links, are clicked the functions executed in this transition will be displayed in the “Source Code Overview” rectangle on the left. The files which those executed functions belong to are shown as link texts in the rectangle. Users can click the file name to visit the source code file saved locally. At the bottom of the interface, there is a link text called "Open". It redirects to a full-screen state machine.

4.2.2. Discussion: Dynamic Interface

Our initial idea was to integrate the aforementioned protocol state fuzzing framework (backend) with D3.js interface (frontend) to develop an interface which can dynamically show the generation of state machine. We expect to see the sequence of states and transition followed by the the learning process through the interface. We name this as a dynamic interface. This achievement will help people who are working on SSH protocol to have an intuitive understanding of the program logic and state machine learning process. However, our first attempt in this direction did not produce satisfactory results, it will be explained below. Hence, we decided to develop a static interface with sufficient interactions to let people have a good understanding. We also expect this development can be used as a debugging tool for the development of SSH implementations. In this section, we will explain our work and present the final design of the interface.

When using protocol state fuzzing framework to learn a state machine from a protocol, users can check the communication between the Learner and the SUT. However, it is invisible for users, they can not see how the state machine is being built under the guidance of L* algorithm. Only when the learning process is terminated, the learned state machine is generated as a result to the users. Hence, we proposed the following design shown in Figure 4.3 to dynamically display the learning process to them:

To make the prototype in Figure 4.3 fully automated, two issues should be dis-
Figure 4.3: Integration of Protocol State Fuzzing framework and D3.js. The components used to perform protocol state fuzzing are highlighted by a blue rectangle (Backend). The extension, presented in the green dashed-line rectangle (Frontend), dynamically visualizes the learned state machine on D3.js interface. The state machine, generated by MAT framework, is written by DOT text and it is the input for D3.js implementation. A python script is used to transform the DOT file to a fixed JSON format. The interface is written in a html file, which loads the external JSON file and renders it on the browser.

cussed:

- **How to find and identify the latest version of the SSH state machine file produced?**

- **How can we synchronize the data of the MAT framework with the D3.js implementation?**

We came up with a hypothesis to solve the first issue: “The final learned result is saved in a DOT file. So it is possible to see the progressing state machine in the DOT file during the learning process.” This would benefit us since the real-time data in the DOT file is used as the input of the D3.js implementation. To verify our hypothesis, we conducted the learning process again. However the DOT file was created at the end of the learning process, shown as a complete state machine instead of showing a progression. We were mislead to this hypothesis because there is another log file, out.txt, which is updated in real time. This file only records the traces sent by the Learner and the replies but not the state machine.

To answer the second question, we have to assume that the first issue can be solved. Under this assumption, the frontend deploys a python script to fetch real-time output data from the backend and transforms the data into JSON. The frontend repeats this operation whenever the external data updates.

This raises another issue, how to render the dynamic JSON data on the browser. To our knowledge, D3.js library does not provide the API to achieve this. Every time the JSON data is updated, the frontend has to be refreshed to reload the external JSON file and render it on the browser.
The above mention problems, stopped our investigation on the development of a dynamic interface. Thus, with the same aims, we shift our focus on developing a static interface.
5

Result and Analysis

5.1. Anomaly Detection
The SSH state machine based development can be used as a useful debugging tool for the study of the SSH protocol. It helps testers locate errors quickly and speedup the software development time. When an unexpected behaviour is observed as an abnormal transition or state, testers can track the relevant code of any interesting execution traces. This helps testers find bugs of the program quicker hence saving time on investigating irrelevant code. It will shorten the time of testing phase during the SSH program development. To prove this in this section, we are going to show a test case, which could appear in the actual testing phase. We firstly modify the source code to interfere with the SSH server behavior, then use the tool we developed to locate the anomaly in the source code. Finally, we explain the challenge which need to be solved by showing an example.

5.1.1. Test Case
This test scenario uses our developed tool to examine the work flow of the Transport Layer Protocol since we believe that the state machine reveals the internal design of a program. The flaws of a program should be seen as anomalous behavior when the program is running. As seen in the introduction of the second chapter the normal execution trace of the Transport Layer Protocol is supposed to follow this happy flow (See Figure 2.3) defined by RFC standard. Hence, the first step of our plan is to “create” a flaw by modifying the source code. Then using the tool developed by us, we will collect the actual execution trace and compare it with the happy flow. We expect the actual execution trace to be different from the happy flow due to the modification, and the inspection of the source code for the anomaly behavior should be faster than manual checks.

Here we give the code segment 5.1 which we are going to perform the experiment
on. It is taken from the file, auth2.c of OpenSSH6.7-p1, which is only used to support the SSH daemon. This function will be triggered when the server side receives a message for requesting a service. Using the code in the 5th line, the server can analyze the type of the received message.

Listing 5.1: do_authentication2 function

```c
void do_authentication2(Authctxt *authctxt)
{
    dispatch_init(&dispatch_protocol_error);
    dispatch_set(SSH2_MSG_SERVICE_REQUEST, &input_service_request);
    dispatch_run(DISPATCH_BLOCK, &authctxt->success, authctxt);
}
```

The 5th line of the above code works by calling the input_service_request function (Shown below). It can be seen that only when the message is a request for authentication the server will accept it. Subsequently, the server replies “SSH2_MSG_USERAUTH_REQUEST” message back to the client. Any request for other services will be denied by the server and it replies “bad service request”.

Listing 5.2: input_service_request function

```c
static void input_service_request(int type, u_int32_t seq, void *ctxt)
{
    Authctxt *authctxt = ctxt;
    u_int len;
    int acceptit = 0;
    char *service = packet_get_cstring(&len);
    packet_check_eom();

    if (authctxt == NULL)
        fatal("input_service_request: no authctxt");

    if (strcmp(service, "ssh-userauth") == 0) {
        if (!authctxt->success) {
            acceptit = 1;
            dispatch_set(
                SSH2_MSG_USERAUTH_REQUEST, &input_userauth_request);
        }
    }
}
```
We replace the 5th line of 5.1 with the following code. The function `input_userauth_request` for processing user authentication request will be deployed instead of `input_service_request`. Thus, the server will process any request as the way of processing an user authentication request.

```
if (acceptit) {
    packet_start(SSH2_MSG_SERVICE_ACCEPT);
    packet_put_cstring(service);
    packet_send();
    packet_write_wait();
} else {
    debug("bad service request %s", service);
    packet_disconnect("bad service request %s", service);
}
free(service);
```

After compiling and running the modified SSH server program, the new work flow of the Transport Layer Protocol is shown below (Visualized by D3.js) in Figure 5.1a

![D3.js visualization of the new work flow](image)

Figure 5.1: (a) The new work flow of the Transport Layer Protocol after modifying the source code. Following the sequence of the inputs of the original happy flow, the outputs are the same as for the happy flow except the last transition SR_AUTH/NO_CONN (zoomed in the red block). (b) The interface displays the executed function during transition SR_AUTH/SR_ACCEPT.

As we expected, an abnormal transition occurred, leading to the state change from kexed to Disconnect. This implies the client’s request for authentication is rejected by the server since it is not recognizable. According to the code segment 5.2, knowing the type of the message is necessary before the server reaches a state where it is ready to accept the authentication request message. However, this step is skipped because of the absence of the `input_service_request` function. In this case, the server sends back a disconnection message creating a transition to a disconnected state.
Presumably during the development of an SSH implementation, the aforementioned abnormal transition is detected by using our tool. Testers may want to know the code that caused this anomaly. Through interacting with the interface, the tester will learn that the executed function during the transition is do_authentication2() (Figure 5.1b). Therefore the tester can inspect the details of this function and find out the error.

5.1.2. Challenges

The test case example shows how our tool provides an intuitive approach on how to handle anomalies in the development of SSH implementations. However, locating errors in the actual development can be very hard due to the high complexity of the SSH code structure. The following example is a case when our tool misplaces an error under a specific situation. It will show that improving the accuracy of tool is very challenging.

Instead of modifying the 5th line in the code 5.1 as in the previous example, we substituted “failures” for “success” in the 6th line. The updated code is shown as:

Listing 5.3: Updated do_authentication2 function

```c
void do_authentication2(Authctxt *authctxt)
{
    dispatch_init(&dispatch_protocol_error);
    dispatch_set(SSH2_MSG_SERVICE_REQUEST, &input_service_request);
    dispatch_run(DISPATCH_BLOCK, &authctxt->failures , authctxt);
}
```

Repeating the same steps, we obtain the new work flow showed in Figure 5.2a. The transitions in green represent the happy flows of the Transport Layer Protocol and the ones in yellow represent the Authentication Protocol. The last transition, CH_OPEN/UNIMPL violates the happy flow of the Connection Protocol, since CH_OPEN_SUCCESS is expected as the output when CH_OPEN is used as an input. To check the executed functions of transition CH_OPEN/CH_OPEN_SUCCESS, we click the transition on the interface, it displays do_authenticated() (See Figure 5.2b), session.c file. Evidently, this result is not what we expect.

We can see that the information provided by our debugging tool can be very misleading in this case. It is no doubt that the function containing the error is do_authenticated() which is triggered by input of SR_AUTH. However, the SSH server then produces the expected outputs from an unmodified OpenSSH implementation. No abnormal replies show up until the server receives a request (CH_OPEN) for opening a channel. The message, UNIMPL is dispatched from the server, implying that the opening channel request cannot be implemented. From the perspective of the
5.1. Anomaly Detection

Figure 5.2: (a) The updated work flow of the SSH protocol. The transitions following happy flows are colored into green and yellow. (b) The executed function during transition CH_OPEN/CH_OPEN_SUCCESS tester, all of these behaviors may result in the misjudge that the error is caused by function do_authenticated(). Finding out where the error actually is becomes extra hard.

To figure out the reason behind this, we inspected the source code and analyzed the logic of all the involved functions of the sequence, SR_AUTH/SR_ACCEPT → UA_PK_OK/UA_SUCCESS → CH_OPEN/UNIMPL. A logic structure is summarized in Figure 5.3.

As we can seen from the diagram, the sshd.c file, contains the main program for the daemon. The two functions mentioned above, do_authentication2() and do_authenticated() are included in this main program. All the arrows are labeled with numbers for a better understanding. There are two logic flows drawn in different colors. The green work flow indicates the working logic of the do_authentication2() function. Similarly, the blue one represents do_authenticated(). Following the green flow, it can be seen that a loop is created when the do_authentication2() function is called. It can only be terminated when the argument authctxt is set as “success” (This condition becomes “failures” after our modification). The arrow 1, 2 and 3 are created by the request SR_AUTH, arrow 4 and 5 corresponds to the request UA_PK_OK. Logically, the code modification has not affected the arrows from 1 to 5. Only the arrow 6 will not execute since the value of authctxt is “success”. This does not meet the condition for closing the loop, causing the main function to halt and wait indefinitely. The server replies UNIMPL since it cannot process the request for opening the channel without calling do_authenticated(authctxt).

This is an example of the several difficulties we faced during the development. Another main concern is how to represent the functions on the interface to speed up the debugging efficiency to the utmost extent. Still taking Figure 5.3 as an example, we can see that one or multiple functions correspond to one or several transitions. If we classify functions into levels, the functions in the main program are named as root level functions. Those in files like auth2.c and session.c are the first level and so forth. The serverloop.c file contains the second level functions.
5. Result and Analysis

Figure 5.3: Logic flows of do_authentication2 and do_authenticated function. The four files from the source code of OpenSSH6.7-p1 are written in pseudo code. The arrows point at the functions to be called. Numbers and colors are used to distinguish relationships between functions. The green lines are all related to the logical flow of the function do_authentication2. The channel opening process is showed in blue. The origin of this flow is do_authenticated.

Presently, our interface only displays functions from the root and the first levels since our development is still at the early stages. Presenting the functions from deeper levels on the interface will be time consuming for the developer of the tool, however testers could benefit from them. This trade-off needs to be discussed in the future development.

5.2. Passive Learning using DFASAT

In section 2.8, we had a discussion about the passive learning and the novel approach DFASAT which is used to solve the DFA identification problem. In this case, we intend to perform an investigation on the SSH implementation by deploying DFASAT as well as passive learning.

The steps we followed are similar to the work flow in Figure 2.10. Firstly, we collected the execution traces. This is derived from running protocol state fuzzing
5.3. Limitation of the Mapper

The study on the state-merging algorithms inspired us to take it into consideration when analyzing the SSH state machine. Two sequences of transitions grasped our attention. Since the original state machine is very large, we extract them out of the model and show them in Figure 5.4.

As we can see, the single sequence transition is separated into two after s5. The sequence highlighted in green is the happy flow. The other sequence, colored in blue, branched out from the happy flow due to the transition CH_OPEN/UNIMPL. The identical transitions in these two sequences are displayed in the dashed-red rectangles.

Although the incoming and the outgoing of state s6 is identical to s10, these two sequences are still not compliant with the standards of state-merging. As the following sequences of them are different, one is followed by transition CH_OPEN/CH_OPEN_SUCCESS, another one is redirected to a disconnected state. It is interesting to see, after two equivalent transitions, both s9 and s13 receive the same input message (CH_OPEN), but the corresponding outputs are different. S13 obtains CH_MAX instead of CH_OPEN_SUCCESS, indicating there is already one channel
opened. Paper [4] describe that there is a buffer with the size of one, for storing opened channel in the Mapper. The channel will be removed from the buffer if the Mapper receives a CH_OPEN message. To prove this, we reviewed the transport.py file in the Mapper. It shows this is actually implemented, thus we arrived at the hypothesis that after the transition CH_OPEN/UNIMPL one channel is opened and the buffer is up to the maximum. There is no CH_CLOSE received before the Mapper sends another CH_OPEN. The Mapper cannot support another channel. Hence, it has to reply a CH_MAX to the Learner. It is worth to mention that there is no CH_OPEN message received before s5, this implies the buffer is initially empty.

The above analysis indicates the insufficient design of the Mapper. The buffer in the Mapper only allows one channel to be opened during the SSH session. This violates the RFC standard since RFC 4254 [36] defines that multiple channels are able to open simultaneously and multiplexed into a single connection. To simulate the actual SSH communication, this should be improved in the future protocol state fuzzing.

5.4. Security evaluators’ Perceptions on the tool

In order to get professional opinions about the effectiveness and practicability of our SSH state machine based debugging tool, we performed a survey. This was conducted on the employees from Brightsight. In this section, we first describe the procedures of the survey, then discuss the threats to validity and give a conclusion in the end.

5.4.1. Survey conducted at Brightsight

In this survey, there were 9 participants involved. Their positions are: 1 technical project manager and 8 security evaluators. There are 2 evaluators specialized in open protocols, 4 in Embedded systems, 1 for software and 1 for hardware.

Each survey takes 15-20 minutes, it is composed by three sections:

- **Background knowledge introduction**: Since the participants have a diverse background it is necessary to introduce the knowledge behind the tool. This will make the usage of the tool more comprehensible.

- **Instructions of using the tool**: We perform a demonstration for participants by executing a test case on the debugging tool. The participants are encouraged to operate the tool and find the bug.

- **Questionnaire**: After the demonstration, the participants are asked to answer a questionnaire containing 4 multiple choices questions, 1 ranking question and 1 discussion/suggestion question.

The first two multiple choice questions of the questionnaire aim to compare the comprehension of the two visualization approaches. The participants are asked questions of “How easy it is to understand the SSH state machine visualized by
5.4. Security evaluators’ Perceptions on the tool

Graphviz/D3.js?”, they can answer the questions by a number from a range of 1 to 5, 1 means very easy and 5 means very hard. The following table 5.1 illustrates the results of the questions.

<table>
<thead>
<tr>
<th></th>
<th>Graphviz</th>
<th>D3.js</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very Easy</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Easy</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Neutral</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Hard</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Very Hard</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1: The comparison of two visualization methods, Graphviz and D3.js. It bases on the results of the survey.

As we can see that participants generally think visualizing the SSH state machine by D3.js is easier for them to understand. One evaluator who choose Neutral for both Graphviz and D3.js explains that (s)he thinks the preference of using these two methods should depend on the needs and the specific situation.

The third question is “Do you think displaying the corresponding executed functions of the transition on the interface is helpful for the debugging?”. Similar to the answers of the previous questions, five options are provided, from “Very Helpful” to “Not At All”. The feedback from the participants are very positive, we received 5 “Very Helpful” and 4 “Helpful”.

The last question is “Do you think this testing approach can really help the development of the SSH implementation?”. Three people voted for “Definitely helpful”, four people think “It will help in many situations”. One person replied “I don’t know” since he does not have a concrete knowledge of the SSH protocol.

In the second part of the survey, the participants are asked to rank the following situations where they think the debugging tool can be used, from the most to the least likely. The ranking questions are referenced from paper [37].

1. To have a high level understanding of the program
2. To investigate a specific part of the program
3. To understand the behavior of the program
4. To find unintended behavior of the program, i.e. bugs
5. To share the understanding of the program with other testers
6. To be used as part of the documentation of the program
7. To compare different versions of the program
8. To find violations against the standards of the program

We collected the answers and summarized them by assigning weights to each situation. The most likely situation is assigned with a value of 1, the least likely one is given an 8. The results are illustrated in the matrix in Figure 5.5. From this matrix, we can see that the situation “To have a high level understanding of the program” receives most votes.

![Figure 5.5: The matrix of the results of the ranking question. The x axis is using the numbers to represent questions. The y axis displays the position of participant at Brightsight. The numbers in the matrix are the order of preference for each situation.](image)

In the end, we asked the advice from the participants. Their suggestions are mainly about the display of the interface and the design of input alphabets. We summarized all the feedback as follows:

- Learn the state machine of the SSH server under other situations, i.e. sending packets messages with variant payloads.
- Add new functions on the D3.js implementation, i.e. highlighting mouseover parts. Write a new D3.js library which is specialized for the SSH protocol for a better visualization effect. The current functions like dragging & dropping on the interface are not very ideal for the complex protocols.
- Add a configure file in the D3.js implementation. In this file, users are able to customize the display of the executed functions.

5.4.2. Threats to Validity

This section we give a discussion of the threats to the validity of this study and how we minimize them.
5.4. Security evaluators’ Perceptions on the tool

Internal validity concerns the only explanation of the results. The threats to the internal validity means the variable (independent variable) we did not consider which may effect the results.

1. Diverse background knowledge of the respondents. This research topic is regarding to the specific fields which might out of the bounds of the respondents’ background knowledge. For example, one respondent implies that he has completely no experience with the state machine or the SSH protocol. To mitigate the impact of this, there is always an introduction about the involved knowledge at the start of the survey.

2. Observer bias (Participant-expectancy effect). The expertise of the participants is varied, they range from project manager to evaluator. This might lead the respondents to look for the information that conforms to their positions or overlook the information that argues against them during the survey. It is hard to eliminate this factor, therefore we tried to minimize this influence by asking questions under specific conditions. We tried to select the conditions that most likely happen in the actual testing phase. For example, one question about the usage of the tool is “Imagine you want to test if the SSH implementation works normally in the transport layer protocol, how would you operate the tool?”

External validity concerns the generalizability of the results

1. This SSH state machine based debugging tool was developed from the state machine of OpenSSH6.7-p1, due to the significant user base of OpenSSH. When applying this tool to other SSH implementations, the modifications to the prototype of the protocol state fuzzing and the algorithms in the D3.js implementation are needed to suit different working environments.

2. We mentioned earlier that some questions in this survey are asked under specific conditions. Therefore the results can only be generalized under similar situations.

5.4.3. Conclusion

This section depicts the survey conducted on the evaluators from Brightsight and the threats to validity about it. Based on the results, we believe that our SSH state machine based debugging tool can indeed bring benefits for the development of the SSH implementations (i.e. whether an implementation follows the RFCs). Especially for the question about displaying the code review results on the interface, we received the positive feedback from all the participants. Furthermore, the professional suggestions from the participants’ experience in the actual work should be valued in the future development.
Conclusion and Future Work

In this chapter we give a summary of our findings and provide some ideas for further research.

6.1. Conclusion

RQ1: How good are the existing fuzzing techniques for SSH implementations?

The available SSH fuzzing tools can be classified into open source and the proprietary tools (i.e. beSTORM). In this work, we used three open source SSH fuzzing tools, Backfuzz, Paramiko-sshfuzz and protocol state fuzzing on the SSH servers. We observed the logs during the fuzzing process, then analyzed and compared the performance of these tools. The fuzzing results of Backfuzz and Paramiko-sshfuzz indicate that these two approaches are still in an early stage. Compared to them, protocol state fuzzing technique is relatively mature and we think it is the most suitable methodology to fuzz SSH implementations at present.

RQ2: To what extent can the state machine learning methodologies be used to improve the understanding and testing of SSH implementations?

As a starting point in our investigation, we first obtained the state machine of OpenSSH6.7-p1 by using two automate learning methodologies (Protocol state fuzzing and Passive learning). To perform protocol state fuzzing on the OpenSSH server, we used the L* algorithm implemented framework, called MAT framework. It is composed by three components, the Learner, the Mapper and the SUT (OpenSSH implementation). The Learner infers the state machine by sending inputs to SUT and observing the outputs of it. Passive learning was conducted by first collecting the execution traces of the server, then processing the data with the passive learning tool, DFASAT. The result indicated that DFASAT is as effective as the protocol
state fuzzing since their results were identical.

After obtained the SSH state machine, we conducted code review on OpenSSH6.7-p1 and used the code review results to analyze the SSH state machine. The source code inspection was performed by using the tool-based static analysis and dynamic analysis. The former was supported by Understand, a proprietary software to comprehend and maintain code with visualizations and metrics. The latter was achieved by analyzing the logs of the server itself after turning on the SSH server debugging mode. Subsequently, we identified the interesting behaviours on the state machine and explained the programming logic behind them by analyzing the source code. By using this method, we proved the limitation of the Mapper in this protocol state fuzzing prototype. This limitation will be further discussed in the future work.

Our work proves that the state machine learning technique is difficult to learn, since it is a novel approach and the setup is complicated. However the automated learning process can significantly improve the efficiency of fuzzing, and it is a useful technique to systematically analyze SSH implementations. The fuzzing result shown as a state machine makes the logic of the protocol implementation easier for testers to understand and easier to spot the spurious behaviours as well. Moreover, the code review methodologies we implemented during the experiments also provided a practical approach to analyze the abnormal behaviors in the state machine from the perspective of the source code.

RQ3: How can we take advantages of the current protocol state fuzzing technique to obtain a useful (easy) debugging tool for SSH implementations?

To answer this question, we designed an interface to visualize the protocol state fuzzing result. Firstly, we visualized the obtained SSH state machine on a D3.js implementation. This development solved the readability problem of the complex SSH state machine as well. Secondly, we identified the executed functions corresponding to the transitions on the SSH state machine and displayed them on the interface. We used two test cases to explain the usage of this tool. Moreover, the first test case proves that it can speed up the testing phase of the development of SSH implementations. The second test case proves the limitations and challenges of the tool.

Additionally, we also conducted a survey to find out the professionals’ perception of our development. In this survey, we first introduced the background knowledge behind the tool to the participants, then demonstrated the usage of it by showing an actual test case. The participants were encouraged to interact with the tool for a concrete understanding. Furthermore, we provided a questionnaire to collect the feedback. From the results, we are glad to see that most participants think our tool is helpful for the development of the SSH implementations. They also had a positive attitude towards the implemented techniques, especially our idea about displaying the source code information on the interface.
6.2. Future work

A few other developments have been left for the future due to lack of time. Some possible future improvements are:

**Designing the dynamic interface:** Although our efforts on dynamic interface did not turn out a satisfying result, we still believe this subject should be further investigated in the future. If it can be achieved, it would benefit the SSH protocol study, development and state machine learning research. The key to actualizing this is finding a tool which can automatically track the executed functions, without testers reviewing the internal code.

**Optimizing the static interface:** There are two ways to improve the static interface. Currently, our interface is only suitable for the OpenSSH6.7-p1. Different SSH implementations may have various state machines, from simple to complex. In some situations, the interface would not render data as we expected (i.e. the state machine is too complicated, there will be missing transitions due to the insufficient of data visualization algorithms). To improve the compatibility of the interface, we suggest to optimize the algorithm written in JavaScript, or write libraries specialized for different types. The state machine data in JSON format also need to be adjusted to fit the change.

**Improving the displayed information on the interface:** Previously we explained the complexity of the source code of OpenSSH6.7-p1, i.e. one or multiple functions correspond to one or several transitions. This is also confirmed by the static analysis tool Understand. Therefore we only added the lower level functions (functions called first by the main program when a transition occurs), since it is impossible to append all the related functions or other source code information under the current design of the frontend. We hope this can be optimized in the future since the more information displayed the easier it will be for the tester to locate the bug. The future developer who intends to work on this can consider to deploy the web service APIs, i.e. REST API and database to store the source code information.

**Re-designing the Mapper:** In the section 5.3, we analyzed a special case from the obtained state machine. This example indicates the insufficient design of the Mapper as the size of the buffer in the Mapper is one. This means only one channel is allowed to be opened at any moment. However, RFC 4254 [36] defines that multiple channels are able to open simultaneously and multiplexed into a single connection. Thus it is necessary to re-design the Mapper to simulate the actual SSH communication.
References


Appendix A

The input alphabets for SSH protocol state fuzzing were originally defined in paper [6]. Later, they were updated in paper [4]. To suit the needs of our work, we combined them together and show as follows.

<table>
<thead>
<tr>
<th>Input</th>
<th>RFC-defined name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEXINIT</td>
<td>SSH_MSG_KEXINIT</td>
<td>Sends parameter preferences</td>
</tr>
<tr>
<td>KEX30</td>
<td>SSH_MSG_KEXDH_INIT</td>
<td>Initializes the Diffie-Hellman key exchange</td>
</tr>
<tr>
<td>NEWKEYS</td>
<td>SSH_MSG_NEWKEYS</td>
<td>Requests a new key</td>
</tr>
<tr>
<td>SR_AUTH</td>
<td>SSH_MSG_SERVICE_REQUEST</td>
<td>Requests the authentication protocol</td>
</tr>
<tr>
<td>SR_CONN</td>
<td>SSH_MSG_SERVICE_ACCEPT</td>
<td>Requests the connection protocol</td>
</tr>
</tbody>
</table>

Table 6.1: Input alphabet for the transport layer

<table>
<thead>
<tr>
<th>Input</th>
<th>RFC-defined name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UA_PK_OK</td>
<td>SSH_MSG_USERAUTH_REQUEST</td>
<td>Provides a valid name/key pair</td>
</tr>
<tr>
<td>UA_PK_NOK</td>
<td>SSH_MSG_USERAUTH_REQUEST</td>
<td>Provides an invalid name/key pair</td>
</tr>
</tbody>
</table>

Table 6.2: Input alphabet for the authentication layer

<table>
<thead>
<tr>
<th>Input</th>
<th>RFC-defined name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH_OPEN</td>
<td>SSH_MSG_CHANNEL_OPEN</td>
<td>Opens a new channel</td>
</tr>
<tr>
<td>CH_CLOSE</td>
<td>SSH_MSG_CHANNEL_CLOSE</td>
<td>Closes a channel</td>
</tr>
<tr>
<td>CH_EOF</td>
<td>SSH_MSG_CHANNEL_EOF</td>
<td>Implies that no more data will be sent</td>
</tr>
<tr>
<td>CH_DATA</td>
<td>SSH_MSG_CHANNEL_DATA</td>
<td>Sends the data through the channel</td>
</tr>
<tr>
<td>CH_REQUESTPTY</td>
<td>SSH_MSG_REQUESTPTY</td>
<td>Requests a pseudo terminal</td>
</tr>
</tbody>
</table>

Table 6.3: Input alphabet for the connection layer
Appendix B
Figure 6.1: Butterfly diagram of SSH exchange identification function in sshd.c file.
Figure 6.2: The obtained state machine of OpenSSH6.7-p1 using DFASAT