RADBOUD UNIVERSITEIT

BACHELOR THESIS COMPUTER SCIENCES

Automatic modeling of SSH implementations with state machine learning algorithms

---

Author:
Max TIJSSEN

Supervisors:
Erik POLL
Joeri de RUITER

June 24, 2014

Contents

1 Introduction **1**

2 Background **3**
2.1 SSH . . . 3
2.1.1 Usage . . . 3
2.1.2 The four layers . . . 4
2.1.3 SSH Packets . . . 5
2.1.4 Diffie Hellman key exchange . . . 6
2.2 Automatic state machine learning . . . 7
2.2.1 Mealy Machines . . . 7
2.2.2 L* . . . 7

3 Implementation **9**
3.1 Learner Mapper Teacher . . . 9
3.2 Abstractions . . . 10
3.3 Implementing the mapper . . . 11
3.3.1 Difficulties . . . 11
3.3.2 Building the mapper . . . 12
3.3.3 The SSHJ client implementation . . . 13

4 Results and Analysis **15**
4.1 OpenSSH . . . 16
4.1.1 Standard input . . . 16
4.1.2 Non-standard input . . . 17
4.1.3 Strange behaviour . . . 17
4.2 Bitvise . . . 18
4.2.1 Standard input . . . 18
4.2.2 Non-standard input . . . 18
4.2.3 Strange behaviour . . . 18
4.3 freeSSHD . . . 19
4.3.1 Standard input . . . 19
4.3.2 Non-standard input . . . 19
4.3.3 Strange behaviour . . . 20

5 Related work **24**
5.1 State machine learning . . . 24
5.2 Analysis of SSH . . . 24

6 Future Work **26**

7 Conclusions **28**

Appendices **32**

A Message number translation **33**

B Full OpenSSH model **34**

C Full Bitvise model **35**

Acknowledgements

Firstly I would like to sincerely thank my two supervisors Erik Poll and Joeri de Ruiter for their great support while writing this thesis. Both their feedback on my writing and their technical help has been invaluable. I would also like to thank Marc Smids for reading and providing feedback on this thesis. Lastly I would like to thank Jelle de Gier for allowing me to use his server.

Abstract

A protocol like SSH is specified in RFCs which state how it should behave under different circumstances. Not only are these RFCs not complete or totally unambiguous, but there is also no guarantee that a implementation actually follows them. This research presents a way to automatically extract models from the transport layer of different SSH server implementations by using a learning algorithm. This learning algorithm will produce a model of a finite state machine which shows how well the implementation follows the RFC standard and possibly discover other unexpected behaviour. We modelled three different server implementations in this way and found only one out of three following the RFC standard. Besides this result we also uncovered some unique behaviour for each server.

Chapter 1

Introduction

In our current information age we have become more and more reliant on security protocols in order to protect our information and communications. Understanding these protocols is vital, because of the very sensitive data information they often times handle. This way we can guarantee that our data really is safe.

History has shown that severe vulnerabilities turn up from time to time in security protocols, also in the protocol this research focuses on: SSH. In SSH 1.5 for instance a vulnerability found in 1998 allowed attackers to insert content into an encrypted SSH stream [3]. A vulnerability found in 2008 allowed recovery of up to 32 bits of plaintext from a block of ciphertext [4]. Important to note is that these are weaknesses found in the formal specification of SSH, while this research will look at real implementations. This means that this research would not find these kinds of vulnerabilities. It should not be hard to imagine that if the formal specification of SSH contain vulnerabilities that some implementations of SSH will as well. One example of such a vulnerability in an implementation was presented at the 30th IEE symposium on Security and Privacy and showed a plaintext recovery attack against the OpenSSH implementation [9]. In theory this research could uncover similar vulnerabilities in other server implementations.

In order to understand these protocols you can use either large books [1] or structured RFCs [10, 11, 12, 13, 14], in which they are described extensively.

There are two problems that arise with these books and RFCs. The first is that these documents are so long that it becomes difficult to keep perspective on what it is you are actually reading. With hundreds of pages of text it is often not easy to find just what it is you are looking for or keep the larger picture in mind. For many protocols great strides have already been made in order to better summarise the actual important contents of these descriptions. An example exists of this being done for SSH [2], where three of the RFCs [10, 13, 14] describing the transport layer of SSH were used to make a much more readable paper and some much needed models. These greatly improve a reader's ability to understand the protocol.

The second problem is that these kinds of official specifications (such as the RFCs) are just that, specifications of how a protocol *should* work. There is no guarantee that an implementation of these specifications follows these rules. Differences will always exist, whether accidental or not. These differences could be a serious security flaw, since people only looking at the formal specifica-

tions will often overlook the fact that the implementation in reality could be functioning differently.

This research will attempt to solve these problems by using the L* state machine learning algorithm in order to automatically create formal models, in the form of state machines, of the SSH protocol. This model is a finite state machine which (at a certain level of abstraction) describes the implementation.

This thesis will first give some necessary background on both SSH and automatic state machine learning before showing how we used these concepts on the SSH protocol. After this we will discuss the produced models and show which conclusions can be drawn from these. We will also discuss some possible future work and other related academic work.

Chapter 2

Background

This chapter will give some necessary background information on SSH and state machine learning, which are used in the research.

2.1 SSH

This section will describe SSH in general, by giving both a short overview of how the protocol works and some common uses.

SSH is a way for two machines to communicate securely with each other over an insecure channel. This has, as might you might be able to imagine, many uses.

SSH does this by means of a server/client architecture. Some server somewhere is running an SSH server implementation which receives requests from one or multiple SSH client implementations. In Figure 2.1 a graphical representation from "SSH The Secure Shell, The Definitive Guide" [1] is included. These requests often are commands like downloading a certain file or executing some command on the server. The server will then respond with some requested information or confirmation that a command was executed, or it can respond that a certain request was denied.

2.1.1 Usage

SSH can obviously be used for a great number of applications which require some secure data being send between two machines. Here we will highlight some of the more often used usages.

Probably the most common usage of SSH is remotely logging in to another machine and using it. This can of course be useful in a wide variety of situations, whether when working from home and using the company network or when you are away from home but want to have a program do some work while you're gone.

Secure file transfer is another big feature of SSH, and is often used in conjunction with the secure remote log in feature described above in order to retrieve some file you need from a remote machine. Big file sharing services like GIT can often times also be run through SSH, which not only can help secure the

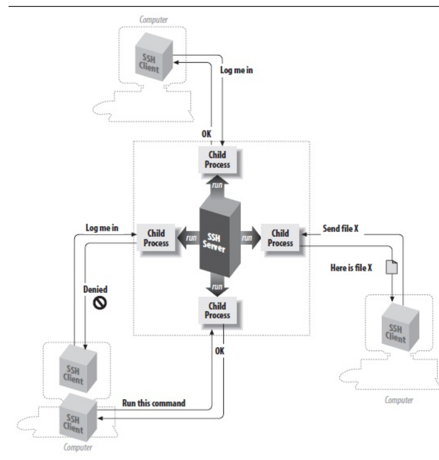

Figure 2.1: A graphical representation of the architecture of SSH [1].

transfer but when using keys also takes away the need to continuously type your password.

According to "SSH: the definitive guide", Chapter 1.4.5, [1] SSH can also be used for access control in order to allow users certain rights on your machine, without have to explicitly grant them this right every time. This can for instance be used to allow people you are working with to retrieve certain files you have on your machine without them being able to access your personal files. It is a bit unclear what is meant by this exactly, since remote log in seemingly already has this feature(since a user has certain rights).

2.1.2 The four layers

SSH is build up of four different layers which all work together in order to make the above described process possible [13]. These are: the transport layer [14], the user authentication layer [11], the connection layer [12] and the SSHFP DNS record [15].

The lowest layer is the transport layer [14] which handles the initial key exchange and thus aides in the authentication of server and client but after this also offers an interface for the other layers to send the different messages between the client and server. The transport layer in essence creates the secure channel which the other layers use to securely send their data. This research in the end models this layer.

The commands (and the responses from the server) described above are confidential so you wish to be sure with whom you are communicating when using SSH. This is handled by the layer above the transport layer, the user authentication layer [11]. This layer uses many common authentication methods such as passwords or RSA key pairs in order to authenticate that a user is indeed who he claims to be.

The connection layer [12], which is the highest layer, handles the channels through which these commands and other communications between server and client occurs. One SSH connection is compromised of a number of channels, which are used for both the sending and receiving of data.

The SSHFP DNS record [15] is a simple layer which stores public host key fingerprints so that the authenticating of hosts which are visited multiple times can be established more easily and quickly.

These layers work on top of each other, with the higher layers needing the functionality of the lower layers in order to function. SSHFP DNS is a bit of a different case since this really only stores data, and is used by the transport layer. A graphical representation can be found below.

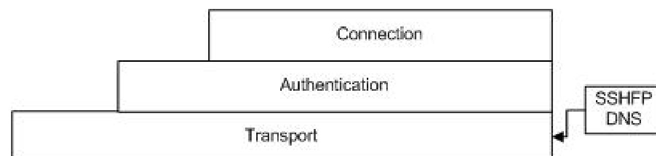

Figure 2.2: A graphical representation of the layers of SSH.

2.1.3 SSH Packets

The server and client communicate with each other by sending packets back and forth. A normal SSH packet has the following contents (in order)[17]:

- *sequence number*: Used to order the incoming packets.
- *pktl*: The length of the packet.
- *pdl*: Padding length, each packets is padded by a certain number of bits in order to make sure it is the correct cipher block size(for the encryption and decryption that follows).
- *(compressed) Payload*: The actual contents of the package. It is this part of the packet we are interested in. It contains a number which is called the message type, this determines the type of message being send. Later on in Section 3.2 we will describe how this is used.
- *Padding*: The padding mentioned in pdl.

This packet is encrypted as a ciphertext, a MAC is added to verify the integrity upon arrival at the destination and sent to either the server or the client. A graphical repesentation of such a packet taken from work of W. Stallings [17] can be found in Figure 2.3.

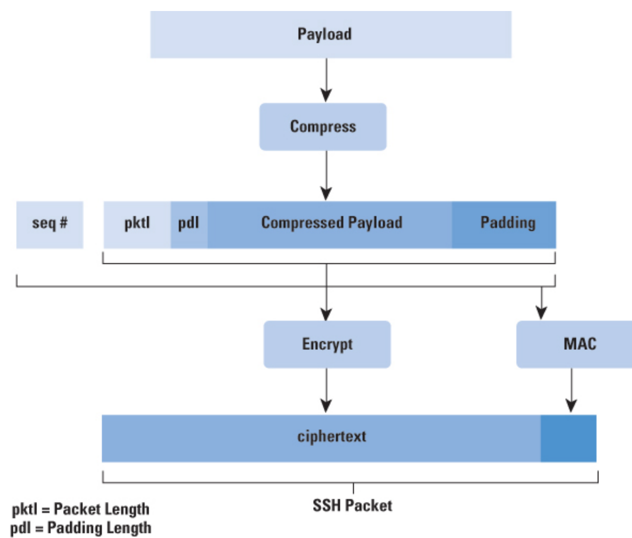

Figure 2.3: A graphical representation of an SSH packet.[17]

2.1.4 Diffie Hellman key exchange

Diffie Hellman(DH) was the key exchange algorithm used to test our servers, so it is important to quickly show how normal DH key exchange takes place. Note that it is not important to understand the contents of the different packets, only the order in which they are send and by whom (client or server).

Figure 2.4 shows how DH key exchange is supposed to take place according to the RFCs as taken from [2] which describes the SSH transport layer. In this figure "C" is the client and "S" the server. Two quick notes about this picture: the order in which identical packets are send by either the server or client is not specified, so either the server or client can send its version number first according to the RFCs. Next is that the KEXDH_REPLY in this picture and KEXDH_31 which we will later show in our own output are the same package, just different names.

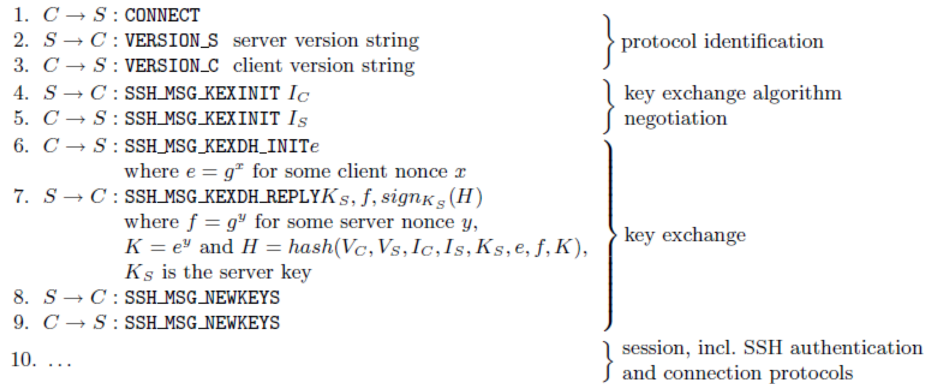

Figure 2.4: Diffie Hellman key exchange in the SSH transport layer [2]

2.2 Automatic state machine learning

This section will give a brief explanation of what Mealy Machines and the L* learning algorithm are.

2.2.1 Mealy Machines

Mealy machines are finite state machines where every transition has both an input and a resulting output [6]. They are described by a tuple $M = \langle I, O, Q, q^0, \delta, \lambda \rangle$. In this tuple:

- I = The list of input symbols.
- O = The list of output symbols.
- Q = The list of states.
- q^0 = The initial state.
- δ = The transition function.
- λ = The output function.

At any given moment a Mealy Machine M is in a state q which has a number of transitions, each of these transitions has both an input at an output symbol. The Mealy Machines produced by L* are both *complete* and *deterministic*, so in every state it there is exactly one transition for every input symbol.

An example such a Mealy machine can be found in Figure 3.1. The machine has input and output alphabet {a,b} and three states {A,B,C}. Each transition has a corresponding x/y where x is the input symbol and y the output symbol.

2.2.2 L*

The method we use to model the SSH implementations largely depends upon the L* learning algorithm which is a learning algorithm first described by Dana Angluin in 1987 [7] and later adapted in 2003 by Niese [8]. It is an active

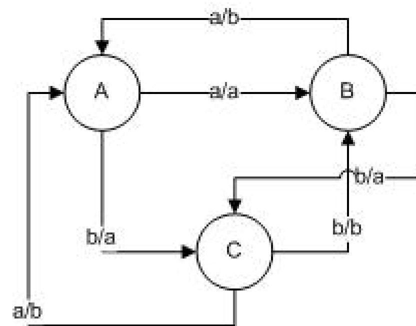

Figure 2.5: A example of simple deterministic and complete Mealy Machine.

learning algorithm which puts query's consisting of an input string to a certain teacher and uses the returned output (and state change) to model a Mealy Machine. This algorithm can learn models of enormous systems which would result in a model consisting of thousands of states, but we are more interested in using it to model a quite small model, so it stays readable. L* has a nice Java implementation called LearnLib[1] which we will be using in this research.

[1] http://learnlib.de/wordpress/

Chapter 3

Implementation

In this chapter we will show how we used the L* algorithm to model different SSH server implementations.

3.1 Learner Mapper Teacher

Our method for modelling will work as follows: The L* algorithm(the Learner) wishes to know what the generated output for a certain input symbol x is. It asks the SSH server (the Teacher) what the output symbol y for this x is, the Teacher responds with symbol y. The Learner processes this new data. At this point the Learner can either decide its done, and asks the Teacher to evaluate the model it has (by comparing its model to the one the teacher knows) or decide it has more questions, and start the process again.

This leads to a few issues. First of all: the Learner and the Teacher can not directly communicate with each other. L* will want to send a certain input symbol x and receive a certain input symbol y but the SSH server doesn't work like that. The SSH server uses, both for input and output, concrete packets which consist of many (dynamic) components.

This also leads in the next issue, we will want a certain level of abstraction so the model can be understandable by humans. The concrete packets send between an SSH server and client are completely unreadable for humans. For our model we would rather have something more similar to what is used in the formal description of SSH [2] which uses commands.

Lastly we can not easily check our end result with the Teacher, since the resulting Mealy machine is an approximation with abstraction of how the SSH server behaves.

For the first two problems we will have to build a so called mapper that will implement the necessary abstraction and translate the Learner's commands to usable packets for the SSH server, and also translates the received output packets back to for the Learner usable symbols. This mapper will be based upon a existing implementation of a SSH client, since this it is only logical to have the mapper behave like an SSH client in order to communicate with the server.A graphical representation of this can be found in figure 3.1.

The last issue can also be overcome. L* can check if its model is correct by sending the mapper certain requests and seeing if the server output it receives

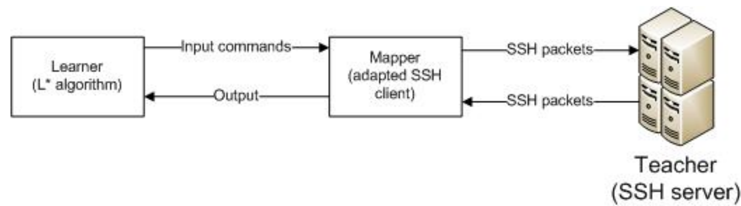

Figure 3.1: A graphical representation of the Learner Mapper Teacher setup.

is the same as it has modelled. The model will be correct as long as the chosen level of abstraction is correct and the mapper is implemented correctly, since the L* algorithm has been shown to work correctly [6] [5] before.

The first two parts are quite easy to implement. The learner will have to be our implementation of the L* algorithm, fortunately there is a very competent Java implementation of this algorithm called LearnLib[1] which has been shown to work in earlier research [6] [5] so its a natural choice to use this.

The teacher is also rather straightforward, since this will be some SSH server. For this research we will be using a couple of servers. We will test a few servers running locally on a windows machine (FreeSSHD[2] and Bitvise SSH[3]) as well as a OpenSSH[4] server running on a remote Linux server.

A quick note about the teacher. Who fulfils role can be interpreted in different ways. For clarity's sake we chose the server for this role. Another interpretation is calling the combination of the mapper and the server the teacher but we found this to over complicate things.

3.2 Abstractions

As said before in Section 3.1 we will need a certain level of abstraction from the concrete packets send between the client and server in order for our model to be workable.

As stated in Section 2.1.3 , it is the payload of packets we are interested in, the other fields are simply control fields used in order to make communications possible, but the payload contains the message. This message will be a sequence of bits which we will want to abstract from. Luckily the makeup of the messages helps a great deal with choosing an abstraction level. Every message contains a byte called the message number, which determines the type of the message [2]. Using these message types would thus be a good level of abstraction, since these types are not only readable but also finite, so the L* knows what the input and output alphabets are.

Since this message type is stored in a byte it can have values ranging from 0 to 255. Within these numbers certain numbers are reserved for certain layers of the protocol. 1 to 49 are reserved for the transport layer, 50 to 79 for the user authentication layer, 80 to 127 for the connection protocol and the remaining numbers are used for either client protocols or local extensions [10].

[1] http://learnlib.de/wordpress/
[2] http://www.freesshd.com/
[3] http://www.bitvise.com/ssh-server-download
[4] http://www.openssh.com/

The table used to translate these message numbers to the corresponding message types is described in RFC 4250 [10] and has also been added to the appendix of this thesis. This will be used as the abstraction level. This means that any data after the message number will be ignored by the L* algorithm, since it will never get to see it. It's the type of message we are interested in, not what it says.

The following message types will be send to the server in order to see how it responds to these at different moments. These messages are basically all the message types belonging to the transport layer, when using Diffie Hellman key exchange.

- SSH_MSG_DISCONNECT
- SSH_MSG_IGNORE
- SSH_MSG_UNIMPLEMENTED
- SSH_MSG_DEBUG
- SSH_MSG_SERVICE_REQUEST
- SSH_MSG_SERVICE_ACCEPT
- SSH_MSG_KEXINIT
- SSH_MSG_NEWKEYS
- SSH_VERSION
- SSH_MSG_KEXDH_INIT
- SSH_MSG_KEXDH_31

3.3 Implementing the mapper

As explained before in Section 3.1 we will need to have 3 separate parts that work together in order to complete this research: the learner, mapper and teacher. The first two parts were already discussed in the same section but since the mapper is the only part that we had to build ourselves it will be discussed in further detail here.

3.3.1 Difficulties

The difficulties with implementing the mapper stem from it involving a large amount of writing code based upon a large and unfamiliar software project.

The mapper will be based upon a existing SSH client implementation, since it will take on the role of a client when communicating with the server. The difference between a regular client and the mapper that is to be created is that this mapper has to take its input from LearnLib (instead of following the SSH specifications) and send this to the server. Then is has to translate the response from the server back to a for LearnLib useable format. The existing client the mapper used in this research is based upon the upon source Java SSH client

SSHJ [5]. Multiple SSH clients were considered but this one offered the most clear structure which was believed to help in creating the mapper.

An example how this mapper will work is as follows. Learnlib wishes to know what kind of output the input symbol "SSH_MSG_KEXINIT" generates. Learnlib send this input symbol to the mapper. The mapper will at this point be connected to the SSH server and is seen as a normal client by the server. The mapper then creates a packet for a KEXINIT message, this will among other things include what kind of encryption and key exchange protocol the mapper(so the client it uses) can use. It sends this packet to the server and awaits its response. The server will respond with a packet which the mapper will *fully process*. Although we only wish to send the message type back to Learnlib we might still need to use the information send by the server for future packets. Something to look out for is to stop the client from automaticly sending follow up packets, since we only wish to sends packets when Learnlib prompts us to. The mapper then sends the message type of the package the server send him back to Learnlib. A picture describing this example can be found in Figure 3.2.

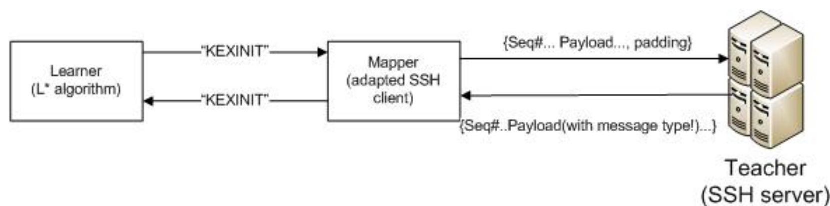

Figure 3.2: A graphical representation of the mapper example

3.3.2 Building the mapper

In this section we will give some helpful information that we learnt while building the mapper used for modelling our SSH implementations, which future researchers seeking to duplicate results or model their own protocol might find helpful.

The first step is finding a client on which to base your mapper. This first choice will greatly influence how difficult the further steps will be so be sure to choose carefully. The client will of course have to be open source and it will be easier if it's in the same programming language as your mapper. The complexity of the client is the big thing to look out for, simpler clients are better in most cases since these are easier to oversee and you will not need many higher level functions. A decent measure for complexity is simply the number of classes or files the project uses. Last but certainly not least is looking at how clearly the code is written and entire project are structured. In order to measure this take a look at how much documentation different functions/classes have and how clearly different used packages are labelled. Try and find a client implementation which maximises these criteria and the rest of the research will go much smoother.

[5]https://github.com/shikhar/sshj

After finding a client implementation you will have to alter this to only send the packets LearnLib asks you to, and send back the response. Doing this will require understanding how the client implementation works, and finding the right functions to use. Helpful tools to do this include looking at some examples at how the client normally behaves, by following how a certain command that you give it is translates into concrete packets. Another tool is the code search included in many programming enviroments to search for certain key words, such as "send" or "handle". It might also be useful look for the names of specific type of packets you wish to send. For example for the SSH_MSG_KEXINT you could search for kexInt and the sendKexInt() method might turn up, which as you can probably imagine is quite useful. Understanding a client implementation such as this is no easy task, with many (for this research) redundant classes and functions it is hard to find just which pieces of code you will want to use.

After you understand your client implementation remove all automatically sent packets and make explicit methods to send these when requested. A thing to look out for when doing this is that some packets might depend on ones send or received earlier, so you will have to include some default initialisation for these values. It is very important to document all changes you make to the implementation, so that further on down the line during debugging you can look back and see just what changes you made and what could cause current behaviour.

3.3.3 The SSHJ client implementation

For this research the mapper we created used modified versions of the following Java classes (and their dependencies) of the SSHJ client:

- TransportImpl:which implements the transport layer for sending and handling packets.
- KeyExchanger: which handles packets used for keyexchange.
- Message: which enumerates the message types.
- SSH Packet: which implements a SSH packet
- AbstractDHG: which implements the DH keyexchange.
- Reader: which reads the inputstream and gives the received packets to the TransportImpl to handle.
- SocketClient: which sets up the connection with the server.

Besides these modified classes we also needed some classes build from the ground up. These classes were made into two different packages: Learner, which handles Learnlib setup, input and output and Mapper which handles the client and the translations between abstract and concrete. This setup was based on code provided by Joeri de Ruiter which was used to make models of banking cards [6] .

The learner package contains the following:

- Mapper: This class tells LearnLib the input and output alphabets.

- MembershipOracle: This class handles input and output to Learnlib
- SutWrapper: This class takes the input given from MembershipOracle and sends it off to the other package to be processed, it then returns its reply.
- Main: This class handles the others.

The mapper package contains the following:

- SUT : This class takes on the role of the SSH client, it uses the modified classes mentioned before in order to send packets to the server and read its reply.
- InputReader: This class translates message types into their corresponding command and the other way around.
- Main: This communicates with the other package, so it takes the input send to it by the learner package and replies with responess. This class is also responsible for resetting the connection to the server.

Chapter 4

Results and Analysis

This section presents the results and analyses these. For the research we modelled three different servers: FreeSSHD[1] and Bitvise SSH[2] running on a local Windows machine as well as a OpenSSH[3] server running on a remote Linux server.

The images consist of state and transitions, with the circles being the states and arrows the transitions. Each transition has the form x/y where x is what the mapper sends to the server and y is what the server responds with. A "a + b" output means that the server replied with both a message of type a and a message of type b. Besides the standard SSH message types (which can be found in Appendix A.) the images also include SOCKET_CLOSED and TIMEOUT. These are not actually responses from the server. SOCKET_CLOSED means that the server has closed the socket the mapper used to communicate, so no more packets can be send. TIMEOUT simply means the server has not replied to the message.

The figures shown in this section are modified versions of the ones produced by the algorithm, these changes were made in order to make the result more readable.

The changes that were made to the original images are the following:

- Clustering multiple edges together with terms like "other" or "any". "Other" meaning any of the messages from Section 3.2 that are not yet mentioned in the current state, "any" meaning any message type.
- All edges where the input was either SSH_MSG_IGNORE , SSH_MSG_UNIMPLEMENTED or SSH_MSG_DEBUG where omitted since these (as long as the socket was open) always resulted in a time out.
- The prefix SSH_MSG has been omitted from all messages, since it doesn't add any information and only serves to bloat the image further.
- Added state "s-1" to the models. This is the state you are in before connecting to the server. As soon as you connect to the server it will send its SSH_VERSION, so not as a response to any packet send by the

[1] http://www.freesshd.com/
[2] http://www.bitvise.com/ssh-server-download
[3] http://www.openssh.com/

mapper. In the original images "s0" is the first state, since you will have to connect before being able to send any packets. This transition can thus been seen as implicit. It has been added for completeness.

The original models made by LearnLib can be found in the appendixes. One thing to note here is that the original model for FreeSSHD is missing. This is because we were unable to create one, a more detailed explanation can be found at Section 4.3 where the picture is discussed. One more thing to note is that the SSH_MESSAGE_SERVICE_REQUEST needs some service to request, we chose "ssh-userauth" for this purpose, which is the standard service that should take place after key exchange.

For every model we will first look at how it handles standard Diffie Hellman key exchange, followed by how it deals with non standard input and lastly discussing the strange things we discovered about each server.

4.1 OpenSSH

Figure 4.1 on page 21 shows the output generated when testing an openSSH server[4] running on a remote Linux server.

4.1.1 Standard input

Here we will look at how the server responds during standard Diffie Hellman key exchange as described in Section 2.1.4. In Figure 4.1 we can clearly see Diffie Hellman key exchange taking place. In s0 the server first sends its KEXINIT as soon as we send our version number since whatever we're going to do we'll first need to exchange keys to encode our communications with. This KEXINIT contains its preferences for things like encoding and key exchange algorithm. We respond to this with our own KEXINIT in s1. After these two messages are sent the client determines to use Diffie Helmman based on the two KEXINITs, which makes sense since our client only supports this algorithm. This is a trend we'll see in all the different models. We do not receive a response to the KEXINIT sent in s1 since the client should send the next message, which is the KEXDH_INIT, which is send in s3. This KEXDH_INIT contains the information needed for the key exchange. The server replies to this with a KEXDH_31 and quickly after this sends a NEWKEYS package which signals the end to the keyexchange, which we have to reply to with our own NEWKEYS package in s4 in order to confirm this.

A thing to note here is that it is dependent on how long the server has to wait before it receives a NEWKEYS package whether or not it will sends its own first. If the client is quick enough it can send its NEWKEYS before the server. It wil then receive only KEXDH_31 as a response to the KEXDH_INIT and the NEWKEYS as a response to its own NEWKEYS. For the protocol itself this doesn't really matter since they can be send in any order but it shows us a bit about the implementation of the server. It seems to imply some kind of timer waiting for the client to send its NEWKEYS and otherwise the server just sends its own.

[4]http://www.openssh.com/

4.1.2 Non-standard input

We can see that any packets sent before the SSH_VERSION are simply ignored by the server, which makes sense since at this point there is no formal connection. In s1 the server is expecting a KEXINIT and will close the socket for some other packets it does not expect. This will cause the model to end up in s2, which is the state where the server has closed the socket, so no more packets can be send and the connection to the server has to be reset. But there are quite a few packets it simply ignores which are very odd among these SERVICE_REQUEST and KEXDH_INIT. This directly violates RFC 4253 [14] which states:

> If the server rejects the service request, it SHOULD send an appropriate SSH_MSG_DISCONNECT message and MUST disconnect.

The server should reject these service requests since they are requested before key exchange finished and according to the same RFC the description of a SERVICE_REQUEST is :

> After the key exchange, the client requests a service.

The key exchange has not been completed, so this SERVICE_REQUEST should not be allowed. So not only does the server not send a DISCONNECT which it should, but also doesn't close the connection which it *must* according to the RFC.

Even stranger is that the server simply ignores packets it should *never* receive from a client, the SERVICE_ACCEPT and KEXDH_31 packets.The RFCs actually do not describe what a server should do when it receives this kind of input. A more defensive server would probably close the socket when it receives this kind of input from a client. This behaviour gets rectified after the server receives the KEXINIT, any wrong packets will cause the socket to close at this point. This points to there being some strange "waiting state" for the server when its waiting for the KEXINIT. In this state it is simply reading packets until it gets a KEXINIT and only looking for a few key unexpected ones to close the socket, but ignores all others that do not fit this criteria. The OpenSSH server also only starts sending SSH_MSG_DISCONNECTs, which tell the client why the server is closing the socket, in s2 so after receiving the KEXINIT. This is more evidence for some weird "waiting state" while it has not yet received a KEXINIT.

4.1.3 Strange behaviour

In s4 any packet sent will cause a time_out, after which the socket closes. This shows a problem we encountered with two of the servers(this one and Bitvise), where after sending the NEWKEYS packet signalling the end of key exchange the server replies with a very strange packet. This packet has a packet length of some absurdly large positive or negative number, so has obviously been damaged somehow. We have not been able to find what causes this to happen, but after this packet is received we are unable to continue communication with the server which explains why all the models end at the key exchange algorithm. But since key exchange is the largest part of the transport layer protocol we still find this research to give some very interesting and useful results.

4.2 Bitvise

Figure 4.2 on page 22 shows the model of a Bitvise[5] SSH server implementation. This server was running on a local Windows machine.

4.2.1 Standard input

There is little to say about this server other then it seems implement DH key exchange the same way that the OpenSSH server did, as described in Section 4.1.1.

4.2.2 Non-standard input

If the server at any point in the algorithm receives some unexpected input it will send a DISCONNECT packet which states why it is closing the socket and then close it. This shows a big contrast with the previous server which ignored some input, closed the server with reply for some others or send a DISCONNECT for yet other packets. This server seems to be the only following the RFCs, although it still has some strange behaviour. An example of the server responding strangely to non-standard input is when you send a NEWKEYS packet after the KEXINIT the server will start ignoring all your input. This is similar to the problem that described in the previous section. So while it seems that sending a NEWKEYS or KEXDH_INIT while in s3 will lead to the same result s4, this is not exactly true. From the clients perspective this is correct, the server will start ignoring input but the server will be in a different state if you first send the KEXDH_INIT namely the state where key exchange has happened. This is confirmed by the NEWKEYS packet the server sends.

4.2.3 Strange behaviour

Another interesting thing we found while testing is that the server would occasionally send a IGNORE message. This message would not come as a reply to anything being send (which is why it is not in the model) and seemed to occur at random. Our best hypothesis is that this is the server testing if the connection is still functional.

The last observation we made while modelling the server is that after the testing had been going on for a while the server would sometimes take a very long time(30 secs to a minute) to respond to a new version being sent, at the start of the protocol. But when it did respond it did so normally and would function like nothing happened.

[5]http://www.bitvise.com/ssh-server-download

4.3 freeSSHD

Figure 4.3 on page 23 is based upon a freeSSHD[6] server running on a local Windows machine.

Figure 4.3 was made by hand and not based upon a existing model as the others were. The reason this picture was not automatically generated is because after exactly 351 "sessions" (SSH_VERSION followed by some other packets) the server would crash. It did not matter if the server was being reset in the meantime. For instance if the server was reset after 200 sessions, it would simply crash at the 151th session after this reset. Letting the server crash and resetting it after this 351 sessions simply causes it to start crashing very frequently after this, making testing impossible. We have not been able to find what causes this behaviour to occur, it is a problem exclusive to this server implementation and was not found anywhere else. The fact that resetting the server does not help means that it is not simply a case of lingering sockets, since these are broken when the server is shut off and it only occurs with this specific server. There is some mysterious link between the server and a testing session that persists even after the server shuts down which will inevitably crash the server. The same result was found when running the server on a remote machine, effectively making the program a way crash any freeSSHD server we can connect to.

This figure was made by making tests consisting of different orders of packets being send and looking at how the server responded. This means that this figure should be considered less trustworthy then the other two, since it was made by hand instead of automatically which makes it a lot more error prone. Nonetheless we believe this picture is largely correct, and are confident that most of the weird behaviour of this server was captured.

4.3.1 Standard input

The server seems to largely function as expected, although this server waits with sending its NEWKEYS until the client sends his. This causes the KEXDH_31 to come as a reply to KEXDH_INIT and NEWKEYS as a reply to NEWKEYS. This is different from the last two servers, who both send the two packets together.

4.3.2 Non-standard input

The server isn't fully consistent with sending SSH_DISCONNECTS but it's better at it then the first server and for the most part does not accept any strange input. But where the first server ignored some input it shouldn't this server takes it a step further, it actively accepts service requests in the middle of key exchange. This behaviour can be seen by the fact that in s1 and s2 SERVICE_REQUESTs are answered with SERVICE_ACCEPT. The RFC describing the transport protocol [14] which was cited in Section 4.1.2 as well even explicitly states:

> "Once a party has sent a SSH_MSG_KEXINIT message for key exchange or re-exchange, until it has sent a SSH_MSG_NEWKEYS message (Section 7.3), it MUST NOT send any messages other than:

[6] http://www.freesshd.com/

> Transport layer generic messages (1 to 19) (but SSH_MSG_SERVICE_REQUEST and SSH_MSG_SERVICE_ACCEPT MUST NOT be sent);
>
> Algorithm negotiation messages (20 to 29) (but further SSH_MSG_KEXINIT messages MUST NOT be sent);
>
> Specific key exchange method messages (30 to 49)."

So not only does the server not enforce the fact that it should not accept these packets it even breaks this rule itself by sending a SERVICE_ACCEPT.

Something else we can learn about the implementation of this server can be seen in s3. Here you can either send a NEWKEYS packet which will cause you to get a NEWKEYS packet in return and complete key exchange. Alternatively you can send something else, and the server will still send you a NEWKEYS packet anyway but close the server socket afterwards. This indicates that the server probably will reply any packet it receives with NEWKEYS, before even checking what this packet is. If this packet is not the NEWKEYS it was expecting it will then close the socket. A nicer implementation would of course have been to first check the received packet and then reply with either a NEWKEYS if it was a NEWKEYS or a DISCONNECT if it was something else, before closing the socket.

4.3.3 Strange behaviour

The strange behaviour of this server has already been discussed. In Section 4.3.2 the server was shown accepting services it should not and in the introduction of this section we discussed the server crashing.

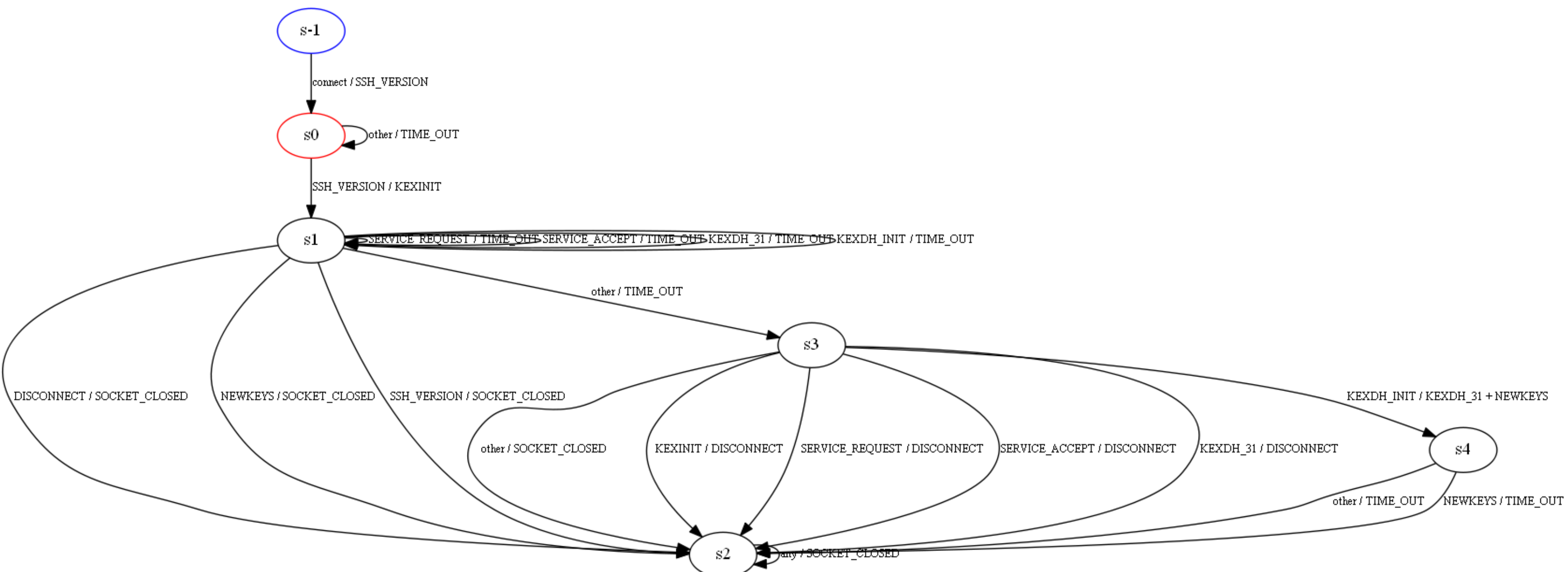

Figure 4.1: An inferred model of the OpenSSH server implementation. At every transition what's left of the '/' is what the mapper send, to the right is the server's reply.

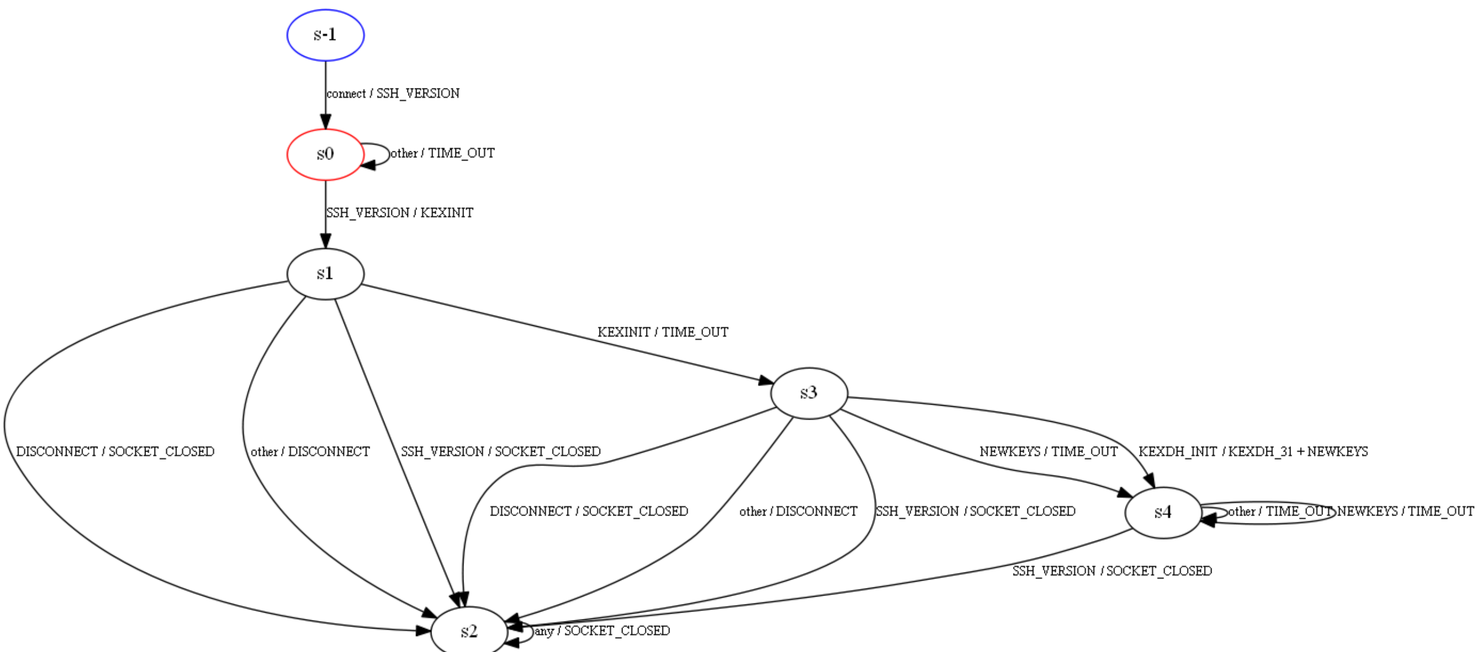

Figure 4.2: An inferred model of the Bitvise server implementation. At every transition what's left of the '/' is what the mapper send, to the right is the server's reply.

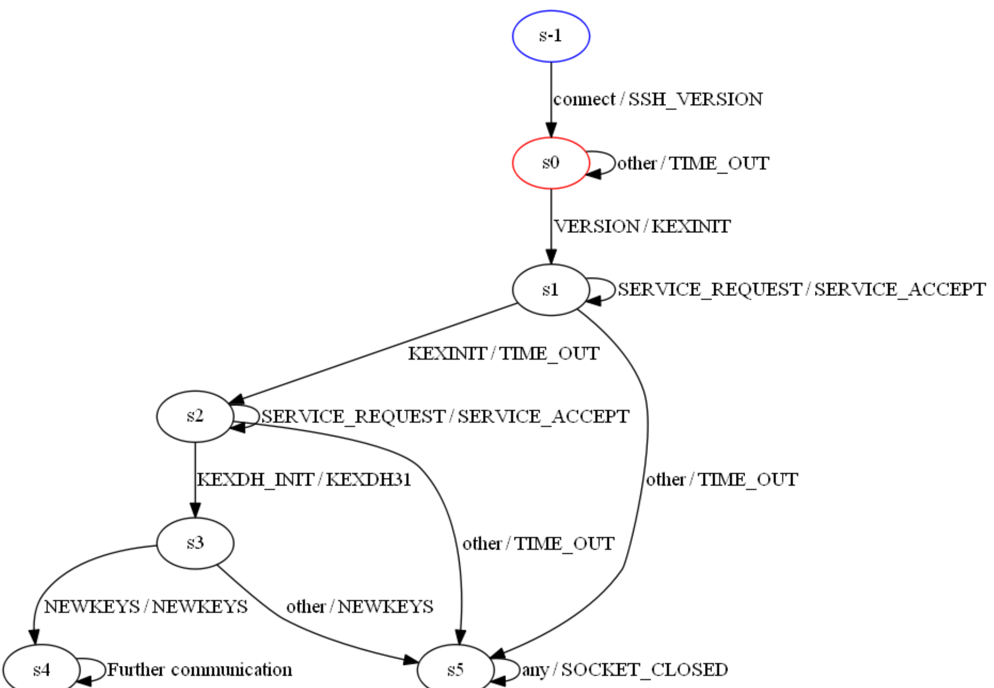

Figure 4.3: An inferred model of the freeSSHD server implementation. At every transition what's left of the '/' is what the mapper send, to the right is the server's reply.

Chapter 5

Related work

In this section we will discuss some other academic work related to this research, to give a better picture of where exactly this research fits in the larger academic world.

Work related to this research comes in many forms and some of the ones mentioned below have also been mentioned in other places in this thesis. We will divide the related work in two different categories. The first category we will discuss is work related to automatic model based learning/testing and the other will be work more specific to the SSH protocol.

5.1 State machine learning

Two papers with a similar approach to the one used in this research that model some other protocols are a bachelor thesis [5] which models TCP and a research paper [6] which models different kinds of bank cards. Both of these papers used a similar setup as this research, using the Learnlib implementation and some mapper in order to model their respective protocols. These both serve as evidence that the used techniques are correct and do in fact give a correct model.

Another interesting article is the one about Athena, a way to efficiently and automatically analyse security protocols [19]. This research shows a different approach in order to achieve the same goals; to automatically check some behaviour of a security protocol. This research also uses a model based approach but instead of Mealy machines relies on an extended version of the Strand Space Model. It combines this with theorem proving to check this model for some behaviour.

5.2 Analysis of SSH

A very relevant piece of research by Erik Poll and Aleksy Schubert [2] was already mentioned multiple times in this thesis. This research gives a formal specification of the transport layer of the protocol based upon the three RFCs describing this. They also present some handmade models, whereas this research seeks to automatically generate these same kind of models for specific implementations.

A bachelor thesis [18] shows how to use models of SSH implementations like the ones we created with this research in order to test if an implementation follows the official specifications. An important difference is that the models he uses were created by hand, whereas this research shows to create these automatically.

Chapter 6

Future Work

There is still much work that could be done to improve or extend this research, in this section we will discuss some possibilities.

Firstly, as shown in Chapter 4 there are still some errors and mysteries that a future research could take a look at. The most glaring one is trying to fix the case where after key exchange both the openSSH and Bitvise servers send a corrupted packet and afterwards close the server. If this was fixed you could model the transport layer more completely, since you could take a look at things such as refreshing the key by starting another key exchange after the first one or try some service requests. Taking a look at why exactly the freeSSHD server crashes could also be very interesting, since for the moment this is still unexplainable.

Another possibility is looking at some different key exchange protocols or SSH versions. By taking an older client that runs SSH1 as the basis for your mapper you could model how servers behave when presented with an older SSH version. A client which uses a key exchange other then Diffie Hellman could be interesting in a similar way, since most servers will expect to simply use Diffie Hellman. An example of such a algorithm is described in RFC 4432 [16] which describes how to use RSA for key exchange.

SSH is of course a lot more complex then only the transport layer. Research similar to this one could try to model another layer like the authentication layer. You would first need to setup the transport layer but after this you could model the authentication layer similarly to how we modelled the transport in this research.

Using the models created by this research with a model checker, for instance Uppaal, to check certain conditions which are necessary for the secure functioning of the implementation is another possibility. By checking these conditions we can with more certainty guarantee the security of the implementation. If one of these conditions fails it can be analysed why it failed, and hopefully find and fix vulnerabilities in the implementation before they are exploited. Besides using model checking tools to analyse the model it can also be greatly beneficial to simply study the model in order to help understand just how a certain implementation of SSH works, since analysing such a picture is much easier than a large amount of text. An example of this kind of work is the thesis described in Related Work [18]. This research and ours can in theory be combined, in order to fully automatically test if a protocol implementation follows some standard.

This could be done by first using the techniques described in this research to produce models, which can then be used by the work by Boss [18] to test if they follow the standard.

Something else that could be done is to mess with the values of some of the other fields of a packet. For instance you could look at how different implementations handle packets with a MAC address or packet length that has been altered. In the same vein you could also change the value of the payload, by sending some non-existing key exchange preferences or encrypting your messages with a wrong key.

Lastly, there are of course many other interesting network security protocols that could be modelled using this technique. Some possible protocols that could be used for this are SSL, HTTP or possibly FTP. You would need to write a different mapper but other then that the setup would likely be largely the same. There are also still many more SSH servers that could be modelled the same way as the three used in this research.

Chapter 7

Conclusions

As shown in Chapter 4 our research has produced some very interesting results. In short these are:

- The research gives further evidence this technique can be used to model different kinds of security protocols.
- The research can be used to find out information about how a certain unknown server is implemented.
- The research can be used to find possible flaws in your implementation and fixing these before they get abused.
- This research can be used to test if a server follows the RFC standard.
- The research found a very strange flaw in the FreeSSHD server implementation which allowed us to crash it.

In more detail:

- This research provides further evidence how this technique can be used to model different kinds of security protocols. After other examples such as banking cards [6] and TCP [5] we show that SSH can also be modelled in the same fashion.
- We discovered quite a lot about the way the different servers were implemented based upon their responses. The possible "waiting state" that the openSSH server might have could be a thing for attackers to exploit, since the server will only close the socket for only a few select packets while ignoring the rest when it is in this state.
- The previous result show the kind of information that a attacker could be interested in that this research produced. This information is of course not only useful to a possible attacker. The developers of the server could also look at the results and find places to improve their work.
- Another result is that both the freeSSHD and OpenSSH server do not follow the RFC standard for the transport layer as specified in RFC 4253 [14]. FreeSSHD breaks this standard in two ways. Firstly by accepting a

SERVICE_REQUEST that should be denied and secondly by sending the SERVICE_ACCEPT for this request during key exchange, which is not allowed. OpenSSH breaks the RFC by not denying a SERVICE_REQUEST during key exchange, but instead ignoring it. We also found the OpenSSH server to ignore packets it should never receive from a client. This isn't breaking the RFC since the appropriate response is not mentioned in them, but this is probably not the safest solution.,

- The last result of this research was quite unexpected but interesting nonetheless. We found a way to crash any remote freeSSHD server as long as we can connect to it. This shows some inherent flaw in the server implementation. It should be clear that a server should not crash this easily by one machine sending packets to it. The fact that these crashes occur even if the server is reset in the meantime is further cause for worry because this means that on the server side there is very little you can do to prevent this attack, except trying to block all communications from the machine sending you the packets, but at this point the server does not.

Bibliography

[1] Barret, D. J., Silverman, R. E. & Byrnes, R. G.(2005) , *SSH, the Secure Shell The Definitive Guide 2nd edition*. Sebastopol: O'Reilly Media, Inc.

[2] Poll, E. & Schubert, A. (2011). Rigorous specifications of the SSH Transport Layer. *Technical Report ICIS-R11004, Radboud University Nijmegen*

[3] SSH insertion attack. Retrieved on March 15, 2014, from http://www.coresecurity.com/content/ssh-insertion-attack

[4] Taschner, C. SSH CBC vulnerability. Retrieved on March 15, 2014, from http://www.kb.cert.org/vuls/id/958563

[5] Janssen, R. (2013). *Learning a State Diagram of TCP Using Abstraction.* Bachelor Thesis. Radboud Universiteit.

[6] Aarts, F., Ruiter, de J. & Poll, E(2013). *Formal models of bank cards for free.* Paper presented at 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Luxembourg. Retrieved on March 16, 2014, from IEEE Xplore. 10.1109/ICSTW.2013.60

[7] Angluin, D(1987). Learning regular sets from queries and counterexamples, *Inf. Comput., 1987* (vol. 75, no. 2), pp. 87–106

[8] Niese, O(2003). An integrated approach to testing complex systems, Dortmund University, Tech. Rep., doctoral thesis.

[9] Albrecht, M. R., Paterson, K. G. & Watson, G. J.(2009) Plaintext Recovery Attacks Against SSH. *2009 30th IEEE Symposium on Security and Privacy.* Berkeley, CA.

[10] Lehtinen, S. The Secure Shell (SSH) Protocol Assigned Numbers. RFC 4250, The Internet Engineering Task Force, Network Working Group.

[11] Ylonen, T. (2006). The Secure Shell (SSH) Authentication Protocol. RFC 4252, The Internet Engineering Task Force, Network Working Group.

[12] Ylonen, T. (2006). The Secure Shell (SSH) Connection Protocol. RFC 4254, The Internet Engineering Task Force, Network Working Group.

[13] Ylonen, T. (2006). The Secure Shell (SSH) Protocol Architecture. RFC 4251, The Internet Engineering Task Force, Network Working Group.

[14] Ylonen, T. (2006). The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, The Internet Engineering Task Force, Network Working Group.

[15] Schlyter, J. & Griffin, W. (2006). Using DNS to Securely Publish Secure Shell (SSH) Key Fingerprints. RFC 4255, The Internet Engineering Task Force, Network Working Group.

[16] Harris, B. (2006). RSA Key Exchange for the Secure Shell (SSH). RFC 4432, The Internet Engineering Task Force, Network Working Group.

[17] Stallings, W. (2009). Protocol Basics: Secure Shell Protocol. *The Internet Protocol Journal, Volume 12, No.4*. Retrieved from http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_12-4/124_ssh.html

[18] Boss, E. (2012). *Evaluating implementations of SSH by means of model-based testing.* Bachelor Thesis. Radboud Universiteit.

[19] Song, D. X., Berezin, S. & Perrig, A.(2001) Athena: A novel approach to efficient automatic security protocol analysis*Journal of Computer Security, 2001* (Vol 9), pp. 47–74

Appendices

Appendix A

Message number translation

This table is based on the table found in RFC 4250 [10]. It describes the correlation between message numbers and the command that is associated with this number. All messages belonging to the transport layer (1 to 31 in the table) are being send by our mapper. The other numbers are added since they could potentially be received as ouput.

Message number	Command	Layer
1	SSH_MSG_DISCONNECT	Transport
2	SSH_MSG_IGNORE	Transport
3	SSH_MSG_UNIMPLEMENTED	Transport
4	SSH_MSG_DEBUG	Transport
5	SSH_MSG_SERVICE_REQUEST	Transport
6	SSH_MSG_SERVICE_ACCEPT	Transport
20	SSH_MSG_KEXINIT	Transport
21	SSH_MSG_NEWKEYS	Transport
30	SSH_MSG_KEXDH_INIT	Transport
31	SSH_MSG_KEXDH_31	Transport
50	SSH_MSG_USERAUTH_REQUEST	User Authentication
51	SSH_MSG_USERAUTH_FAILURE	User Authentication
52	SSH_MSG_USERAUTH_SUCCESS	User Authentication
53	SSH_MSG_USERAUTH_BANNER	User Authentication
80	SSH_MSG_GLOBAL_REQUEST	Connection
81	SSH_MSG_REQUEST_SUCCESS	Connection
82	SSH_MSG_REQUEST_FAILURE	Connection
90	SSH_MSG_CHANNEL_OPEN	Connection
91	SSH_MSG_CHANNEL_OPEN_CONFIRMATION	Connection
92	SSH_MSG_CHANNEL_OPEN_FAILURE	Connection
93	SSH_MSG_CHANNEL_WINDOW_ADJUST	Connection
94	SSH_MSG_CHANNEL_DATA	Connection
95	SSH_MSG_CHANNEL_EXTENDED_DATA	Connection
96	SSH_MSG_CHANNEL_EOF	Connection
97	SSH_MSG_CHANNEL_CLOSE	Connection
98	SSH_MSG_CHANNEL_REQUEST	Connection
99	SSH_MSG_CHANNEL_SUCCESS	Connection
100	SSH_MSG_CHANNEL_FAILURE	Connection

Appendix B

Full OpenSSH model

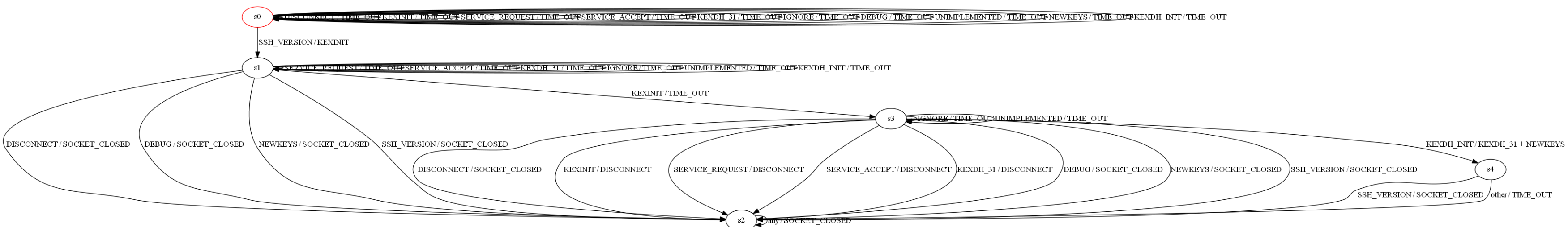

Figure B.1: An inferred model of the OpenSSH server implementation. At every transition what's left of the '/' is what the mapper send, to the right is the server's reply.

Appendix C

Full Bitvise model

Figure C.1: An inferred model of the Bitvise server implementation.At every transition what's left of the '/' is what the mapper send, to the right is the server's reply.