Radboud University

Master Thesis Computer Science

Inferring SSH state machines using protocol state fuzzing

Author:
P. Verleg
s3049701

First supervisor/assessor:
Dr. ir. E. Poll
`erikpoll@cs.ru.nl`

Second assessor:
Prof. dr. F.W. Vaandrager
`f.vaandrager@cs.ru.nl`

February 10, 2016

Abstract

In this thesis, a technique called protocol state fuzzing is used to infer state machines for six SSH servers. We found that all tested SSH servers implement a secure state machine. However, implemented state machines differ significantly. These variances allow anyone to effectively fingerprint the tested servers.

Our results show that opening multiple channels is not properly implemented on CiscoSSH and PowerShell. OpenSSH contains a bug which can result in connection closure after rekeying in some circumstances. Both Tectia and OpenSSH implement a liberal message acceptance policy in the first phase of the protocol. Such a liberal policy is unwise in this error-prone stage.

The SSH protocol defines three layers: the transport, user authentication and connection layer. These layers are executed sequentially, although rekeying invokes the transport layer during execution of higher layers. Besides the mentioned OpenSSH bug, our research has not revealed unwanted interaction between layers.

If the SSH RFC authors would have sketched a state machine, we would expect the standards to be less ambiguous and implemented state machines to have fewer differences. Most RFCs would benefit from added state information, and we would generally encourage RFC authors to append a reference state machines to protocol specifications.

Acknowledgements

I would like to thank my supervisor Erik Poll for his help and useful feedback while writing this thesis. His knowledge of SSH and ideas about inferring state machines helped to shape this research. Furthermore, I would like to express my gratitude to everyone who gave feedback when discussing the learning setup, including Ramon Janssen who offered various very useful hints.

Contents

1 Introduction

The SSH protocol, short for secure shell, is widely used to securely interact with remote machines. Alongside TLS and IPSec, SSH is amongst the most frequently used security suites [1]. Due to its significant user base and sensitive nature, flaws in the protocol or its implementation could have major impact. It therefore comes as no surprise that SSH has attracted scrutiny from the security community. This led to the discovery of cryptographic issues, of which plain text recovery is the most notable [1]. Nonetheless, the protocol has been able to withstand cryptographic analysis quite well, with only a handful of serious flaws found since the introduction of the version that is subject of this thesis: SSHv2. Throughout this thesis, we will use SSH to refer to the second version of the protocol.

Given the limited number of cryptographic issues that has been discovered, the specification has proven to be rather solid. An unblemished protocol specification alone, however, does not suffice. Several studies have looked into the protocol's implementations. The approaches used in these studies can be divided into two categories. White-box approaches, for example used in [2], rely on accessing and understanding an application's source code. Black-box testing approaches, on the other hand, examine an implementation without any prior knowledge of the internal workings [3].

Fuzzing (also known as fuzz testing) is a black-box testing approach in which unexpected input data is sent to a system under test (SUT) in the hope that this triggers anomalies. These abnormalities could expose security vulnerabilities [3]. Different fuzzing techniques are available, and they serve different purposes. Providing a system with a longer input than is expected, for example, is a frequently-used method to detect buffer overflow errors.

In this thesis, we will be fuzzing on the *order* of otherwise correctly-formed messages. This technique is known as protocol state fuzzing [4]. We will infer a state machine by returning the SUT responses to an off-the-shelf learning algorithm. A proper state machine forms the basis of a solid and secure protocol implementation. It should allow all transitions as defined by the grammar of the protocol, and react appropriately to input that is not within the happy flow of a protocol's execution. In a thorough fuzzing exercise, forbidden state machine transitions should be included [5].

Once a state machine has been inferred, security-related logical flaws are usually easily spotted by an auditor with some knowledge about the protocol [4]. An example of a logical flaw is exchanging user credentials before an encrypted connection has been established. Although spotting logical errors is relatively easy, checking for full compliance with the standards is much harder, since this requires a reference state machine which is used to judge every transition. In the case of SSH, the architecture and basic components are defined in four RFCs [6, 7, 8, 9]. Together with their numerous extensions, the standards total to several hundreds of pages, and no reference state machine is provided.

Besides security-related logical flaws, inferred state machines can show quirks such as superfluous states. Although these might not be directly exploitable, OpenBSD auditors illustrate why these small bugs should be resolved: "we are not so much looking for security holes, as we are looking for basic software bugs, and if years later someone discovers the problem used to be a security issue, and we fixed it because it was just a bug, well, all the better"[1].

Knowledge of the implemented state machine also allows type and version fingerprinting. It could be argued, however, that this has limited impact since the protocol's first message's comment string is frequently used to announce this information. This is the case for all tested implementations.

The goal of this research is to conduct a structured and thorough analysis of the state machines used in popular SSH implementations. The results of this structured analysis will provide insight in implementation decisions and potentially implementation flaws. Furthermore, we will reflect on the usage of state machines and their relation to RFCs. We will therefore try to answer the following research question:

> Can protocol state fuzzing be used to reveal incompatibilities or flaws in SSH servers, and in what way should state-related information be described in the RFCs?

Related work on SSH. Most security-related research has been centred around the security primitives in the SSH protocol. Formal analysis has been performed on the key exchange phase, which revealed no serious issues [10]. Problems with information leakage via unencrypted MACs were identified [11], which led to mandatory rekeying at certain stages in the latest RFCs. A security analysis showed possibilities for plain text recovery when a block cipher in cipher block chaining (CBC) mode was used [1]. This led OpenSSH to prefer counter mode to CBC mode in parameter negotiation. The binary packet protocol, which provides confidentiality and integrity to all exchanged messages, has been analysed in [12]. The resulting model would have been able to correctly identify the CBC-mode issues in SSH. It revealed no further vulnerabilities.

Effort has also been made to verify the correctness of SSH software implementations (for example in [2]). Academic researchers, however, have so far focussed more on the theoretical aspects than on implementations of the protocol.

[1] `http://www.openbsd.org/security.html`

Related work on protocol state fuzzing. Active learning techniques have been successfully used to infer state machines of EMV bank cards [13], electronic passports [14] and hand-held readers for online banking [15]. Furthermore, implementations of TCP [16] and TLS [4] have been subject to protocol state fuzzing. Results ranged from interesting insights to more serious implementational flaws. In case of TCP, fingerprinting possibilities for a remote operating system were found. Performing state fuzzing on TLS revealed security flaws in three different implementations. The first steps towards protocol state fuzzing on SSH were taken in [17], from which the general approach was the starting point for this research. Furthermore, some non-state based fuzzers for SSH servers are available online[2].

Organization. An outline of the SSH protocol will be provided in Chapter 2. Chapter 3 covers background information on Mealy machines, the learning algorithm and techniques used for inferring state machines. The experimental setup is discussed in Chapter 4. Chapter 5 contains analysis of the inferred state machines and provides some observations on their relation to the RFCs. Our main conclusions are summarized in Chapter 6, as well recommendations for future research.

[2]For example, Backfuzz, which is available at `https://github.com/localh0t/backfuzz/`.

2 Secure shell

This chapter covers the SSH protocol in more depth. We will briefly look at SSH's history in Section 2.1. A global architecture outline can be found in Section 2.2, and the security guarantees that the protocol aims to provide in Section 2.3. In order to familiarize the reader with the messages being exchanged, a typical protocol run will be described in Section 2.4.

2.1 History

The development of SSHv1 started in 1995 when Finnish researcher Tatu Ylönen drafted the first version of the software and its protocol at the Helsinki University of Technology. It was later standardized by the Internet Engineering Task Force, but quickly succeeded by SSHv2 because it contained design flaws that could not be fixed without losing backwards compatibility. Most notably, it was discovered in 1998 that an attacker could inject malicious packets into an SSHv1 encrypted stream, which might allow execution of arbitrary code [18].

Worldwide adaptation of the second version of the protocol was hampered because Ylönen released it with a more restrictive license. However, its usage quickly took off after the OpenBSD Project released their own implementation: OpenSSH. With over eighty percent market share in 2008, OpenSSH has remained the most popular implementation ever since [1].

2.2 Architecture

The architecture of SSH is laid out in RFC 4251 [6]. It follows a client-server paradigm consisting of three components. These can be seen as the layers of the protocol, although outer layers do not wrap inner layers. Instead, they are distinguished by the ranges of their message numbers. RFC 4250 [19] summarizes the numbers and symbolic names used in the protocol. The three main components are as follows:

- The *transport layer protocol* (RFC 4253 [8]) forms the basis for any communication between a client and a server. It provides confidentiality, integrity and server authentication as well as optional compression.

- The *user authentication protocol* (RFC 4252 [7]) is used to authenticate the client to the server. Strictly speaking, this component is optional and can be omitted if no user authentication is needed. In practise, however, user authentication is almost universally required.

- The *connection protocol* (RFC 4254 [9]) allows the encrypted channel to be multiplexed in different channels. These channels enable a user to run multiple processes over a single SSH connection. For example, a user could request terminal emulation on one channel and a file transfer on another, with both processes using the same connection.

An overview of the different SSH components is shown in Figure 2.1. Note that the shown SSH implementation relies on a TCP/IP stack. While this is the most common scenario, SSH can in theory be run over any reliable data stream [6, p. 3].

Different layers are identified by their message numbers. Numbers up to 19 are reserved for the transport layer, 50 to 79 for the user authentication layer and the remainder up to 127 for the connection layer. Numbers from 128 upwards can be used for client protocols and local extensions. This range does not seem to be commonly used, with OpenSSH not mentioning support for any of them on their specifications page[1].

The message numbers will form the basis of the state fuzzing used in this thesis, as we will see in Chapter 4. The SSH protocol is especially interesting because outer layers do not encapsulate inner layers. This means that different layers can interact. One could argue that this is a less systematic approach, in which a programmer is more likely to make state machine-related errors.

On the lowest level, data are sent as binary packets over a networking stack. This format is known as the binary packet protocol and has been defined in [8]. Every binary packet starts with the sequence number and length of the packet and padding, followed by the actual payload and padding. The aforementioned parts can be encrypted. A message authentication code (MAC) completes the packet, and is based on the entire unencrypted packet. A schematic packet overview is shown in Figure 2.2.

The disadvantages of the chosen compute-then-encrypt-and-MAC[2] method have been extensively discussed in [11]. As a result of this approach, SSH

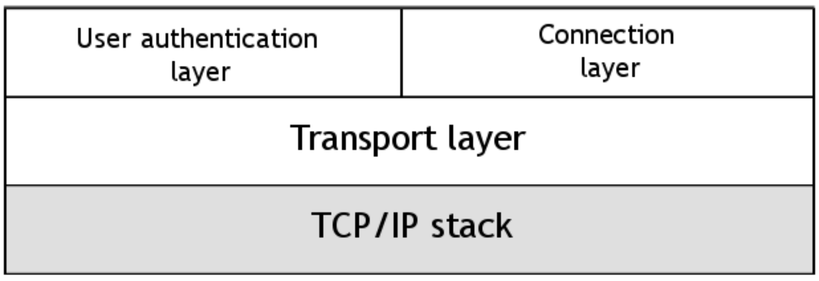

Figure 2.1: SSH protocol components running on a TCP/IP stack.

[1]`http://www.openssh.com/specs.html`

[2]Note that this is neither a encrypt-then-MAC or a MAC-then-encrypt approach. Note that only encrypt-then-MAC is good practise.

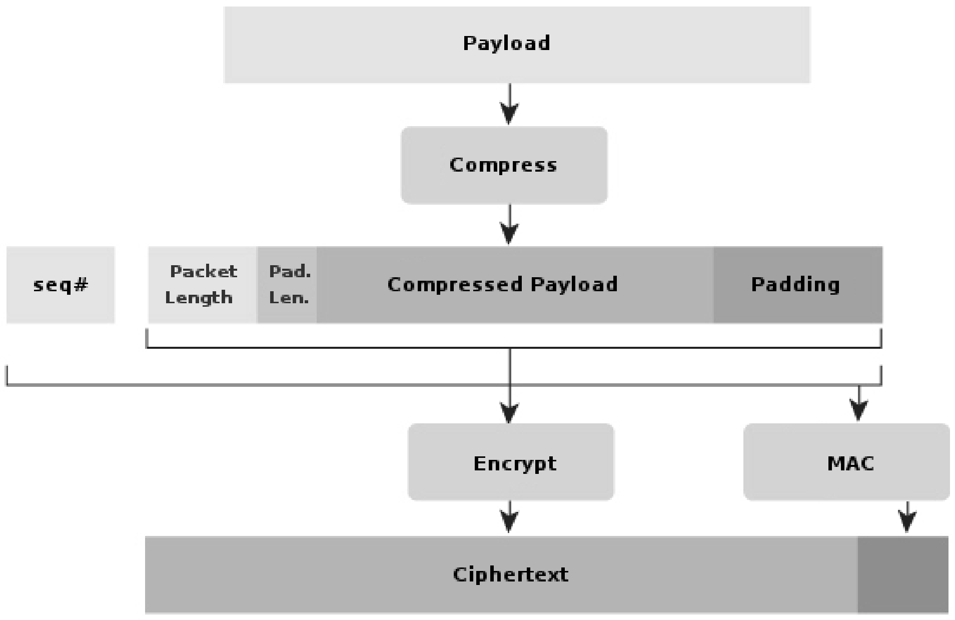

Figure 2.2: SSH packet construction with compression, encryption and message authentication.

does not provide integrity on the cipher text. Furthermore, MACs are prone to sensitive information leakage. Timely rekeying and refraining from using block ciphers in CBC mode will prevent information leakage and chosen-cipher attacks. Using an encrypt-then-MAC approach would have been a better solution.

2.3 Security

SSH aims to provide various security guarantees [6, section 9]. Confidentiality is provided by means of encryption. Message integrity is ensured by the MACs, while message numbers ensure the integrity of the sequence of messages. The end-to-end encryption is based on pseudo-random keys which are securely negotiated during key exchange and discarded on session termination. SSH can use many commonly-used ciphers such as AES, 3DES and Blowfish. Developers are free to choose which ones they support, although 3DES in CBC mode must be supported by all implementations to ensure interoperability. A "none"-cipher is provided if no encryption is needed, although this obviously violates the intended security guarantees. Some implementations, such as OpenSSH, have opted to not supporting the "none"-cipher [20]. Because of the chosen compute-then-encrypt-and-MAC approach, the choice of encryption has no effect on the MAC at all (see Figure 2.2)

The client and the server will negotiate the used ciphers according to their preferences. Both client and server can pick their most preferred cipher, provided that the other party knows how to handle it. Custom ciphers can also be used, as long as both parties support them.

The extensible and flexible negotiation process is one of the main improvements over SSHv1. The negotiation results in non-symmetric parameters: the cipher used for packets travelling from the client to the server need not be the same as the cipher used for messages the other way around. The same asymmetric approach is taken the negotiation of MAC and compression algorithms. Messages can therefore, for example, be compressed in one direction but not in the other.

As we have seen in the packet description, MACs are provided with each packet to detect integrity violations. Because the session key and sequence number are part of the MAC, it effectively prevents replay attacks. As with the cipher suites, a variety of MAC algorithms can be used, with HMAC-SHA1 being required by all implementations. Because the 32-bit message sequence number wraps when it overflows, a sequence number might reoccur within a session. This would make the MAC prone to sensitive information leakage because they are based on the unencrypted payload [11]. The RFC requires rekeying after 2^{28} outgoing packets. This guarantees a fresh and unpredictable session key before the sequence numbers wrap, which in turn prevents information leakage by the MAC.

SSH provides server authentication by public key exchange. Diffie-Helman group1-sha1 and group14-sha1 must be supported by all implementations. If one of these Diffie-Helman key exchanges is used, perfect forward secrecy is ensured. Unlike TLS, SSH does not provide a certificate chain-approach to verifying a host's public key. Although RFC 4255 [21] does provide a means to retrieve a host's public key via DNS, in general it is unspecified how public keys should be checked. Moreover, the protocol does not specify how they should be stored by the client [20, p. 55]. The lack of specified public key infrastructure can be seen as a security risk, since users might not check key validity. However, the Network Working Group believed ease of use was critical to end-user acceptance, with this solution being preferable over not having any encryption at all [6, p. 4]. Authentication of the client can be achieved my multiple means, such as password, host-based or public key. The server can be configured to require a combination of methods, and plug-in modules allow for custom authentication methods.

SSH is not designed to prevent every attack one could think of. Password attacks and attacks based on the underlying network fall outside its scope. The protocol has not been designed to guarantee the absence of covert channels to pass information [6, p. 20]. Furthermore, while an outside observer is unable to decipher traffic, an observer can still analyse network traffic, resulting in traffic fingerprints [6, p. 21]. Although SSH supports traffic padding, it has been shown that these fingerprints can in practise be used to reveal the type of applications that are run over the SSH connection [22].

2.4 Protocol run

We will describe a typical protocol run in this section. By doing so, we hope to familiarize the reader with the different messages and layers defined by the RFCs. For each layer, we will include:

- The *happy flow*. This is the sequence of messages that is expected by the server and leads to a satisfying result for that layer.
- A *visual representation* of this happy flow as a Mealy machine[3].
- The *security definition* that defines what kind of behaviour would result in a secure state machine.

Appendix A provides a mapping between the (shorter) message names used in this thesis and the official RFC message names.

2.4.1 Transport layer

All messages follow the binary packet protocol, with the noteworthy exception of the first message: the exchange of the SSH version and comment string. Since this message does not use the binary packet protocol, it does not have a message number. The comment part of the message is generally used to provide the product name and version. It should be noted that this decision can be objected to from a security perspective, since it allows for rapid identification of vulnerable software versions.

After the client and server have exchanged version information, parameter negotiation begins with the KEXINIT message. For each of the negotiable parameters, a list is included in which is ordered by the implementation's preference. Before the KEXINIT step completes, the "none" options for encryption, message authentication and compression are used. The KEXINIT message is therefore never encrypted and can be replayed, but this does not allow for vulnerabilities. No new messages need to be sent to confirm the chosen parameters, because both parties assume that the other party will use the most preferable option that is understood by them both.

Both parties are now set to engage in the actual key exchange using the negotiated algorithm. Message numbers 30 to 49 are reserved for this purpose, and these numbers can be reused for different authentication methods [19, p. 5]. A typical key exchange will require two round-trips (which includes parameter negotiation), while three are required in the worst-case scenario [8, p. 3].

[3]Refer to Section 3.2 for more on Mealy machines.

The key exchange leads to a shared secret and an exchange hash [8, p. 19], with the latter serving as session identifier. Encryption and authentication keys are also derived from this hash. The keys are used from the moment the NEWKEYS command has been issued by both parties. Either the user authentication or the connection protocol can now be invoked by the client.

Happy flow. We define the happy flow as the sequence that results in a successfully parameter negotiation, key exchange and authentication protocol request. The trace KEXINIT; KEX30; NEWKEYS; SR_AUTH typically triggers this behaviour. A visual representation of the happy flow can be found in Figure 2.3[4]. *Security definition.* We consider an transport layer state machine secure if there is no path from the initial state to the point where the authentication service is invoked without exchanging and employing cryptographic keys.

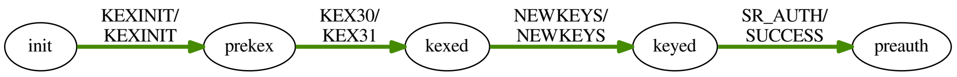

Figure 2.3: The happy flow for the transport layer.

2.4.2 User authentication layer

The user authentication layer is centred around the user authentication request and its response. The RFC define four authentication methods (password, public-key, host-based and none), which are all sent using the same message number [7, p. 8]. The authentication request includes a user name, service name and authentication data.

The service name that is included in the authentication request refers to a protocol: the authentication protocol provides the authentication service and the connection protocol provides the connection service. Since the protocol is already at the user authentication stage, the only sensible service to request is the connection service.

The provided authentication data includes both the authentication method as well as the data needed to perform the actual authentication, such the password or public key.

Happy flow. The happy flow for this layer is defined as the sequence that results in a successful authentication. The queries UA_PW_OK and UA_PK_OK achieve this for respectively password or public key authentication[5,6]. A

[4]More information on the used Mealy machine representation can be found in Section 3.2.

[5]The reason for not implementing host-based authentication can be found in Section 4.3.

[6]Appendix A provides for a mapping between the used names and the RFC message

visual representation of the happy flow can be found in Figure 2.4.

Security definition. We consider a user authentication layer state machine secure if there is no path from the unauthenticated state to the authenticated state without providing correct credentials.

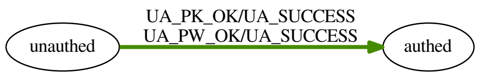

Figure 2.4: The happy flow for the user authentication layer.

2.4.3 Connection protocol

The connection protocol provides multiplexing in order to run different communication channels over the same SSH connection. The connection protocol's requests can be either global or channel-related. The only global request that is defined in the RFC is TCP forwarding [9, p. 16]. Channel-related requests include opening and closing channels, requesting a process over that channel and sending data to a requested process. Channel requests can invoke processes such as terminal emulation, X11 window forwarding and file transfer.

Happy flow. Because the connection protocol offers a wide range of functionalities, it is hard to define a single happy flow. Requesting a terminal is one of the main features of SSH and has therefore been selected as the happy flow. This behaviour is typically triggered by the trace CH_OPEN; CH_REQUEST_PTY. A visual representation of the happy flow can be found in Figure 2.5.

Security definition. Its hard to define which behaviour would result in a state machine security flaw in this layer. We will therefore take a more general approach and look at unexpected state machine transitions that can point towards potential implementation flaws.

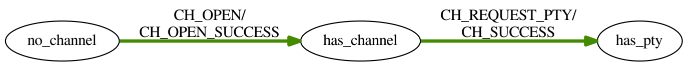

Figure 2.5: The happy flow for the connection layer.

names.

3 Preliminaries

This chapter covers background information about techniques, algorithms and formalisms relevant to this thesis. Different methods of state machine inference will be discussed in Section 3.1. Inferred state machines are represented as Mealy machines. The basic properties of Mealy machines are the subject of Section 3.2. Section 3.3 will cover the L* learning algorithm, the off-the-self algorithm used for inferring state machines.

3.1 Inferring state machines

There are two fundamentally different ways in which state information can be derived for a certain protocol. In a passive approach, messages originating from two or more communicating systems are analysed to derive information. Since most protocols communicate over a network, these messages usually originate from network traffic. On the other hand, an active approach injects or alters messages to reveal protocol information. Both techniques can use learning algorithms to infer information, but the former has a focus on statistical methods to categorize messages, while the latter uses fuzzing.

Both techniques have successfully been applied. The passive approach was used to extract state machines from real-world network traffic by making use of both package heads and body in [23]. This information was fed to a clustering algorithm, and its output was used to construct a state machine. The authors specifically mention that their inferred state machine might not include all transitions of the actual protocol's state machine. This is an observation that is true for any passive technique, since a clustering algorithm can only learn from traces that occur in the observed network traffic. Transitions that are not part of the collected traces can thus never be predicted. Another fundamental restriction to passively inferring state machines is the inability to learn protocols that communicate over an encrypted channel, at least without having access to the security keys. Encryption will mask the information needed to infer the state machine.

These restrictions do not apply to active learning systems. Active learning approaches are less generic than passive learning systems. Since they need to take part in the protocol communication, some knowledge of the protocol is a prerequisite and therefore a tailor-made protocol participant has to be created. Encryption stresses the need for tailor-made systems even more. Firstly, because encryption prevents any automatic deduction of message format to use later on. Secondly, because encrypted systems have a natural protection against fuzzing, since the encryption-handling outer layers of a protocol will discard any message that is damaged or malformed [5].

In general, the more extensive the protocol, the more custom logic needs to be added to the fuzzer to successfully participate in the protocol's execution. In fact, it has been suggested that one of the reasons that the Heartbleed bug in OpenSSL's TLS implementation remained undiscovered for so long is due to the fact that only tailor-made fuzzers could be used [5]. In this thesis, we will use a tailor-made fuzzer to actively participate in the protocol's execution. By fuzzing on the order of otherwise correctly-formed messages we will deduct a state machine, which will be represented as a Mealy machine.

3.2 Mealy machines

Mealy machines provide a representation for the internal state machine of an SSH implementation. We will refrain from covering all the available literature on Mealy machines, and instead discuss the necessary information needed for understanding the relevant representation restrictions and interpreting this thesis' results. For further reading, please refer to [24], in which the relation between Mealy machines and inferring state machines is more thoroughly discussed and from which we will adopt terminology.

A Mealy machine can be represented as a tuple $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$ in which I and O represent the input and output symbols, Q is the set of states, with q_0 being the starting state. The two transition functions denoted by δ and λ represent respectively the edges ($Q \times I \to Q$) and output ($Q \times I \to O$) with respect to a certain state and input. To spot logical flaws, a graphical representation is usually better-suited. Such a graphical representation typically depicts states as nodes and state transitions as edges. A Mealy machine for which $\lambda(q_0, i') = o'$ and $\delta(q_0, i') = q_0$ will therefore show an edge from q_0 to q_0 with label i'/o', meaning that input i' results in output o' without changing the state. An example of a graphical representation for a simple Mealy machine is shown in Figure 3.1.

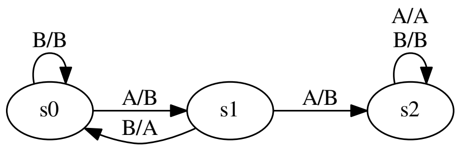

Figure 3.1: A Mealy machine with three different states.

A Mealy machine is a deterministic finite state transducer, implying it is both deterministic and finite. The transition functions therefore allow only a single output for a given state and input, and the number of input symbols and states is finite. Therefore, Mealy machines do not allow effective representation of infinitely large buffers in a protocol. Consider a state q in which an input i' always results in o', but as soon as i'' is processed the SUT sends o'' once for each i' that has been processed so far. A Mealy machine representation of this behaviour would need at least as many states as the (infinitely large) buffer. We will observe buffer-like behaviour in some SUTs. The adaptations made to still be able to model these SUTs as Mealy machines are described in Section 4.4.

3.3 L* learning algorithm

The L* algorithm is used to build a model of the state machines and decide which queries need to be sent to a SUT. The usage of L* to infer state machines has first been demonstrated in [25]. In this thesis, we will use L* as an off-the-self algorithm and thus only briefly cover its properties.

Given a certain input alphabet, L* dictates which input to send to the SUT. The resulting output is used to infer states and decide which subsequent input symbols to send. Between the different traces, the SUT is reset to the initial state. In case of SSH, this is effectively done by terminating the connection and initiating a new one.

If the SUT's state machine is deterministic and finite, the algorithm produces a state machine hypothesis in a finite number of queries. The number of needed queries depend on the complexity of the state machine that is to be inferred. Unfortunately, neither the state machine's complexity nor its deterministic or finite properties are known beforehand. Running the algorithm on non-terminating or non-deterministic SUTs might result in non-termination of the L* implementation. In our testing setup, we automatically detect such cases (Section 4.4.3).

As soon as a first hypothesis has been formed, it is passed to a testing oracle. This oracle will search for a counterexample. Since our testing-approach is black-box, no oracle algorithm can guarantee that existing counterexamples will be found in finite time. Section 4.1 describes the considerations for the selection of our oracle algorithm.

If the testing oracle finds a counterexample, the counterexample and hypothesis are used as a starting point for further learning. This process repeats until no more counterexamples are found. The final hypothesis is assumed to correctly define the SUT's minimal state machine [25, p. 12].

4 Experimental setup

This chapter will cover the setup used to infer the state machines. We provide a general setup outline in Section 4.1. The tested SSH servers are described in Section 4.2, which were queried with the alphabet described in Section 4.3. Section 4.4 will cover the challenging SUT behaviour faced when implementing the mapper, and the adaptations that were made to overcome these challenges. Section 4.5 will discuss the relation between state machines for individual layers and the state machine of the complete SSH protocol. The conventions on visualisation of the inferred state machines are described in Section 4.6.

Throughout this chapter, an individual SSH message to a SUT is denoted as a *query*. A *trace* is a sequence of multiple queries, starting from a SUT's initial state. Message names in this chapter are usually self-explanatory, but a mapping to the official RFC names is provided in Appendix A.

4.1 Components

Our setup consists of three components: a learner, mapper and SUT. In order to infer state machines, the leaner needs to query the SUT. We use LearnLib[1], a Java library for automata learning, as a basis for our learner. The L* algorithm implemented in LearnLib can only handle abstract messages, and can neither output nor process actual SSH network traffic. The mapper translates abstract message representations to well-formed SSH messages. A graphical representation of our setup is shown in Figure 4.1.

Drafting a mapper proved to be non-trivial. Because the mapper needs to participate in an SSH session, it should be able to engage in processes such as key exchange and authentication, and has to support features such as

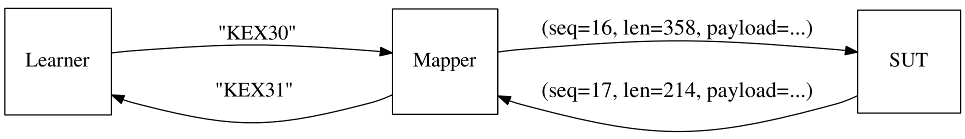
Learner
"KEX30"
"KEX31"
Mapper
(seq=16, len=358, payload=...)
(seq=17, len=214, payload=...)
SUT

Figure 4.1: The setup consists of a learner, mapper and SUT.

[1]More information on LearnLib is available at `http://learnlib.de/`

encryption and compression. For this thesis, an open source SSH implementation was used as a starting point for the implementation of the mapper. We used the Python-written Paramiko package[2] because its code is relatively well-structured.

The convenience of using existing code comes at a cost. Since Paramiko features a state machine of its own, normal execution of the Paramiko code will result in messages being sent, received, interpreted and rejected without an explicit order to do so. We therefore altered the package so that it only sends queries when explicitly told to by the learner, and accepts all response messages. The message numbers as defined by the RFC [19] are used to map response messages to an abstract representation that the learner can interpret.

Although all messages are accepted and returned to the learner, the mapper still needs a minimal state machine of its own. The mapper state will only be changed if necessary for participation in the protocol's execution. In other words: the mapper only processes and records the contents of a SUT's response when strictly needed. For example, when a SUT sends an DISCON or DEBUG message, the mapper does nothing but return the abstract representation to the leaner. Keeping the mapper's message interpretation to a minimum allows the learner to interpret the SUT's state machine without the mapper cluttering the results. Some response message, however, do need to be interpreted by the mapper in order to participate in the SSH protocol. These messages can be found in Table 4.1. The mapper's own state consists of the information saved upon receiving these messages.

The mapper communicates with the leaner over a socket connection and implements a simple protocol. It interprets abstract message representations (for example, "KEXINIT") as well as a special reset command, which is used to revert the SUT to the initial state. This is effectively done by terminating the connection to the SUT and initializing a new one.

Message	**Influence on mapper state**
KEXINIT	Saves SUT's parameter preferences[3].
KEX31	Saves the exchange hash resulting from key exchange.
NEWKEYS	Takes in use new keys for all outgoing messages[4].
CH_ACCEPT	Saves the channel identifier, used in some queries[5].
any	Saves the sequence number, used for the UNIMPL query[6].

Table 4.1: State-changing responses implemented by the mapper. These combinedly result in the mapper's state.

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

As soon as the learner builds a state machine hypothesis, it uses the equivalence oracle to check the hypothesis' adequateness. We used a random-walk algorithm, which uses 2000 randomly constructed traces containing ten to fifteen queries each. Although this does not guarantee model correctness (no oracle can be absolutely conclusive in a black-box testing setup), it gives us reasonable confidence that the model is correct. More complex state machines will need other algorithms[7] or parameters to find counterexamples, but we deem these parameters adequate for the state machines found in Chapter 5.

4.2 SUTs

Six SSH servers have been tested. They can be found in Table 4.2. Unfortunately, no recent figures are available on their market share. There is little doubt, however, that OpenSSH is the market leader. The OpenSSH project reported[8] over 80% market share for their server in 2008 and it has been the default server on many UNIX-based operating systems for years. DropBear is an alternative to OpenSSH and was designed to be a drop-in replacement for low resource systems. It is the server of choice for routers running OpenWRT. Bitvise and PowerShell are Windows-only clients, with the latter providing Window's PowerShell as emulated terminal. The company of SSH's founding father Tatu Ylönen markets Tectia, which is available on various platforms. Cisco's high-end networking hardware ships with their IOS operating system, which has a build-in proprietary SSH client.

Name	**Developer**	**Version**	**Platform**	**License**
OpenSSH	OpenBSD Project	6.9p1-2	UNIX-based	BSD
DropBear	Matt Johnston	2014.65-1	UNIX-based	MIT
Bitvise	Bitvise	6.45	Windows	Proprietary
PowerShell	N Software	6.0.5732	Windows	Proprietary
Tectia	SSH Comm. Sec.	6.4.12.353	Various	Proprietary
CiscoSSH	Cisco	1.25	Cisco IOS	Proprietary

Table 4.2: The SSH implementations tested in this thesis.

[3]The parameters that must be supported according to the RFCs to ensure interoperability are used if no KEXINIT has been received.

[4]Silently ignored when key exchange has not yet been completed.

[5]Zero is used if no CH_ACCEPT has been received.

[6]Zero is used if no message has been received.

[7]An overview of LearnLib's equivalence oracles can be found on `https://learnlib.github.io/learnlib/maven-site/0.9.1/apidocs/de/learnlib/eqtests/basic/package-summary.html`

[8]Data are available on `http://www.openssh.com/usage/graphs.html`

In our setup, we ran OpenSSH and DropBear on the same operating system as the mapper. CiscoSSH ran on a Catalyst 3550 switch, and other SUTs were executed in a virtual machine. Of course, SUTs can run on any system as long as it accepts incoming connections from the mapper. However, querying over localhost is preferable because it reduces timing differences.

In their default configuration, all of the SUTs support (only) SSH version 2. Furthermore, all SUTs support our alphabet defined in Section 4.3, with the exception of CiscoSSH, which does not support public key authentication[9].

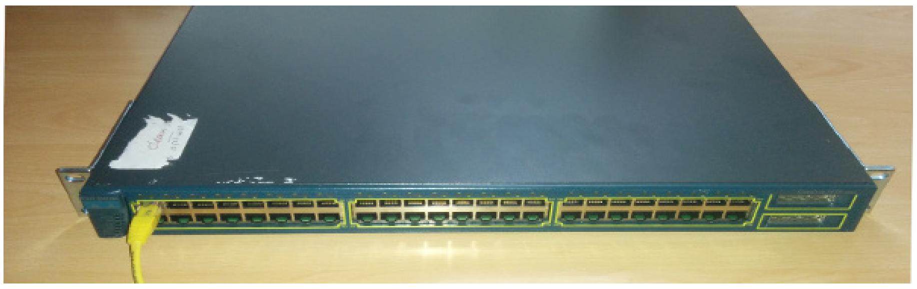

Figure 4.2: The Catalyst 3550 used for testing.

4.3 Alphabet

Learning time tends to grow rapidly as the input alphabet grows. It is therefore important to focus on messages for which interesting state-changing behaviour can be expected. As a general principle, we therefore chose not to query protocol messages that are not intended to be sent from a client to a server[10].

Applying this "outgoing only" principle to the transport layer results in the messages of Table 4.3[11]. The only message that is out of the ordinary is GUESSINIT. This is a special instance of KEXINIT for which `first_kex_packet_follows` is enabled [8, p. 17]. Our mapper can only handle correct key guesses, so the wrong-guess procedure as described in [8, p. 19] was not supported. When needed, SUTs were configured to make this guess work by altering their cipher preferences. The SSH version and comment string (described in Section 2.4.1) was not queried because it does not follow the binary packet protocol.

[9]IOS supports public key authentication since version 15, while the Catalyst switch runs on IOS 12.2(46)SE.

[10]Applying this principle to the RFC's messages results in not including SERVICE_ACCEPT, UA_ACCEPT, UA_FAILURE, UA_BANNER, UA_PK_OK, UA_PW_CHANGEREQ, CH_SUCCESS and CH_FAILURE in our alphabet.

[11]A mapping to the official RFC names is provided in Appendix A.

Message	**Description**
DISCON	Terminates the current connection [8, p. 23]
IGNORE	Has no intended effect [8, p. 24]
UNIMPL	Intended response to an unimplemented message [8, p. 25]
DEBUG	Provides other party with debug information [8, p. 25]
KEXINIT	Sends parameter preferences [8, p. 17]
GUESSINIT	A KEXINIT after which a guessed KEX30 follows [8, p. 19]
KEX30	Initializes the Diffie-Hellman key exchange [8, p. 21]
NEWKEYS	Requests to take new keys into use [8, p. 21]
SR_AUTH	Requests the authentication protocol [8, p. 23]
SR_CONN	Requests the connection protocol [8, p. 23]

Table 4.3: Alphabet used to query the transport layer.

For the user authentication layer, applying our "outgoing only" principle results in just one message: the authentication request [7, p. 4]. Its parameters contain all information needed for authentication. As stated in Section 2.4.2, four authentication methods exist: none, password, public key and host-based. Our mapper supports all methods except the host-based authentication because various SUTs lack support for this feature. As shown in Table 4.4, both the public key as well as the password method have a OK and NOK variant which provides respectively correct and incorrect credentials.

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

Table 4.4: Alphabet used to query the user authentication layer.

The connection protocol allows the client to request different processes over a single channel. Our mapper only implements requesting terminal emulation because availability of other processes depends heavily on a SUTs configuration. Moreover, little security-relevant information is expected to be gained by thoroughly testing other process requests. Combining this premise with the aforementioned "outgoing only" principle resulted in the alphabet of Table 4.5.

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

Table 4.5: Alphabet used to query the connection layer.

4.4 Challenging SUT behaviour

Creating a mapper proved to be difficult, with three types of SUT behaviour being especially challenging. We will elaborate on how our mapper handles issues with regard to non-determination, receiving multiple responses and non-termination.

4.4.1 Non-determinism

The learner expects that the a SUT behaves deterministically. In reality, however, the SSH protocol and its implementations exhibit non-deterministic behaviour. Sources of this behaviour can be divided into three categories:

1. SSH's *protocol design* is inherently non-deterministic. Firstly, because underspecification leads to multiple options for developers, from which one can be selected in a non-deterministic manner. Secondly, because non-deterministic behaviour directly results from the specifications. An example of the latter is allowing to insert DEBUG and IGNORE messages at any given time.

2. *Response timing* is a source of non-determinism as well. If a SUT does not reply within a predefined time-out, the mapper assumes that no response will follow. If the time-out is of similar order of magnitude as the SUTs response time, timing variances will cause some queries to result in a response while others do not. In general, we want the time-out to be significantly higher than the average response time[12].

3. Other *timing-related quirks* can cause non-deterministic behaviour as well. Some SUTs behave unexpectedly when a new query is received

[12]Note that if the time-out is lower than the response time, the SUT behaves deterministically but the resulting model will note adequately describe the SUT's state machine.

shortly after the previous one. For example, a trace in which a valid user authentication is performed within five milliseconds after an authentication request on DropBear can cause the authentication to (wrongly) fail.

With regard to the *protocol design*: although the protocol allows for quite some non-deterministic constructs in theory, most SUTs seem to behave deterministically when it comes to what message they send. Tectia and Bitvise sometimes send seemingly random IGNORE and DEBUG messages, but these could be easily filtered without influencing the structure of the state machine.

With regard to the *timing*: timing issues proved to be difficult to tackle. To detect non-determinism, the learner has been extended with a SQLite-based trace and response log. For any (sub)trace, the learner checks if the SUT's response matches earlier responses. In case of non-determinism, an exception is thrown and manual investigation is needed. This manual investigation typically leads to changing the delays and time-outs. For debugging purposes, the mapper also accepts complete traces as input. For example, the command "20 KEXINIT KEX30 NEWKEYS reset" repeats the key exchange 20 times, so that variances in responses can be easily spotted. Because responses to some queries (such as authentication requests) need substantially more time, the mapper allows the time-out to be set based on the type of query.

4.4.2 Multiple responses

In a Mealy machine, an input from a given state leads to a single output. In practise, however, a SUT might respond with more than one message. For example, Tectia sends three messages (IGNORE, UA_BANNER and UA_SUCCESS) in response to a single successful authentication request. If our mapper would only read the first message, other messages would appear as if they are responses to subsequent queries.

The mapper has been altered to successfully deal with multiple response-behaviour. The abstract representation returned to the learner is the concatenation of all responses received within the message time-out period. The response to the aforementioned authentication query on Tectia will thus be presented to the learner as IGNORE+UA_BANNER+UA_SUCCESS. By using this method, we make sure that there are no subsequent messages in the socket pipeline when querying for new responses.

Waiting for multiple responses comes at a cost. Whereas a single response-approach allows to return the message to the learner as soon as a response is received, our altered mapper has to wait for subsequent messages. The waiting times quickly accumulate and result in a significantly slower learning process. To speed up learning, the learner has been altered so that it can query the trace and response log mentioned in Section 4.4.1. If a cached trace is available, the mapper need not be queried. This did not only improve learning speed, it also provides a way to store learned traces to quickly resume the learning process later on.

An important observation has to be made with regard to the asynchronous nature of message exchange in SSH [26]. Neither the client nor the server needs to wait for a certain timeslot in order to transmit. Moreover, there is no requirement to receive and process queued messages before transmitting new ones. This results in a protocol in which, for example, both the server and the client can have the impression that they sent their version number before receiving the version number of their counterpart. This is an inherent property of the SSH protocol, and receiving multiple responses does not change this behaviour. This did not prove to be a major limitation in our setup, since this behaviour does not alter the structure of inferred state machines.

4.4.3 Non-termination

Mealy machines are unable to adequately model buffers in a protocol. Automata such as register machines are able to effectively model buffers, but are not supported by L*. Buffers should therefore be removed at the mapper-level.

We have encountered buffers in two occasions. Firstly, some implementations buffer certain responses, such as the ACCEPT message, when in key re-exchange. As soon as rekeying completes, these queued messages are released all at once. This leads to a NEWKEYS message (indicating rekeying has completed), directly followed by all buffered messages. Buffer-behaviour can also be observed when opening and closing channels, since a SUT can close only as many channels as have previously been opened.

Buffers are hard to detect since LearnLib does not release intermediate results while building a state machine hypothesis. In other words, an observer has no way to know whether the learner is expanding on a buffer and will consequentially never terminate. To detect this behaviour, we let the learner regularly extract and display the response alphabet size from the trace and response log.

The multiple responses resulting from buffers are concatenated into one message as described in Section 4.4.2. An example of such a message is NEWKEYS+ACCEPT+ACCEPT+ACCEPT. Consequently, the response alphabet will quickly grow. As soon as this happens, the mapper gives a warning and the learning process can be halted to investigate the buffer.

Generic countermeasures were also added to correct buffering behaviour. If a response is identical to the one received before within the same message time-out, an asterisk is appended. Subsequent identical messages are discarded. The NEWKEYS+ACCEPT+ACCEPT+ACCEPT response will thus be returned to the learner as NEWKEYS+ACCEPT*. From the learners perspective, this effectively removes the buffer.

A SUT can only close as many channels as previously have been opened, which faces the learner with a buffer as well. We therefore restricted the number of simultaneously open channels to one. The mapper returns a custom response CH_MAX for every subsequent CH_OPEN. These CH_MAX messages are filtered from the state machine representation.

4.5 Inferring individual layers

Initially, the setup was used to infer a state machine in which all three layers (transport, user authentication and connection) of the SSH protocol were combined. There are multiple reasons as to why this has proven to be infeasible.

Firstly, the number of states tends to grow quickly when combining all layers. Given the timing restrictions we had to keep in mind to prevent non-determinism, inferring and validating the combined state machines could easily end up taking days. This would not have been an insurmountable problem if the implementations behaved fully Mealy machine-compliant but, as described in Section 4.4.3, in many cases they do not. Unexpected buffers in the protocol's implementation become increasingly difficult to detect and prevent as different layers start interacting.

The lack of choke points between protocol layers makes SSH an especially difficult protocol to infer. This means that there is no point after which it can be safely assumed that previously ran layers will no longer interact. Rekeying poses the most notable example, in which a sequence of three transport layer messages could be sent at any given time during execution of a higher layer.

In one of our earlier attempts to infer a combined state machine, our setup was able to detect rekeying, but unable to detect that rekeying preserves the state in the higher layers. A schematic overview of this behaviour is shown in Figure 4.3. A model in which the state is not preserved after rekeying is erroneous, since it implies that the server has no information on authentication and channels any more as soon as rekeying completes.

Although a human observer will easily notice the rekeying pattern (KEXINIT; KEX30; NEWKEYS), LearnLib's L* cannot recognize nor anticipate on the repetition of such patterns. It is trivial to see that this behaviour results in an exploding number of states as soon as the number of higher-layer states increases. In order to still make some observations on the security-wise interesting rekey operation, we add a REKEY to the user authentication and connection alphabet. This operations implements the messages KEXINIT; KEX30; NEWKEYS as an atomic operation. By performing an entire rekey procedure at once, we can deduce when rekeying succeeds and whether this correctly preserves state.

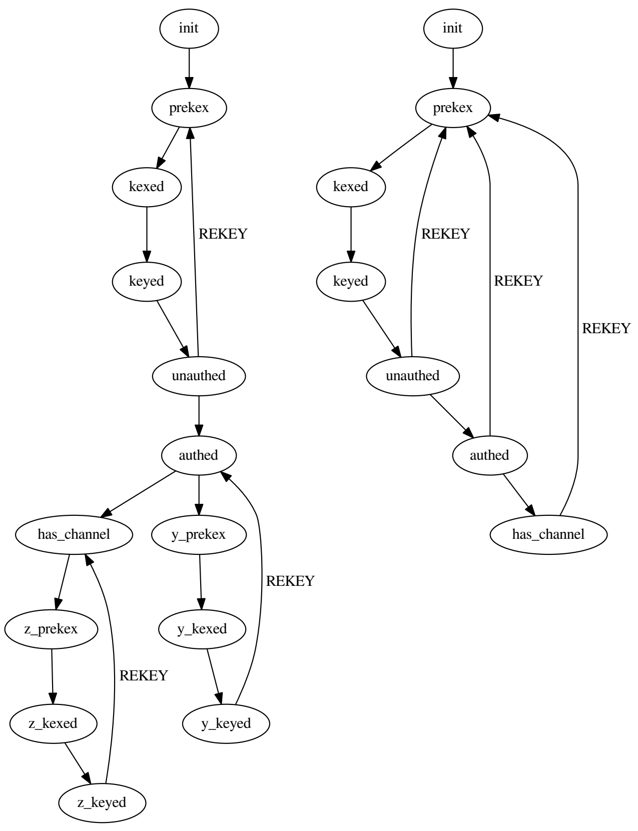

Figure 4.3: State machines showing state preserving (left) and non-state preserving (right) rekeying. A adequate model should correctly preserve state.

One could wonder what the combined state machine for the three different layers would look like. On secure implementations, the transport and user authentication layer need to complete before the connection layer starts. It will thus not be a parallel composition of three layers running independently, as would be the case when, for example, three different OSI-layers would be considered. The key re-exchange, however, causes the protocol to not be entirely sequential either. Without providing an airtight formal definition, the combined state machine SM_{ssh} would look something like:

$$SM_{ssh} = SM_{trans}; (SM_{trans} \times SM_{auth}); (SM_{trans} \times SM_{conn})$$

4.6 State machine visualisation

As soon as our setup finishes learning, it outputs the state machine as a GraphViz[13] file. Various alterations are applied to improve readability while keeping the machines unequivocal. State machines in various stages of this visualisation process have been attached in Appendix B for the interested reader.

A slightly altered version of Python's Pydot[14] package is used to merge labels of edges that span between the same nodes. Subsequently, OTHER and ANY representations are added in order to merge queries that result in similar responses. To keep the machine unambiguous, a node can have at most one outgoing edge with an OTHER label. If no confusion is possible, the ANY/NO_CONN label will be omitted.

Queries using the IGNORE and DEBUG message are removed from the representations because they never resulted in a state change. The same applies to the UNIMPL message on all SUTs except DropBear and Tectia. Green edges were added to denote each layer's happy flow[15].

[13]More information on GraphViz is available at `http://www.graphviz.org/`

[14]Pydot is available at `https://pypi.python.org/pypi/pydot`

[15]Happy flows are defined in Section 2.4.

5 Results

This chapter discusses the state machines inferred by our testing setup. The transport layer, user authentication layer and connection layer will be discussed in Sections 5.1-5.3. For each layer, we will discuss the state machines in order of increasing complexity. A few concluding observations will be made in Section 5.4.

5.1 Transport layer

The transport layer is arguably the most interesting layer from a security perspective because it involves parameter negotiation and key exchange. The SUTs were queried with the alphabet described in Table 4.3. The inferred state machines discussed in this section reveal major implementation differences. The majority of the observed differences has one of the following causes:

1. *Rekeying* during the first execution of the transport layer is either allowed or not allowed.

2. Responses to *authentication service request* while in key exchange are handled differently. The RFC explicitly disallows sending an acceptance response to any service request while in key re-exchange [8, p. 19]. Some developers have implemented this requirement by buffering responses: acceptance responses are kept back until key re-exchange has completed. Other developers have chosen to simply discard these responses during rekeying.

3. The presence of *unresponsive and superfluous states.* SUTs in an unresponsive state ignore all incoming messages. Superfluous states are states which serve no real purpose: if they would be removed, the layer would still be working properly[1].

Learning statistics for the transport layer are shown in Table 5.1. Note that equivalence queries have not been saved to the query log, and are therefore not included in the statistics tables.

[1]It is hard to give a formal definition of a superfluous state, since a reader's interpretation is needed to see whether a state serves a *useful* purpose.

All inferred state machines in this section are secure with regard to the security definition given in Section 2.4.1.

SUT	**Traces**	**Queries**	**Time ([h:]m:s)**[2]
OpenSSH	895	5584	16:57
DropBear	562	2634	8:38
Bitvise	766	3091	16:03
PowerShell	866	5413	19:14
Tectia	1224	8203	1:01:05
CiscoSSH	522	2689	51:41

Table 5.1: Statistics for the transport layer extracted from the query log.

[2]Time per query differs because of the differences in time-outs. In general, time-outs for SUTs running on the same machine as the learner could be lower than time-outs for remote or virtualized servers.

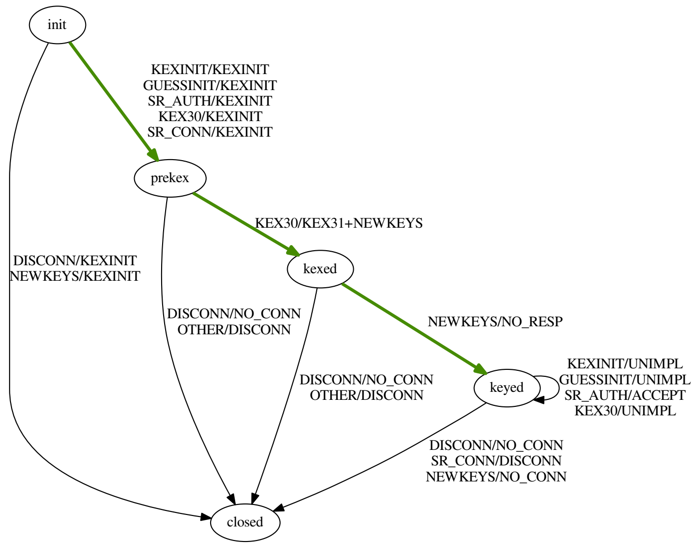

Figure 5.1: Inferred state machine of the transport layer for OpenSSH 6.9p1-2.

OpenSSH (Figure 5.1) implements a simple state machine. It does not allow rekeying in this phase of the protocol, and therefore does not need a rekeying response buffer. OpenSSH is extremely liberal when it comes to the parameter negotiation: the KEXINIT message need not to be sent at all. OpenSSH seems to do a proper job when it comes to guessing all of the used parameters for the client-to-server connection. OpenSSH's developers have been notified of this issue, but have not (yet) shed light on why they chose to implement such a liberal acceptance policy during parameter negotiation.

The parameter negotiation behaviour in OpenSSH is in line with Postel's law, which has been formulated in an early version of the TCP standard [27, p. 21] and boils down to "be conservative in what you send, be liberal in what you accept from others". It has been argued that this approach is unwise in a security-sensitive context [26]. Interesting enough, OpenSSH does clearly not follow Postel's law when it comes to rekeying, since it is one of the few clients that disallow rekeying at this stage of the protocol.

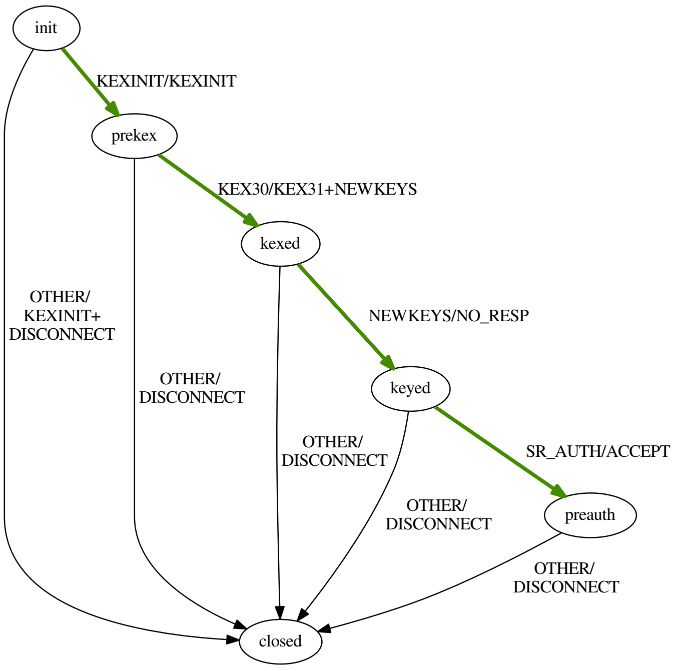

Figure 5.2: Inferred state machine of the transport layer for CiscoSSH 1.25.

CiscoSSH's state machine (Figure 5.2) resembles OpenSSH's, although it does requires a KEXINIT message from the client before starting the key exchange. In other words, a transition from the "initial" state to the "prekex" state is not possible without mutually exchanging preferred parameters. The GUESSINIT is not supported.

Contrary to OpenSSH, the user authentication service can only be requested once, and the connection is closed on subsequent requests. Although the RFC does not mention limiting the number of SR_AUTH messages, this is a more restrictive implementation.

Just like OpenSSH's state machine, CiscoSSH does not allow rekeying in this stage of the protocol. Cisco's developers have not responded to inquiries as to why they chose this behaviour.

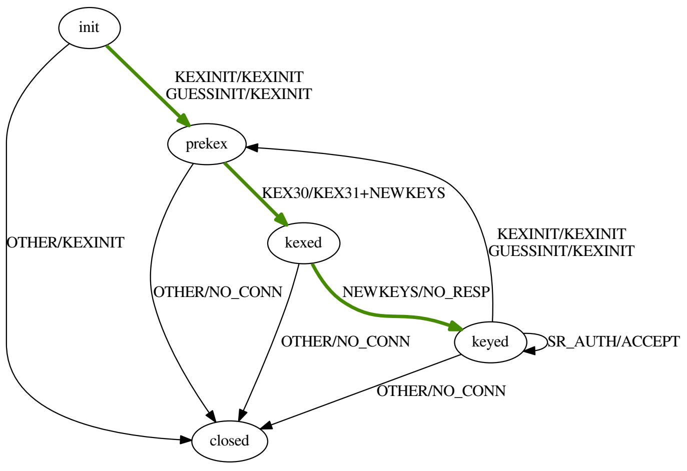

Figure 5.3: Inferred state machine of the transport layer for Dropbear 2014.65-1.

DropBear's state machine (Figure 5.3) also resembles OpenSSH's, but allows rekeying after the initial key exchange has completed.

Just like OpenSSH and CiscoSSH, DropBear does not buffer ACCEPT messages. In reply to my inquiry on the reason behind this, DropBear's developer Matt Johnston stated that according to him, buffering these messages is not allowed by the RFCs. Indeed, the description in [8, p. 19] is ambiguous, and can be interpreted as a requirement to temporary withhold these messages, *or* as a requirement to discard them.

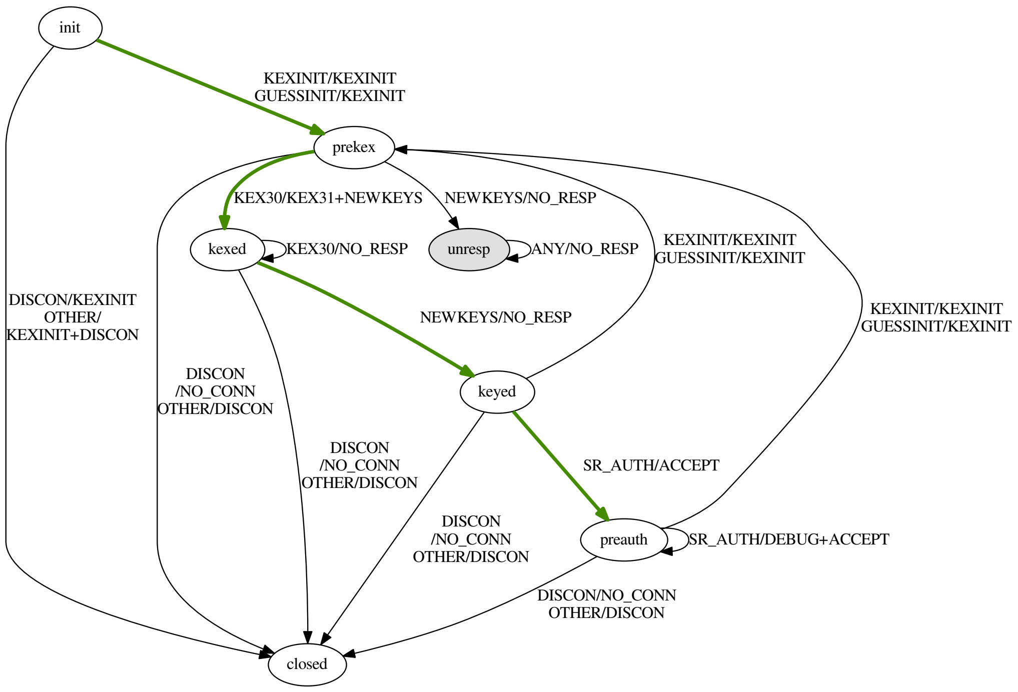

Figure 5.4: Inferred state machine of the transport layer for Bitvise 6.45.

Bitvise's state machine (Figure 5.4) stops responding when a NEWKEYS directly follows parameter negotiation. We have tested this unresponsive state for a denial-of-service scenario, but it seems that the "unresp" state closes connections after one minute. The DEBUG message the SUT sends on a second authentication request results in a separate state "preauth".

In response to an inquiry about the unresponsive state, Bitvise developers started an investigation which led to the conclusion that "the reason for the unresponsive state is in our asynchronous, component-based, message-passing architecture; and the way we use this to implement a kex-handler-agnostic transport layer". Although Bitvise agrees that an error message would have been better in this case, they also note that "other mechanisms, such as various time-outs, should continue to operate in this circumstance, so it seems an error message would be the only benefit." They conclude that the unresponsive state results from an "architectural limitation", but it does little harm.

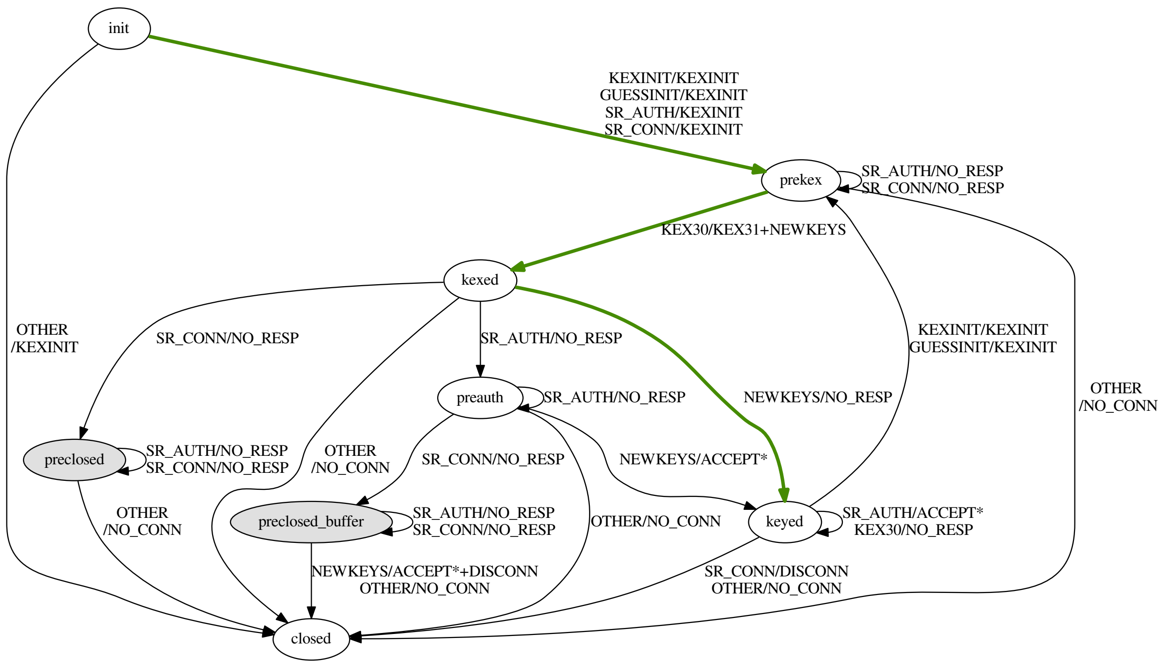

Figure 5.5: Inferred state machine of the transport layer for PowerShell 6.0.5732.

The state machine of PowerShell (Figure 5.5) is more complex because of its buffering behaviour and the existence of multiple superfluous states. The grey superfluous states are caused by PowerShell's odd interpretation of the SR_AUTH and SR_CONN messages. As can be seen in Figure 5.5, these messages can cause state changes but frequently do not result in a response. The resulting "preclosed"-states do not seem to serve any purpose. Other SUTs simply close the connection upon unexpectedly receiving these messages.

Recall that the asterisk in ACCEPT* is used to indicate an arbitrary non-zero number of identical responses to a single query. These responses originate from buffered responses to SR_AUTH messages during key re-exchange.

PowerShell's developers responded to these findings by stating that they would further investigate the particularities when time allows.

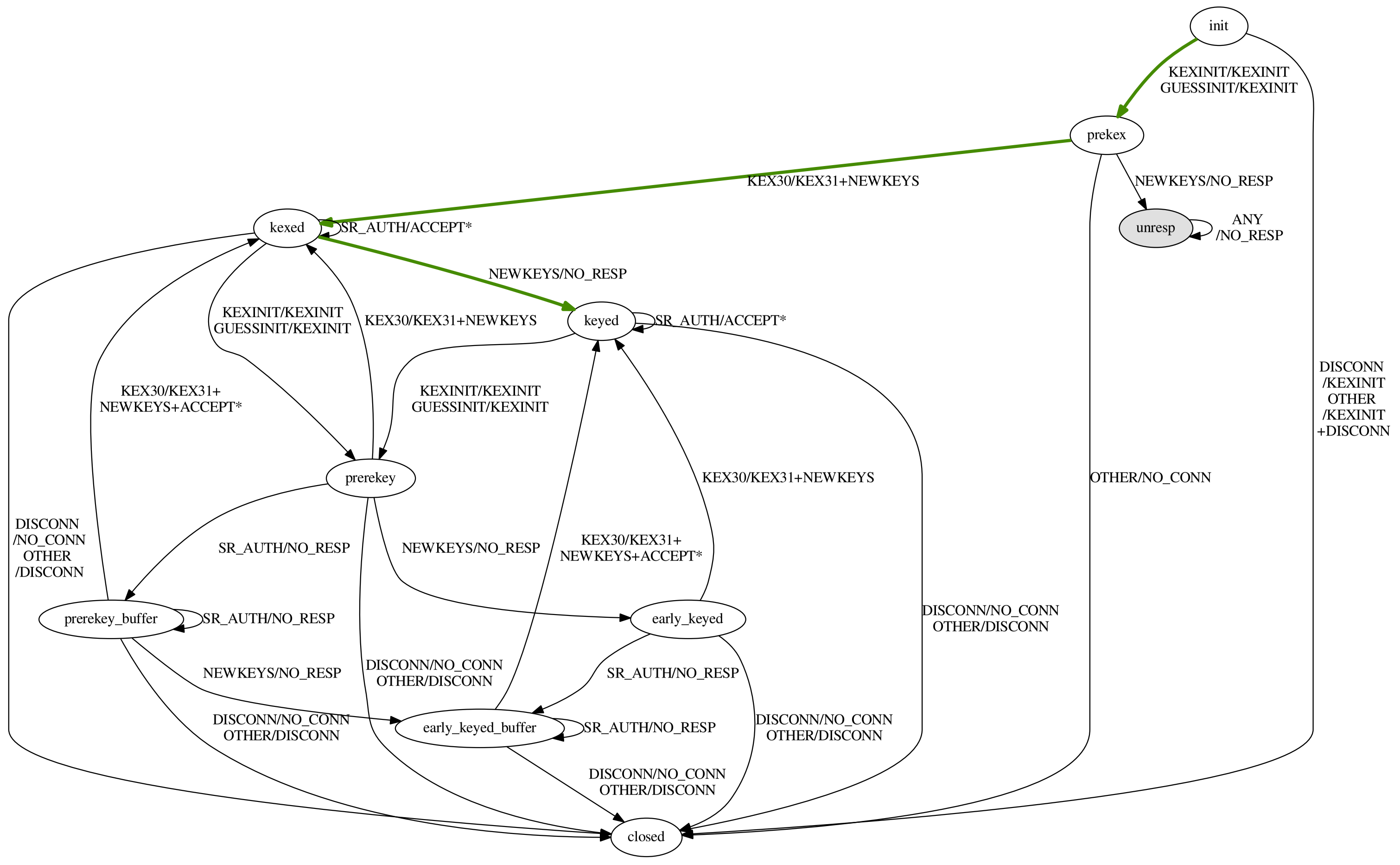

Figure 5.6: Inferred state machine of the transport layer for Tectia 6.4.12.353.

Tectia's state machine (Figure 5.6), just like DropBear's, contains an unresponsive state when NEWKEYS is sent too early. Whether this is caused by the same architectural limitation cannot be answered, since Tectia's developers have not taken the opportunity to respond to the results.

Tectia accepts the NEWKEYS message before renewed keys have actually been exchanged. The SUT's extremely liberal rekey sequence acceptance policy in combination with buffering behaviour results in a tangled web of interrelationships. Actual rekeying is performed as soon as a KEXINIT, KEX30 and a NEWKEYS have been received, regardless of the order. While this does not directly lead to a vulnerability, this behaviour is neither wise nor allowed by the RFC, which clearly state that "key exchange ends by each side sending an NEWKEYS message" [8, p. 21].

Given this behaviour, it is likely that the state machine has been implemented as a combination of variables which combinedly form the machine's state. A state machine could also be implemented using a single state variable. It should be noted that no single method is superior, although the multi-variable approach can result in an cluttered state machine representation with many seemingly unnecessary states.

Note that the majority of states in Figure 5.6 have not been marked as superfluous. Although the state machine implementation would be better off with fewer states, all of the states (besides the "unresp") allow recovering to a state like "kexed" or "keyed". These states are the result of the aforementioned implementation decisions, but are strictly speaking not superfluous.

5.1.1 Concluding remarks on the transport layer

The inferred transport layer state machines differ substantially. While causes for these changes fall into three major categories described on page 26, many more subtle variances can be observed in both structure of the state machine and the used response messages. Differences in implemented response messages are especially notable when considering unexpected messages: some SUTs send UNIMPL, others send NO_RESP, DISCON, or simply immediately close the connection. Querying with the GUESSINIT message did not result in peculiar behaviour, and all SUTs expect CiscoSSH support it.

All state machines are secure, since none of the state machines show a path to the authentication service request without a proper exchange of cryptographic keys[3].

From a security perspective, a liberal message acceptance policy in the key exchange phase is undesirable. OpenSSH has a liberal policy when it comes to parameter negotiation because it also accepts other messages than KEXINIT. Tectia allows rekeying in arbitrary message order. This liberal behaviour could easily lead to an obscure combination of states in which errors are easily introduced and hard to detect. Other SUTs are more restrictive what kind of messages they accept.

Both Tectia and PowerShell buffer ACCEPT messages during rekeying, resulting in extra states with ACCEPT* transitions. Buffering ACCEPT messages seems to be implied by the specifications, which state that a client or server "must not send any messages other than" the ones provided on [6, p. 19]. Given that key re-exchange may take place at any given moment, it is likely that the RFC authors meant holding back other messages -such as ACCEPT-rather than discarding them. However, the specifications remain ambiguous and are interpreted differently by different readers. This is directly reflected by the response from DropBear's developer, but also observable by looking at how developers have translated the RFCs to state machines.

It is hard to say which state machine is the "best". This depends on one's interpretation of the RFCs, and on whether the most compliant implementation is always preferable. We would recommend developers to not allow rekeying and not buffer ACCEPT messages. Choosing a simple state machine over a strict implementation of specifications seems sensible in this error-prone stage of the protocol, since it results in a simpler state machines which, in turn, allows for fewer state-related bugs. For this reason, we would argue that CiscoSSH's state machine (Figure 5.2) is preferable.

[3]The security definition can be found in Section 2.4.2.

5.2 User authentication layer

Inferring state machines for the user authentication layer proved to be relatively easy compared to inferring the transport layer because the implemented state machines were less complex. Although all SUTs implement a secure user authentication layer with regard to the security definition given in Section 2.4.2, various differences can be observed.

The used alphabet consists of the REKEY query[4] and the queries described in Table 4.4. Throughout this chapter, the KEYOK response message was used as a shorthand for the sequence of the three expected result messages (KEXINIT; KEX31; NEWKEYS) in a key re-exchange.

The initial state for this layer is "unauthed". In order to reach this state, the sequence KEXINIT; KEX30; NEWKEYS; SR_AUTH has been executed in between every trace. This performs a key exchange and requests the user authentication service. Besides the atomic REKEY message, no other transport layer messages are used in this layer in order to keep learning times feasible. Resulting learning statistics for the authentication layer can be found in Table 5.2.

SUT	**Traces**	**Queries**	**Time (m:s)**
OpenSSH	78	618	4:38
DropBear	107	969	9:34
Bitvise	103	993	6:57
PowerShell	87	672	6:39
Tectia	95	728	11:17
CiscoSSH	105	758	3:44

Table 5.2: Statistics for the authentication layer extracted from the query log.

[4]More information on the REKEY message is available in Section 4.5.

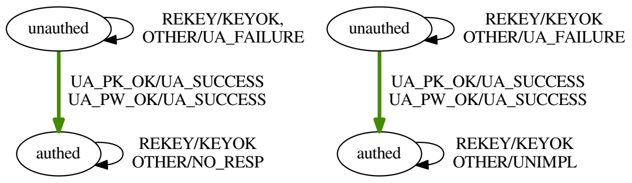

Figure 5.7: Inferred state machine of the user authentication layer for DropBear 2014.65-1, Tectia 6.4.12.353 and PowerShell 6.0.5732 (left) and Bitvise 6.45 (right).

The structure of the state machines for Bitvise, Tectia, DropBear and PowerShell (Figure 5.7) are almost identical and arguably as simple as the authentication layer could theoretically be. Their state machines have two states, and rekeying is allowed during the entire protocol.

Tectia sends a UA_BANNER [7, p. 7] as soon as the SUT connects, which can be used to inform connecting clients of relevant (legal) information. This message has been filtered from the representation in Figure 5.7, because it would falsely gives the impression that this message is a response to the first query, while it might have already been sent[5].

[5]Section 4.4.2 provides more information on this behaviour.

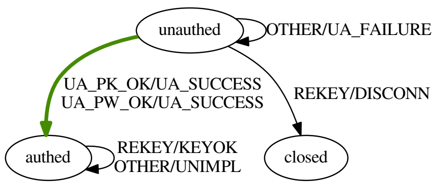

Figure 5.8: Inferred state machine of the user authentication layer for OpenSSH 6.9p1-2.

OpenSSH's state machine (Figure 5.8) does not allow rekeying until user authentication has been completed. Besides that, the state machine is identical to Bitvise's state machine. OpenSSH and Tectia do not allow requesting authentication with a changed username. In other words: subsequent authentication requests have to involve the same user. This behaviour is not defined by the RFCs, and initially led our mapper to infer incorrect state machines. OpenSSH's debug logs revealed this issue, after which our mapper has been altered to use identical usernames. We do not know the reason for this implementation decision, and the OpenSSH developers did not elaborate on it.

As soon as a SUT successfully authenticates, OpenSSH transmits a global request featuring their own protocol extension (`hostkeys-00@openssh.com`), which is used to inform clients of all the server's protocol host keys[6]. This is a OpenSSH-specific extension using the "`@`" notation as defined in [19, p. 9].

[6]A complete list of OpenSSH's deviations and extensions can be found on `https://anongit.mindrot.org/openssh.git/plain/PROTOCOL`

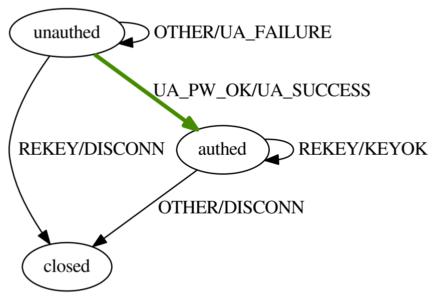

Figure 5.9: Inferred state machine of the user authentication layer for CiscoSSH 1.25.

Just like OpenSSH, CiscoSSH does not allow rekeying until the user has been successfully authenticated. As stated in Section 4.2, public key authentication is not supported in the tested version op CiscoSSH. Just like Cisco's state machine in the transport layer, its inferred state machine for the authentication layer is the most restrictive one: unexpected authentications requests in the "authed" state result in a closed connection.

5.2.1 Concluding remarks on the user authentication layer

The state machines for the user authentication layer are less complex than their transport layer counterparts. The state machines differ in two aspects: whether they allow rekeying in an "unauthed" state and how implementations deal with authentication attempts in the "authed" state. OpenSSH and CiscoSSH do not allow rekeying for unauthenticated users. Other SUTs seem to correctly preserve their state after rekeying. Most implementations send no response or an UNIMPL message when faced with unexpected authentication attempts, while CiscoSSH immediately closes the connection. Furthermore, OpenSSH and Tectia do not allow to switch usernames between subsequent requests.

All inferred state machines implement a secure state machine: none of the SUTs allow reaching an authenticated state without providing correct credentials. Furthermore, none of the SUTs show unresponsive or superfluous states[7]. Bitvise's state machine (Figure 5.7) most closely follows the RFCs. However, just like in the transport layer, developers might have good reasons to disable rekeying until user authentication.

[7]The security definition can be found in Section 2.4.2.

5.3 Connection layer

The connection layer allows a user to request different processes over a single SSH connection[8]. Querying the SUTs with the alphabet described in Table 4.5 and the REKEY query resulted in the learning statistics shown in Table 5.3. The initial state for this layer is the "no_channel" state. The key exchange sequence (KEXINIT; KEX30; NEWKEYS; SR_AUTH) is followed by successful authentication (UA_PW_OK) in between every trace in order to reach this state.

SUT	**Traces**	**Queries**	**Time (m:s)**
OpenSSH	540	5115	10:50
DropBear	490	4595	9:55
Bitvise	294	2315	6:43
PowerShell	289	2343	6:39
Tectia	160	1109	6:00
CiscoSSH	264	1810	5:40

Table 5.3: Statistics for the connection layer extracted from the query log.

Looking at the RFC [9], we notice that it gives few clues on how to handle unexpected messages. Compared with the other layers, the connection layer RFC focusses more on what is allowed rather than what is disallowed. A possible explanation could be that this layer is less security critical, since it does not involve exchanging keys or passwords. We would argue that the connection layer is, more than the other layers, underspecified. For example, it does not specify how to deal with requesting a second process over a single channel (CH_REQUEST_PTY) or sending data after an end-of-file message (CH_EOF). We will see that each implementation takes it own approach in these cases.

[8]More information on the connection layer is available in Section 2.4.3.

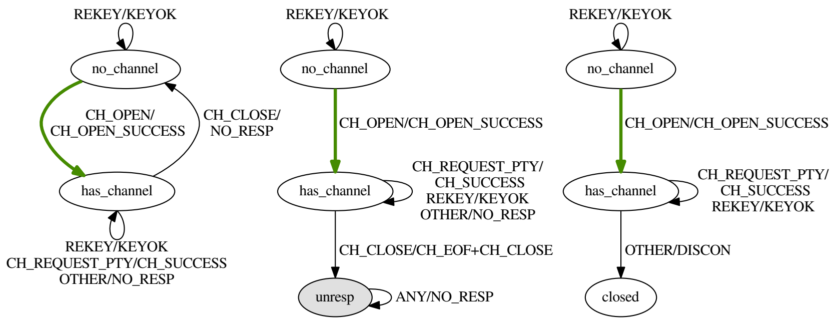

Figure 5.10: Inferred state machine of the connection layer for Tectia 6.4.12.353 (left), PowerShell 6.0.5732 (middle) and CiscoSSH 1.25 (right).

PowerShell, Tectia and CiscoSSH implement a fairly simple state machine (Figure 5.10). They support rekeying and requesting a terminal, and do not respond to other messages at all. The latter is in line with the RFC, which does not specify a response to CH_DATA, CH_EDATA, CH_WINDOW_ADJUST or CH_EOF.

PowerShell and Tectia have a rather liberal acceptance policy in two aspects. Firstly, they accept multiple terminal emulation requests over a single channel. Secondly, they preserve their state when receiving an end-of-file message (CH_EOF). They could also have chosen to take measures (for example, close the connection) if a client sends data after CH_EOF, but they did not.

PowerShell and CiscoSSH do not support opening multiple channels[9], and jump to an unresponsive or closed state as soon as a channel has been closed. This deviation from the RFCs does no justice to the intention of the connection layer, which is to multiplex multiple channels into a single connection [9, p. 5]. Developers of PowerShell and CiscoSSH did not respond to queries about this bug, but it could be a deliberate decision to implement only part of the specification. After all, both SUTs are marketed to perform one specific task only: PowerShell's selling point is providing the Windows PowerShell terminal, while CiscoSSH provides management access to networking appliances.

[9]Technically, our learning setup can only infer that these SUTs do not support closing and subsequently opening another channel, since our mapper does not support simultaneously opened channels.

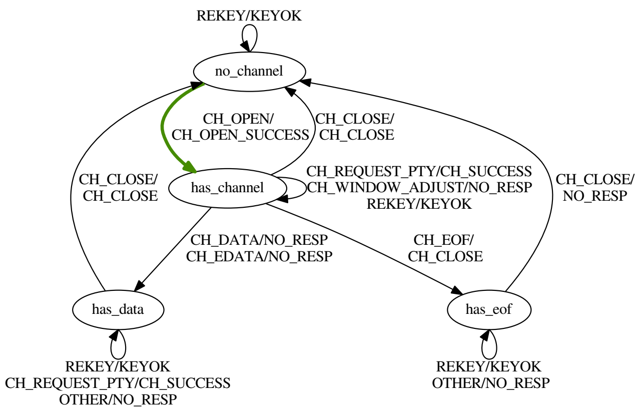

Figure 5.11: Inferred state machine of the connection layer for Bitvise 6.45.

The state machine of Bitvise (Figure 5.11) is different from the three aforementioned because it does not respond to any connection-layer messages after CH_EOF. Although the RFC does not specify what to do when data is received after an end-of-file, not responding to this data seems more in line with the intention of the client sending such a message. After all, if a server has to anticipate on receiving more data after an end-of-file message, the message serves no real purpose.

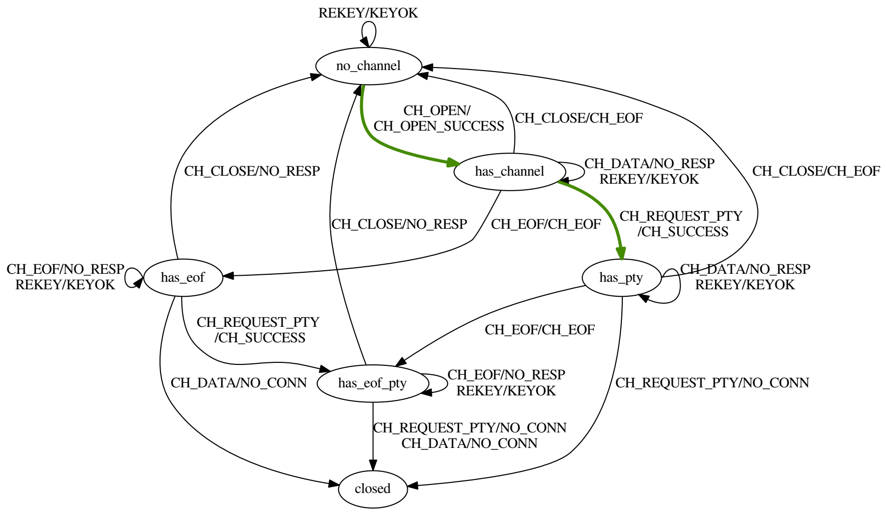

Figure 5.12: Inferred state machine of the connection layer for Dropbear 2014.65-1.

DropBear's state machine (Figure 5.12) looks more complex at first sight, which is caused by two implementation decisions. Firstly, just like Bitvise, DropBear does not respond on any connection layer messages when an end-of-file has been issued. Secondly, it allows only one terminal emulation request per channel. Non-compliance with these two restrictions results in a closed connection.

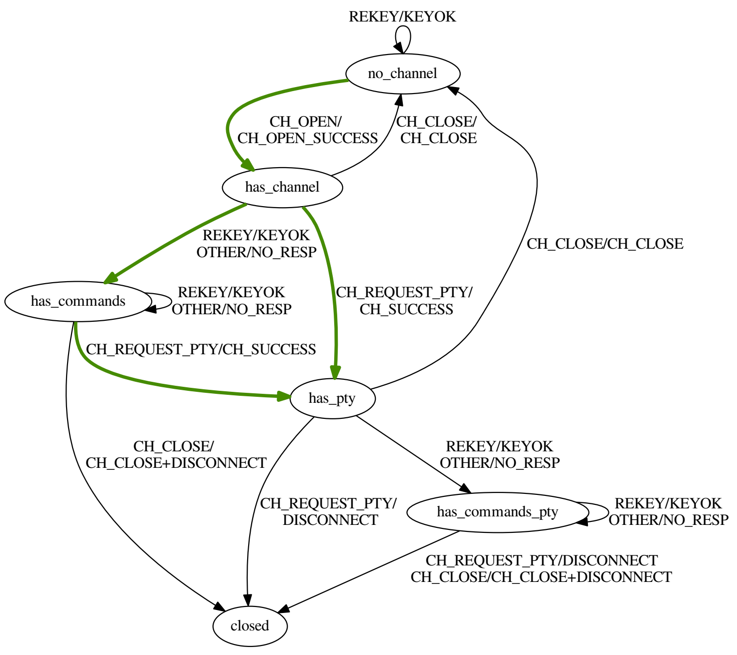

Figure 5.13: Inferred state machine of the connection layer for OpenSSH 6.9p1-2.

OpenSSH (Figure 5.13) also closes the connection on a second terminal emulation request for the same channel. Furthermore, its state machine seems to allow multiple channels, but only until as long as no non-service request query has been received yet. As soon as another query has been received, it changes to the “has_commands” state, after which the closing of a channel results in connection termination. The same behaviour can be observed after a terminal emulation request has been accepted, resulting in the “has_commands_pty” state. This behaviour is rather strange and cannot be explained using the RFC or OpenSSH’s debug logs. As a consequence of this behaviour, OpenSSH fails to close a channel after rekeying, and instead closes the entire connection. OpenSSH developers have been notified, but have so far not provided an explanation.

5.3.1 Concluding remarks on the connection layer

From a state machine security perspective, the transport and user authentication layer are more interesting than the connection layer. The connection layer could contain security vulnerabilities, but it is unlikely that a vulnerability can be revealed by interpreting a SUT’s state machine.

PowerShell and CiscoSSH do not seem to support multiple channels.

Rekeying is allowed by all SUTs, but for OpenSSH seems to result in closing the entire connection upon receiving a CH_CLOSE. This indicates a state machine-related bug.

It comes as no surprise that the fewer constructions are explicitly disallowed by the RFCs, the wider the ranger of resulting state machines. The RFCs for the connection layer almost entirely lack *explicit* description of state-related behaviour. Even with the limited number of requests[10]. we employed when inferring state machines for this layer, various ambiguities are revealed. The CH_EOF is exemplary, since the RFC only specify that it should be sent "when a party will no longer send more data to a channel" [9, p. 9]. Why this is useful and what the consequence is if data is sent afterwards is not explained.

Of the wide range of state machines, DropBear's interpretation of the RFCs is arguably preferable because it disallows dubious service request and end-of-file constructions.

5.4 State machines and RFCs

Many differences can be observed when comparing the inferred state machines. In fact, only in the authentication layer did we observe two identical state machines. Fingerprinting an unknown SSH server therefore seems to be a trivial task. One of the major reasons for these observed variances is underspecification in the RFCs. Underspecification can be the result of deliberately allowing multiple options, having ambiguously protocol descriptions, or not discussing certain possibilities at all.

Underspecification accounts for at least three observed differences amongst the inferred state machines: the buffering behaviour during key re-exchange, dealing with multiple services over a single channel and different handling of CH_EOF messages. Other differences seem to be caused by explicit choices which are not part of the RFCs, or are simply caused by programming errors.

We strongly believe that if the SSH protocol authors would have sketched a conceptual state machine while drafting the standards, the resulting RFCs would have been more clear, and inferred state machines would show fewer differences. Sketching conceptual state machines forces one to explicitly think about the impact of a protocol rule on the state machine. This might lead to omitting certain features (key re-exchange during transport layer would be a good candidate) or messages (the CH_EOF message can be omitted without any consequence).

Adding a reference state machine to the RFCs (ASCII-based or otherwise) has advantages. It would allow for easier interpretation by developers, and might result in simpler and more consistent state machine implementations.

[10]We restricted ourselves to terminal emulation, as explained in Section 4.3.

This might, in turn, ensure better compatibility and leave less room for state-related security vulnerabilities.

Although adding a reference state machine to the SSH protocol architecture would be a significant improvement, appending state machines might not be suitable for every protocol. It implies that the protocol behaves (almost) Mealy-machine compliant. While non-deterministic Mealy machines allow for more flexible constructions (such as the ones made with the "MAY" keyword), RFCs which allow a wide range of possible state machines might become hard to interpret. It is, however, our belief that adding a state machine to current and future RFCs would generally have added value.

Whether a Mealy machine is the best way to depict a state machine is open for debate. In general, we feel that one of the major advantages of Mealy machine representation is that it provides an interpretable overview in the blink of an eye. Other formalisms, such as register automata or the formalism used in [26] can be harder to interpret but do provide more expression power (for buffers and order of messages respectively). In the end, our choice for Mealy machines was based on a rather practical limitation: these are the only machines L* can infer.

6 Conclusions

In this thesis protocol state fuzzing was used to infer state machines for six SSH servers. Our setup (Chapter 4) uses a mapper to translate abstract queries from the learner to actual network packets that could be understood by an SSH server. Message numbers were used to convert response messages back to abstract representations.

Creating the mapper proved to be non-trivial. Handling non-determinism, non-termination and multiple responses were challenging (Section 4.4). The mapper was extended with a trance and response log to detect non-determinism and improve performance. Delays were added to the mapper, as well as means to handle multiple responses to a single query and detect buffers within those responses.

All of the tested SUTs implement secure state machines (Chapter 5). PowerShell and CiscoSSH do not support multiple channels (Section 5.3). Besides that, no incompatibilities were found. This does not mean that the inferred state machines show little difference. On the contrary: unresponsive and superfluous states, different interpretation of the RFCs, and unexplained quirks are the rule rather than the exception. In fact, only in the authentication layer did we find two identical state machines. These variances allow anyone to fingerprint the tested SUTs.

OpenSSH and Tectia implement a liberal message acceptance policy (Section 5.1). OpenSSH does not require a KEXINIT message from the client, and makes a guess on the used parameters. Tectia allows the three rekeying messages to arrive in arbitrary order. These liberal acceptance policies are unwise from a security perspective, since which they allow for state machine-related bugs which are easily introduced and hard to detect. CiscoSSH's state machines is the most restrictive, and closes the connection on any unexpected message.

OpenSSH seems to contain a bug which leads to a terminated connection when closing a channel if a rekey has been performed earlier. Besides this bug, our results do not show that different protocol layers interact unexpectedly.

The RFC for the connection layer provides relatively little state-related information (Section 5.4). Developers used this freedom of interpretation to implement state-machines which differ significantly. If the SSH RFC authors would had sketched a state machine, we would expect the standards to be less ambiguous and implemented state machines with fewer differences. Adding a reference state machine to the standards would definitely have added value for SSH. Most RFCs would benefit from added state information, and we would generally encourage authors to append a reference state machines to protocol specifications.

6.1 Future work

We had to make strict decisions on which messages we implemented to keep the learning times feasible. Although we aimed to pick messages that are most likely to produce interesting state-changing behaviour[1], the alphabet could be further extended. Testing key exchange algorithms other than Diffie-Hellman is especially interesting in this regard.

The input alphabet can also be extended by fuzzing on message parameters. Some parameters were included (for example, the GUESSINIT message is a KEXINIT with slightly altered parameters), but further extending parameter fuzzing might result in interesting behaviour. Extending the mapper to implement the wrong-guess procedure[2] would also provide more information.

We limited the number of SUTs to six. Other frequently-used servers could be investigated as well.

Finally, a practical limitation of our setup is that it infers SSH's protocol layers individually. As a result, we cannot prove that there is no undetected, unwanted interaction between different layers. Trying more transport layer messages in higher layer could reveal unwanted interaction. A combined state machine for all three layers could also reveal more interesting interaction, although learning a combined machine would most likely need another experimental setup[3].

[1] An "outgoing only" message policy was used, as described in Section 4.3.
[2] Section 4.3 provides more information on the GUESSINIT message.
[3] The grounds for inferring the layers indivudally are covered in Section 4.5.

Bibliography

[1] M. R. Albrecht, K. G. Paterson, and G. J. Watson, "Plaintext recovery attacks against SSH," in *Security and Privacy, 2009 30th IEEE Symposium on*, SP '09, (Washington, DC, USA), pp. 16–26, IEEE, May 2009.

[2] E. Poll and A. Schubert, "Verifying an implementation of SSH," *Proceedings of the 17th Workshop on Information Technology and Systems (WITS'07)*, pp. 164–177, Dec. 2007.

[3] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery.* Addison-Wesley Professional, 1 ed., July 2007.

[4] J. de Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *24th USENIX Security Symposium (USENIX Security 15)*, (Washington, D.C.), pp. 193–206, USENIX Association, Aug. 2015.

[5] D. A. Wheeler, "How to prevent the next Heartbleed." Online. `http://www.dwheeler.com/essays/heartbleed.html`.

[6] T. Ylonen, "The secure shell (SSH) protocol architecture. RFC 4251, the internet engineering task force, network working group," Jan. 2006.

[7] T. Ylonen, "The secure shell (SSH) authentication protocol. RFC 4252, the internet engineering task force, network working group," Jan. 2006.

[8] T. Ylonen, "The secure shell (SSH) transport layer protocol. RFC 4253, the internet engineering task force, network working group," Jan. 2006.

[9] T. Ylonen, "The secure shell (SSH) connection protocol. RFC 4254, the internet engineering task force, network working group," Jan. 2006.

[10] S. C. Williams, "Analysis of the SSH key exchange protocol," in *Cryptography and Coding* (L. Chen, ed.), vol. 7089 of *Lecture Notes in Computer Science*, pp. 356–374, Springer Berlin Heidelberg, 2011.

[11] M. Bellare, T. Kohno, and C. Namprempre, "Breaking and provably repairing the SSH authenticated encryption scheme: A case study of the Encode-then-Encrypt-and-MAC paradigm," *ACM Trans. Inf. Syst. Secur.*, vol. 7, pp. 206–241, May 2004.

[12] K. G. Paterson and G. J. Watson, "Plaintext-Dependent decryption: A formal security treatment of SSH-CTR," in *Advances in Cryptology – EUROCRYPT 2010* (H. Gilbert, ed.), vol. 6110 of *Lecture Notes in Computer Science*, pp. 345–361, Springer Berlin Heidelberg, 2010.

[13] F. Aarts, J. de Ruiter, and E. Poll, "Formal models of bank cards for free," in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pp. 461–468, IEEE, Mar. 2013.

[14] F. Aarts, J. Schmaltz, and F. Vaandrager, "Inference and abstraction of the biometric passport," in *Leveraging Applications of Formal Methods, Verification, and Validation* (T. Margaria and B. Steffen, eds.), vol. 6415 of *Lecture Notes in Computer Science*, pp. 673–686, Springer Berlin Heidelberg, 2010.

[15] G. Chalupar, S. Peherstorfer, E. Poll, and J. de Ruiter, "Automated reverse engineering using LEGO," *WOOT'14 Proceedings of the 8th USENIX conference on Offensive Technologies*, pp. 1–10, 2014.

[16] R. Janssen, "Learning a state diagram of TCP using abstraction.," Master's thesis, Radboud University, 2015.

[17] M. Tijssen, "Automatic modeling of SSH implementations with state machine learning algorithms," *Radboud University*, 2014.

[18] A. Futoransky and E. Kargieman, "An attack on CRC-32 integrity checks of encrypted channels using CBC and CFB modes." Online. `http://www.coresecurity.com/files/attachments/CRC32.pdf`.

[19] S. Lehtinen, "The secure shell (SSH) protocol assigned numbers. RFC 4250, the internet engineering task force, network working group," Jan. 2006.

[20] D. J. Barrett, R. E. Silverman, and R. G. Byrnes, *SSH, The Secure Shell: The Definitive Guide.* O'Reilly Media, second edition ed., May 2005.

[21] J. Schlyter, "Using DNS to securely publish secure shell (SSH) key fingerprints. RFC 4255, the internet engineering task force, network working group," Jan. 2006.

[22] M. Dusi, F. Gringoli, and L. Salgarelli, "A model for the study of privacy issues in secure shell connections," in *Information Assurance and Security, 2008. ISIAS '08. Fourth International Conference on*, pp. 311–317, IEEE, 2008.

[23] Y. Wang, Z. Zhang, D. Yao, B. Qu, and L. Guo, "Inferring protocol state machine from network traces: A probabilistic approach," in *Applied Cryptography and Network Security* (J. Lopez and G. Tsudik, eds.), vol. 6715 of *Lecture Notes in Computer Science*, pp. 1–18, Springer Berlin Heidelberg, 2011.

[24] F. Aarts, B. Jonsson, and J. Uijen, "Generating models of Infinite-State communication protocols using regular inference with abstraction," in *Testing Software and Systems* (A. Petrenko, A. Simão, and J. C. Maldonado, eds.), vol. 6435 of *Lecture Notes in Computer Science*, pp. 188–204, Springer Berlin Heidelberg, 2010.

[25] C. S. Păsăreanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer, "Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning," *Form. Methods Syst. Des.*, vol. 32, pp. 175–205, June 2008.

[26] E. Poll and A. Schubert, "Rigorous specifications of the SSH transport layer." Radboud University, 2011.

[27] Information Sciences Institute, University of Southern California, "DOD standard internet protocol. RFC 760, the internet engineering task force, network working group," Jan. 1980.

A Message abbreviations

Abbreviation	**RFC-defined name**	**Reference**
DISCON	SSH_MSG_DISCONNECT	[8, p. 23]
IGNORE	SSH_MSG_IGNORE	[8, p. 24]
UNIMPL	SSH_MSG_UNIMPLEMENTED	[8, p. 25]
DEBUG	SSH_MSG_DEBUG	[8, p. 25]
KEXINIT	SSH_MSG_KEXINIT	[8, p. 17]
GUESSINIT	SSH_MSG_KEXINIT	[8, p. 19]
KEX30	SSH_MSG_KEXDH_INIT	[8, p. 21]
KEX31	SSH_MSG_KEXDH_REPLY	[8, p. 22]
NEWKEYS	SSH_MSG_NEWKEYS	[8, p. 21]
SR_AUTH	SSH_MSG_SERVICE_REQUEST	[8, p. 23]
SR_CONN	SSH_MSG_SERVICE_REQUEST	[8, p. 23]
ACCEPT	SSH_MSG_SERVICE_ACCEPT	[8, p. 25]

Table A.1: Message abbreviations for the transport layer.

Abbreviation	**RFC-defined name**	**Reference**
UA_NONE	SSH_MSG_USERAUTH_REQUEST	[7, p. 7]
UA_PK_OK	SSH_MSG_USERAUTH_REQUEST	[7, p. 8]
UA_PK_NOK	SSH_MSG_USERAUTH_REQUEST	[7, p. 8]
UA_PW_OK	SSH_MSG_USERAUTH_REQUEST	[7, p. 10]
UA_PW_NOK	SSH_MSG_USERAUTH_REQUEST	[7, p. 10]
UA_SUCCESS	SSH_MSG_USERAUTH_SUCCESS	[7, p. 6]
UA_FAILURE	SSH_MSG_USERAUTH_FAILURE	[7, p. 5]

Table A.2: Message abbreviations for the user authentication layer.

Abbreviation	**RFC-defined name**	**Reference**
CH_OPEN	SSH_MSG_CHANNEL_OPEN	[9, p. 5]
CH_CLOSE	SSH_MSG_CHANNEL_CLOSE	[9, p. 9]
CH_EOF	SSH_MSG_CHANNEL_EOF	[9, p. 9]
CH_DATA	SSH_MSG_CHANNEL_DATA	[9, p. 7]
CH_EDATA	SSH_MSG_CHANNEL_EXTENDED_DATA	[9, p. 8]
CH_WINDOW_ADJUST	SSH_MSG_CHANNEL_WINDOW_ADJUST	[9, p. 7]
CH_REQUEST_PTY	SSH_MSG_CHANNEL_REQUEST_PTY	[9, p. 11]
CH_SUCCESS	SSH_MSG_CHANNEL_SUCCESS	[9, p. 4]

Table A.3: Message abbreviations for the connection layer.

B State machine formatting

This appendix shows the formatting process applied to CiscoSSH's transport layer state machine.

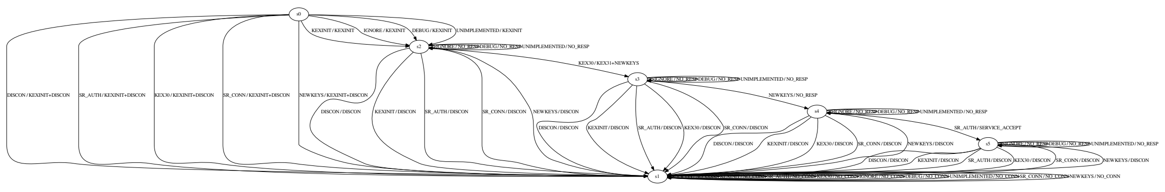

Figure B.1: Unformatted state machine.

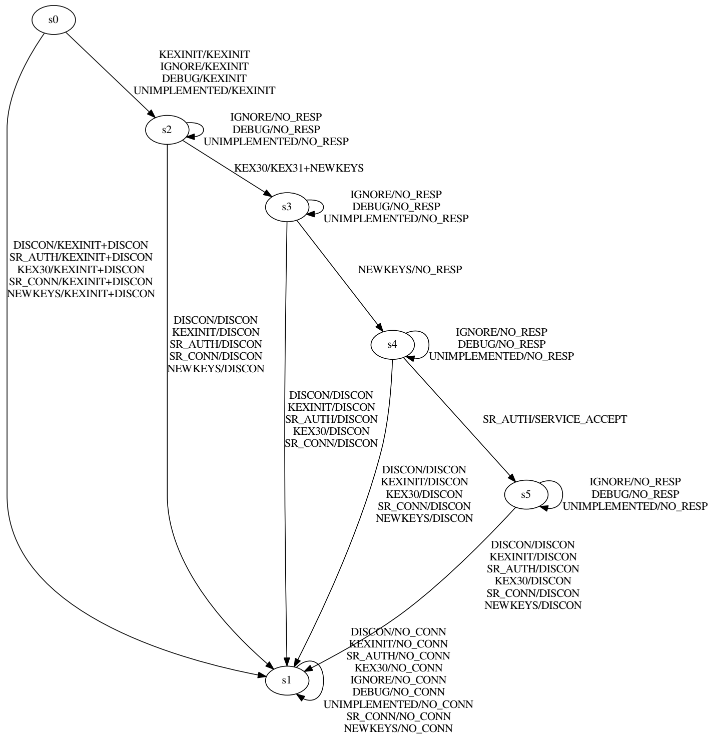

Figure B.2: State machine after merging edges between same nodes.

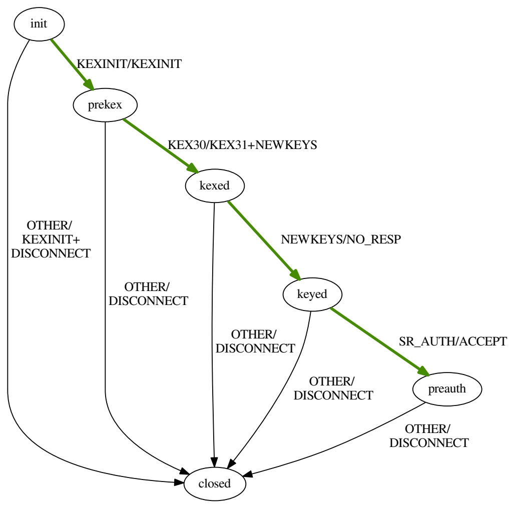

Figure B.3: State machine after manually applying conventions from Section 4.6.