Master's Thesis

Submitted to the System Security Research Group
in Partial Fulfillment of the Requirements
for the Degree of Master of Science

Automated Analysis of SSH Client State Machines

Tim Leonhard Storm

Supervisors: Prof. Dr.-Ing. Juraj Somorovsky
Prof. Dr. Jörg Schwenk
Fabian Bäumer, M.Sc
Date: February 23, 2026

Abstract

State Machine Learning (SML) is a powerful technique to analyze the behavior of network protocol implementations. In this thesis, we apply SML to 12 SSH clients to examine their adherence to the SSH protocol specification. While SML enables automatic inference of protocol state machines, the manual analysis of these models is often time-consuming. Instead, we explore a differential-based analysis approach, where two different state machines are compared to highlight behavioral differences. For this, we apply the LTSDiff algorithm to SSH client state machines and systematically evaluate its effectiveness.

We find that plain LTSDiff is insufficient, but that it can be tailored to work with Secure Shell (SSH) and used to identify meaningful differences. These differences reveal minor violations of the SSH transport protocol and hint at a broader issue of under-specification in the SSH standard. Fully automated detection of non-compliance remains challenging, as our differential analysis still requires expert interpretation in the absence of a comprehensive reference model.

Contents

1 Introduction

Secure Shell (SSH) is the de facto standard for secure remote access to computer systems and is widely used, both directly by IT professionals and as part of automated deployment tools [33–37]. While desktop operating systems like Windows typically prefer GUI-based remote access solutions like Remote Desktop Protocol (RDP) [42], SSH dominates the market for Linux-based systems. Since Linux deployments make up the majority of all server systems, especially with an increase in cloud computing and IoT devices, SSH is arguably one of the central security mechanism for modern computing environments. This makes secure SSH implementations essential.

The SSH ecosystem has been thoroughly cryptographically examined over the years, and many weaknesses have been identified and mitigated [2, 12, 46, 57]. Yet even today, 20 years after the standardization by the IETF, new vulnerabilities are still being discovered. While cryptographic guarantees are helpful, the security of real-world deployments ultimately depends on the correctness of the implementation. There are different techniques like fuzzing or static analysis that can help identify implementation flaws, but we focus on a different approach: learning a model of the implementation.

Learning implementation behavior. State Machine Learning (SML) is a technique to derive a model of an implementation's behavior, called State Machine (SM), which shows how a system's output behavior changes in response to different inputs. These models can be used to find behavioral bugs and give correctness guarantees. This typically involves identifying unexpected states and transitions based on knowledge about the underlying protocol specification.

There are different approaches to SML, including active and passive learning, that differ in how they interact with the target implementation, called System Under Learning (SUL). Active black-box learning can infer the state machine of any SUL, regardless of the underlying code, by crafting messages and observing the responses. This has been successfully applied to different protocols [1, 4, 20–22]. SSH too has been previously analyzed using SML by Fiterău-Broştean et al. [23], who learned SMs for three SSH server implementations. Recently, Bäumer et al. also applied SML to find bugs specifically in different servers' implementation of a countermeasure to their Terrapin attack [9, 10]. Their implementation called SSH-State-Learner is publicly available.

Client-side challenges. Servers are a natural target for SML: They make up a significant portion of the SSH ecosystem and are often publicly exposed, offering a large attack

surface. In terms of SML, they are easy to automate as non-interactive, Linux-compatible applications dominate the landscape.[1]

In SSH clients on the other hand, there is more diversity, although actual usage statistics are hard to come by. Many SSH clients are implemented as graphical applications with complex user interfaces, and cover a range of platforms and programming languages. Also, they often include advanced features, such as file transfer.

Applying SML requires a reproducible test harness, where the SUL repeatedly interacts with the learning application. For SSH clients, this means automating the actions typically performed by a human user, which makes SML challenging for SSH clients. But even when SML works, the results still require careful manual interpretation.

Differential-based state machine analysis. Recently, Ang et al. [4] demonstrated that a differential-based approach is effective in identifying behavioral differences between implementations: They compared learned QUIC [31] implementation state machines to find protocol non-compliance, by examining behavior differentials between pairs of implementation. Their differentials were generated using the LTSDiff algorithm [56], which given two state machines, identifies states and transitions that are different between the two. Based on the standard for QUIC, they were able to pick a set of conforming implementations to serve as reference models for future applications.

Goals. This thesis applies differential-based analysis to study SSH client implementations with SML. While prior work has focused on server implementations, a systematic evaluation of SSH clients has not been done before. Differential analysis in general has also not yet been applied to SSH.

Unlike modern protocols, like QUIC or TLS 1.3 [31, 50], whose specifications explicitly specify the underlying state machine, the specification for SSH is less formal and lacks such rigor. This leaves more room for interpretation and potential divergence across implementations. It remains an open question, whether a differential-based approach is effective for such a protocol. Due to the complexity of SSH, this work focuses exclusively on the SSH transport layer as defined in RFC 4253 [37].

In particular, we want to answer two research questions:

RQ1 : Is LTSDiff-based analysis effective in identifying differences between SSH client state machines and if so to what extent?

RQ2 : Are there violations of RFC 4253 in SSH clients?

Contributions. In this thesis, we present the first large-scale empirical study of SSH client state machines using SML.

- We automate 12 SSH clients to serve as SML SULs, including GUI clients, whose automation requires more individual setup.

1 The `Censys.io` search engine reports that $> 75\%$ of public SSH deployments use OpenSSH on Linux, while `Shodan.io` estimates close to 90%.

- We adapt and extend Bäumer's SSH-State-Learner implementation to learn state machines for these different SSH client implementations.
- We integrate the LTSDiff algorithm to produce state machine differentials and define a usefulness criterion for differentials
- We optimize LTSDiff for SSH state machines to maximize the usefulness of the produced differentials
- We uncover multiple forms of specification non-compliance across all SSH clients

Structure. The thesis is split into 7 chapters. In Chapter 2, we provide necessary background on SSH, SML, and the LTSDiff algorithm. In Chapter 3, we present design choices made in terms of configuring the SSH clients, the SSH-State-Learner and the LTSDiff analysis. In Chapter 4, we show how we implemented the SML into the existing SSH-State-Learner artifact. In Chapter 5, we evaluate different LTSDiff configurations and their effectiveness in analyzing SSH clients. We also present standard violations obtained through our analysis. In Chapter 6, we interpret the results of our evaluation and discuss their broader implications. Finally, we conclude the thesis in Chapter 7 and discuss future work.

1.1 Related Work

SSH. SSH is a protocol with a long history and widespread use, which means it has undergone extensive analysis over the years. In 2004, Bellare, Kohno, and Namprempre first proposed changes to SSH that would make sure the authenticated encryption used by SSH's Binary Packet Protocol (BPP) would provide *provable* security guarantees [12]. Albrecht, Paterson, and Watson showed that this security notion could be broken in practice, leading to a plain-text recovery attack in 2009 [3]. Their attack exploited the fact, that an adversary may be able to distinguish between different types of decryption errors, "in any conceivable implementation", because in BPP the success of a decryption can depend on the decrypted content itself. This had not been considered in the original security proofs. Paterson and Watson remedied this in 2010 and extended the security notion, and proved the security of SSH with counter mode encryption. In 2011, Williams extended security proofs to also consider the key exchange before BPP [57]. More theoretical work followed, proving security when considering more than one cipher suite [14] or in a post-quantum setting [13].

Nonetheless, there have still been practical attacks on SSH, like the recent Terrapin attack by Bäumer, Brinkmann, and Schwenk [9], which showed that there are still gaps between formal models and real-world implementations. A larger group of attacks however, did not break formal security guarantees, but exploited implementation and deployment flaws instead. Attacks have exploited the difficulty of secure randomness [8, 29] or timing side-channels [54]. Lastly, there have been bugs in implementations themselves, that can have disastrous consequences. This includes relatively simple bugs, like the `CVE-2001-0144`

buffer overflow[2], enabling arbitrary code execution. But it also includes the interesting group of state machine bugs, like libssh's `CVE-2018-10933` server authentication bypass or the recent remote code execution vulnerability `CVE-2025-32433` against Erlang/OTP [10]. Both are vulnerabilities, that arise from the way implementations respond to unexpected messages in their internal state machine. SML is a technique that can and has been used to find such bugs.

SML. (Active black-box) State Machine Learning is a technique first applied to network protocols by De Ruiter and Poll to Transport Layer Security (TLS) in De Ruiter and Poll. Various work followed, applying SML to other aspects of TLS like DTLS [22], TLS in Windows [58] and TLS outside a lab setting [38], but also to other protocols like OpenVPN and QUIC [4, 20, 49]. However, SML itself, has a long history in the field of formal methods and software testing. One of the most popular algorithms for SML is the L* algorithm by Angluin [5], who proposed it for learning regular languages. With it, he introduced the Minimally Adequate Teacher (MAT) framework to formalize the capabilities of any learning algorithm, where the learner learns any language by issuing two specific types of queries. Later learning algorithms by Kearns and Vazirani [32] and the TTT algorithm Isberner, Howar, and Steffen [30] also use the MAT framework, but optimize the amount of required queries with new data structures. Although the MAT framework was designed for regular languages, it was also extended to different types of Finite State Machines (FSMs), which is how it is used today [39, 48, 52].

Part of the MAT framework's success is due to its connection to the field of software testing: Angluin's learning algorithm requires an oracle to determine whether a hypothetical model is consistent with the actual target. Techniques such as the W-method [18] and partial W-method [25] can be used. They were designed to test software systems where a reference FSM already exists, but they can also be adapted for use with SML.

SSH State Machine Learning. In 2017 Fiterău-Broştean et al. applied SML to SSH server implementations and presented state machines for three different implementations [23]. Their work focused on inferring complete state machines that cover the entire SSH protocol, including SSH Transport Layer Protocol (SSH-T) and SSH Authentication Protocol (SSH-A). This meant limiting the alphabet to reduce learning times by excluding optional messages and messages not normally sent by the client. They modelled functionality and security properties from the SSH specification in the Linear Temporal Logic, which can reason about the behavior of a system over time [47]. Using the NuSMV symbolic model checker [19], they uncovered violations in the implementation.

A more recent application of SML to SSH was by Bäumer et al. in 2025 [10]: Rather than inferring complete state machines, they focused only on the key exchange phase to analyze the adoption of the Strict KEX extension, the countermeasure they proposed to the recent Terrapin attack [9]. They used a custom equivalence oracle specifically tailored to this use case, which allowed them to learn with an alphabet comprised of all available SSH messages.

2 As referenced in "The Matrix"

Differential analysis of state Machines. In 2025, Ang et al. first applied the LTSDiff algorithm to QUIC state machines as a means of identify standard violations. They were able to curate a suite of reference state machines that capture *all* valid protocol behavior. This allows automatically finding standard deviations in new state machines by comparing them against the curated reference set [4].

2 Background

In this chapter, we provide the necessary background for this thesis. In Section 2.1, we introduce the SSH protocol with a focus on the transport layer. We then introduce State Machine Learning (SML) in Section 2.2. We explore both theoretical foundations and practical considerations. Concretely, we also introduce the SSH-State-Learner, which the implementation in Chapter 4 is built upon. Lastly, in Section 2.3 we introduce the LTSDiff algorithm, as well as the generalized gLTSDiff framework. Our differential analysis is based on this algorithm.

2.1 SSH

SSH is an important protocol to securely access remote systems, which is widely used today. It provides confidentiality, integrity, and authenticity over an insecure network and allows for a wide-range of applications, including interactive shell sessions, file transfers, and tunneling [36].

SSH technically consists of multiple sub-protocols, the SSH Transport Layer Protocol (SSH-T) [37], SSH Authentication Protocol (SSH-A) [34], and SSH Connection Protocol (SSH-C) [35] protocols, which are layered on top of each other.

SSH-T provides *server* authentication, confidentiality, and integrity. It provides a secure channel which is used by the SSH-A protocol to provide different modes of client authentication, like password-based authentication or public key authentication. Finally, the SSH-C protocol multiplexes logical channels over the same channel. These logical channels can be used for different purposes, e.g., to provide an interactive shell session, to forward ports, or to transfer files.

For the purposes of this work, we focus on the SSH-T protocol. We very briefly introduce the other protocols, but omit most details.

2.1.1 Transport Layer Protocol

The SSH-T protocol is a "secure, low level transport protocol [that] provides strong encryption, cryptographic host authentication, and integrity protection." [34]. SSH-T consists of two parts: The underlying transport protocol, known as the BPP and the SSH key exchange, which establishes the algorithms and key material used by the BPP.

The key exchange begins with the client and server sending an ASCII string identifying the protocol version, followed by a series of key exchange messages, encoded in the BPP. BPP packets encapsulate a given payload and add a header containing metadata such as the packet length and padding, to make them suitable for encryption, applying message authentication codes (MACs) for integrity and optional compression. Different message

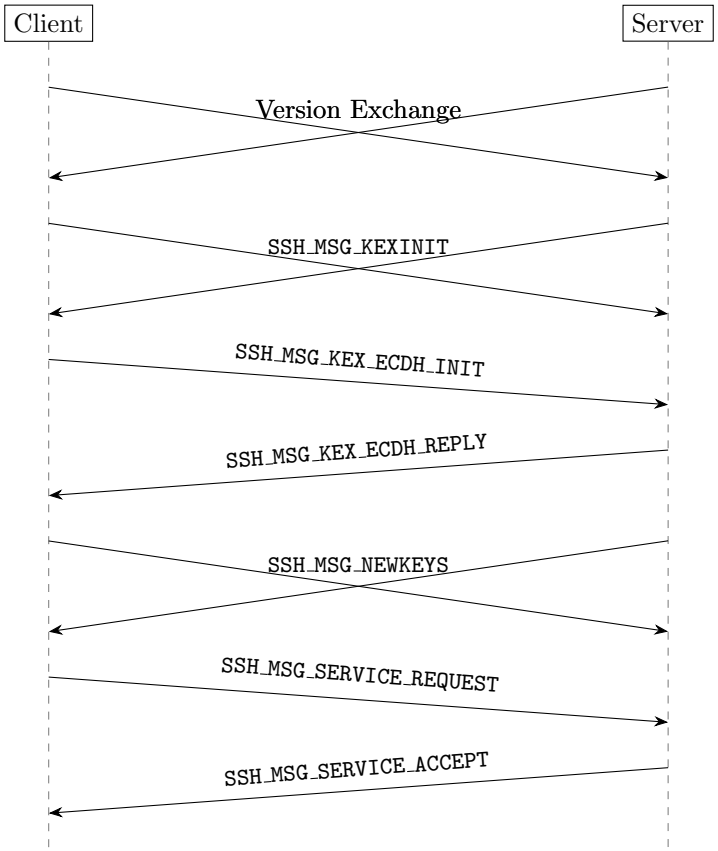


Figure 2.1. Typical SSH transport protocol message sequence with the ECDH key exchange. After negotiating the algorithms using `KEXINIT`, both parties perform the key exchange, and transition to encrypted communication using `NEWKEYS`. Finally, the client can request a specific service to be run over the now encrypted and server-authenticated channel. Adapted from Bäumer [11].

types are identifiable with a unique byte identifier at the beginning of the message, at least for all messages defined in RFC 4253 [37]. Message types also have labels of the form `SSH_MSG_*` in the specification, for brevity we omit the `SSH_MSG_` prefix when referring to messages, or use the more concrete symbols in Table 3.3

After switching to the BPP, both parties negotiate the relevant algorithms for encryption, integrity protection, and compression, by including their preferences in a `KEXINIT` message. Both parties then perform a cryptographic key exchange (KEX) using the negotiated algorithm: We only show the ECDH [55] key exchange here, but alternative key exchange methods, such as RSA or traditional DH, have been standardized [24, 26]. Although the concrete messages vary, they all negotiate a shared secret to derive keys from. In our case the client sends `KEX_ECDH_INIT` with its ECDH share, and the server replies with its share in `KEX_ECDH_REPLY`, which both lie on the curve negotiated within the `KEXINIT` message. Note that instead of waiting for the other parties' `KEXINIT` message, implementations may optimistically guess the negotiated KEX algorithm and

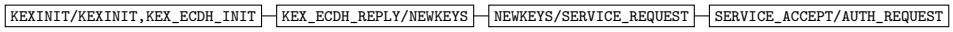


Figure 2.2. Happy flow for the ECDH key exchange, excluding version exchange. This shows the interaction as sent messages/received messages *from the server's perspective*. Unintuitively, the first `KEXINIT` sent by the client tends to chronologically occur before the server's `KEXINIT`, but we consider them to be part of the same flight. We choose this representation, because it is better aligned with the server's perspective shown later.

send the KEX messages before receiving the `KEXINIT` message to potentially save on latency. There is no strict client-server order for the `KEXINIT` messages.

The key exchanges make use of a server's static *host key*, to authenticate the server to the client. The host key should be distributed securely to the client beforehand, but in practice users are often prompted to verify a given host key on first use.

Upon completing the key exchange, both parties derive session keys from the negotiated secret, which are used for encryption and MACs going forward. This is confirmed by both parties sending a `NEWKEYS` message. Once again there is no enforced order for the `NEWKEYS` messages. At this point, both parties share an encrypted, integrity protected channel, where the server is authenticated. Then the client can request a specific service to be run over this channel with `SERVICE_REQUEST`: Defined services are `ssh-userauth`, which performs user authentication using SSH-A, and `ssh-connection`, which provides general-purpose channels via SSH-C. In practice, `ssh-userauth` is almost always requested, which in turn requests the `ssh-connection` service. See Fig. 2.1 for a visual representation of the message sequence in the transport layer protocol.

There are additional, but optional messages defined in the transport layer: For example, the `IGNORE` message may be sent at any time, or implementations may send `DEBUG` messages to provide debugging information to either party. `DISCONNECT` messages may be used to terminate the connection at any time, and include a reason code and an optional description. There are also various message IDs reserved for future use. Some of these have already been standardized, e.g., different KEX methods, but many of them remain unused. The `UNIMPLEMENTED` message is used to indicate that a received message is not implemented by the receiving party, and may be used to gracefully handle unsupported messages.

Happy flow. The *happy flow* (for a KEX) is the sequence of messages exchanged during a successful key exchange. It is shown in Fig. 2.1. It represents the idealized case, where no errors or special cases occur. Any implementation that does not support the happy flow is not functional.

Note, that the happy flow is not explicitly defined, but can be derived from the specification. Assuming a client-server back-and-forth communication pattern, our reference happy flow is shown in Fig. 2.2. This flow is only one of a family of happy flows, because SSH-T allows interleaving messages like `IGNORE`, but our representation is consistent with prior work [23].

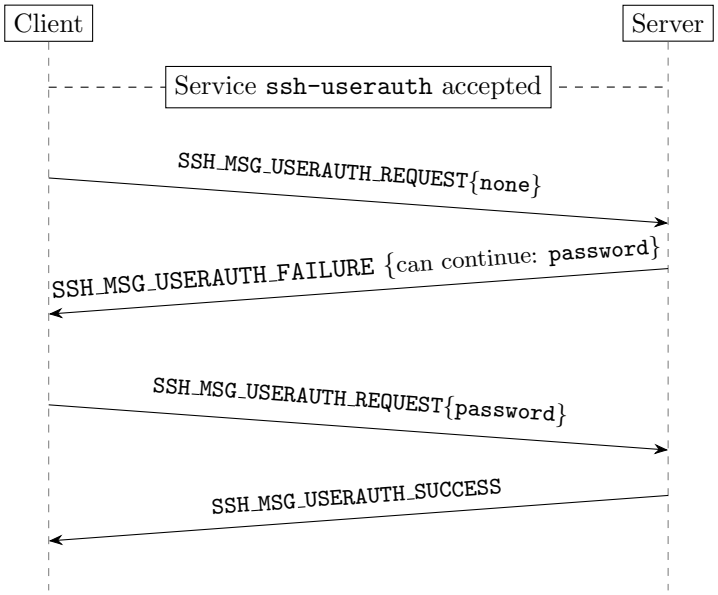


Figure 2.3. Typical SSH authentication protocol message sequence. After establishing a channel using SSH-T, the client first sends a `none` authentication request to query the server's supported authentication methods. The server indicates support for `password` authentication, which the client then uses to successfully authenticate. Adapted from Bäumer [11].

Re-keying. The key exchange described above, starting with `KEXINIT` and ending with `NEWKEYS`, can be repeated during a session to replace existing cryptographic session keys. This can be initiated by either party, and is recommended based on time and transmitted data. The feature is called key re-exchange (REKEX).

Extensions. With RFC 8308 [15], SSH also has a mechanism for securely negotiating extensions. By including the pseudo-KEX `ext-info-*` either party may signal that they support this. If their peer indicate support, both parties can then send `EXT_INFO` to communicate extension-specific information. These are sent after KEX immediately following `NEW_KEYS`. Additionally, the server may send `EXT_INFO` after user authentication, if the information should not be visible to unauthenticated clients.

Like extension negotiation itself, support for other extension to SSH-T is often communicated through pseudo-KEXes, e.g., strict key exchange (Strict KEX) [44].

Extensions can be security relevant, e.g., the `server-sig-algs` extension informs the client about the supported signature algorithms when using public keys for authentication [15]. Additional vendor-specific extensions exist, e.g., the `ping@openssh.com` extension, which indicates support for a newly-defined keep-alive message.

2.1.2 Authentication Protocol

The SSH-A protocol is intended to be run over the encrypted and integrity-protected channel established by the transport layer protocol. It provides a framework for different authentication methods for the client.

Authentication starts when the client sends a `USERAUTH_REQUEST` message to the server in which the client specifies a username, the service to be accessed after authentication (usually `ssh-connection`), and the authentication method to be used. The remainder of the message can contain additional method-specific data, such as passwords or public keys. In the original specification, `publickey`, `password`, and `hostbased` authentication methods are defined. Additionally, the (not recommended) `none` method can be used for no authentication. Clients are permitted to try authentication methods in any order: The server responds with a `USERAUTH_FAILURE` message if the authentication fails. The message will also include a list of other possible authentication methods with which the client may continue. Here the `none` method can be useful to query the server for supported authentication methods.

As soon as the client successfully authenticates, the server responds with a `USERAUTH_SUCCESS` message and should ignore future authentication attempts. At this point the requested service is started on top of SSH-A and respective messages (identified by the message number) are passed onto the service. A possible interaction is shown in Fig. 2.3.

2.1.3 Connection protocol

The `ssh-connection` service provides a general-purpose channel multiplexing protocol. Either party may request a channel to be opened, which is identified by a unique channel number on both sides. Channels can have different types, e.g., `session` for interactive shell sessions, which introduce additional semantics. Different channel types have different pre-defined `CHANNEL_REQUEST`, such as the `exec` request for executing a command on the remote end. Such an interaction is shown in Fig. 2.4. There are also channel-independent `GLOBAL_REQUEST` messages, e.g. the `tcpip-forward` request for port forwarding.

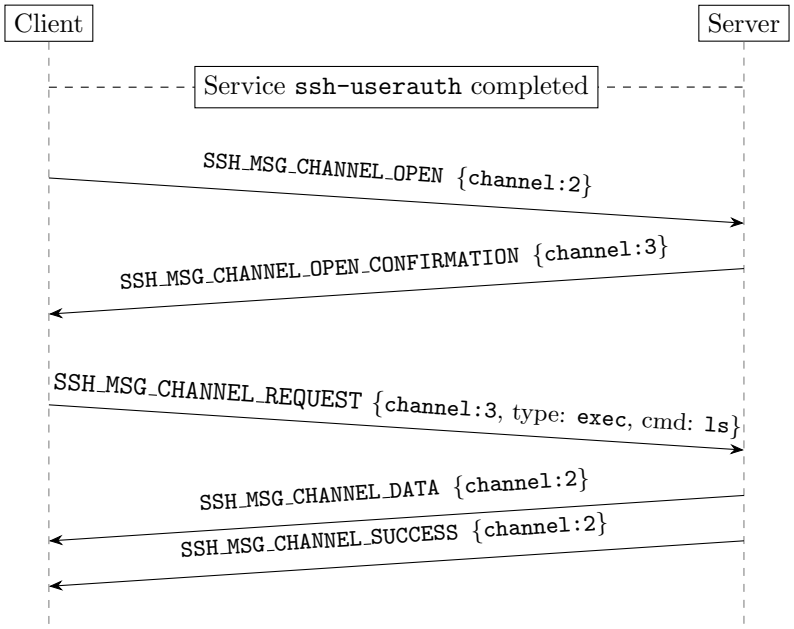


Figure 2.4. Typical SSH connection protocol message sequence. After authentication, the client now requests a `session` channel to the server. It uses the `exec` channel request to execute a command on the server, and the server responds with the command's output. Adapted from Bäumer [11].

2.2 State Machine Learning

State Machine Learning (SML), also referred to as *protocol state fuzzing*, is a technique to automatically infer a FSM for a (possibly black-box) system's behavior. Different flavors of SML exist, but we specifically focus on active black-box learning, which can be formalized using the Minimally Adequate Teacher (MAT) framework by Angluin [5]. We give a brief overview of the theoretical background and describe the practical aspects of applying SML.

2.2.1 Finite State Machines and Mealy Machine

FSMs are a formalism that model a system's behavior using a finite amount of states and transitions between these states based on input to the system. Multiple computationally equivalent definitions exist, we use Mealy Machines (MMs) in this work. MMs are well-suited to model protocol implementations, as they directly model the input/output nature of such systems. Modelled as a MM, any implementation is assumed to keep some internal state (e.g., session parameters or connection status) and react to incoming protocol messages (input symbols) with outgoing messages (output symbols) and then possibly change its internal state.

Definition 2.2.1. A MM is a 6-tuple $M = (Q, q_0, \Sigma, \Gamma, \delta, \lambda)$, where:

- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- Σ is a finite set of input symbols,
- Γ is a finite set of output symbols,
- $\delta : Q \times \Sigma \to Q$ is the state transition function,
- $\lambda : Q \times \Sigma \to \Gamma$ is the output function.

Given a sequence of symbols $w = i_1 \dots i_n \in \Sigma^*$, which we refer to as the input word, the MM produces an output word $w' = o_1 \dots o_n \in \Gamma^*$. The output word is generated by the MM starting in q_0 and processing input symbols sequentially. Each input symbol i_k is applied to the current state q_k, resulting in $q_{k+1} = \delta(q_k, i_k)$ and $o_k = \lambda(q_k, i_k)$.

2.2.2 Minimally Adequate Teacher

Our SML approach builds upon the MAT framework by Angluin [5]. In this framework, a *learner* (algorithm) tries to learn the language L (in our case represented via the MM) of a deterministic SUL via a teacher T. The learner can learn the behavior of the SUL using two types of queries:

- **Membership Queries** The learner sends a sequence of input symbols to T. Given this input word $w \in \Sigma^*$ the learner can ask the teacher for the corresponding SUL's output word $w' \in \Gamma^*$.
- **Equivalence Queries** The learner can also ask T whether a given hypothesis about the SUL is equivalent to the actual SUL. T responds with either "yes" or a counterexample (i.e., a sequence of input symbols in which the output of the hypothesis differs from the output of the SUL).

The teacher T answering these queries implies two oracles answering these queries which we call *membership oracle* $\mathcal{O}_M$ and *equivalence oracle* $\mathcal{O}_E$ respectively, adhering to Bäumer [11]. T serves as an abstraction of the SUL, which has complete knowledge of its behavior.

Note that this definition of a membership oracle does not directly match the original formulation of Angluin's MAT framework [5], because it was only designed for regular languages. Our modified version is based on LearnLib, which implements a FSM-compatible version of different MAT algorithms and is our library of choice [11, 39, 48].

Using the oracles, a learning algorithm like TTT [30] continually refines a hypothetical SM of the SUL until no counterexample can be found. When no more counterexamples can be found, the current hypothesis is returned as the learned SM of the SUL. This process is illustrated in Fig. 2.5.

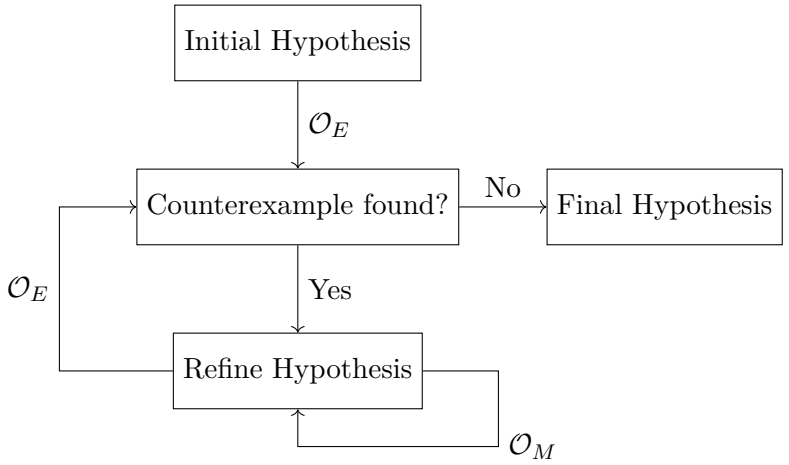


Figure 2.5. The basic learning loop of a MAT-based learning algorithm, such as TTT. The learner starts with a hypothesis about the SUL and iteratively refines it using the membership oracle $\mathcal{O}_M$. The equivalence oracle $\mathcal{O}_E$ then either disproves the current hypothesis with a counterexample, where the true output differs from the hypothesis, or terminates and outputs the latest hypothesis.

For an unbounded black-box SUL, $\mathcal{O}_E$ can only be approximated, because a true teacher is not available. Depending on this approximation, the resulting SM may not be truly equal, but a useful approximation (see Section 2.2.5).

2.2.3 Practical Considerations

In practice, there is no dedicated teacher, and we rely on the SUL itself. Because the abstract input and output symbols of a learning algorithm cannot be directly applied to the SUL, we employ an interface layer that translates abstract symbols to concrete protocol messages and vice versa, which we call *mapper*. As an example consider the abstract input symbol for `KEXINIT` compared to its real-world instantiation in Listing 2.1. The learning algorithms input alphabet will contain only a single symbol for `KEXINIT` and translates the symbol to the appropriate byte representation, by generating the random cookie and filling in desired algorithms.

```
1 byte         SSH_MSG_KEXINIT
2 byte[16]     cookie (random bytes)
3 name-list    kex_algorithms
4 name-list    server_host_key_algorithms
5 name-list    encryption_algorithms_client_to_server
6 name-list    encryption_algorithms_server_to_client
7 ...
```

Listing 2.1. Excerpt from the definition of `KEXINIT` message from RFC 4253 in SSH data type representation [37].

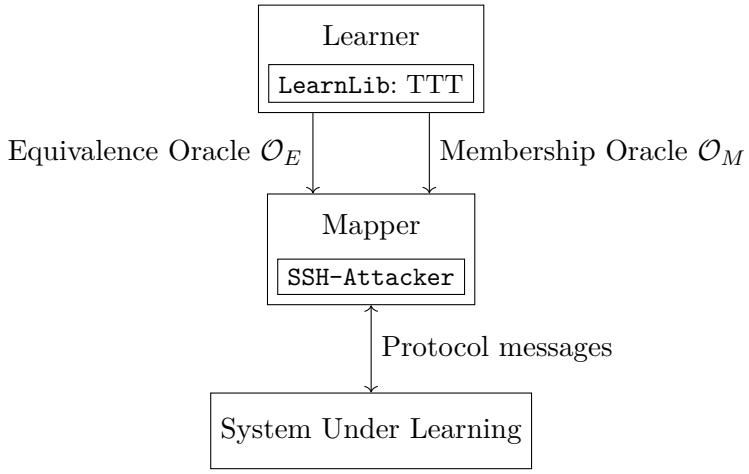


Figure 2.6. Overview of the learning process using the mapper to connect the learning algorithm with the real-world SUL. The learning algorithm (in our case TTT) only interacts with an equivalence oracle $\mathcal{O}_E$ and $\mathcal{O}_M$ using abstract symbols. All communication with the SUL is performed through the mapper (using the SSH-Attacker library), which translates abstract symbols to concrete SSH messages and vice versa.

Using the mapper, we can then implement $\mathcal{O}_E$ and $\mathcal{O}_M$ through querying the real-world SUL as seen in Fig. 2.6. The SUL itself will only interact with concrete SSH messages and answer with messages that are mapped back to abstract symbols.

Note, that a single symbol does not necessarily correspond to a single network message: It can be mapped to multiple messages (e.g., a full handshake exchange) or even none at all (e.g., for modelling delays).

Non-determinism. Real-world SULs can exhibit (seemingly) non-deterministic behavior, which is incompatible with most learning algorithms. While technically possible, this is unlikely for a well-defined protocol implementation. Instead, factors apart from the sent messages, such as internal timeouts, can manifest as non-determinism. The mapper has to mitigate these effects. An example countermeasure to non-determinism is to repeat inconsistent queries multiple times and use the majority output.

2.2.4 SSH-State-Learner

We base our work on the existing SSH-State-Learner artifact by Bäumer [11], which uses SSH-Attacker as a mapper. SSH-Attacker is a Java-based framework for analyzing SSH implementations, based on TLS-Attacker [7]. It allows creating arbitrary SSH messages flows and executing them against implementations.

The SSH-State-Learner integrates SSH-Attacker with LearnLib [48] and already implements mechanisms to speed up the learning process and analyzing the resulting state machines. For details, we refer to the original work [11].

Unless otherwise mentioned, we use SSH-State-Learner in its base configuration: It uses the TTT learning algorithm and a chained equivalence oracle of a cache consistency, a happy flow oracle and a random walk oracle. We describe them below.

2.2.5 Membership and Equivalence Oracles

Implementing $\mathcal{O}_M$ using the mapper requires no additional effort: Given an input word $w \in \Sigma^*$, the membership oracle sends each input symbol to the mapper, which translates it into concrete SSH messages and sends them to the SUL.

As hinted above, implementing $\mathcal{O}_E$ is often only an approximation: If the true underlying model is unknown, only exhaustive testing can guarantee equivalence. This is infeasible for all but the most trivial SULs. The SSH-State-Learner chains together two equivalence oracles for these queries:

Happy flow mutation oracle. The *happy flow mutation oracle* [10] searches for counterexamples by comparing the output for hypothesis and SUL for inputs along the happy flow.

Definition 2.2.2 (Happy flow mutation oracle)**.** Given a happy flow sequence $h = (h_1, \ldots, h_n) \in \Sigma^*$ and a configurable number of insertions $x_1, \ldots, x_k \in \Sigma$ the happy flow oracle searches all sequences of the form

$$(h_1, \ldots, h_{i_1}, x_1, h_{i_1+1}, \ldots, h_{i_k}, x_k, h_{i_k+1} \ldots, h_n) \text{ for all } 0 \leq i_1 \ldots \leq i_k \leq n$$

for counterexamples to the hypothesis.

Because the sequences will stay close to the happy flow, this oracle ensures that the hypothesis is correct for the happy flow itself and for nearby states. We instantiate it with the happy flow sequence described in Section 2.1.1.

Random words oracle. The random words oracle is one of the simplest approaches to equivalence testing, which tests random sequences of input symbols for counterexamples. While its effectiveness decreases for large alphabets and or when new counterexamples require longer inputs, it is easy to implement.

Definition 2.2.3 (Random words oracle)**.** The random words oracle tests n sequences x uniformly sampled from Σ^* where $b_l \leq |x| \leq b_u$ for counterexamples, given fixed bounds b_l and b_u.

Cache consistency oracle. The cache consistency oracle is not a true equivalence oracle: It re-queries inputs for which responses are cached. When learning fails due to nondeterminism, the hypothesis can be quickly recovered from the cache. This does not work

for a truly non-deterministic system, but is useful to mitigate transient errors. It is part of the `ResponseTreeCache` presented by Bäumer [11].

2.3 LTSDiff

LTSDiff is an algorithm that can compute the difference between state machine. We give a brief overview of the algorithm as presented by Walkinshaw and Bogdanov [56], as well as the improved gLTSDiff [27].

2.3.1 The LTSDiff Algorithm

LTSDiff is an algorithm first presented by Walkinshaw and Bogdanov in 2013 [56] . It can be used to compare state machines and identify differences between them. LTSDiff takes two Labelled Transition Systems (LTSs) LHS and RHS as input and outputs a set of changes to transform LHS into RHS. We call these changes *diff(erential)s*. See Fig. 2.7 for an example.

Definition 2.3.1. A LTS is a tuple $L = (S, \Sigma, \Delta, I)$ where S is a set of states, Σ is a set of labels, $\Delta \subseteq S \times \Sigma \times S$ is a set of labelled transitions, and $I \subseteq S$ is a set of initial states.

Remark. Any MM $M = (Q, q_0, \Sigma, \Gamma, \delta, \lambda)$ can be interpreted as a LTS.

We give an intuitive construction here. Let $S = Q$, $I = \{q_0\}$ and $\Sigma_{LTS} = \{i||o \mid i \in \Sigma_{MM}, o \in \Gamma\}$. Then for $\delta(q_x, i_k) = q_y$ with $\lambda(q_x, i_k) = o_k$, let $t = (q_x, i_k||o_k, q_y) \in \Delta$ be a transition in the LTS.

Given LTSs LHS and RHS, the LTSDiff algorithm works in three main steps:

1. Calculate state similarity scores
2. Match “landmark” states
3. Derive Diff LTS

State Similarity Scores. LTSDiff first computes similarity scores for all state pairs in RHS and LHS. The goal is to identify states that are “equal” in both LTSs, so-called *landmark states*. They liken this to the process of finding recognizable landmarks on a map, which a human might use to orient themselves. Landmark states are pairs of states that fill a similar role in their surroundings.

As an example, consider a sink state where all transitions loop to itself. It would be easily recognizable as a landmark state, as it has the same distinctive structure in both LTSs.

Walkinshaw and Bogdanov present a state similarity metric to identify landmark states, which we explain here. We give a slightly simplified version of their metric here, because their original definition also accounts for non-deterministic transitions, which we do not need for the LTS we are dealing with.

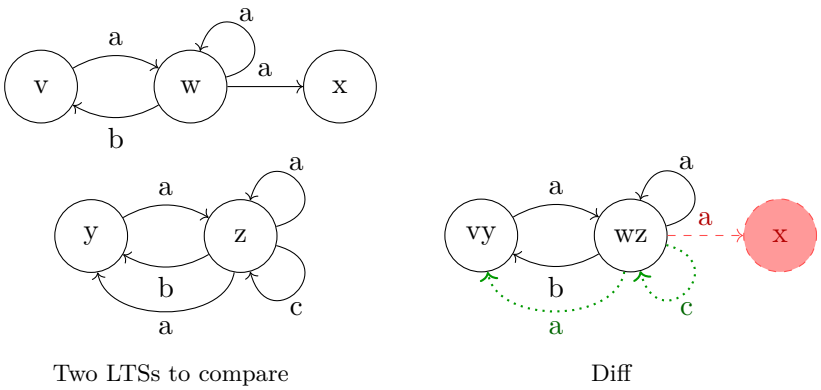


Figure 2.7. Example of a diff computed by gLTSDiff, adapted from [27]. Removing state x from the upper LTS and adding the missing transitions transforms it into the lower LTS.

First they define local similarity, which is high when the outgoing transitions of two states are similar (e.g., v and y in Fig. 2.7).

Definition 2.3.2 (Local similarity)**.** Let $\Sigma_X(a)$ be the set of outgoing transition labels of state a in LTS X. Then the local similarity of a state a in LTS LHS and b in LTS RHS can be defined as[a]:

$$S^L_{ab} = \frac{|matchingAdjacentTransitions|}{|allAdjacentTransitions|} = \frac{|\Sigma_{LHS}(a) \cap \Sigma_{RHS}(b)|}{|\Sigma_{LHS}(a) \cup \Sigma_{RHS}(b)|}$$

When either state has no outgoing transitions, $S_{ab} = 0$.

[a] We use the fact that our LTS are deterministic here.

While landmark states should have a high local similarity, this is not sufficient. Depending on the structure of the LTS, two states may have similar outgoing transitions but that lead to very different successor states. A good pair of landmark states should have transitions to similar successor states, which Walkinshaw and Bogdanov capture in a global similarity metric. We first define

$$Succ_{a,b} = \{(c, d, \sigma) | (a, \sigma, c) \in \Delta_{LHS}, (b, \sigma, d) \in \Delta_{RHS}\},$$

which is the set of all pairs of successor states reachable via the same transition label σ in their respective LTS. Note that we can calculate

$$S_{ab} = \frac{|Succ_{a,b}|}{|\Sigma_{LHS}(a) \cup \Sigma_{RHS}(b)|}.$$

The global similarity score now extends the idea of local similarity by taking shared transitions and aggregating the similarity of the corresponding successor states as well.

Definition 2.3.3 (Global successor similarity)**.** The global successor similarity $S^G_{Succ}(a, b)$ of state a in LTS LHS and state b in LTS RHS is defined as follows:

$$S^G_{Succ}(a, b) = \frac{1}{2} \frac{\Sigma_{(c,d,\sigma) \in Succ_{a,b}} (1 + k \cdot S^G_{Succ}(c, d))}{|\Sigma_{LHS}(a) \cup \Sigma_{RHS}(b)|}$$

Because S^G_{Succ} is calculated recursively, not only successor states, but also their respective successors are taken into account and so on (i.e. all reachable neighbor states). The parameter k gives precedence to the similarity of directly adjacent states and prevents far away states from skewing the similarity score. This means that highly similar states are similar in the context of all outgoing transitions in the entire LTS.

Definition 2.3.4. The global similarity of states a in LTS LHS and b in LTS RHS is now defined as:

$$S^G(a, b) = \frac{(S^G_{Succ}(a, b) + S^G_{Pred}(a, b))}{2}$$

where $S^G_{Pred}(a, b)$ is defined analogously to $S^G_{Succ}(a, b)$ by using predecessor states and incoming edge labels instead.

This makes state pairs with high global similarity, states that are similar in the context of all their outgoing and incoming transitions in their entire LTS. $S_{a,b}$ can be calculated as a system of linear equations, because the respective denominator for $S^G_{Succ}(a, b)$ and $S^G_{Pred}(a, b)$ is known for any state pair.

Matching landmark states. Given the global similarity $S_{a,b}$ between all states, the most similar states are mapped to each other and all their shared transitions are considered `UNCHANGED`. These states are then no longer considered for further matching. See the state vy in Fig. 2.7 which represents v and y respectively[1].

To prevent dissimilar states from being matched, two criteria are applied:

- a threshold t ignores all possible matchings with a similarity score below t.
- a ratio r ensures that for states with a high number of possible matchings, they are only matched to states where the similarity is significantly higher than with the next best candidate.

This matching process then repeats until no more states can be matched. By the end, all states are either matched with a likely counterpart or remain unmatched.

Deriving the Diff LTS. All unmatched states and their transitions remaining in LHS are considered `REMOVED`, while those only in RHS are considered `ADDED`. Together with the `UNCHANGED` matched states, these form the final diff LTS.

1 Note that LTS states are not actually labelled and that this is only for visual clarity.

Failure Cases. LTSDiff is deterministic and will always produce a diff, given two inputs LTSs. The quality of a diff depends on the chosen landmark states: If two states are matched that should not actually be similar, most of their transitions will be classified as `ADDED` or `REMOVED`. They state pairs are chosen based on the similarity score, which takes into account the output behavior and relationship to other states. This does not guarantee that all matched states are similar in the sense of being the same state in the underlying system. Two states that behave very similar can still serve very different roles in their respective systems.

2.3.2 gLTSDiff

gLTSDiff is an algorithm "which generalizes and extends LTSDiff", which was presented by Hendriks and Oortwijn in 2023 [27]. Their main contribution is an improved implementation of LTSDiff which allows tailoring the matching process to specific use cases [28].

The most important change for this work is their notion of *combinability* between states and transitions. When calculating the state similarity scores, LTSDiff requires transitions to be equal for two states to be considered similar. gLTSDiff instead allows the user to define whether two transitions are *combinable* via a custom combination operator. As an example, Hendriks and Oortwijn use this to annotate transitions with version information, which is then ignored during the matching process, but still included in the final diff LTS.

Similarly, states can also include custom combinability logic to determine whether they should be able to be matched. This can be used to improve matching accuracy by including additional metadata. An example of this is excluding known error states from being matched with non-error states. Other changes include post-processing for refining the visualization, improved performance, and the allowing multiple LTSs to be compared at once.

Going forward, we will refer to LTSDiff when we talk about the algorithm, and only gLTSDiff when we talk about the specific implementation by Hendriks and Oortwijn.

2.3.3 Differentials as Similarity Measures

Differentials are very useful when interpreted visually, but do not directly provide a numerical similarity score. Walkinshaw and Bogdanov also present a way to compute a similarity score based on differentials [56, Section 4.4]. They construct a confusion matrix which treats RHS as a prediction of LHS, where shared transitions are considered true positives (TP) and superfluous or missing transitions are considered false positives (FP) or false negatives (FN), respectively. The confusion matrix can be computed from a differential, as seen in Fig. 2.8. This allows applying classification metrics to compute a similarity score. The F_1 score is a typical measure of prediction performance [51] calculated as

$$2\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{2TP}{2TP + FP + FN}.$$

We will be using the F_1 score as our similarity measure as well.

	in RHS	not in RHS
in LHS	$TP =$ `UNCHANGED`	$FN =$ `REMOVED`
not in LHS	$FP =$ `ADDED`	$TN = \emptyset$

Figure 2.8. Confusion matrix based on differentials. By treating RHS as a prediction of LHS, we can treat superfluous and missing edges as false positives or false negatives respectively. This then allows applying the F_1 score as a similarity measure.

3 Methodology

To answer **RQ1**, whether LTSDiff differentials can effectively identify meaningful differences between SSH implementations, we first infer client SMs using the SSH-State-Learner. In Section 3.1, we describe the selection of the SSH clients in our study we used for this. In Section 3.2, we cover the learning process itself and:w describe the rationale behind our configuration of the SSH-State-Learner. To use the output SMs with LTSDiff, we need to bring them into a compatible format, which we describe in Section 3.3. In Section 3.4.1, we define what we consider to be useful LTSDiff differentials. They serve as a baseline to apply LTSDiff, but also to explore different refinements of the differential process in Section 3.4.2. Finally, in Section 3.5, we describe how we analyze differentials with respect to the SSH-T specification. This enables us to use the refined differentials to answer **RQ2**, whether we can detect deviations from RFC 4253. Together, these steps form the foundation for our evaluation presented in Chapter 5. The combined process is illustrated in Fig. 3.1

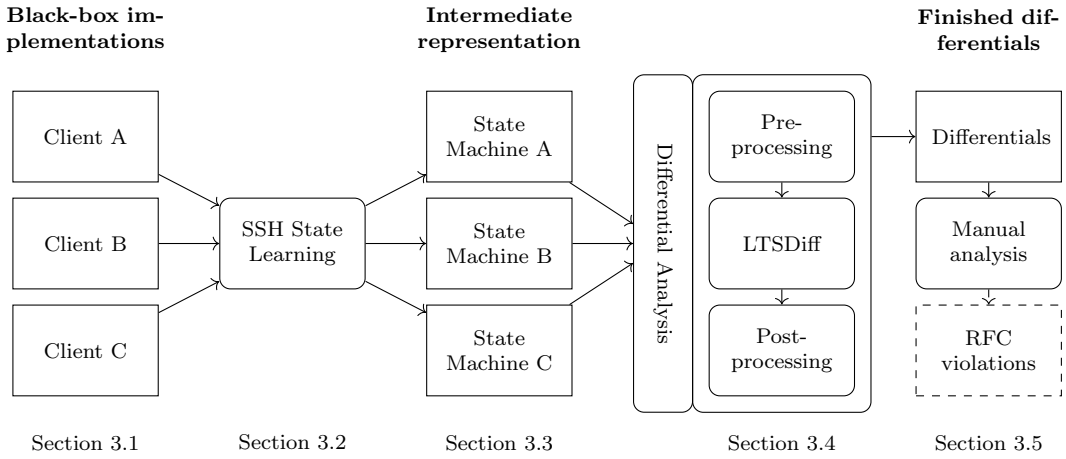


Figure 3.1. Overview of the evaluation pipeline. Rounded boxes represent processing steps, and rectangles products from them. Each step is mapped to its corresponding section in the methodology.

3.1 Choice of Clients

We select 11 different SSH clients for our evaluation. Key criteria for selection are popularity, recency, different use cases, different operating systems, different interaction

paradigms and availability of source code. The selection aims for a diverse set of clients, covering various use cases. Our selection includes terminal-based clients, GUI clients and

Software	Use case	OS	Interface	Open-source	Latest release
AsyncSSH	Library	Win, Linux	-	✔ (MIT)	2025-12-21
Bitvise	Terminal, SFTP	Win	GUI	✘	2026-02-01
Dropbear	Terminal	Linux	CLI	✔ (MIT)	2025-12-16
FileZilla	SFTP	Win, Linux	GUI	✔ (GPL)	2025-11-12
GoSSH	Library	Win, Linux	-	✔ (BSD-3)	2026-01-12
OpenSSH	Terminal	Win, Linux	CLI	✔ (BSD-3)	2025-10-10
Win32-OpenSSH	Terminal	Win, Linux	CLI	✔ (BSD-3)	2025-10-22
Paramiko	Library	Win, Linux	-	✔ (GPL)	2025-08-03
PuTTY RuSSH	Library	Win, Linux	-	✔ (Apache-2.0)	2026-01-24
Termius	Terminal, SFTP	Win, Linux	GUI	✘	2026-02-05
WinSCP	SFTP	Win	GUI	✔ (GPL)	2025-11-18

Table 3.1. Overview of our selected SSH clients.
GUI = GUI, CLI = CLI, Win = Windows, Linux = Linux

SSH libraries with client support. We also aim to cover different levels of user expertise: WinSCP and FileZilla are clients that focus on file transfer and are usable to non-experts. The different libraries like GoSSH, RuSSH, Paramiko and AsyncSSH are intended to be used by developers with some level of expertise. The remaining clients fall somewhere in the middle and include some GUIs, but also CLI usage. Note that WinSCP and FileZilla partially rely on PuTTY for its SSH functionalities, although public documentation is ambiguous on the extent of this dependency. From what we can tell, PuTTY code is used alongside custom code in these clients. Here, we can observe whether a shared implementation leads to structural similarities in the learned state machines. We also include both OpenSSH and OpenSSH for Windows (Win32-OpenSSH) - the latter being a fork of the former. Note that Win32-OpenSSH[1] has become a multi-platform port to both UNIX-like systems and Windows, which is maintained separately from OpenSSH.

All clients are actively maintained and have had recent releases within the past six months. Most clients are open-source, but we also included some proprietary clients to ensure a diverse evaluation set. Although user-facing features differ, these differences do not impact the transport-layer behavior we analyze.

We used the latest stable version for each client at the time of testing, and for Docker clients we also included the recent major releases for testing purposes[2], see Table 3.2.

1 Also known as openssh-portable.

2 We accidentally included older Dropbear versions than intended, but since they are not part of the main evaluation, we decided to keep them.

Software	Version
AsyncSSH	2.21.0, (2.20.0, 2.19.0)
Bitvise	9.47
Dropbear	2025.88, (2024.84, 2022.83)
FileZilla	3.69.5
GoSSH	0.43.0, (0.42.0, 0.41.0)
OpenSSH	9.9p2, (9.8p1, 9.7p1)
Win32-OpenSSH	9.5p1
Paramiko	4.0.0, (3.5.1, 3.4.1)
RuSSH	0.56.0, (0.55.0, 0.54.0)
Termius	9.34.4
WinSCP	6.5.5

Table 3.2. We evaluated the latest stable version for each SSH client at the time of testing. For clients available as Docker images, we also included recent major releases, which are shown in parentheses, for testing purposes.

3.1.1 Automation

For SML, we need to automate the interaction with the SSH clients and simulate how a human user would interact with them. The goal here is to simulate establish a SSH connection with minimal overhead. Our clients fall in two main categories: Primarily command-line clients, and clients that require a GUI.

For all CLI-based clients or libraries it is possible to run them in a lightweight Docker container. For GUI-based clients (and specifically those running on Windows), we employ GUI automation using the Appium framework. We discuss this in more detail in Section 4.2.

As our focus lies on the SSH protocol implementation rather than user interface semantics, we disregard the potential differences arising solely from different invocation methods. For each client, we choose the most reliable automation, even if it does not represent typical end-user usage. Regardless of automation, all clients were subjected to identical learning conditions.

3.2 State Machine Learning

Apart from the implementation itself, the configuration of the state learning components determines the resulting state machines and thus the differentials. Here we outline key design decisions that influenced our resulting state machines.

3.2.1 SSH Configuration

We only support `ecdh-sha2-nistp521` key exchanges to have common ground for differentials. The algorithm is supported by all clients and is considered a secure choice by the German Federal Office for Information Security [17]. It is also the default key exchange algorithm for SSH-State-Learner.

All SULs are used with their default configuration to simulate realistic deployments. Because we are not interested in how clients implement host key verification, the server-side host key is accepted prior to learning. Other than that, no further configuration is applied.

During our evaluation, we found that this was sufficient for most SULs, but some additional mitigations were necessary to enable the learning, see Section 4.2.3. These are purely on the learner side and do not affect the underlying implementation behavior.

Strict KEX. Strict KEX is an extension to the SSH-T key exchange, proposed by OpenSSH as a countermeasure to the Terrapin attack. It forces "implementations [to] terminate the connection if they receive a non-KEX message during the initial key exchange" [44]. We intentionally do *not* signal Strict KEX support: Since it strongly restricts the allowed messages, it could potentially obscure attack surfaces that are caused by unexpected messages sent during the key exchange.

Note, that proper implementation of Strict KEX is interesting in its own right and has been shown to produce new attack surfaces [10]. This has been explored in detail by the aforementioned Bäumer et al. and is ultimately not the goal of this thesis. This does not mean we intentionally disabled Strict KEX in any SULs, it only means we do not include the signaling suite on the server-side.

3.2.2 Oracles

The SSH-State-Learner uses a chained equivalence oracle consisting of the happy flow mutation oracle and a random walk oracle as a fallback, also used by Bäumer et al. [10]. The equivalence oracle will yield a reliable hypothesis in and around the happy flow. Looking at prior work that inferred server state machines indicates that the number of states for SSH-T is relatively small, and that the happy flow covers a significant portion of the state machine [11, 23]. We therefore expect our oracle to be effective here. Our happy flow mutation oracle is configured to insert up to $k = 2$ messages, and our random walk oracle tests $n = 1000$ sequences of length between $b_1 = 5$ and $b_u = 15$.

3.2.3 Alphabet

Previous work by Fiterău-Broştean et al. has shown that learning full SSH SMs can take a prohibitively long time [23]. Their work uses a small alphabet, for which they can learn a SSH state machines including SSH-T, SSH-A and SSH-C, but which may miss behavior that only occurs with messages that are not included in the alphabet.

Therefore, we restrict our scope to learning SSH-T state machines and instead attempt to cover a large alphabet. In particular, we aim to cover the largest possible set of SSH-T messages.

We hope to explore edge cases that arise from clients facing unexpected messages. We include *all* KEX message types (even those not supported/negotiated by the client or server). All KEX variants share the same message IDs, so when receiving a KEX message, the client has to parse the message according to the negotiated algorithms. This opens up interesting corner cases, when negotiation is still ongoing or when message IDs not normally used in ECDH are received. For the same reason, we also include all optional messages, and messages that emulate not yet defined message IDs. We include messages regardless of whether they are supposed to be sent by the server (e.g., `SERVICE_REQUEST`). We also include the messages for extension negotiation, defined in RFC 8308, and one extension, `ping@openssh.com` [15].

Note, that for the purposes of this thesis, we omit the version exchange phase from the transport protocol, because Binary Packet Protocol messages during version exchange may be interpreted as ASCII, leading to unpredictable behavior. Instead, we only consider the messages exchanged in BPP and any shown state machines will reflect this.

See Table 3.3 for a comprehensive overview of our input alphabets. We call the configuration described above alphabet Medium. We provided two additional configurations: Reduced is a stripped down version that focuses on the messages essential to the happy flow. More importantly, we disabled re-keying in this configuration to reduce the size of the state machine. Extended is a version that includes some SSH-C messages that would be highly dangerous if processed during SSH-T. Such weaknesses would be easily visible in a SM and are not unrealistic, as shown by CVE-2025-32433[3] [10].

We are referring to the Medium configuration, unless stated otherwise.

3 `https://nvd.nist.gov/vuln/detail/CVE-2025-32433`

			Sent by		Alphabet		
Symbol	Message type	ID	C	S	R	M	E
DEBUG	SSH_MSG_DEBUG	4	●	●	✓	✓	✓
DISCONNECT	SSH_MSG_DISCONNECT	1	●	●	✓	✓	✓
IGNORE	SSH_MSG_IGNORE	2	●	●	✓	✓	✓
KEXINIT	SSH_MSG_KEXINIT	20	●	●	✓	✓	✓
NEWKEYS	SSH_MSG_NEWKEYS	21	●	●	✓	✓	✓
SERVICE_ACCEPT	SSH_MSG_SERVICE_ACCEPT	6	○	●	✓	✓	✓
SERVICE_REQUEST_CONNECTION	SSH_MSG_SERVICE_REQUEST	5	●	○	✓	✓	✓
SERVICE_REQUEST_USERAUTH	SSH_MSG_SERVICE_REQUEST	5	●	○	✓	✓	✓
UNIMPLEMENTED	SSH_MSG_UNIMPLEMENTED	3	●	●	✓	✓	✓
KEX_RSA_DONE	SSH_MSG_KEX_RSA_DONE	32	○	●		✓	✓
KEX_RSA_PUBKEY	SSH_MSG_KEX_RSA_PUBKEY	30	○	●		✓	✓
KEX_RSA_SECRET	SSH_MSG_KEX_RSA_SECRET	31	●	○		✓	✓
KEXDH_INIT	SSH_MSG_KEXDH_INIT	30	●	○		✓	✓
KEXDH_REPLY	SSH_MSG_KEXDH_REPLY	31	○	●		✓	✓
KEX_DH_GEX_GROUP	SSH_MSG_KEX_DH_GEX_GROUP	31	○	●		✓	✓
KEX_DH_GEX_INIT	SSH_MSG_KEX_DH_GEX_INIT	32	●	○		✓	✓
KEX_DH_GEX_OLD_REQUEST	SSH_MSG_KEX_DH_GEX_REQUEST	30	●	○		✓	✓
KEX_DH_GEX_REPLY	SSH_MSG_KEX_DH_GEX_REPLY	33	○	●		✓	✓
KEX_DH_GEX_REQUEST	SSH_MSG_KEX_DH_GEX_REQUEST	34	●	○		✓	✓
KEX_ECDH_INIT	SSH_MSG_KEX_ECDH_INIT	30	●	○	✓	✓	✓
KEX_ECDH_REPLY	SSH_MSG_KEX_ECDH_REPLY	31	○	●	✓	✓	✓
UNKNOWN_ID_ALGORITHM_NEGOTIATION	n/a	22	○	○	✓	✓	✓
UNKNOWN_ID_KEY_EXCHANGE_SPECIFIC	n/a	49	○	○	✓	✓	✓
UNKNOWN_ID_RESERVED_0	n/a	0	○	○	✓	✓	✓
UNKNOWN_ID_TRANSPORT_GENERIC	n/a	9	○	○	✓	✓	✓
VERSION_EXCHANGE	n/a	n/a	●	●	✓	✓	✓
PING_OPENSSH	PING	192	●	◐		✓	✓
PONG_OPENSSH	PONG	193	◐	●		✓	✓
EXT_INFO	EXT_INFO	7	●	●	✓	✓	✓
NEWCOMPRESS	NEWCOMPRESS	8	●	○	✓	✓	✓
GLOBAL_REQUEST_TCPIP_FORWARD	SSH_MSG_GLOBAL_REQUEST	80	●	●			✓
GLOBAL_REQUEST_HOSTKEYS_OPENSSH	SSH_MSG_GLOBAL_REQUEST	80	○	●			✓
CHANNEL_OPEN_SESSION	SSH_MSG_CHANNEL_OPEN	90	●	◐			✓
CHANNEL_EXEC_REQUEST	SSH_MSG_CHANNEL_REQUEST	98	●	◐			✓

Table 3.3. Alphabets used in our experiments. They are designed to cover SSH-T as completely as possible, and includes optional or not yet defined messages as well. Apart from the concrete messages themselves, we show whether each party normally sends the message (●) or not (○). For some messages, sending is discouraged, but not forbidden (◐). In the last column, we show which messages are included in the different configurations Reduced, Medium and Extended respectively. Messages are grouped by corresponding standard.

3.3 Intermediate Format for Diffing

The SSH-State-Learner produces a human-readable output of the learned state machine, which is useful for manual inspection, but not suitable for applying LTSDiff. Some elements of the state machines are not preserved in the human-readable output, such as the exact socket state after each transition. Also, edges between the same states are merged, which improves readability.

We serialize the state machine into a machine-readable format. All transitions are kept intact separately. Transitions and states are annotated with the analysis results from the SSH-State-Learner. At a minimum, this includes whether the state/transition are part of the happy flow and whether a state is a sink state.

The graph is serialized in GraphViz DOT format and annotated with metadata like alphabet or SUL name.

3.4 State Machine Differentials

Given our state machines, we apply LTSDiff to produce state machine differentials. The differentials are analyzed to identify potential violations. To address our research questions, we first define what we expect of a differential so that it is usable with regard to **RQ1**. Subsequently, we describe different strategies to optimize calculated differentials to improve their usefulness.

3.4.1 Heuristic for Usable Differentials

Differentials produced by LTSDiff are, by definition, never wrong: They will always show the differences between LTSs, but they may not be useful as a shortcut to finding deviations. A trivial example: If two LTSs only differ in a single transition, a differential might show all of LHS being removed and all of RHS being added.

To properly answer **RQ1**, we therefore need to ensure that differentials can actually help a user find relevant deviations between SMs. We identify three baseline criteria of a desirable differential, which we use during our evaluation to rate the applicability of LTSDiff.

1. **The happy flow states must be matched**: Almost by definition, the happy flow must be contained within each state machine, regardless of implementation. The states involved in the happy flow should be functionally equivalent across state machines. If LTSDiff is unable to match these states, key behavior will be marked as deviations, which is not helpful.
2. **The sink state must be matched**: Any state machine will contain at least one terminal sink state that represents closed connections and where further input is ignored. For each individual state, there is only a limited amount of intended legal behavior, and we expect most transitions outside that to lead to the sink state. If LTSDiff cannot match the sink state, every error transition leading to that sink will be marked as a deviation even if it is functionally equivalent, which is not helpful.

3. **Most transitions must be matched**: A core assumption of our approach is that SMs are in essence similar, because they implement the same specification. Thus, most states and transitions should be shared between them. Matching happy flows and sink states already guarantees that most states can be matched, but transitions may not be. As a coarse similarity measure, we consider differentials more useful when a high proportion of transitions (e.g., more than half) are matched. However, this is not a strict requirement, because it is possible that a few matched transitions already cover the most important behavior, while the unmatched transitions are mostly noise.

We will apply these criteria to the differentials we generate to give a more objective measure of their usefulness.

3.4.2 Applying LTSDiff

LTSDiff calculates differentials for arbitrary LTSs, purely based on the structures and labels. We first apply plain LTSDiff to establish a baseline, using the recommended default parameters $t = 0.25$ and $r = 1.5$.

To refine the differentials there are also ways to tailor the algorithm to SSH specifically in several dimensions. There are different optimization strategies that we can employ at different steps in the process, which we describe here. Each strategy influences the quality of the resulting differential, and we evaluate their impact in Section 5.2.

Preprocessing. Before applying LTSDiff, we can already preprocess the learned state machines. Rather than using the raw state machine, we have the ability to change the SM input to LTSDiff. We consider removing certain transitions and states that carry little information and are not able to influence regular SSH behavior, like error edges and states whose transitions do not return to the happy flow. Removing transitions risks excluding important details, so we also consider keeping them, but replacing their labels with meta-symbols instead. This way, they are still present in the SM, but we can reduce their impact on the similarity calculations in LTSDiff.

Configuring LTSDiff matching. gLTSDiff offers option to determine how transitions and state similarities are calculated, by overriding the combinability operator and the default similarity metric. We can use meta-knowledge from SSH-State-Learner, to restrict or enforce certain state matchings. Given our heuristic, we can consider preventing differentials from matching happy or non-happy flow states with each other. We can even fully pre-determine the matching of the happy flow, and only use normal matching for the rest. Instead of relying solely on the default similarity metric, which uses the edge labels as a whole, we can also compare only specific parts of the labels.

Postprocessing. For readability purposes, we postprocess our differentials into a human-readable format. We merge all edges of the same type between the same states, to reduce

clutter. This change does not contribute to our heuristic, but it is necessary to later examine the differentials for violations.

3.5 Analyzing State Machines

The ultimate goal of the differential analysis is to identify behavioral discrepancies that indicate violations of the standard. Since our work focused on the SSH Transport Layer Protocol, our primary reference is RFC 4253, which directly defines it. RFCs specify normative requirements using the "MUST", "SHALL" and "SHOULD" keywords, as defined in RFC 2119 [16]. Since SMs only show input/output behavior, we only include requirements that would be visible, which excludes many cryptographic or semantic requirements. Therefore, we do not consider related standards like RFC 5656, which defines our chosen key exchange [55]. We do include extension negotiation from RFC 8308, which (visibly) happens during SSH-T.

For each implementation, we examine the usable differentials produced by LTSDiff as a starting point to identify how violations manifest. However, we do not limit ourselves to the differentials. We also examine singular isolated state machines, potentially unusable differentials and perform manual testing where necessary. This ensures our analysis for **RQ1** does not rely solely on our differential quality heuristic.

4 Implementation

In this section, we describe how we technically implemented the approach described in Chapter 3. In Section 4.1, we describe changes that were necessary to the SSH-State-Learner to allow client state learning. In Section 4.2, we specifically describe how we automated the different SSH clients, so that we could use them as SULs. Finally, Section 4.4 describes how we integrated gLTSDiff to compare SSH state machines.

4.1 SSH-State-Learner Integration

We first describe the basic lifecycle of a SUL during state learning that is already implemented in SSH-State-Learner [11], shown in Fig. 4.1. We omit some elements that are not relevant to our implementation.

During state learning, the learner will repeatedly query a `MembershipOracle` object from LearnLib, which wraps around a `SshSul` implemented by us. The `MembershipOracle` will do the following:

1. `pre()` is called to set up the SUL
2. for each abstract `SshSymbol` in the query `step(in: SshSymbol)` is called
   a) based on the ongoing query, SSH specific mitigations may be applied
   b) the `SshSymbol` is translated to a concrete `ProtocolMessage` and sent to the SUL
   c) Mapper waits for a response
   d) a `ResponseFingerprint` output symbol is created from received messages and the socket state
3. `post()` closes any open connections and prepares the SUL for the next query

Now in detail: All SULs have to implement LearnLib's `SUL` interface, so they can be used for learning by a corresponding `MembershipOracle` object. Any `EquivalenceOracle` is then created by wrapping that `MembershipOracle`. There is already a `SUL` implementation with the `SshSul` superclass, which contains settings for configuration provided by the state learner. This includes different optimizations that may simplify SUL behavior, e.g., the option to disable re-keying.

The main `SUL` functionality is implemented in the `NetworkSshSul` subclass. There are the three necessary functions: `pre()` for setting up the SUL, `post()` for shutting down and `O step(I in)`, which takes an input symbol `in` of class `I` and returns a corresponding output symbol of class `O`. Within `SshSul`, these abstract symbols are instantiated with the `SshSymbol` and `ResponseFingerprint` class respectively.

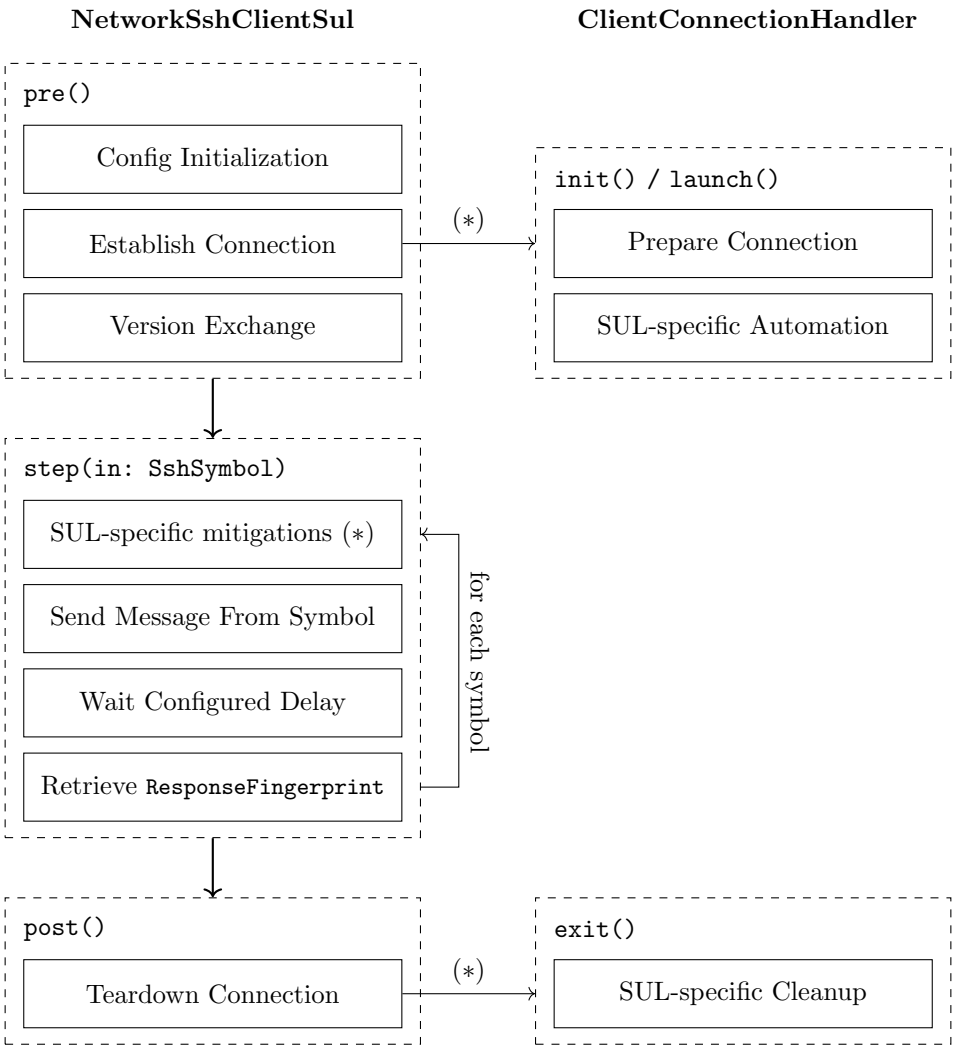


Figure 4.1. Lifecycle of a `SUL` object, showing the different steps and their relation to the `ClientConnectionHandler` interface. Steps marked with (∗) show where we extended the existing implementation.

`SshSymbol` is a pre-defined enum that represents the different SSH message types and their different parameterizations, as shown in our alphabets (cf. Table 3.3). Each `SshSymbol` produces a SSH-Attacker `ProtocolMessage`, which can send a real SSH message over the network. `ResponseFingerprint` consists of a (list of) received `ProtocolMessages`, as well as the state of the underlying TCP socket after receiving. Responses are considered equal if they contain the same `ProtocolMessages` in the same order and the TCP socket state is identical.

4.1.1 Re-introducing `NetworkSshClientSul` to the CLI

The SSH-State-Learner already supports client state learning, partially implemented by Martin [40]. We reuse the existing code, which introduces a client-specific `NetworkSshClientSul` class. Although functional, it is not fully integrated with the main codebase, and we re-introduce appropriate parameters to the CLI, so that users can properly instantiate client SULs.

We solve this by removing unnecessarily tight coupling between the core SSH-State-Learner components and moving functionality to a newly created `cli` package. The new `Main` class is then able to instantiate the `NetworkSshClientSul` with client-specific parameters.

4.1.2 Using `NetworkSshClientSul`

The `NetworkSshClientSul` implements the `pre()` and `post()` functions. Because client software typically initiates connections, it is responsible for establishing a server socket and then waiting for a client connection attempt. Here we implement a re-try mechanism, because we found establishing a new connection would sometimes time out depending on system load.

To implement the actual connection attempt, we implement the `ClientConnectionHandler` interface, which would then be called by the `NetworkSshClientSul`. The interface mirrors the `SUL` abstraction itself: `init()` performs some connection level setup, `connect()` performs the actual connection attempt, and `exit()` handles cleanup.

As described in Section 3.1, we aim for Docker and Appium-based clients, which would implement these differently. We implement two types of `SshClientControllers` that bundles common functionality for each of these categories. This leads to the class structure shown in Fig. 4.2.

Docker-based clients. Using the Docker API, the `DockerSshClientController` can manage SSH client containers, where the Docker daemon can handle most of the lifecycle. `init()` creates the corresponding Docker container, which has to be pre-built on the host machine. For easier debugging, each container shares the host's network stack rather than using a bridge network. `connect()` simply starts the created container and logs its output. Although we could also interact with containers by attaching to the standard input, we instead opted for non-interactive images. We go deeper into this in Section 4.2.1.

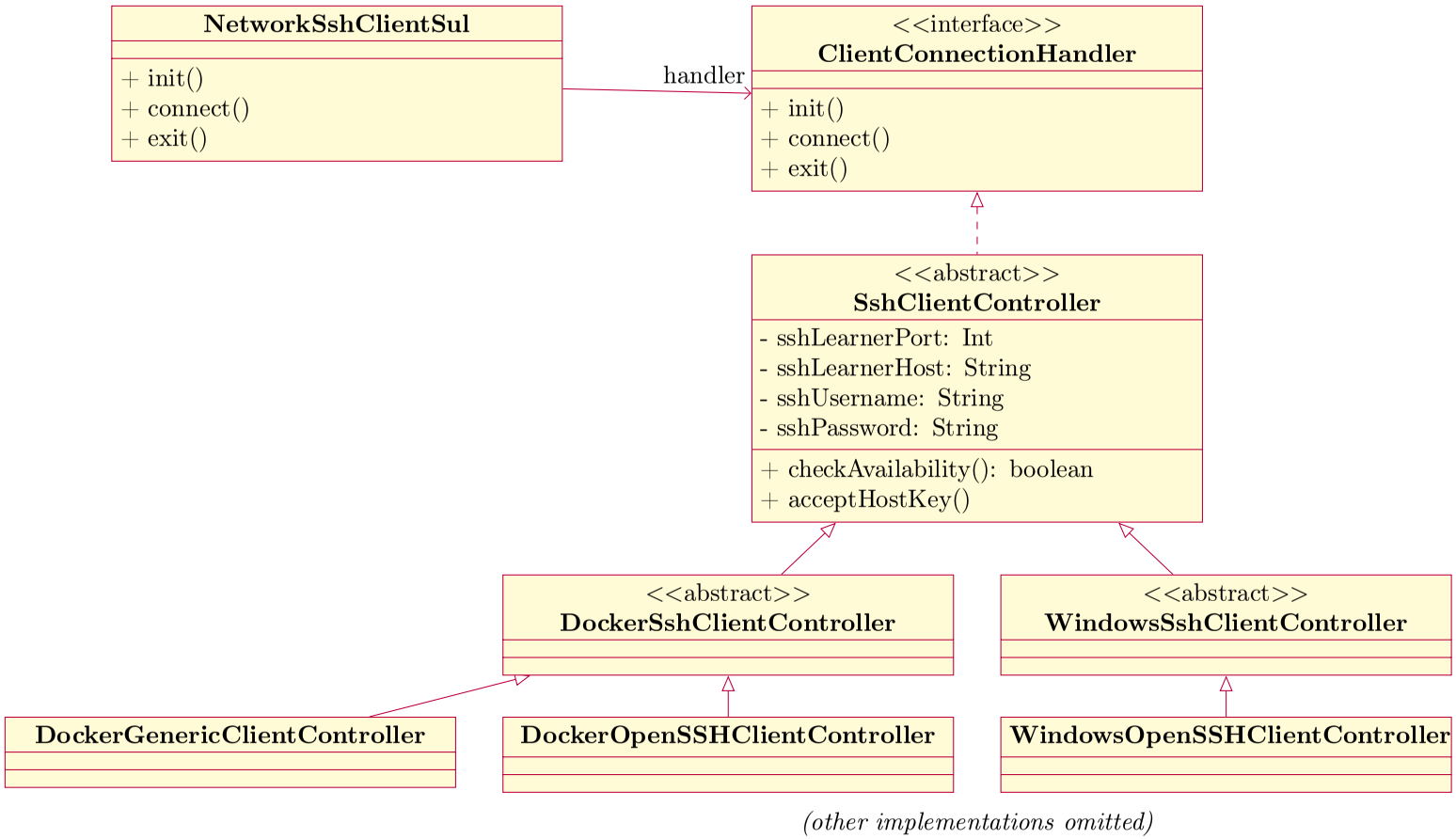


Figure 4.2. SULs implemented in SSH-Attacker, shown in UML notation. We implement the different SULs as subclasses of `SshClientController`, which implements the `ClientConnectionHandler` interface used within a SUL object. To accomodate different SUL we provide Docker and Windows/Appium specific client controllers to bundle common functionality. Leaf classes then represent concrete instantiation with software. The `DockerGenericClientController` can be used for compatible SULs via CLI.

Containers are ephemeral and are set to be removed after the process exits[1]. Once the connection attempt has been performed, `exit()` can generally default to stopping the container without further cleanup.

Windows-based clients. The `WindowsSshClientController` is designed to manage SSH client sessions by attaching to an existing setup Appium environment. Because there are many different ways of actually launching a Windows application, a clear separation between `init()` and `connect()` is not always possible or sensible. Instead, we choose the easiest approach for each respective client and implemented them in individual subclasses. We explore this in Section 4.2.2. The `WindowsSshClientController` itself only offers basic functionality for establishing the Appium driver and assuring cleanup.

4.1.3 Handling Guessed Key Exchanges

The underlying SSH implementation in SSH-Attacker does not support optimistically guessing the KEX algorithm. We encountered that at least Dropbear always performs this optimization, leading to crashes as soon as the learner receives a second KEX message.

To fix this, we made sure that the `KeyExchangeInitMessageHandler` class sets an appropriate flag in the `SshContext` if a `KEXINIT` guess was incorrect. The next respective message handler will then check this flag and simply ignore the message.

4.2 Client Automation

We target various SSH clients, ranging from purely terminal-based clients and libraries to clients with a GUI, on both Linux and Windows. In this section, we describe the automation strategies we used for different client categories.

For portability and reproducibility reasons, we prefer Docker containers when possible. Docker allows us to easily create isolated environments which are reproducible and cleanly removed after. This also allows us to re-use existing container images and configurations, specifically those compiled in the SSH-Docker-Library project.[2]

Docker is not suitable for all our needs: Due to the limited support of Docker for Windows and GUI applications[3], we chose to employ the Appium framework [6] on a fully-virtualized Windows deployment for GUI clients.

Appium is a set of platform-specific libraries that allow controlling GUI applications programmatically. It implements the (Selenium) WebDriver protocol which provides an HTTP API to control and interact with GUI elements. WebDriver has been specified for *web-based* user agents, but Appium maps the semantics of web applications to its different supported platforms. For Windows it also offers some platform-specific capabilities, e.g., launching applications.[4]

1 Using `docker run`, this is achieved by the `--rm` flag
2 `https://github.com/tls-attacker/SSH-Docker-Library`
3 While Windows-based Docker containers are possible, they can only run on a Windows-based host machine and specifically do not support GUIs [43].
4 `https://appium.readthedocs.io/en/latest/en/drivers/windows/#capabilities`

To support both Docker and Appium, we run the SSH-State-Learner on a Linux host machine, which can then interact with the Docker daemon and the Windows VMs running on the same host. The Windows VMs share the same base image and are controlled by `libvirt`[5] using QEMU[6] as a hypervisor. See Fig. 4.3.

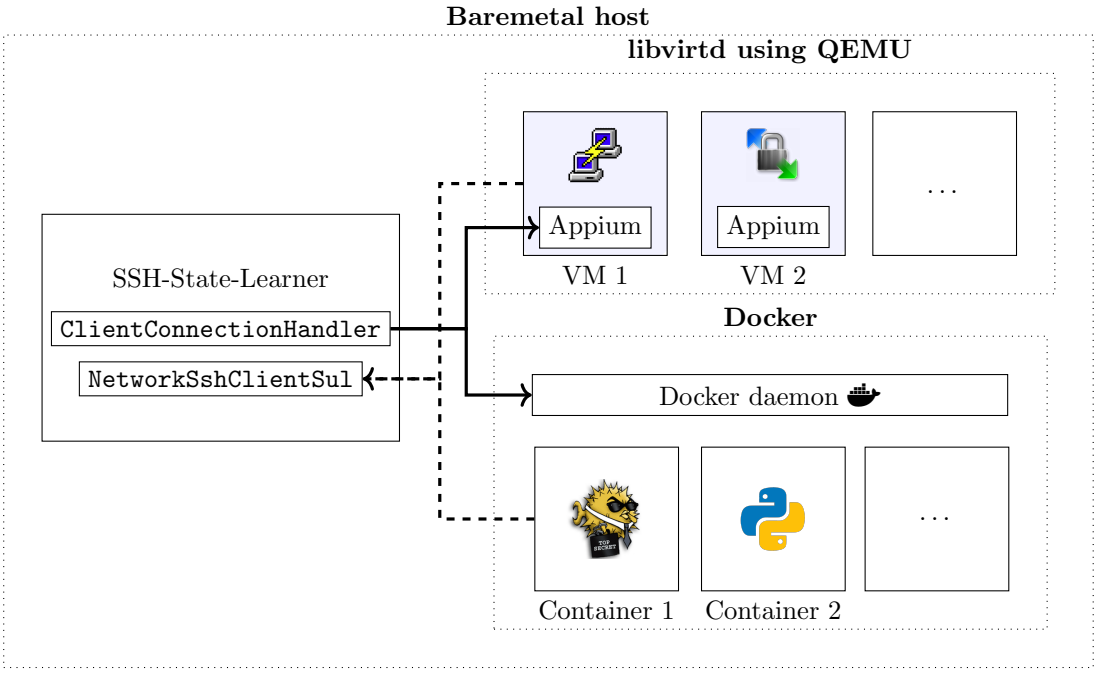


Figure 4.3. The SSH-State-Learner only ever interacts with the Docker daemon and Appium to trigger connection attempts (normal arrows). All SSH network communication is then initiated by the respective SUL, which connects to the learner SSH server (dashed arrows).

All clients follow the same basic automation procedure, as illustrated in Fig. 4.4. Clients will always perform a realistic connection attempt and then attempt to perform a basic action, given their use case. With our alphabets, authentication cannot actually succeed, but we included it for possible future extensions.

Host key handling differs between clients. We enabled automatic acceptance whenever possible. For other clients, we automated accepting the host key, and persist this for the duration of the experiment.

Also, the basic action is optional for some clients: Especially for rich GUI applications (e.g., FileZilla) the most basic action involves transferring a file, which is more complex than a simple command execution.

5 `https://libvirt.org/`
6 `https://www.qemu.org/`

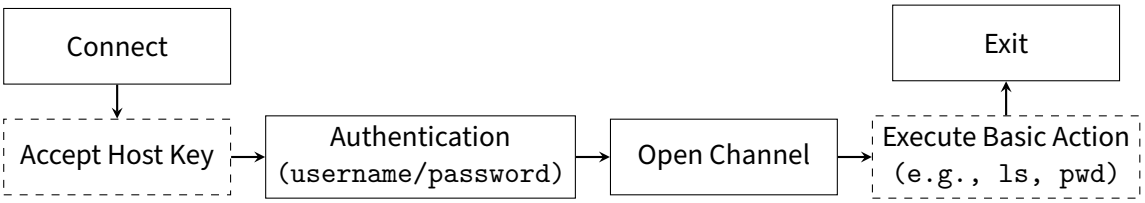


Figure 4.4. The basic automation loop for SSH clients. The learner initiates a connection attempt, which is then handled by the corresponding client controller. The client controller then simulates user interactions to perform the connection and execute a basic action (e.g., running a command). Finally, the client is exited to reset the state for the next query.

4.2.1 Dockerized Automation Strategies

Our dockerized clients should implement the automation flow, such that all interactions are handled upon launching the container and the container cleanly exits afterwards (regardless of the success of the connection attempt). For Dropbear and OpenSSH there already exist fully usable CLI clients, which we used. For AsyncSSH, GoSSH, Paramiko and RuSSH we implemented minimal clients based on the respective libraries.

Wrapping existing CLI clients. We employ `expect`[7] scripts to automate existing terminal-based SSH clients and use Docker containers to run them. This has been partially implemented in the SSH-Docker-Library already, from which we used the OpenSSH container. We also provide an automated container for Dropbear. For an example of an `expect` script, see Dropbear's automation in Listing A.1.

Building custom clients. For library-style clients, we still follow the same automation flow as before (see Fig. 4.4), and keep as close to examples provided by the respective libraries as possible. We made sure to include the least amount of functionality required. See Listing A.2 for the client we use with Paramiko.

4.2.2 GUI Automation Strategies

We use Appium to automate the GUI clients. In our setup, we use a pre-configured Windows VM image, that automatically starts Appium and exposes the API, accessible to the learner. There are different drivers compatible with Appium: We chose the FlaUI[8] driver, because it has better compatibility with newer Windows versions than the Appium-maintained driver.[9]

On the learner side, we use the Appium Java binding to interact with the Appium server. Every Appium interaction is isolated into a separate session, where only certain windows are accessible. Appium offers different capabilities, which are key-value pairs that serve as parameters for the automation session. Using different capabilities, we can

7 `https://core.tcl-lang.org/expect/index`
8 `https://github.com/FlaUI/FlaUI.WebDriver`
9 `https://github.com/appium/appium-windows-driver`

spawn new processes or interact with already running processes. For all clients, we choose the easiest way to reliably automate the GUI, described below.

Launching via start-up parameters. Even though most GUI clients are not built for automation, many offer some degree of automation via command-line parameters. WinSCP, Bitvise and PuTTY[10] all allow specifying connection parameters (e.g., hostname, username) as command-line arguments. For these clients, there is no separate `init()` and `connect()` phase, because the connection attempt is already performed during the launch of the application.

Simulating user interactions with metadata. When start-up parameters are insufficient, we use Appium to simulate user interactions with the GUI: Ideally GUI elements (mostly text fields or buttons) have been annotated with automation labels by the developers, which allow for easy automation. When using UI metadata, we can select certain GUI elements and trigger interactions via the Windows API, without simulating mouse movement. This also allows for more reliable automation, as we can define events such as a specific button appearing.

We use this strategy for FileZilla: While start-up parameters can be used to pre-fill certain fields, it will not start the connection on launch. We ended up using the metadata to select the button required to initiate the connection.

This strategy is suitable for most clients. Even when GUI elements are not annotated, element types (e.g., button, textfield, checkbox) or text content can also be queried to select GUI elements.

Simulating user interactions with coordinates. For the Termius client, we use primitive hard-coded coordinates, as it is Electron-based and its GUI does not expose standard Windows GUI elements compatible with our automation setup. This form of automation is far more brittle, because any GUI changes (e.g., resizing) may break the automation. It also provides less feedback to the programmer, because we cannot rely on UI events. As a result, we implement a mix of explicit wait times and changes of the window itself to achieve robust automation: While this approach works, we find it to be error-prone when dealing with dynamic content. We observed unpredictable loading times dependent on system load.

We decided against using Electron's native WebDriver interface to automate the GUI more reliably, because integrating the additional tooling would have required setting up an additional automation driver. Future work could add a set of Electron-specific automations to improve the learning for such clients.

Automating `cmd.exe`. Not all windows applications are GUI-based: The Win32-OpenSSH is a command-line application that runs within `cmd.exe`. We found that automating `cmd.exe` is far more difficult, as none of the tested automation drivers could be used to directly spawn a `cmd.exe` window. Invoking `ssh.exe` directly is also not an option.

10 Technically the PuTTY command line client `plink.exe`

Although it executes the application itself, this does not spawn a new window, stalling the driver. We believe this to be an unintended interaction with the driver itself, which also runs within a `cmd.exe` context.

We found the following workaround: Using the special `appium:app=Root` capability, we can attach to the Desktop session, which allows us to query all windows and their metadata. We then spawn a `cmd.exe` window with FlaUI's `powershell` module. We retrieve the internal window handle, which we then use to attach to the window within a new Appium session. Lastly, we send commands to the `cmd.exe` window by simulating keyboard input. While this approach works reliably, simulating keyboard input can be slow compared to other methods.

Limited concurrency. Due to shortcomings of the FlaUI driver and its integration with the Appium library, we cannot reliably use multiple threads for client SULs. We found that Appium timeouts and errors are not reliably propagated to the client code, which can lead to deadlocks, when the UI behaves unexpectedly. Most of the existing SSH-State-Learner code uses Kotlin coroutines, which could timeout, but the underlying library calls are blocking and cannot be cleanly interrupted.

This means, given the current implementation, any hangups in the UI automation are fatal. We found that the load of multiple VMs caused unpredictable delays in the GUI (e.g., a window taking longer to appear), which then triggered timeouts within the VM, which we then could not handle properly in the learner application. While this can be partially mitigated with longer delays between actions, experimentation showed the speedup gained with multiple SULs did not justify the added instability. Therefore, we did not run multiple client SULs in parallel, when learning GUI-based clients.

4.2.3 SUL-specific Mitigations

As mentioned above, the `SshSul` can also implement SSH-specific mitigations to improve learning. As described in Section 3.2.3, we disabled re-keying for the REDUCED alphabet. This had already been implemented in the `SshSul`: It can keep track of the current state of the connection and can close the connection if it detects a re-keying attempt. Because the learner only observes the input and output through the mapper, this will simplify the resulting state machine. During testing, we observed unexpected behavior that would prevent the state learner from learning proper state machines for OpenSSH and Termius. We add additional mitigations to work around these issues, which we describe below.

4.2.3.1 OpenSSH Ping-pong

The `ping@openssh.com` extension mandates that pings are not answered during re-keying "in which case they are queued until rekeying completes" [45]. This causes issues within the state machine, as the buffering behavior will result in pseudo states representing the queued pings. The large number of states would delay termination and also increase differences to other implementations. We restrict each connection to a single `Ping` or `Pong` message to mitigate this issue.

4.2.3.2 Termius Delays and Disconnection Behavior

We found the Termius implementation to behave inconsistently in terms of delays between messages and its disconnection behavior. We increased the time allotted for receiving an answer from the SUL to 600 milliseconds (6 times the default), until we found a stable configuration. Lower than that, we regularly observed race conditions between attempting to receive data and querying the connection state.

We also observed inconsistencies in whether the connection would be closed with after a `DISCONNECT` or not, even with the increased timeout. We decided to run all Termius-related experiments with three majority votes per query.

4.3 Intermediate Representation

Our intermediate representation is implemented in addition to the existing representation as a separate `DOTVisualizationHelper`, which is already contained in LearnLib. GraphViz DOT[11] is a language to define graph structures in a textual format. Because it is already integrated into LearnLib and also has interfaces for other programming languages (e.g., `pydot`[12]), we decided to embed additional information into the DOT representation.

We (ab)use the fact that DOT allows custom attributes to include SSH-State-Learner metadata, serialized as JSON, while still being valid DOT. Listing 4.1 shows how this looks in practice.

```
digraph g {
 _sshstatelearner="{
     \"title\":\"TERMIUS_GUI\",\"protocol_stage\":\"[TRANSPORT]\",
     \"kex_algorithm\":\"ecdh-sha2-nistp521\",\"strict_kex\":\"false\",
     \"alphabet\":
         \"MSG_DEBUG;MSG_DISCONNECT;MSG_EXT_INFO;MSG_IGNORE;MSG_KEXINIT;
         MSG_NEWKEYS;MSG_NEWCOMPRESS;MSG_SERVICE_ACCEPT;MSG_SERVICE_REQUEST_USERAUTH;
         MSG_SERVICE_REQUEST_CONNECTION;MSG_UNIMPLEMENTED;MSG_KEX_ECDH_REPLY;MSG_KEX_ECDH_INIT;\"
     }"
 s0 [shape="circle"
     _sshstatelearner="{\"initial_state\":\"true\",\"happy_flow_state\":\"0\"}"
     label="s0"];
 s0 -> s1 [_sshstatelearner="{\"happy_flow\":\"true\"}"
           label="MSG_KEXINIT / KeyExchangeInitMessage,EcdhKeyExchangeInitMessage <UP>"];
 ...
}
```

Listing 4.1. Excerpt of a DOT representation of an SSH state machine with embedded SSH-State-Learner metadata. The overall state machine contains metadata such as the client name, protocol stage and used KEX algorithm. Each state and transition also contains additional metadata, e.g., whether a state is initial or part of the happy flow.

11 https://graphviz.org/doc/info/lang.html
12 https://github.com/pydot/pydot

4.4 State Machine Differentials

As described in Section 3.4.2, we considered strategies configurations to improve LTSDiff results. Each option in this section corresponds to the pre-processing, configuration and post-processing there. We give an overview of what and where we implemented these modifications. In Section 5.2 we show how we actually applied them in practice.

4.4.1 Preprocessing

Given the intermediate representation, we can already apply some basic transformations to the state machine during parsing. We implemented this in the `SSHDotParser`, which generates the appropriate gLTSDiff automaton from the DOT representation. It can be configured to omit all transitions leading to the error state or overwrite their labels to a common meta symbol. We also implemented the possibility to omit transitions that do not lead to a new state.

4.4.2 gLTSDiff

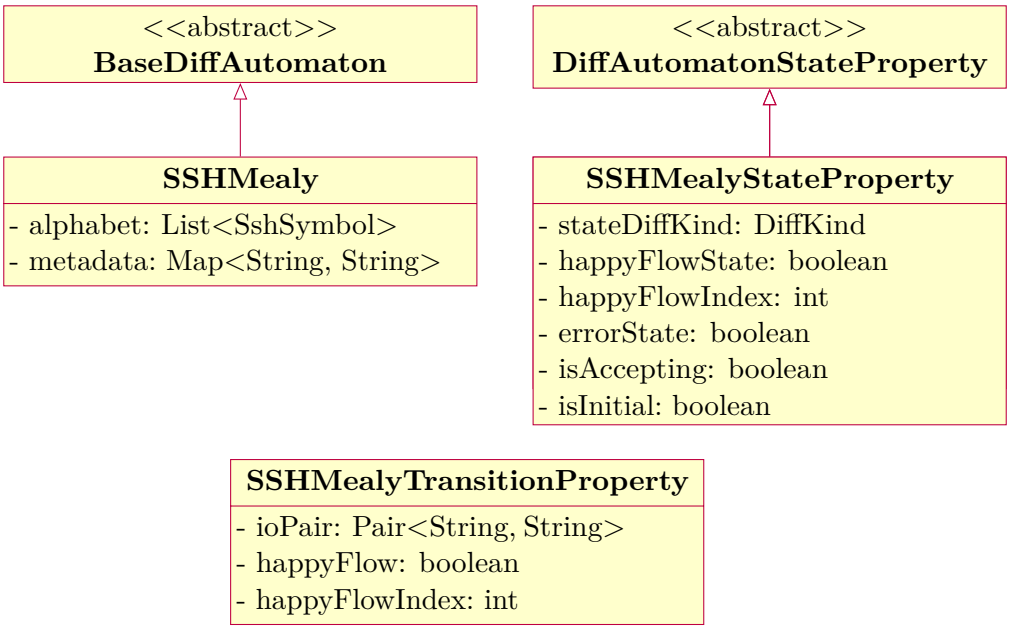


Figure 4.5. Class diagram for the gLTSDiff integration. We had to inherit from the existing base classes to add additional information, which we could then access for our custom matching and scoring implementations. The `SSHMealy` class represents the state machine, which uses the `SSHMealyStateProperty` and `SSHMealyTransitionProperty` to represent states and transitions. We also implemented custom combiners for states and transitions, as well as a custom scorer to force certain matches.

As a base, we use gLTSDiff's abstract `BaseDiffAutomaton` class, which represents a LTS where states are annotated with `isInitial` and `isAccepting` properties. All states

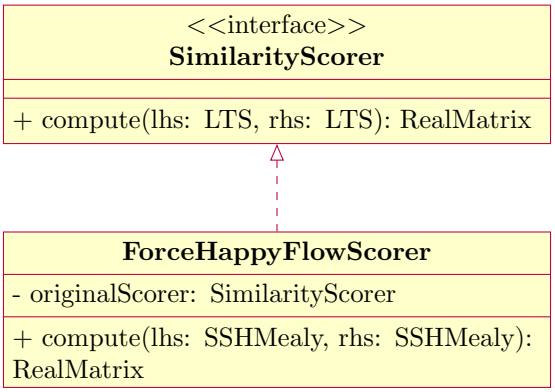


Figure 4.6. Class diagram for the `ForceHappyFlowScorer` class, implementing gLTSDiff's `SimilarityScorer` interface.

and transitions are also annotated with a `DiffKind` to express whether they have been added, removed or remain unchanged. We add the `SSHMealy` subclass, which then also includes the alphabet and other potentially arbitrary metadata.

The `SSHMealy` class uses the `SSHMealyStateProperty` and `SSHMealyTransitionProperty` to represent states and transitions in our state machine. The states and transitions can then use the SSH-specific information from the SSH-State-Learner's analysis for the matching process: Currently, this uses *happy flow* information from the `HappyFlowClassifier(s)` [11] and whether a state is identified as the error state.

During LTSDiff's state matching process, objects are compared based on a `Combiner` implementation: The default implementation for states allows all states to be combined, but with a custom `SSHMealyStatePropertyCombiner` we can decide based on the state attributes. We implemented a combiner that only combines states if they are both happy flow states, or both error states. For edges, the default implementation considers edges equal if their labels are identical, but we used a custom `SSHMealyTransitionPropertyCombiner` to implement alternatives. Here we allowed comparing only part of the label, e.g., only the input. This leads to the classes shown in Fig. 4.5.

gLTSDiff also allows custom implementations to calculate similarity scores based on the state and transition properties. We provided the `ForceHappyFlowScorer` class, which implements a specialized similarity scoring mechanism for SSH state machines, where we use the happy flow information to pre-determine certain matches. We pair the sequence of happy flow states and each pair is assigned a similarity score of 1.0, while other states receive a score of 0.0 to clear the internal thresholds set by LTSDiff. We need this when we want to force certain states to be matched like we describe in Section 3.4.2: Even when the `Combiner` only permits the desired combinations, LTSDiff may still not match them when the similarity score is below the threshold.

4.4.3 Postprocessing

After the matching process, instances of the `Rewriter` class can be used to transform the state machine differential. Visually, we use this to improve readability, by combining edges between the same states into a single edge combining their labels.

We can also use the `Rewriter` to apply further filtering: One possible filtering step is to remove error edges from the final result.

The other filtering step is to remove unmatched states that do not have a path back to the happy flow. This is similar to filtering out the states before the matching process, but it will keep states that are not part of the happy flow, if they are included in both implementations.

Lastly, we implemented multiple `Printer` subclasses, which can be used to visualize the state machine and its properties. gLTSDiff also uses the DOT format for visualization, although independently of implementation in LearnLib. In our context, `ADDED` states/transitions are drawn green and dotted, while `REMOVED` is drawn red and dashed in the resulting state machine. All happy flow states and edges between them are bolded.

5 Evaluation

In this chapter, we evaluate the applicability of LTSDiff for SSH client state machine analysis. First, we describe the experimental setup used to obtain our set of state machines in Section 5.1. In Section 5.2, we evaluate LTSDiff's performance according to the metric defined in Section 3.4.1. We explore different strategies designed to improve the quality of the differentials. In Section 5.3, we show various standard violations that can be detected using these differentials. Lastly, Section 5.4 discusses de facto protocol semantics not explicitly given by the standard. They are important for understanding the behavior of SSH clients in practice.

5.1 Learned state machines

Our evaluation experiments were performed on a host machine with an Intel i7-8700K CPU and 48 GB of RAM, running Debian 12 on version 6.1.159-1 of the Linux kernel. Docker based clients were limited to six concurrent instances, while Appium-based clients were limited to one (see Section 4.2.2). The Windows VM, running Windows 10 Enterprise LTSC Build 19044, was assigned two of the 12 logical CPU cores and 6 GB of RAM. The VM was assigned a static IP address and communicated with the learner via a software-defined network managed by `libvirt`.

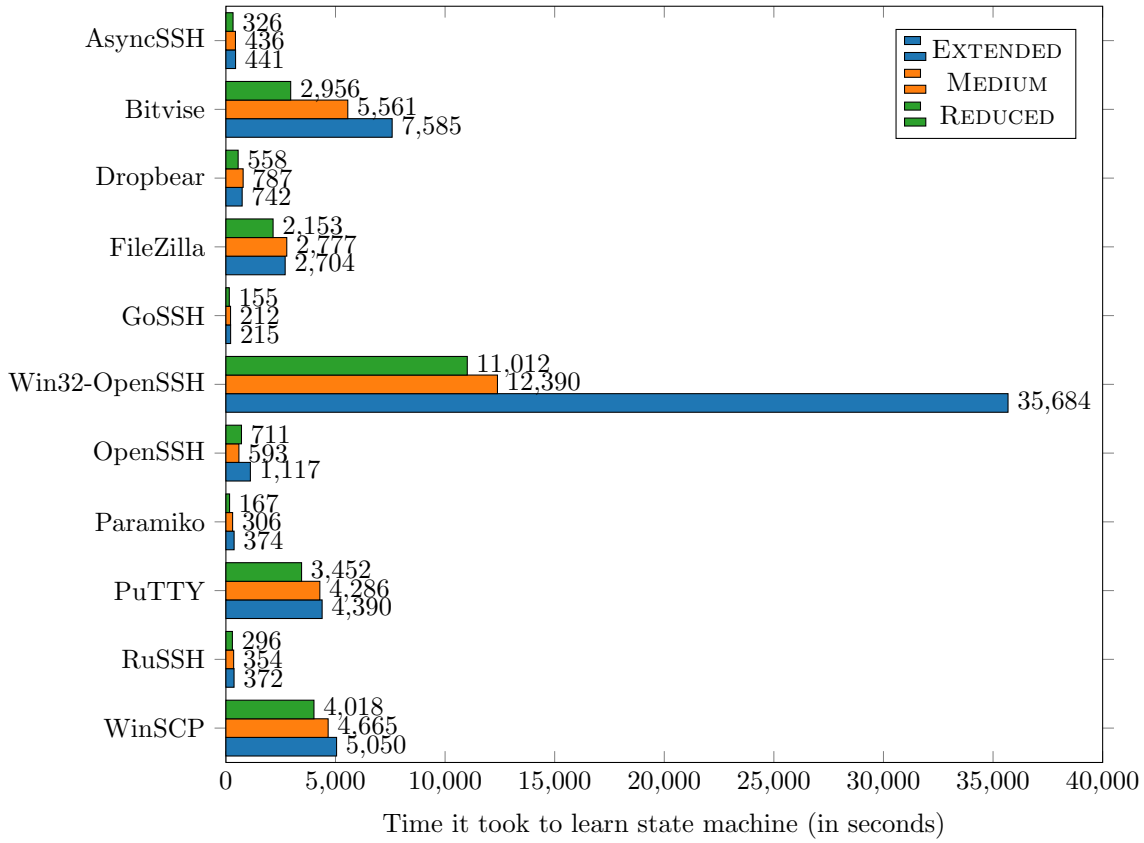
Each SML run was limited to 12 hours of learning time, except for Termius which we limited to 36 hours to account for majority voting (cf. Section 4.2.3.2).

Unsurprisingly, the learning time for GUI-based clients was significantly higher than for terminal-based clients. Even though learning time ranges from several minutes to up to 10 hours, we consider this acceptable. Learning duration (cf. Fig. 5.1a), can be partially attributed to the impact of automation. However, Fig. 5.2 shows that some SULs also required significantly more oracle queries, which correlates with larger learned state machines. Termius is an outlier in both size and learning duration and total queries. It timed out in every configuration, so we use the latest hypothesis as its state machine instead. Despite our additional mitigations, this indicates that the resulting SUL is too unstable for learning. Under the MEDIUM configuration, the number of states ranges from 6 to 13 states, averaging around 11 states. Dropbear and OpenSSH are among the larger state machines, but remain within the general range. In contrast, Termius consistently shows substantially larger machines in all configurations.

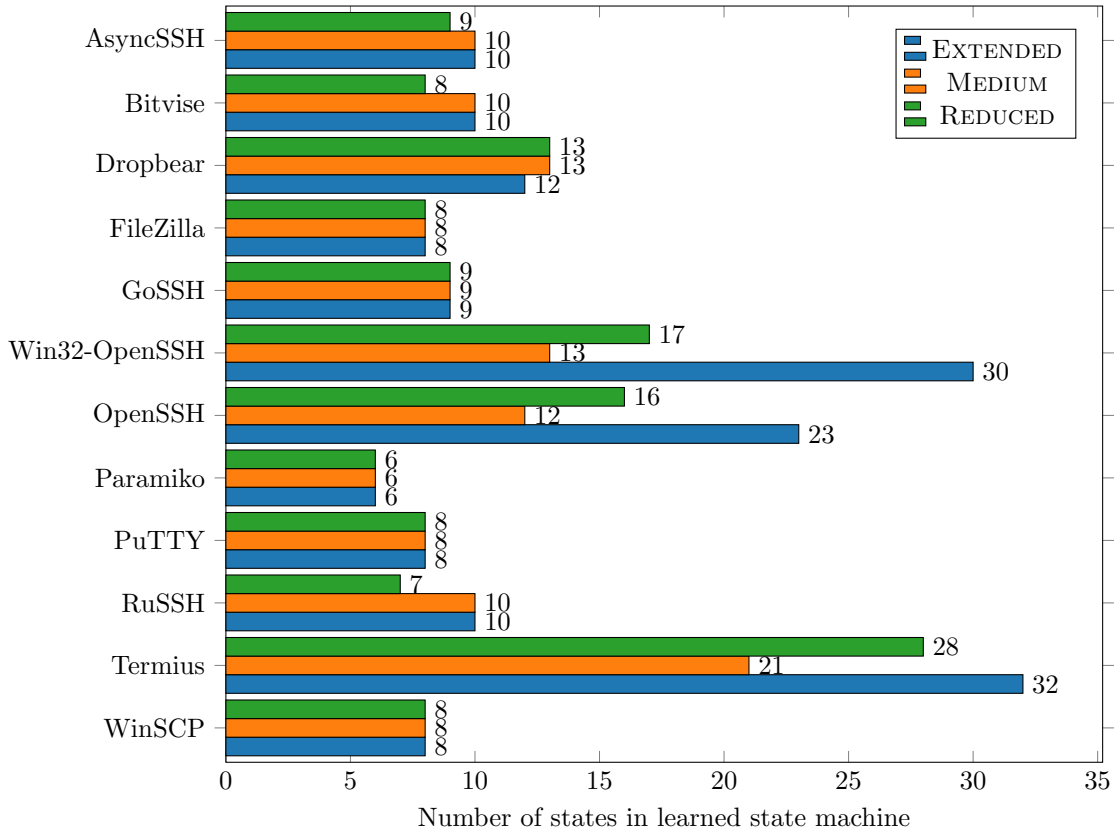
Under the EXTENDED configuration, we see some significant increases in Win32-OpenSSH/OpenSSH, where it almost doubles the size of the state machine from 13 to 30 and 12 to 23 states, respectively. The other implementations show little to no change, peaking at three additional states for RuSSH. This either indicates that EXTENDED

explores genuinely more complex behavior or that the approximated regions outside the happy flow are more easily triggered.

Interestingly for OpenSSH and Termius, the MEDIUM alphabet actually showed the opposite effect. This suggests an interesting interaction between the input alphabet and our equivalence oracle, where some additional messages can create counterexamples that work better for approximated regions of the state machine. The number of resets also supports this, as the MEDIUM configuration for OpenSSH and OpenSSH did not require significantly more resets than the REDUCED configuration, like we see for the other SULs. In fact, for OpenSSH the MEDIUM configuration required fewer resets than the REDUCED configuration. Overall, the state machine size indicates that despite our large input alphabets, most SSH clients implement the transport layer compactly.



(a) The total time it took to learn the state machine for each SUL under the different alphabets. We generally see an increase with the larger input alphabets, although differences are mostly minor. Win32-OpenSSH is the clear outlier, where Extended triples the learning time. Termius is excluded because every run timed out within the allotted time.



(b) Number of states in the learned state machines for each SUL under the different alphabets. Most implementations remain compact, while OpenSSH and Termius are sensitive to the larger input alphabets, because of our oracle.

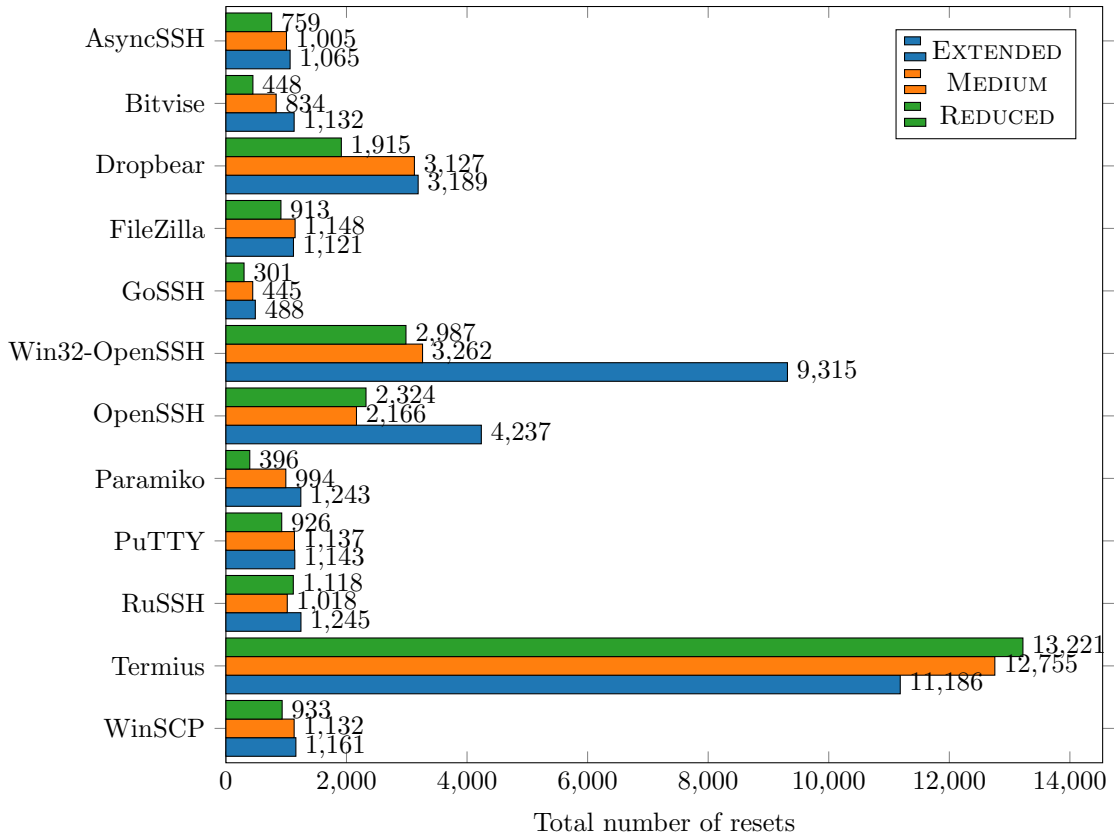


Figure 5.2. Number of SUL resets performed during learning for each SUL under the different alphabets. This metric reflects the time required for the learning process, independent of automation overhead. Implementations with larger state machines (notably OpenSSH and Dropbear) require more resets. We see little impact of increasing the input alphabet, except for OpenSSH and Win32-OpenSSH, where the Extended configuration requires significantly more resets. Termius is a clear outlier due to its unstable behavior, but since all configurations timed out, relative differences between the individual measurements are not meaningful.

5.2 LTSDiff for implementation comparison

To answer **RQ1**, we evaluated LTSDiff's performance given the criteria outlined in Section 3.4.1, namely matching the happy flow and error state, and a high proportion of total matched states and transitions.

Over the course of our evaluation, we explored different strategies of pre- and post-processing the SMs, and how to configure LTSDiff. In this section, we will describe the different strategies and explore error cases that we encountered.

For each strategy, we calculated the percentage of usable differentials, according to our criteria, over all unique implementation pairs. Recall that our final criterion is matching at least half of all transitions. To gain better insight into the effect of our modifications, we will examine the differentials both with and without this last requirement. To also account for the complexity of the underlying SMs, we also compare the results for both the Reduced and Medium configuration. Because our focus remains on SSH-T, we decided to exclude the Extended alphabet for this. All results are summarized in Table 5.1.

5.2.1 Naive LTSDiff

We first applied Plain LTSDiff directly without any modifications, using the intuitive label mapping also described in Section 2.3. These baseline results indicated, that LTSDiff is largely ineffective for our use case, given our criterion for useful differentials. Both the Reduced and Medium configuration produced relatively few usable differentials, at 15.15% and 16.67%, respectively. These account for 10 and 11 of the total of 66 unique implementation pairs. Based on these initial differentials, we had two main observations, which motivated further experiments.

Error edges. Because the set of *intended* messages at any point is relatively small, almost every state had many error edges leading to the sink state. In fact, we found that depending on the implementation up to 80% of all edges were error edges. These error edges are a significant factor in the quality of LTSDiff's matching step: If the error edges appear dissimilar, states may be deemed less similar, impacting matching quality. We found that this often occurred, because implementations would behave differently when closing the connection (we explore this further in Section 5.4). An example of this can be seen in Fig. 5.3. Here we found that a strategy of rewriting edges with a meta-label could increase similarities, which we call Rewrite. While the actual messages sent during disconnection may differ, all transitions resulting in a closed connection are functionally equivalent. We can therefore normalize the labels of all edges leading to the error state to a single meta-label `ERROR`, before applying LTSDiff. We also evaluated the approach of removing error transitions entirely, which we call Remove. While normalization prevents different ways of error signaling from reducing similarity, the sheer amount of error edges could in fact artificially inflate state similarities and drown out more meaningful edges.

Happy flows. We assumed that states along the happy flow should be matched between implementations. In practice, we found that many differentials did not match these as

Method	Reduced in %	Medium in %
Plain LTSDiff	15.15 (46.97)	16.67 (21.21)
Domain Knowledge (Section 5.2.2)		
Rewrite	28.79 (68.18)	36.36 (56.06)
Remove	0.00 (0.00)	0.00 (0.00)
Match	15.15 (42.42)	19.70 (27.27)
Force	13.64 (83.33)	24.24 (83.33)
Match+Rewrite	28.79 (75.76)	33.33 (57.58)
Force+Rewrite	25.76 (100.00)	37.88 (100.00)
Input Only (Section 5.2.3)		
Input-only	42.42 (90.91)	60.61 (84.85)
Input-only+HF-only	63.64 (100.00)	60.61 (92.42)

Table 5.1. Usable differential rates for different LTSDiff configurations. Each cell contains two values: The first value is the percentage of usable differentials over all unique implementation pairs. The second value (in parentheses) is the percentage of usable differentials without requiring matching at least half of all transitions.

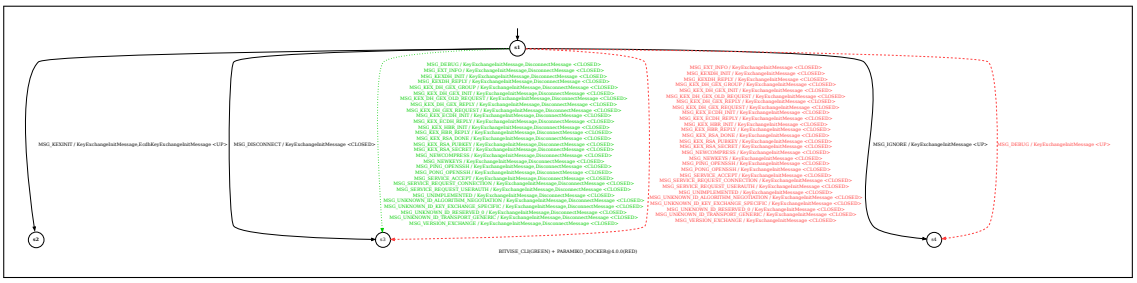


Figure 5.3. An example of how error edges cannot be matched, because one implementation sends a `DISCONNECT` message, while the other simply closes the connection. Even though all but one of the messages leading to the sink state $s3$ are shared between the two, they manifest as different edges, making the diff unnecessarily large. In this case the two initial states leading to $s1$ can be matched because the have high predecessor similarity.

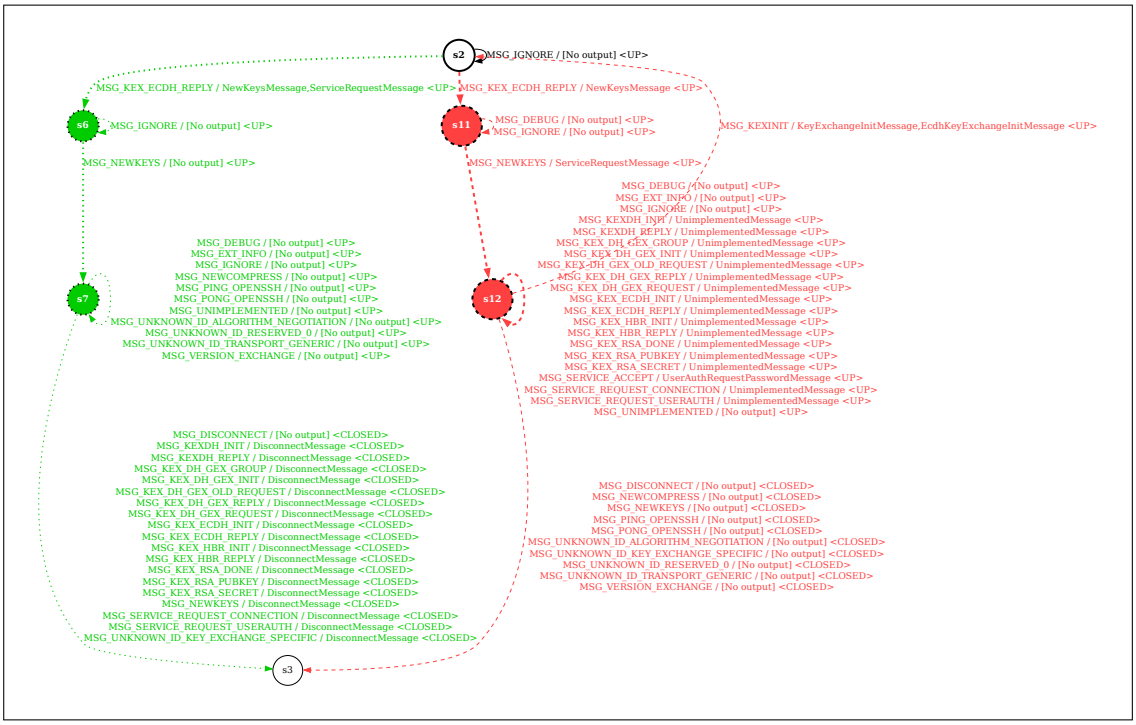


Figure 5.4. A reduced excerpt from a diff between Bitvise (`ADDED`) and Paramiko (`REMOVED`), generated by Plain, in which two happy flow states are not matched. The states $s6/s11$ and $s7/s12$ appear dissimilar because their outgoing edges are different, even though the analysis shows they are each part of the happy flow.

expected or even match happy flow states with non-happy flow states. In the REDUCED configuration, the happy flow was matched in 49.97% of cases, for the MEDIUM configuration, this was only 22.73%. Happy flow states were not matched based on the singular important happy flow edge, but rather on the combination of edges leading to and from the state. We explored two strategies to take this into account. We restricted happy flow states to only allow being matched happy flow states in the Match strategy. This would at least prevent happy flow states from being mistakenly matched to a non-happy flow state, but does not guarantee that these states find their proper match. Therefore, we also used a forced matching strategy Force, where we pre-determine happy flow state pairs.

5.2.2 LTSDiff with Domain Knowledge

Rewrite, Match and Force improved our differentials to varying degrees. Rewrite significantly improved how often the happy flow states were matched, increasing the total rate of usable differentials to 27.27% for REDUCED and 37.88% for MEDIUM Especially in the MEDIUM configuration, the rewriting strategy appears effective, where a larger alphabet means that per state more error edges will be present. Remove on the other hand, generated no usable differentials at all (0.00%). While it diminishes the effect of edges to the error state on

the similarity score, we found that there are also various other error handling strategies, e.g., simply ignoring erroneous inputs. Removing all error edges then makes states appear less similar, and prevents matches. Based on this, we explored error handling behavior further in Section 5.4.

Examining Match and Force also brought surprising insights. We found that some implementations like Dropbear only have four effective states traversed by the happy flow, instead of five, which we implicitly assumed based on our happy flow (cf. Section 2.1.1). This was unexpected, and we explore it further in Section 5.4. Our heuristic checks for absence of unmatched happy flow states, not for semantic equivalence.

For the Reduced configuration, Match reduces the number of differentials that fully match the happy flow, because when they are of unequal length, surplus states remain that are never matched. In the plain configuration, they can still be matched with non-happy flow states, but in the happy flow matching configuration, this prevents matching the fully happy flow in 3 implementation pairs (4.55%). In Force states that do not have a clear match can also be matched with non-happy flow states. This leads to higher happy flow matching rates of 83.33%, but also produces some unusable differentials, because some forced state pairings exhibit many unequal transitions. Still, it produces a higher rate total of usable differentials.

By applying Force+Rewrite, we achieve the most effective strategy, but we still only cleared our heuristic in 39.39% of cases. While we were able to significantly improve the rate of matched happy flows to up to 100.00%, this still produced many unusable differentials, which had many unmatched transitions. We found that many edges were simply different in subtle ways, e.g., OpenSSH was the only implementation to always send an `IGNORE` before `NEWKEYS`. This results in differences as shown in Fig. 5.5, where functionally equivalent behavior is represented differently.

It shows how functionally equivalent behavior may be represented differently, which hinders the matching process when comparing edge labels for equality. Although we could employ meta-labels for situations beyond errors, doing this for all edge types is difficult. Instead, we explore the strategy used by Ang et al. for QUIC [4] instead. They completely ignore the output symbols for each transition and only consider the input symbols and also exclude all edges that do not cause a state transition. This Input-only strategy has interesting implications for LTSDiff: Because the original SM is total (i.e., every input has a corresponding transition in every state), when only the input symbols are considered, states are inherently locally similar. By abstracting from output symbols, Input-only shifts the similarity comparison from syntactical to structural equivalence. This obsoletes the need for the Rewrite and Remove strategies, and should ideally yield happy flow match rates comparable to Force while being less complex.

5.2.3 LTSDiff with Input only

Ignoring the output symbols and comparing transitions only based on input yielded the highest rate of useful differentials, significantly outperforming previous configurations. We were able to match the happy flow similarly close to as often as before, without any domain knowledge required. While it was slightly down compared to Force+Rewrite, 90.91% and

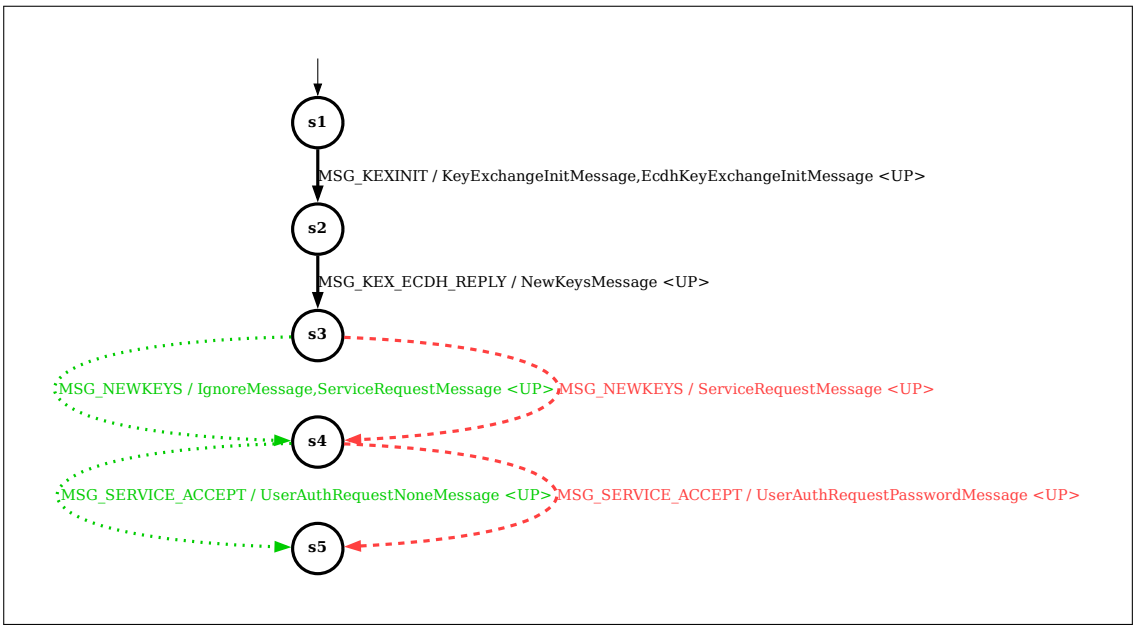


Figure 5.5. A reduced excerpt from a diff between OpenSSH (`ADDED`) and RuSSH (`REMOVED`), generated using Force+Rewrite, in which the states belonging to the happy flow are correctly matched, but not all transitions are. OpenSSH sends an `IGNORE` message before `NEWKEYS`, while RuSSH does not. They also send different `USERAUTH_REQUEST`, causing the final transition to show a deviation.

84.85% versus 100.00%, we achieved a far higher rate of usable differentials, because disregarding the output would match all cases like Fig. 5.5. For both REDUCED and MEDIUM, this improves the rate of usable differentials to 43.94% and 60.61%, respectively, which is a significant improvement over the previous best of 39.39%, using Force+Rewrite on MEDIUM.

Based on the size observations in Section 5.1, we considered one additional optimization over Ang et al., HF-only. In it, we excluded all states that do not have a path to return to the happy flow, after seeing implementations like Dropbear, where such states make up 40% of all states. This is a potentially aggressive optimization, as such states can still contain interesting behavior, but due to our equivalence oracle, states far from the happy flow are more likely to be approximated incorrectly. While this optimization did improve the REDUCED case by 19.70%, it did not improve the MEDIUM case, although it successfully matched the happy flow for five more implementation pairs. We deemed this optimization not worthwhile, and use Input-only for the remainder of our evaluation.

5.2.4 Impact of Complexity

As can be seen in Table 5.1, the domain knowledge approaches tend to be more effective in matching the happy flow, when the state machines are smaller in the REDUCED configuration. At the same time all domain knowledge strategies struggle with the large variance in output labels, which only grows when considering a larger alphabet.

The Input-only strategy is not affected by this, and seems to benefit off of larger state machines with more state changes to focus on. We therefore expect it to scale more robustly for even larger state machines.

We can also examine which implementations tend to produce unusable differentials, by looking at the pairwise matrix, see Fig. 5.6. Here we find that Termius, Dropbear and OpenSSH, the three implementations with the largest state machines, also tend to produce the most unusable differentials. Dropbear and OpenSSH mostly fail at matching half of all transitions, while Termius often fails at the happy flow matching.

This shows that the third part of our heuristic (at least 50% of transitions matched) is not well suited for all pairs of state machines: There are implementation pairs with a large difference in state machine size. In these cases, there are inevitably states that will not get matched, and the associated transitions will likewise remain unmatched, violating the heuristic. Therefore, we disregarded the last criterion when choosing which differentials to examine manually.

5.2.5 Differentials for comparing software versions

While we focus on comparing different implementations, it is important to point out, that different software versions can also induce state machine changes. In our experiments, we also compared different versions of the same implementation to test our software. Recall, that we only learned different versions for dockerized implementations.

Generally, none of the previously described issues, are a problem for LTSDiff when considering the same software: Output behavior is much more consistent between versions,

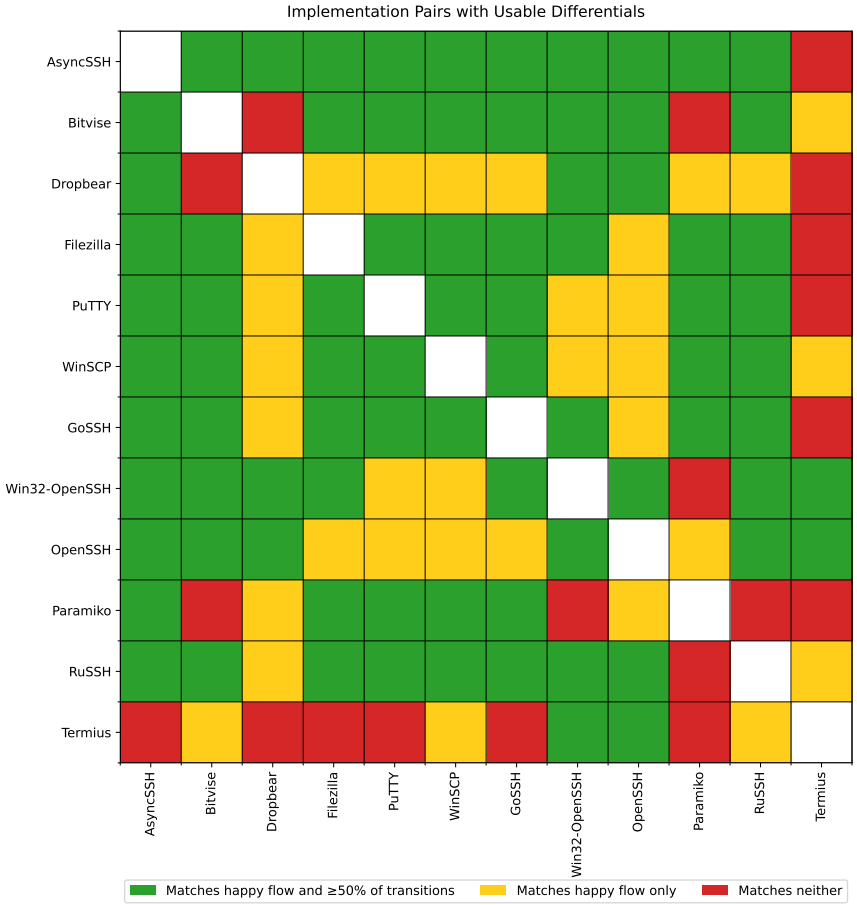


Figure 5.6. Matrix showing the quality of differentials for each pair of implementations. We see that Termius is by far the strongest outlier, only generating usable differentials with both OpenSSH variants. Dropbear and OpenSSH/Win32-OpenSSH tend to produce unusable differentials, because they often fail the transition matching, which is likely due to their large state machines.

so naive LTSDiff already performs well - of 24 unique version pairs, 91.67% (22) were usable according to Section 3.4.1. Applying the input-only strategy improves this to 100%, so only output labels seem to remain a problem between versions, although a minor one.

Looking at the differentials confirms this and indicates that changes to the state machine are mostly very small and localized (i.e., addressing specific corner cases) In fact, GoSSH, Paramiko, and RuSSH did not show any differences between versions at all. Dropbear and OpenSSH indicated changes in one state, but turned out to be false positives from approximation far from the happy path. AsyncSSH is the only exception, where output behavior changed globally, which we also show in the following section.

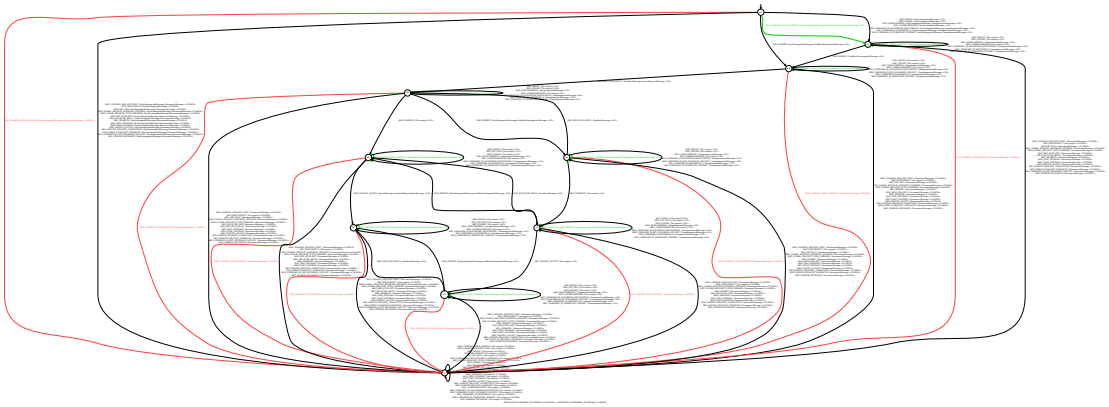


Figure 5.7. Example of a constructed differential between two versions of AsyncSSH, showing a vulnerability in which an unauthenticated server can open a channel at any time. This is generated by Plain LTSDiff. The vulnerability manifests itself as several added self-transitions that indicate the client confirms opening the channel. On the other hand, each removed edge shows the original secure behavior, in which the connection would be closed.

5.2.6 Example Differentials

We first show some examples of inter-version differentials, which are typically easier to interpret, to familiarize the reader with the concept.

In Fig. 5.7, we show a diff for a vulnerability allowing an unauthenticated server to open a channel at any time. We specifically constructed this diff using a custom fork of AsyncSSH, because it is very manageable visually. It also ties in with our Extended alphabet and shows the kind of vulnerability we would like to detect with it. Note, that this simple example is especially outrageous: Apart from being illegal during SSH-T, opening a session from server to client is actively discouraged [35]. However, it demonstrates well how simple changes can introduce large differences in the state machine, and how these can be easily spotted in the differential.

Fig. 5.8 and Fig. 5.9 both show a real change to AsyncSSH, introduced in commit `b88eed6`[1], which removes `IGNORE` messages that were previously sent during happy flow. Sending `IGNORE` violated connections with Strict KEX, so they simply removed them altogether. The former diff is generated using naive LTSDiff, while the latter uses the input-only strategy. Naive LTSDiff generates a large diff because it is unable to combine different output labels, while input-only fully matches both state machines. Note, that in the visualization we keep both labels to the edge, but they are being treated as the same edge during matching.

Lastly, we show some of our final inter-implementation differentials, generated using Input-only, which showcases usable differentials, as well as those, where our heuristic fails. In Fig. 5.10 we see our ideal differential, which we correctly classify as usable. Fig. 5.11 shows a usable differential, that we incorrectly classify as unusable due to unequal happy flows. Fig. 5.12 presents another unusable differential, where the happy flow is fully

1 `https://github.com/ronf/asyncssh/commit/b88eed615a583eab77987bd213d15b1cc9a63a3a`

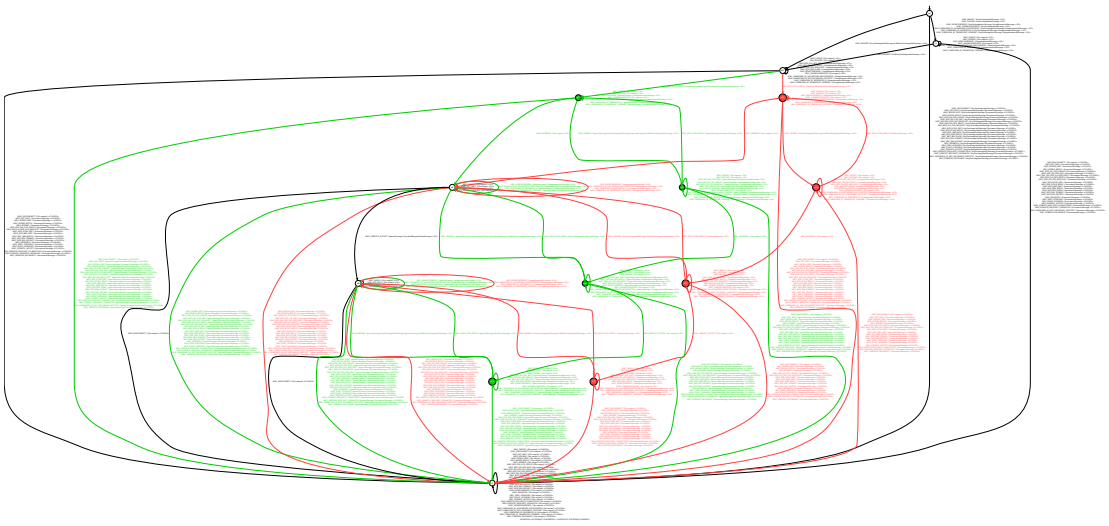


Figure 5.8. Example of a real differential between two versions of AsyncSSH which removed `IGNORE` messages during happy flow, generated using Plain. Almost every state and edge is affected by this change and can no longer be matched.

matched, but there are many unmatched transitions because of a size difference. We also include two actually unusable differentials in the appendix. Fig. A.3 illustrates a false positive case, where the happy flow states are matched with non-happy flow states. Finally, Fig. A.2 shows a true negative case, where the happy flow is not matched at all.

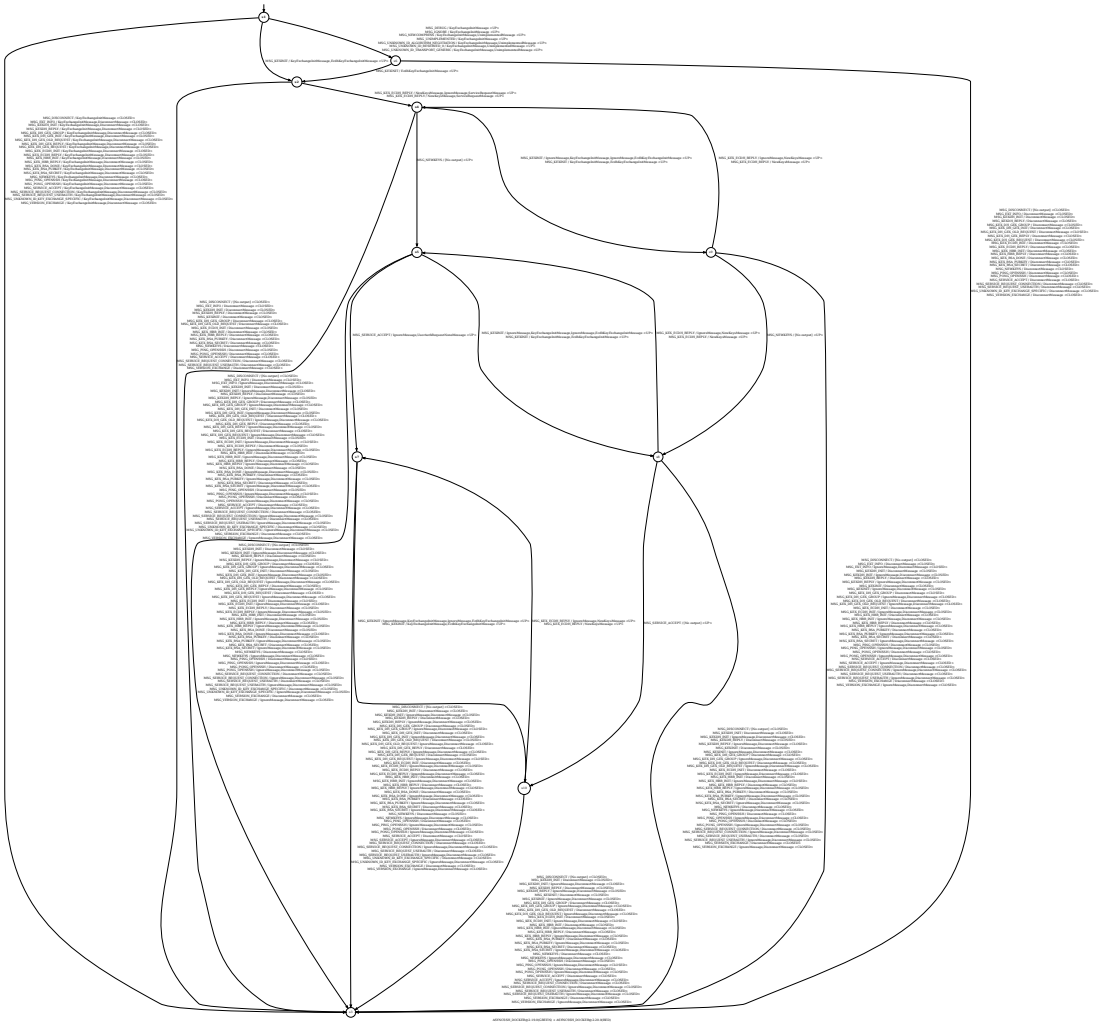


Figure 5.9. The same differential as Fig. 5.8, but generated using the Input-only strategy. Here we see that the Input-only strategy is able to match the complete state machine, because the output labels are no longer considered. The merged edges display both the original and modified behavior, but they are being treated as the same edge during matching.

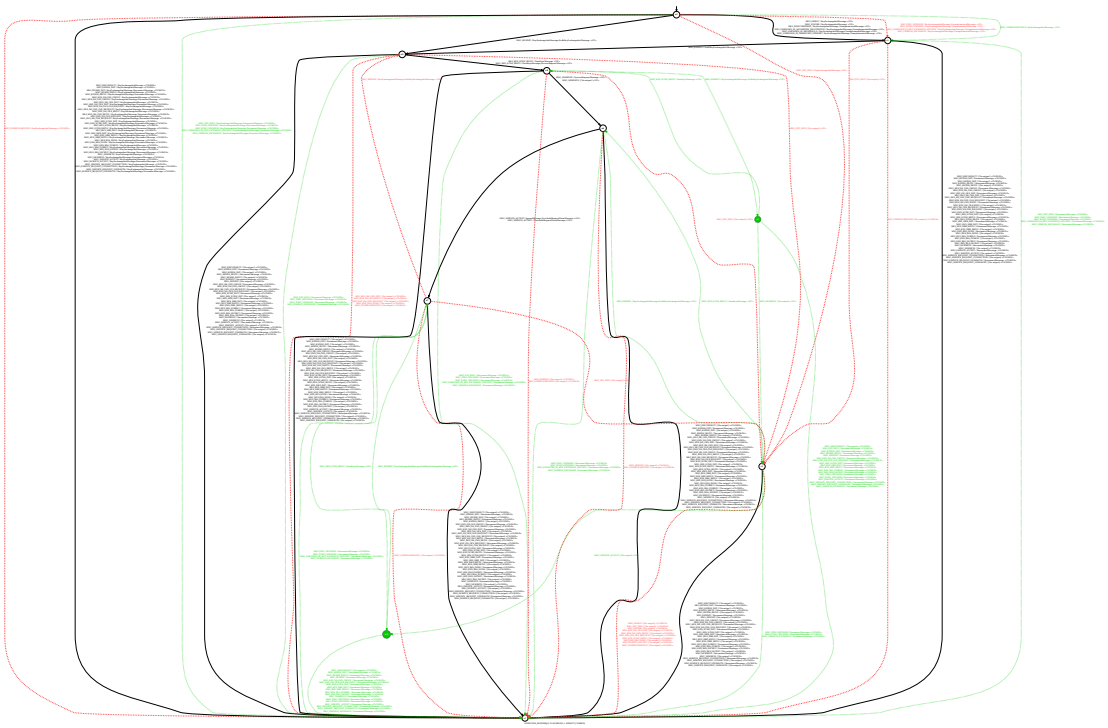


Figure 5.10. A usable differential between AsyncSSH (`ADDED`) and WinSCP (`REMOVED`), according to our heuristic. The happy flow is fully and correctly matched in the differential and overall, there are only few surplus states that were not matched and deviating edges are mostly composed of singular words.

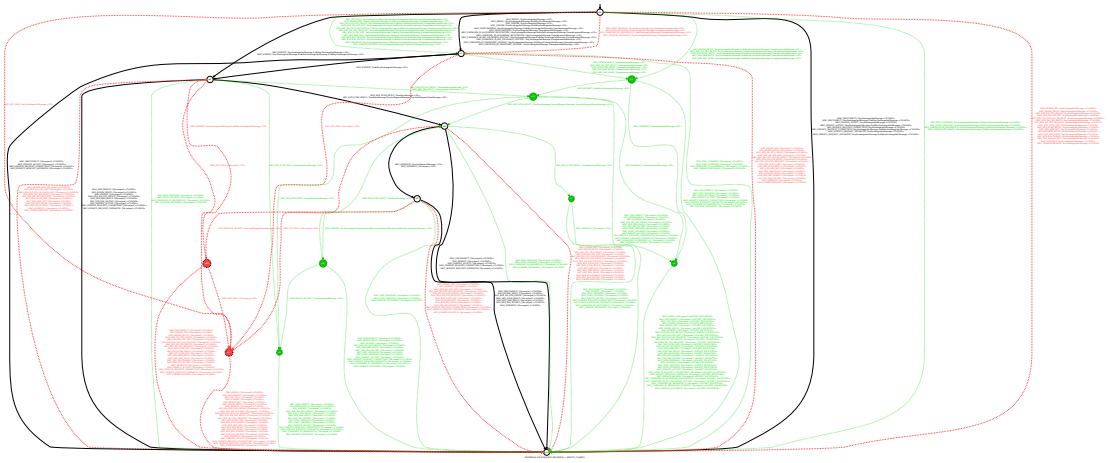


Figure 5.11. An unusable differential between Dropbear (`ADDED`) and WinSCP (`REMOVED`), according to our heuristic. The happy flow is not fully matched, because Dropbear only has four effective states along the happy flow, while WinSCP has five.

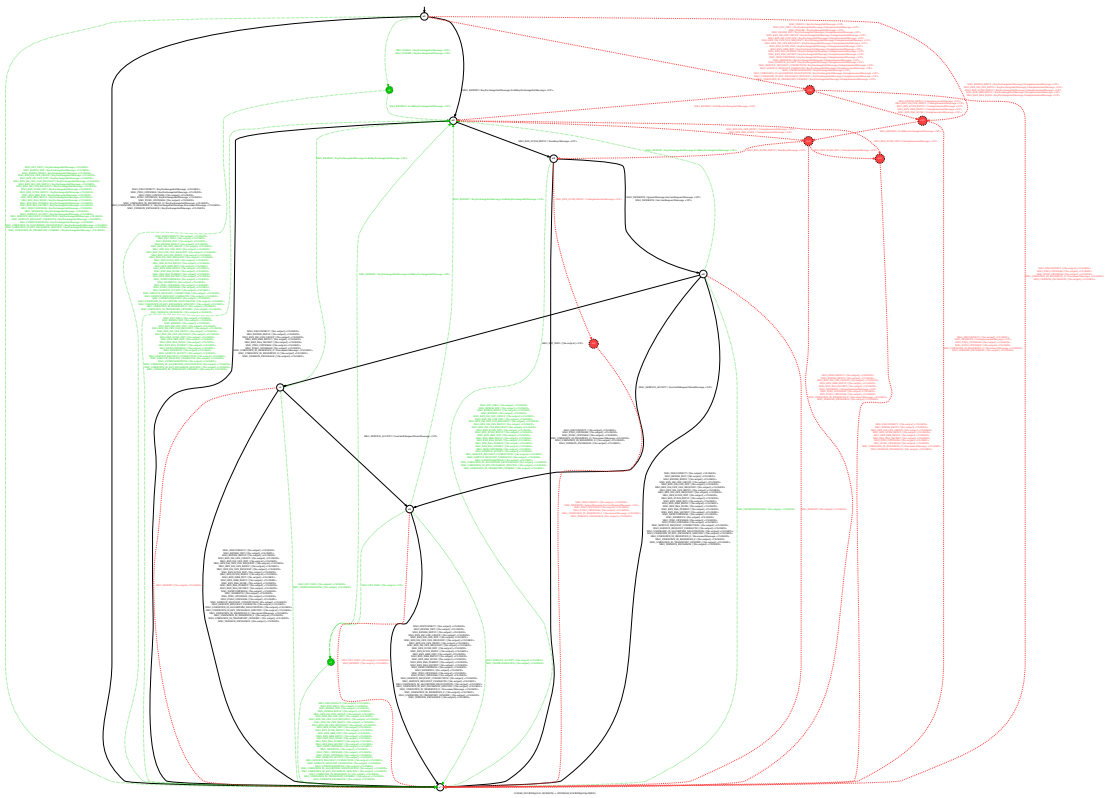


Figure 5.12. An unusable differential between GoSSH (`ADDED`) and OpenSSH (`REMOVED`), according to our heuristic, when considering transition matching. The happy flow is fully matched, but there is an isolated cluster of surplus state that introduce transitions that cannot be matched, because OpenSSH's SM is significantly larger.

5.2.7 Implementation Similarity

A usable differential can still contain a large amount of differences, which is why we also calculated the similarity of different implementations according to Section 2.3.3. The results can be seen in Fig. 5.13.

None of the pairs are particularly surprising. OpenSSH and Win32-OpenSSH are effectively equal: While the input-only strategy accounts for such differences, Win32-OpenSSH has a false surplus state, introduced by the way the mapper handles closed connections during state learning. PuTTY, FileZilla and WinSCP turn out to be identical, and there are no (visible) differences from their individual codebases. Dropbear and OpenSSH/Win32-OpenSSH achieve the highest similarity outside the related implementations, which is likely because of their similar size.

The main take-away is that almost all other implementations are fairly dissimilar, which is surprising given the shared standard. Including different versions (see Fig. A.4) shows the same trend.

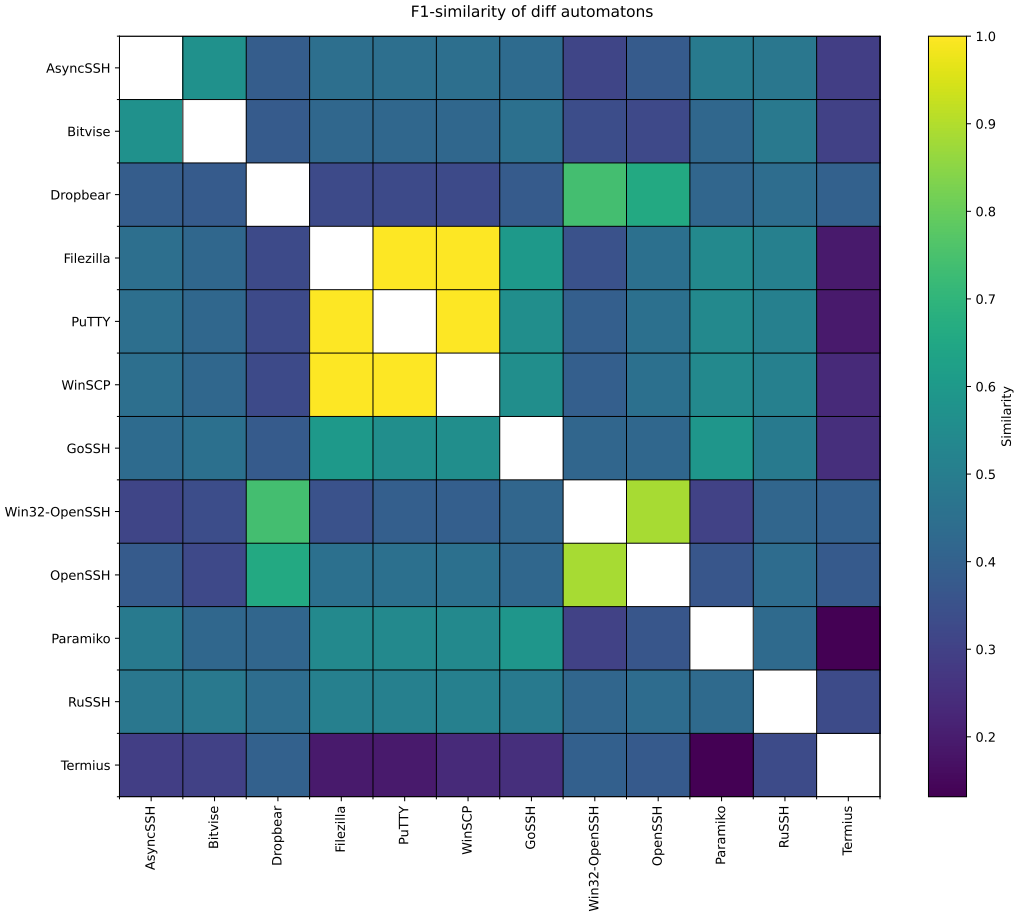


Figure 5.13. Pairwise similarity matrix calculated according to Section 2.3.3. We find low similarity scores throughout all unrelated implementations. The only exception is the pair of OpenSSH and Dropbear, which are similar, presumably because of their similar size. Also, FileZilla, PuTTY and WinSCP achieve a similarity of 1, which is not surprising given their shared codebase.

5.3 Standard Violations

The SSH-T protocol specification uses 85 “MUST”[2], 1 “SHALL” and 34 “SHOULD”[3] requirements as defined in RFC 2119. Most of these define cryptographic behavior or semantic properties of protocol message.

We only found 9 MUST (and no other) requirements that define behavior which would show in our SMs. We paraphrase them as follows:

2 “This word, or the terms "REQUIRED" or "SHALL", mean that the definition is an absolute requirement of the specification” [16]

3 “This word, or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course” [16].

R1 Implementation must answer key re-exchange requests with `KEXINIT` [37, Section 9]

R2 Service requests must be answered with `SERVICE_ACCEPT` or rejected with disconnection [37, Section 10]

R3 Any implementation must handle `DISCONNECT` properly [37, Section 11.1]

R4 All implementations MUST understand (and ignore) `IGNORE` [37, Section 11.2]

R5 All implementations MUST understand `DEBUG` message, but they are allowed to ignore it [37, Section 11.3]

R6 An implementation MUST respond to all unrecognized messages with an `UNIMPLEMENTED` message [37, Section 11.4]

Additionally we include the following requirement from RFC 8308, which happens directly after the key exchange:

R7 A servers `EXT_INFO` may only be sent after `NEWKEYS` or before `USERAUTH_SUCCESS` [15, p.5]

We include it because extension negotiation can be crucial for security and partially shows up during SSH-T.

Implementation	R1	R2	R3	R4	R5	R6	R7
AsyncSSH	●	●	●	●	●	○	●
Bitvise	●	●	◐	●	○	○	●
Dropbear	●	◐	●	◐	◐	◐	●
FileZilla	●	●	●	●	●	●	●
GoSSH	●	●	●	●	●	○	●
OpenSSH	○[a]	◐	●	●	●	○	●
Win32-OpenSSH	○[a]	◐	●	◐	◐	○	●
Paramiko	●	◐	●	●	●	○	●
PuTTY	●	●	●	●	●	●	●
RuSSH	●	◐	●	●	●	○	◐
Termius	●	○	●	●	●	○	◐
WinSCP	●	●	●	●	●	●	●

[1] ●Standard-conforming
[2] ○Standard-violating
[3] ◐Partial violation (likely unintended)
[a] Violates the standard as written, but likely intended

Figure 5.14. Summary of adherence to MUST requirements. We find several implementations that do not fully comply with the requirements from RFC 4253 or at least exhibit behavior that we deem not to be within the specification's intent.

These requirements do not necessarily cover all corner cases, so we make some additional considerations. For **R2**, it is not defined what happens when these messages are sent

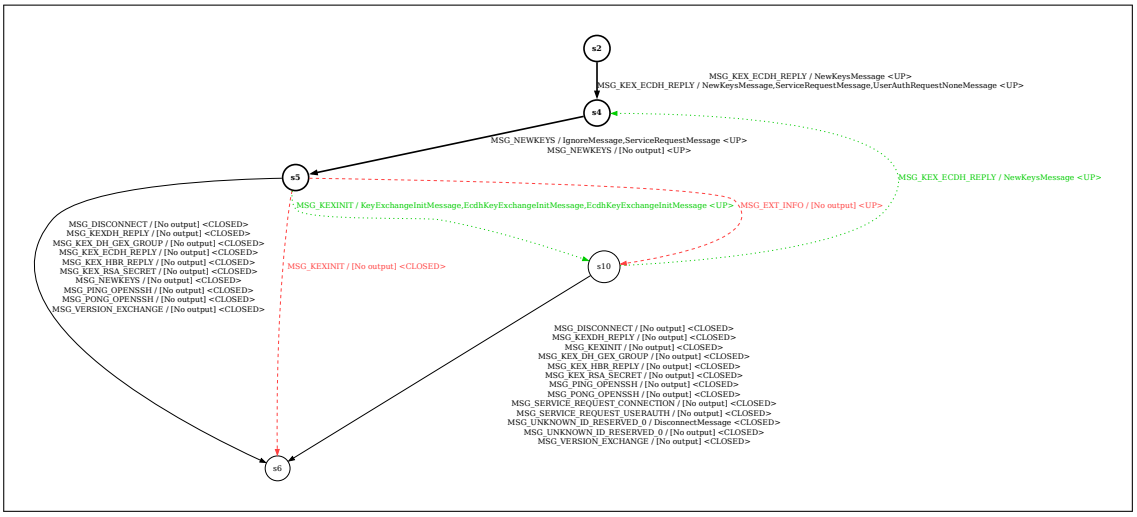


Figure 5.15. A reduced excerpt from a diff between OpenSSH (`REMOVED`) and Dropbear (`ADDED`). After transitioning to the post-KEX state $s5$, Dropbear allows the server to initiate re-keying, transitioning to $s10$ via `KEXINIT` and back to $s4$. OpenSSH simply closes the connection and does not allow re-keying. The state $s10$ is still matched, because it resembles the state OpenSSH is in after receiving a `EXT_INFO` at this time.

outside the intended time. For our purposes, we considered any occurrences in which there is no clear rejection, as a partial violation of what we believe to be the intended behavior. It increases the potential for interoperability issues and may create new attack surfaces where peers become desynchronized. We did the same for **R7**, if the implementation indicated extension negotiation support. For **R3**, **R4**, and **R5**, we consider any state in which either of these requirements is not fulfilled at least a partial violation.

Following the requirements above, we observed various violations and inconsistencies, which we describe below. Each type of violation is illustrated with an example from our differentials.

Allowing re-keying. Almost all implementations allowed re-keying at all legal times, except OpenSSH: OpenSSH and Win32-OpenSSH do not allow server-side re-keying before user authentication. The standard states that re-keying "requires a fair amount of processing power and should not be performed too often" [37], therefore this can be seen not as a full violation, but rather a stricter interpretation of the standard, as there is no valid reason to re-key this early. See Fig. 5.15.

`SERVICE_REQUEST` handling. The handling of `SERVICE_REQUEST` varies significantly between implementations. We find that some implementations (Dropbear, Paramiko) do not actually require the server to confirm the `SERVICE_REQUEST`. They send `USERAUTH_REQUEST` directly with the `SERVICE_REQUEST` and will proceed to a full connection even if no `SERVICE_ACCEPT` is ever received. Both OpenSSH variants require the service acceptance and will, differently from other implementations, answer repeated `SERVICE_ACCEPT`s. See

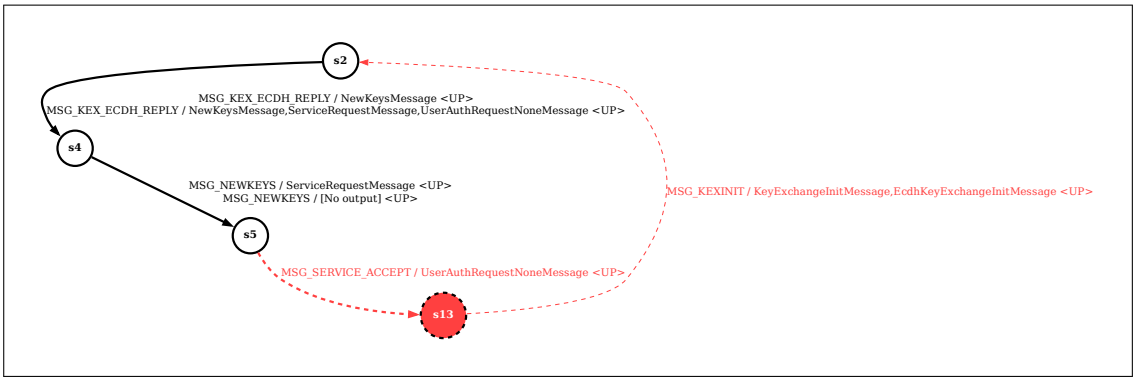


Figure 5.16. A reduced excerpt from a diff between WinSCP (`REMOVED`) and Dropbear. There is simply no fourth happy flow state for Dropbear (compared to the bold edge to $s13$), because in Dropbear there is no actual change in I/O behavior after receiving a `SERVICE_ACCEPT` message, at least not in terms of output behavior. Note, that the respective looping edge is present in the output of the state learning, but is not shown here, because it falls away during pre-processing.

Fig. 5.16. RuSSH will accept `SERVICE_ACCEPT` at any time during key exchange without complaints, but does not actually handle it. Termius will actually buffer `SERVICE_ACCEPT` messages and process them after KEX.

Disconnection. The handling of `DISCONNECT` is mostly consistent, which is unsurprising given its simple nature. During our analysis, we found one exception in Bitvise. We were able to trigger a deadlock state in which the client will not respond to any further messages, but will keep the connection open indefinitely as long as the server is still sending messages. See Fig. 5.17.

IGNORE and DEBUG handling. `DEBUG` and `IGNORE` messages are not arbitrarily allowed by all implementations. With Bitvise, we observed that `DEBUG` messages will close the connection immediately when sent before completing the key exchange. See Fig. 5.18. We also found some unrecoverable states that would disconnect also on `DEBUG` and `IGNORE` after two `NEWKEYS` messages. We explored this further in Section 5.3.1.

Reserved message handling. The handling of "reserved messages" has by far the most inconsistencies. Note that the corresponding section [37, Section 11.4] applies to "all unrecognized messages".

Only PuTTY (and by extension WinSCP and FileZilla) and Dropbear correctly handle reserved messages and always respond with `UNIMPLEMENTED` when receiving an unrecognized reserved message. GoSSH and Paramiko behave consistently, but not conforming and will close the connection on any unrecognized message. The remaining implementations behave inconsistently, depending on message and context. E.g., Bitvise will disconnect on any reserved message, prior to the first `NEWKEYS`. OpenSSH will close the connection for

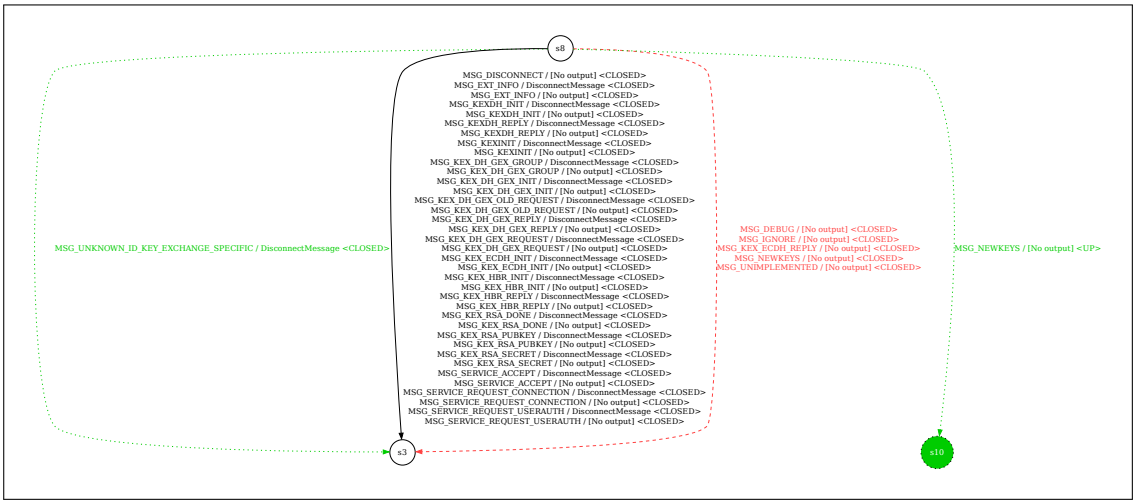


Figure 5.17. An excerpt from a diff between Bitvise (`ADDED`) and WinSCP (`REMOVED`), where Dropbear shows a second sink state $s10$ that is reached after `NEWKEYS` and does not close the connection. It is not directly apparent, that in this state any further message keeps the connection open, although the state machine shows at least no outgoing transitions. Looking at the isolated Bitvise state machine confirms it.

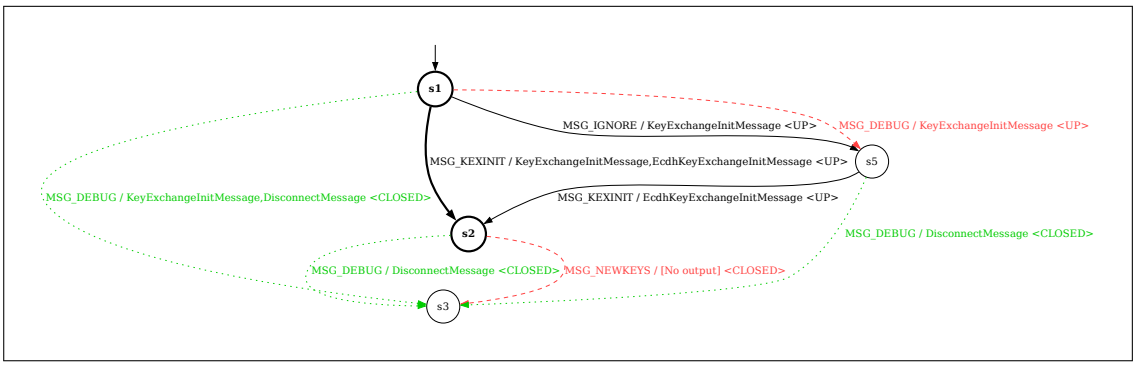


Figure 5.18. An excerpt from a diff between Bitvise (`ADDED`) and Paramiko (`REMOVED`), where Bitvise disconnects on receiving a `DEBUG` message during key exchange, but not on a `IGNORE` message. Both implementations show the buffer state $s5$, when the client is waiting for the server's `KEXINIT`, but only Paramiko reaches it with both `DEBUG` and `IGNORE`. Bitvise only reaches it with `IGNORE`, while all `DEBUG` messages lead to a disconnect.

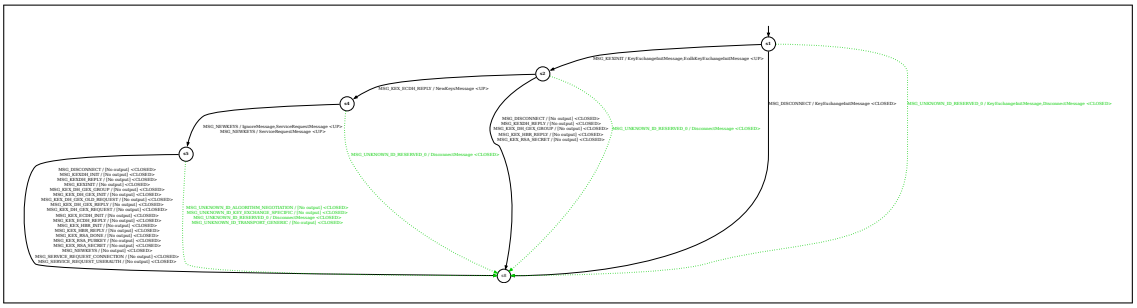


Figure 5.19. An excerpt from a diff between Bitvise (`ADDED`) and WinSCP, where OpenSSH illegally disconnects when receiving messages of unknown (reserved) type without sending `UNIMPLEMENTED`. Internally, OpenSSH seems to map them to the protocol phase, as reserved KEX messages are only rejected after the key exchange finishes with `NEW_KEYS`. The message with ID 0 is consistently rejected with a `DISCONNECT` response.

messages in the *algorithm negotiation* and *Key exchange method specific* range messages before the key exchange.

Dropbear, Termius, and OpenSSH handle the message number 0 distinctly and close the connection with an explicit error message. Since message numbers are technically defined to be in the range $1 - 255$ [36, Section 7] this can be considered behavior not covered by this part of the specification, and we only consider it a partial violation. Going by the specification, most implementations are in clear violation of the standard here, because they do not respond with `UNIMPLEMENTED` to unrecognized messages.

See Fig. 5.19.

`EXT_INFO`. All implementations except RuSSH only allow a server's `EXT_INFO` as intended. RuSSH will also receive a `EXT_INFO` prior to the key exchange, without (visibly) rejecting it. See Fig. 5.20. Inspection of the source code and logging reveals that the `EXT_INFO` is simply ignored, so this is not a security issue. Termius behaves the same way.

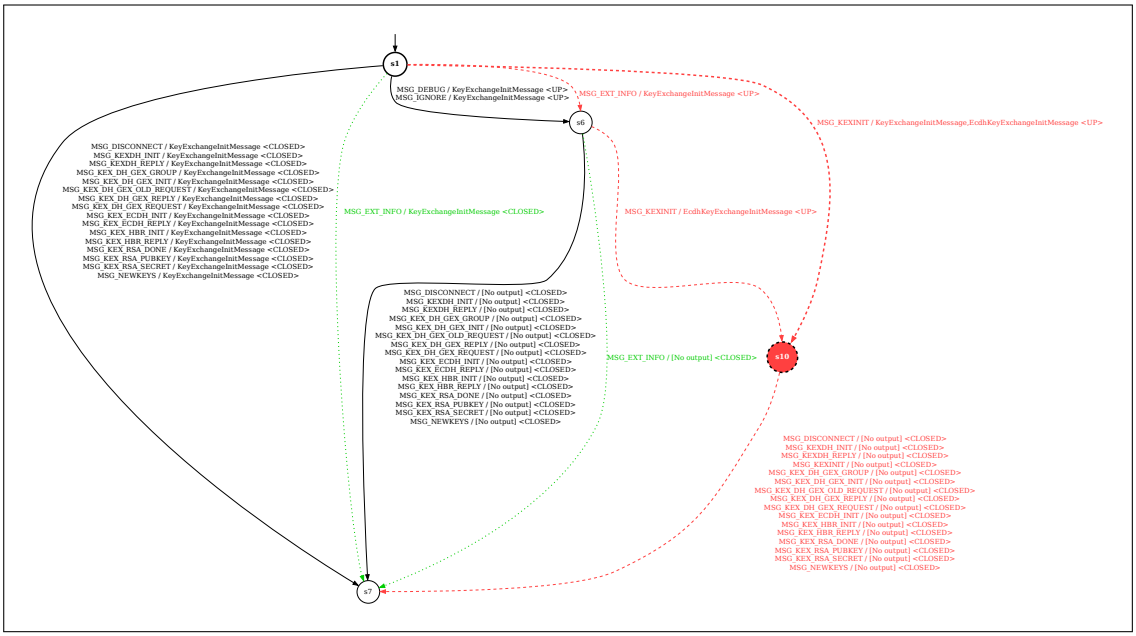


Figure 5.20. An excerpt from a diff between FileZilla (`ADDED`) and RuSSH (`REMOVED`), where RuSSH accepts a `EXT_INFO` message before the key exchange. FileZilla transitions to the sink state when receiving `EXT_INFO` before the key exchange, while RuSSH will receive the message before the key exchange via state $s6$ and still return to its first happy flow state $s10$. In this case the happy flow was not matched, but the differential is still useful for demonstration purposes.

5.3.1 Issues around NEWKEYS

Bitvise, Dropbear, and Win32-OpenSSH all exhibit unexpected behavior when receiving a second NEWKEYS message without it being part of a key re-exchange. In all three implementations, this leads to an unrecoverable state. Dropbear and OpenSSH simply disconnect for any subsequent message, while Bitvise will keep the connection open.

We found that Bitvise will remain in this waiting state indefinitely, as long as the server continues to send messages. Otherwise, the connection will be closed after an OS level timeout of one minute. Termius exhibits similar behavior, although a NEWKEYS message is actually accepted at anytime. Any subsequent messages will be ignored, but the connection remains open. Sending an encrypted message will disconnect with the message "Unable to exchange encryption keys", but also logs "Agreed client-to-server cipher: none", "MAC: none". We did not find a way to directly leverage this for an attack, but it is risky behavior that should be explored further.

5.3.2 Pre-mature Channel Messages

Our Extended alphabet did not identify any outright vulnerabilities, but it did reveal, that some implementations do not cleanly separate the different subprotocols. Most implementations handle the message like any other erroneous message (cf. Section 5.4), but Paramiko and Termius handle global and channel request earlier than intended. Paramiko will only process the message, after the transport layer protocol has been completed. Given that Paramiko does not require a SERVICE_ACCEPT (see above), this indicates some coupling between the authentication and connection protocol. Termius will handle the requests at any time. With our alphabet, neither request actually succeeded, but this potential attack surface should be re-visited in future work. Further manual analysis shows that Termius will buffer SSH-T and SSH-A messages for later (including, as previously mentioned, SERVICE_ACCEPT), even if they are sent pre-KEX. For example, injecting a USERAUTH_FAILURE pre-KEX, will cause Termius to try an additional authentication method during client authentication.

5.4 De facto Protocol Semantics for SSH Clients

Our analysis also highlights the de facto protocol semantics that have emerged among different SSH clients. These differences are not in violation of the SSH standard, but they do show off how different areas of the standard are under-specified. They also explain why plain LTSDiff and LTSDiff with domain knowledge struggle: Implementations are often functionally equivalent, but syntactically different.

5.4.1 Happy Flows

We observed several variations in how implementations handle the happy flow, that go beyond our initial assumption that there is a single happy flow up to interleaved optional messages (cf. Section 2.1.1). While the sequence of messages is the same, there are variations in which messages are bundled together and how the message sequence relates to internal state changes. We find that some implementations tend to send messages earlier than necessary, while others delay messages until the last possible moment. E.g., Bitvise will immediately include both `SERVICE_REQUEST` and `AUTH_REQUEST` with its `NEWKEYS` message, while OpenSSH will only send the `NEWKEYS` message and wait for the server's response before sending the next messages. As expected, some implementations also interleave `IGNORE` messages in the happy flow. We found that KEX guessing is only performed Dropbear. Lastly, we found that some implementations do not use `none` authentication (cf. Section 2.1.2) and will simply attempt the user-supplied authentication method. Fig. 5.21 shows the different happy flows we observed across implementations.

We find, that there is no single canonical happy flow, but rather a family of equivalent but syntactically different happy flows. This goes beyond our initial assumption that there is a single happy flow up to interleaved optional messages.

5.4.2 Implicit Rejection

There are two functional failures in which case explicit disconnection is required: Failure during algorithm negotiation and requesting an unsupported service. Other than that, there is only the requirement for reserved messages, already explored above.

For other errors `DISCONNECT` seems to be the intended mechanism, because it is the only other error signaling mechanism explicitly defined. It offers reason codes for various error conditions, e.g., `MAC_ERROR` or `KEY_EXCHANGE_FAILED`. But even for such cryptographic failures, the standard does not prescribe a specific error handling behavior. For errors explored by SML only the generic `PROTOCOL_ERROR` seems appropriate.

What we found in practice, is that implementations often use implicit rejection mechanisms instead. Concretely, across all implementations, we observed these distinct reactions to error:

- Accepting the message and proceeding as if it were valid
- Ignoring the message
- Deadlock

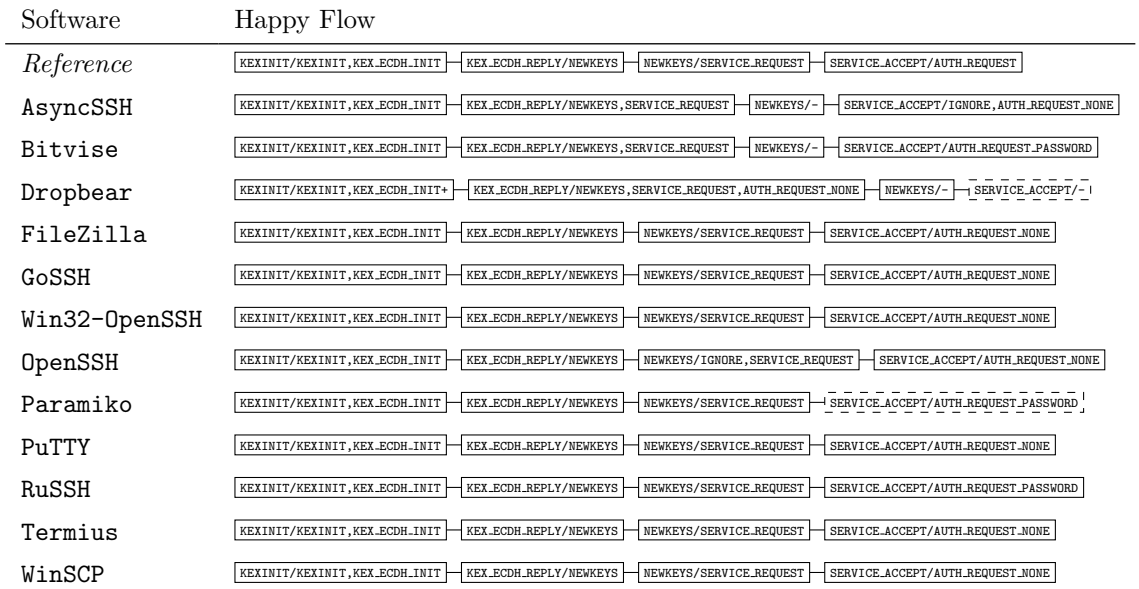


Figure 5.21. Different happy flows observed in implementations. "+" marks where multiple `ECDH_KEX_INIT` messages are sent, because one of them is a failed guess. Dashed boxes indicate transitions that are not accompanied by a state change. Recall that this uses the server's perspective already introduced in Section 2.1.1, because the output state machines are from the server's perspective.

- Responding with `UNIMPLEMENTED`
- Silent disconnection (closed socket)
- Silent disconnection (TCP RST)
- Explicit disconnection with `DISCONNECT`

In Fig. 5.22, we show how different implementations use these strategies in practice, when responding to unexpected messages. For that purpose, we sort into different categories

E1 Reserved messages/Unknown ID

E2 Known message, invalid in current phase (e.g., KEX before `KEXINIT`, `SERVICE_ACCEPT` before completed KEX or `NEW_KEYS` outside key exchange)

E3 Known message, malformed or unexpected repetition (e.g., double `KEXINIT`, double `ECDH_REPLY`, double `NEW_KEYS`, double `SERVICE_ACCEPT`)

E4 Known message, wrong direction (`KEX_ECDH_INIT` and `SERVICE_REQUEST` from server)

We extracted the behavior by hand from the learned state machines with the Medium alphabet and restricted ourselves to the states in and around the happy flow, i.e., no states that do not ever return to the happy flow. The results are shown in Fig. 5.22. Note that rejection behaviors are not mutually exclusive and each category can list multiple

errors. The chosen reaction depends at least on the specific implementation, cause of the error, and the phase within the protocol.

The large variety of error handling behavior presents an underexplored aspect of the protocol, and various corner cases may hide side-channel vulnerabilities and implementation-specific quirks. Disconnection behavior has been used as a side-channel for a plaintext recovery attack against OpenSSH in the past [3].

E1: Reserved messages

Impl	⚠	⊗	ϟ	?	⊘	🔒	✓
AsyncSSH	✓			✓			
Bitvise	✓				✓		
Dropbear		✓		✓			
FileZilla				✓			
GoSSH		✓					
Win32-OpenSSH	✓		✓	✓			
OpenSSH	✓	✓		✓			
Paramiko		✓					
PuTTY				✓			
RuSSH		✓			✓		
WinSCP				✓			
Termius	✓				✓		

E2: Invalid in current phase

Impl	⚠	⊗	ϟ	?	⊘	🔒	✓
AsyncSSH	✓						
Bitvise	✓					✓	
Dropbear		✓		✓			
FileZilla		✓					
GoSSH		✓					
Win32-OpenSSH			✓	✓			
OpenSSH		✓		✓			
Paramiko		✓		✓			
PuTTY		✓					
RuSSH		✓			✓		
WinSCP		✓					
Termius					✓		✓

E3: Unexpected repetition

Impl	⚠	⊗	ϟ	?	⊘	🔒	✓
AsyncSSH	✓						
Bitvise	✓						
Dropbear		✓		✓			✓
FileZilla		✓					
GoSSH		✓					
Win32-OpenSSH			✓	✓			
OpenSSH		✓		✓			
Paramiko		✓					✓
PuTTY		✓					
RuSSH		✓					
WinSCP		✓					
Termius					✓		✓

E4: Wrong direction

Impl	⚠	⊗	ϟ	?	⊘	🔒	✓
AsyncSSH	✓						
Bitvise	✓						
Dropbear		✓		✓			
FileZilla		✓					
GoSSH		✓					
Win32-OpenSSH			✓	✓			
OpenSSH		✓		✓			
Paramiko		✓		✓			
PuTTY		✓					
RuSSH		✓			✓		
WinSCP		✓					
Termius					✓		

⚠ Explicit `SSH_MSG_DISCONNECT`
⊗ Silent disconnection (closed socket)
ϟ Silent disconnection (TCP RST)
✓ Accept message and proceed
? Respond with `UNIMPLEMENTED`
⊘ Ignore message
🔒 Deadlock

Figure 5.22. Error handling strategies for different implementations and error categories. For each error category, we report whether there exist an input that is handled with the specific strategy. Internally, implementations differ in how and when they send which error, leading to multiple occurrences per row.

5.5 Limitations

Ill-defined heuristic. Our heuristic informally describes "matching the happy flow" between implementations, which it checks by looking for happy flow states that have not found a match in the differential. As a result, structurally incorrect mappings may be counted as successful. This effectively measures coverage of the happy flow in the differential, rather than the actual quality of the mapping. We mistakenly assumed that a happy flow state's best match would be the corresponding happy flow state. In practice, our base assumption that all happy flow states even have a counterpart is not necessarily true, because happy flows do not necessarily have the same length or structure.

Furthermore, the transition-based similarity requirement (i.e., matching around half of all transitions) is vulnerable to differences in state machine size. Some state machines have significantly more states than others, which makes it hard to satisfy this criterion even the implementations are similar. In such cases, the heuristic underestimates the usefulness of differentials.

Quality of State Machines. We found that the SSH-State-Learner default configuration was not sufficient to learn accurate SMs for all implementations. We verified behavior indicated in the differentials (shown in Fig. 5.14) manually, and encountered discrepancies between the learned models and the actual behavior of the implementations, which we would have identified as violations otherwise.

Our original assumption assumed that the happy flow would make up a significant portion of the state machine, and therefore we expected the happy flow mutation oracle to be effective. We did not account for the fact, that our larger alphabet would introduce as many new states outside the happy flow as it did. For outliers like OpenSSH, the happy flow mutation oracle was not representative.

This is in line with findings from Fiterău-Broştean et al., who uncovered similar issues as their learning extended deeper into the protocol. They mitigated this by using a more powerful equivalence oracle, called the *Augmented DS-Method* [53]. The Augmented DS-Method is a general purpose oracle, which builds on the well-known W- and partial W-method for software testing[18, 25]. In fact, we implemented a prototype of it ourselves, but were not able to test it thoroughly, so we did not use it for our evaluation.

Learning through a Mapper. Every SUL is learned through our mapper, so it will always be an abstraction of the actual implementation. Certain undefined behavior, like the handling of multiple `NEWKEYS` messages we observed, depends on the mapper's implementation. We also defined SUL-specific mitigation, that actively reduce the detail of the final SMs. While we deemed these necessary, this could also be mitigated by changing the learning algorithm.

Loss of Detail and Manual Analysis. In an effort to improve our differentials, we sacrifice a certain level of detail. The final Input-only strategy discards self-transitions from the learned state machines, which while detrimental to the matching process can still contain

valuable behavior. We also intentionally ignore the output of transitions that lead to the same state, and do not consider them true differences. While that holds for differences like additional `IGNORE` messages, there may be cases where the output is significant. Combined with the fact that we ultimately evaluate these differentials by hand, this risks overlooking important behavioral nuances.

6 Discussion

This chapter interprets the findings of our evaluation in Chapter 5 and discusses its implications with respect to our research questions.

6.1 On the Effectiveness of LTSDiff

With respect to **RQ1**, our results show that naively applying LTSDiff to SSH state machines is insufficient for useful cross-implementation differentials.

The baseline configuration yielded relatively few usable differentials and frequently failed to match even the happy flows. Although implementations follow the same specification, they exhibit significant syntactical differences in practice. As our evaluation showed, many implementations show syntactical variations even on the happy flow, which should be the most similar path throughout all implementations. The Input-only strategy addresses the issue by abstracting from output labels, which significantly improves the quality of the differentials and shifts the comparison towards structural similarity.

In this sense, LTSDiff can be effective for comparing SSH state machines under appropriate abstraction. However, the results also indicate that state machines for different software implementations are not as similar as expected. Even in our restricted scope, state machines can grow large, which can inhibit effective comparison. And even in small state machines, the happy flow is not as canonical as we expected, so caution is warranted when using it as a reference point. Therefore, our heuristic cannot be considered a true indicator of quality, but rather a rough indicator of relative performance.

6.2 RFC Compliance and Under-specification

We identified multiple deviations from the RFC requirements, particularly in the handling of reserved messages. While some implementation differences can be considered stricter or more lenient interpretations of the standard, others are clear violations that contradict explicit MUST requirements. We also identified the `NEWKEYS` message as one source of under-specification that connects to some of the observed violations. Given that `NEWKEYS` is one of the most important messages in the protocol, it is surprising that the standard does not explicitly define how to handle unexpected `NEWKEYS` messages. Even though we restricted ourselves to SSH-T, our results indicate the such issue extend into SSH-A and SSH-C as well. We found that there is no strict behavioral separation, e.g., SSH-C messages being parsed during key exchange. Generally, the specification is not very strict or clear about sending and receiving messages beyond the happy flow. Overall, even mature SSH clients differ in corner cases, which may contribute to future

vulnerabilities or interoperability issues. Our findings suggest that the "real" SSH protocol consists not only of the RFC—which sometimes under-specifies corner cases—but also the different implementations that try to follow it. In this context, comparing real-world implementations against each other can yield valuable insights about the de facto semantics of the protocol.

6.3 Implications for Automated Protocol Analysis

A fully automated SSH non-compliance discovery via LTSDiff is currently still out of reach. Many of the deviations LTSDiff identifies do not constitute clear violations and require an expert for interpretation. Unlike more recent protocols that are specified with explicit state machines, SSH does not have a canonical reference model of its own. Nevertheless, differential analysis remains valuable. For cross-implementation comparison, it can certainly help reduce the effort for an expert. Our implementation was rather simple, but further refinement of the tooling itself could enhance its effectiveness, e.g., by automatically extracting minimal deviations along the happy flow. Differential analysis also seems particularly effective for regression testing within singular implementation, where behavior is largely consistent across versions.

7 Conclusions and Future Work

This thesis has explored the automated analysis of SSH clients through the comparison of 12 state machines learned from real-world protocol implementations via the LTSDiff algorithm. We show that native LTSDiff is ineffective for SSH state machines, but it can be successfully adapted for this purpose. Specifically, abstracting from differences in output messages and focusing on structural similarity allowed us to generate compact and interpretable differentials. These make LTSDiff a practical tool for identifying behavioral differences.

Using the differentials from LTSDiff, we observed highly heterogeneous behavior, both within and outside the confines of the SSH specification. Some of these discrepancies reflect different ways of interpreting the specification, while others contradict explicit MUST requirements. These show large areas of under-specification in the standard itself. Instead, SSH behavior is shaped by the implementation ecosystem itself and differential analysis can serve as a tool to understand this ecosystem better. While full automation remains unlikely, differential analysis can be a valuable tool to human experts.

7.1 Future Work

We showed that there is potential in applying LTSDiff to SSH, which opens interesting new venues to improve its impact.

Improving Matching and Evaluation Metrics. We showed that our evaluation is limited by the fact, that our heuristic for determining the usefulness of differentials is not perfect. Future work should focus on refining this heuristic, especially by giving a clearer criterion on matching the happy flow. Subsequently, Input-only could be further augmented towards the happy flow like with the Match strategy: We merely tried excluding undesirable matches, but one could also improve the similarity of happy flow states, e.g., by giving additional weight to the edges of the happy flow when comparing two such states.

Learning larger state machines. An obvious next step is to extend the evaluation to also include SSH-A and SSH-C. Fiterău-Broştean et al. already showed that with a more sophisticated oracle, they could learn state machines that cover all SSH sub-protocols. By limiting ourselves to the transport layer, the state machines do not cover the user authentication and connection phase, which cannot be ignored for a comprehensive analysis. We also considered this, but found it to be unsuitable for our time constraints. We believe a longer evaluation period would make larger state machines feasible.

Grey-box learning. Speed and quality of the learned state machines are partially limited by the black-box nature of the SUL. Grey-box techniques have been proposed that use additional information about the SUL. With StateInspector, McMahon Stone et al. combined SML with monitoring of the memory at runtime, to identify states not visible purely in I/O behavior. They showed significant speed-ups compared to black-box state learning, which may be applicable to our scenario as well.

Towards true automated SSH analysis. As we showed, a true reference model cannot be defined from the specification, at least not in a way where every state cleanly maps to a state as inferred from an implementation. However, LTSDiff does not require a deterministic state machine. It should be possible to construct a non-deterministic SM that captures different empirically observed behaviors simultaneously. While this would be narrower than the specification as written, it could still be used to identify behavior that falls far outside expectations. By curating such a model once, it could be used to automatically identify protocol violations in the future while relying even less on human interpretation.

Bibliography

[1] F. Aarts, J. De Ruiter, and E. Poll. "Formal Models of Bank Cards for Free". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. 2013 IEEE 6th International Conference On Software Testing, Verification and Validation Workshops (ICSTW). Luxembourg, Luxembourg: IEEE, Mar. 2013, pp. 461–468. ISBN: 978-0-7695-4993-4 978-1-4799-1324-4. DOI: 10.1109/ICSTW.2013.60.

[2] M. R. Albrecht, J. P. Degabriele, T. B. Hansen, and K. G. Paterson. "A Surfeit of SSH Cipher Suites". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. New York, NY, USA: Association for Computing Machinery, Oct. 24, 2016, pp. 1480–1491. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978364.

[3] M. R. Albrecht, K. G. Paterson, and G. J. Watson. "Plaintext Recovery Attacks against SSH". In: *2009 30th IEEE Symposium on Security and Privacy*. 2009 30th IEEE Symposium on Security and Privacy. May 2009, pp. 16–26. DOI: 10.1109/SP.2009.5.

[4] K. K. Ang, G. Farrelly, C. Pope, and D. C. Ranasinghe. "An Automated Blackbox Noncompliance Checker for QUIC Server Implementations". In: *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '25. New York, NY, USA: Association for Computing Machinery, Aug. 24, 2025, pp. 1459–1475. ISBN: 979-8-4007-1410-8. DOI: 10.1145/3708821.3736222.

[5] D. Angluin. "Learning Regular Sets from Queries and Counterexamples". In: *Information and Computation* 75.2 (Nov. 1, 1987), pp. 87–106. ISSN: 0890-5401. DOI: 10.1016/0890-5401(87)90052-6.

[6] *Appium*. URL: https://appium.io/docs/en/latest/.

[7] F. Bäumer, M. Brinkmann, N. Erinola, S. N. Hebrok, N. Heitmann, F. Lange, M. Maehren, R. Merget, N. Niere, et al. "TLS-Attacker: A Dynamic Framework for Analyzing TLS Implementations". In: *Proceedings of Cybersecurity Artifacts Competition and Impact Award (ACSAC '24)* (2024). URL: https://ris.uni-paderborn.de/record/57816.

[8] F. Bäumer, M. Brinkmann, M. Radoy, J. Schwenk, and J. Somorovsky. "On the Security of SSH Client Signatures". In: *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. CCS '25. New York, NY, USA: Association for Computing Machinery, Nov. 22, 2025, pp. 4619–4633. ISBN: 979-8-4007-1525-9. DOI: 10.1145/3719027.3765079.

[9] F. Bäumer, M. Brinkmann, and J. Schwenk. "Terrapin Attack: Breaking SSH Channel Integrity By Sequence Number Manipulation". In: 33rd USENIX Security Symposium (USENIX Security 24). 2024, pp. 7463–7480. ISBN: 978-1-939133-44-1. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/baumer.

[10] F. Bäumer, M. Maehren, M. Brinkmann, and J. Schwenk. *Finding SSH Strict Key Exchange Violations by State Learning*. 2025. Pre-published.

[11] F. C. Bäumer. "Learning State Machines of SSH Server Implementations". MA thesis. Bochum: Ruhr-Universität Bochum, Dec. 6, 2021.

[12] 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". In: *ACM Trans. Inf. Syst. Secur.* 7.2 (May 1, 2004), pp. 206–241. ISSN: 1094-9224. DOI: 10.1145/996943.996945.

[13] B. Benčina, B. Dowling, V. Maram, and K. Xagawa. "Post-Quantum Cryptographic Analysis of SSH". In: *2025 IEEE Symposium on Security and Privacy (SP)*. 2025 IEEE Symposium on Security and Privacy (SP). May 2025, pp. 595–613. DOI: 10.1109/SP61157.2025.00126.

[14] F. Bergsma, B. Dowling, F. Kohlar, J. Schwenk, and D. Stebila. "Multi-Ciphersuite Security of the Secure Shell (SSH) Protocol". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. New York, NY, USA: Association for Computing Machinery, Nov. 3, 2014, pp. 369–381. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660286.

[15] D. Bider. *Extension Negotiation in the Secure Shell (SSH) Protocol*. Request for Comments RFC 8308. Internet Engineering Task Force, Mar. 2018. 14 pp. DOI: 10.17487/RFC8308.

[16] S. O. Bradner. *Key Words for Use in RFCs to Indicate Requirement Levels*. Request for Comments RFC 2119. Internet Engineering Task Force, Mar. 1997. 3 pp. DOI: 10.17487/RFC2119.

[17] *BSI TR-02102-4 "Kryptographische Verfahren: Teil 4 – Verwendung von Secure Shell (SSH)" Version: 2026-01*. Bundesamt für Sicherheit in der Informationstechnik. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102-4.html?nn=132646.

[18] T. Chow. "Testing Software Design Modeled by Finite-State Machines". In: *IEEE Transactions on Software Engineering* SE-4.3 (May 1978), pp. 178–187. ISSN: 1939-3520. DOI: 10.1109/TSE.1978.231496.

[19] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. "NUSMV: A New Symbolic Model Checker". In: *International Journal on Software Tools for Technology Transfer* 2.4 (Mar. 1, 2000), pp. 410–425. ISSN: 1433-2779. DOI: 10.1007/s100090050046.

[20] L.-A. Daniel, E. Poll, and J. De Ruiter. "Inferring OpenVPN State Machines Using Protocol State Fuzzing". In: *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). London, United Kingdom: IEEE, Apr. 2018, pp. 11–19. ISBN: 978-1-5386-5445-3. DOI: 10.1109/EuroSPW.2018.00009.

[21] J. De Ruiter and E. Poll. "Protocol State Fuzzing of TLS Implementations". In: 24th USENIX Security Symposium (USENIX Security 15). 2015, pp. 193–206. ISBN: 978-1-939133-11-3. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter.

[22] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. De Ruiter, K. Sagonas, and J. Somorovsky. "Analysis of DTLS Implementations Using Protocol State Fuzzing". In: 29th USENIX Security Symposium (USENIX Security 20). 2020, pp. 2523–2540. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean.

[23] P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. De Ruiter, F. Vaandrager, and P. Verleg. "Model Learning and Model Checking of SSH Implementations". In: *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. ISSTA '17: International Symposium on Software Testing and Analysis. Santa Barbara CA USA: ACM, July 13, 2017, pp. 142–151. ISBN: 978-1-4503-5077-8. DOI: 10.1145/3092282.3092289.

[24] M. Friedl, N. Provos, and W. A. Simpson. *Diffie-Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol*. Request for Comments RFC 4419. Internet Engineering Task Force, Mar. 2006. 10 pp. DOI: 10.17487/RFC4419.

[25] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. "Test Selection Based on Finite State Models". In: *IEEE Transactions on Software Engineering* 17.6 (June 1991), pp. 591–603. ISSN: 1939-3520. DOI: 10.1109/32.87284.

[26] B. Harris. *RSA Key Exchange for the Secure Shell (SSH) Transport Layer Protocol*. Request for Comments RFC 4432. Internet Engineering Task Force, Mar. 2006. 8 pp. DOI: 10.17487/RFC4432.

[27] D. Hendriks and W. Oortwijn. "gLTSdiff: A Generalized Framework for Structural Comparison of Software Behavior". In: *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS). Västerås, Sweden: IEEE, Oct. 1, 2023, pp. 285–295. DOI: 10.1109/models58315.2023.00025.

[28] D. Hendriks and W. Oortwijn. *TNO/gLTSdiff*. TNO, May 14, 2024. URL: https://github.com/TNO/gLTSdiff.

[29] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. “Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices”. In: 21st USENIX Security Symposium (USENIX Security 12). 2012, pp. 205–220. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/heninger.

[30] M. Isberner, F. Howar, and B. Steffen. “The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning”. In: *Runtime Verification*. Ed. by B. Bonakdarpour and S. A. Smolka. Cham: Springer International Publishing, 2014, pp. 307–322. ISBN: 978-3-319-11164-3. DOI: 10.1007/978-3-319-11164-3_26.

[31] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Request for Comments RFC 9000. Internet Engineering Task Force, May 2021. 151 pp. DOI: 10.17487/RFC9000.

[32] M. J. Kearns and U. Vazirani. *An Introduction to Computational Learning Theory*. The MIT Press, Aug. 15, 1994. ISBN: 978-0-262-27686-3. DOI: 10.7551/mitpress/3897.001.0001.

[33] C. M. Lonvick and S. Lehtinen. *The Secure Shell (SSH) Protocol Assigned Numbers*. Request for Comments RFC 4250. Internet Engineering Task Force, Jan. 2006. 20 pp. DOI: 10.17487/RFC4250.

[34] C. M. Lonvick and T. Ylonen. *The Secure Shell (SSH) Authentication Protocol*. Request for Comments RFC 4252. Internet Engineering Task Force, Jan. 2006. 17 pp. DOI: 10.17487/RFC4252.

[35] C. M. Lonvick and T. Ylonen. *The Secure Shell (SSH) Connection Protocol*. Request for Comments RFC 4254. Internet Engineering Task Force, Jan. 2006. 24 pp. DOI: 10.17487/RFC4254.

[36] C. M. Lonvick and T. Ylonen. *The Secure Shell (SSH) Protocol Architecture*. Request for Comments RFC 4251. Internet Engineering Task Force, Jan. 2006. 30 pp. DOI: 10.17487/RFC4251.

[37] C. M. Lonvick and T. Ylonen. *The Secure Shell (SSH) Transport Layer Protocol*. Request for Comments RFC 4253. Internet Engineering Task Force, Jan. 2006. 32 pp. DOI: 10.17487/RFC4253.

[38] M. Maehren, N. Erinola, R. Merget, J. Schwenk, and J. Somorovsky. “Towards Internet-Based State Learning of TLS State Machines”. In: 34th USENIX Security Symposium (USENIX Security 25). 2025, pp. 7097–7116. ISBN: 978-1-939133-52-6. URL: https://www.usenix.org/conference/usenixsecurity25/presentation/maehren.

[39] T. Margaria, O. Niese, H. Raffelt, and B. Steffen. “Efficient Test-Based Model Generation for Legacy Reactive Systems”. In: *Proceedings. Ninth IEEE International High-Level Design Validation and Test Workshop*. . Ninth IEEE International High-Level Design Validation and Test Workshop. Nov. 2004, pp. 95–100. DOI: 10.1109/HLDVT.2004.1431246.

[40] A. Martin. “Analyse von Android-SSH-Clients Mit State-Machine-Learning”. MA thesis. Sept. 4, 2024.

[41] C. McMahon Stone, S. L. Thomas, M. Vanhoef, J. Henderson, N. Bailluet, and T. Chothia. “The Closer You Look, The More You Learn: A Grey-box Approach to Protocol State Machine Learning”. In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’22. New York, NY, USA: Association for Computing Machinery, Nov. 7, 2022, pp. 2265–2278. ISBN: 978-1-4503-9450-5. DOI: 10.1145/3548606.3559365.

[42] Microsoft. *[MS-RDPBCGR]: Remote Desktop Protocol: Basic Connectivity and Graphics Remoting*. July 4, 2025. URL: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-rdpbcgr/5073f4ed-1e93-45e1-b039-6e30c385867c.

[43] Microsoft. *Lift and Shift to Containers*. URL: https://learn.microsoft.com/en-us/virtualization/windowscontainers/quick-start/lift-shift-to-containers.

[44] D. Miller. *SSH Strict KEX Extension*. Internet Draft draft-miller-sshm-strict-kex-01. Internet Engineering Task Force, Mar. 18, 2025. 13 pp. URL: https://datatracker.ietf.org/doc/draft-miller-sshm-strict-kex-01.

[45] D. Miller, M. Friedl, M. Frysinger, T. C. Miller, and D. Tucker. *OpenSSH’s Deviations to the Published SSH Protocol*. 2024. URL: https://cvsweb.openbsd.org/cgi-bin/cvsweb/src/usr.bin/ssh/PROTOCOL?rev=1.55.

[46] K. G. Paterson and G. J. Watson. “Plaintext-Dependent Decryption: A Formal Security Treatment of SSH-CTR”. In: *Advances in Cryptology – EUROCRYPT 2010*. Ed. by H. Gilbert. Berlin, Heidelberg: Springer, 2010, pp. 345–361. ISBN: 978-3-642-13190-5. DOI: 10.1007/978-3-642-13190-5_18.

[47] A. Pnueli. “The Temporal Logic of Programs”. In: *18th Annual Symposium on Foundations of Computer Science (Sfcs 1977)*. 18th Annual Symposium on Foundations of Computer Science (Sfcs 1977). Oct. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[48] H. Raffelt, B. Steffen, T. Berg, and T. Margaria. “LearnLib: A Framework for Extrapolating Behavioral Models”. In: *International Journal on Software Tools for Technology Transfer* 11.5 (Nov. 1, 2009), pp. 393–407. ISSN: 1433-2787. DOI: 10.1007/s10009-009-0111-8.

[49] A. Rasool, G. Alpár, and J. de Ruiter. *State Machine Inference of QUIC*. Version 1. 2019. DOI: 10.48550/ARXIV.1903.04384. Pre-published.

[50] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. Request for Comments RFC 8446. Internet Engineering Task Force, Aug. 2018. 160 pp. DOI: 10.17487/RFC8446.

[51] C. J. van Rijsbergen. *Information Retrieval*. 2. ed., repr. London: Butterworth, 1981. 208 pp. ISBN: 978-0-408-70929-3.

[52] M. Shahbaz and R. Groz. “Inferring Mealy Machines”. In: *FM 2009: Formal Methods*. Ed. by A. Cavalcanti and D. R. Dams. Berlin, Heidelberg: Springer, 2009, pp. 207–222. ISBN: 978-3-642-05089-3. DOI: 10.1007/978-3-642-05089-3_14.

[53] W. Smeenk, J. Moerman, F. Vaandrager, and D. N. Jansen. “Applying Automata Learning to Embedded Control Software”. In: *Formal Methods and Software Engineering*. Ed. by M. Butler, S. Conchon, and F. Zaïdi. Cham: Springer International Publishing, 2015, pp. 67–83. ISBN: 978-3-319-25423-4. DOI: 10.1007/978-3-319-25423-4_5.

[54] D. X. Song, D. Wagner, and X. Tian. “Timing Analysis of Keystrokes and Timing Attacks on SSH”. In: 10th USENIX Security Symposium (USENIX Security 01). 2001. URL: https://www.usenix.org/conference/10th-usenix-security-symposium/timing-analysis-keystrokes-and-timing-attacks-ssh.

[55] D. Stebila and J. Green. *Elliptic Curve Algorithm Integration in the Secure Shell Transport Layer*. Request for Comments RFC 5656. Internet Engineering Task Force, Dec. 2009. 20 pp. DOI: 10.17487/RFC5656.

[56] N. Walkinshaw and K. Bogdanov. “Automated Comparison of State-Based Software Models in Terms of Their Language and Structure”. In: *ACM Transactions on Software Engineering and Methodology* 22.2 (Mar. 2013), pp. 1–37. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/2430545.2430549.

[57] S. C. Williams. “Analysis of the SSH Key Exchange Protocol”. In: *Cryptography and Coding*. Ed. by L. Chen. Berlin, Heidelberg: Springer, 2011, pp. 356–374. ISBN: 978-3-642-25516-8. DOI: 10.1007/978-3-642-25516-8_22.

[58] T. Yadav and K. Sadhukhan. “Identification of Bugs and Vulnerabilities in TLS Implementation for Windows Operating System Using State Machine Learning”. In: *Security in Computing and Communications*. Ed. by S. M. Thampi, S. Madria, G. Wang, D. B. Rawat, and J. M. Alcaraz Calero. Singapore: Springer, 2019, pp. 348–362. ISBN: 978-981-13-5826-5. DOI: 10.1007/978-981-13-5826-5_27.

A Appendix

A.1 Minimal Clients

```
1  #!/usr/bin/expect -f
2
3  set ip [lindex $argv 0]
4  set port [lindex $argv 1]
5  set user [lindex $argv 2]
6  set password [lindex $argv 3]
7  set command [lindex $argv 4]
8
9  spawn /usr/local/bin/dbclient "$ip" "-p" "$port" "-l" "$user"
10
11 set timeout 5
12 expect {
13     "y/n" {
14         send "y\r"
15     }
16 }
17 expect {
18         "$user@$ip's password:" {
19          send "$password\r"
20     }
21 }
22 expect {
23     "$ " {
24        send "$command\r"
25         }
26 }
27 expect {
28     "$ " {
29        send "exit\r"
30         }
31 }
```

Listing A.1. Expect script to automate Dropbear client. The script takes multiple arguments (lines 3-7), spawns the Dropbear SSH client process and then simulates a user interaction. The different steps match strings in the standard output of the program: Confirming the hostkey (lines 12-16), entering the password (lines 17-21) and running a command (lines 22-26).

```
def client_start(host, port, username, password, command, output, error):
if output:
    print("init Client")
client = paramiko.SSHClient()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

if output:
    print("start connection")
print(
    "Trying to connect to: [host="
    + host
    + ", port="
    + str(port)
    + ", username="
    + username
    + ", password="
    + password
    + "]"
)
client.connect(host, port=port, username=username, password=password)

if output:
    print("start command")
stdin, stdout, stderr = client.exec_command(command)

if output:
    print("finish")
out = stdout.readlines()
err = stderr.readlines()
if output:
    print("output: ", "\n".join(out))
if error:
    print("error: ", "\n".join(err))

stdin.close()
stdout.close()
stderr.close()
```

Listing A.2. Example of a custom Paramiko-based SSH client. The client takes connection parameters as command-line arguments, connects to the server, executes a command and prints the output. This example is simplified for clarity and omits error handling.

A.2 Example of Learned State Machine

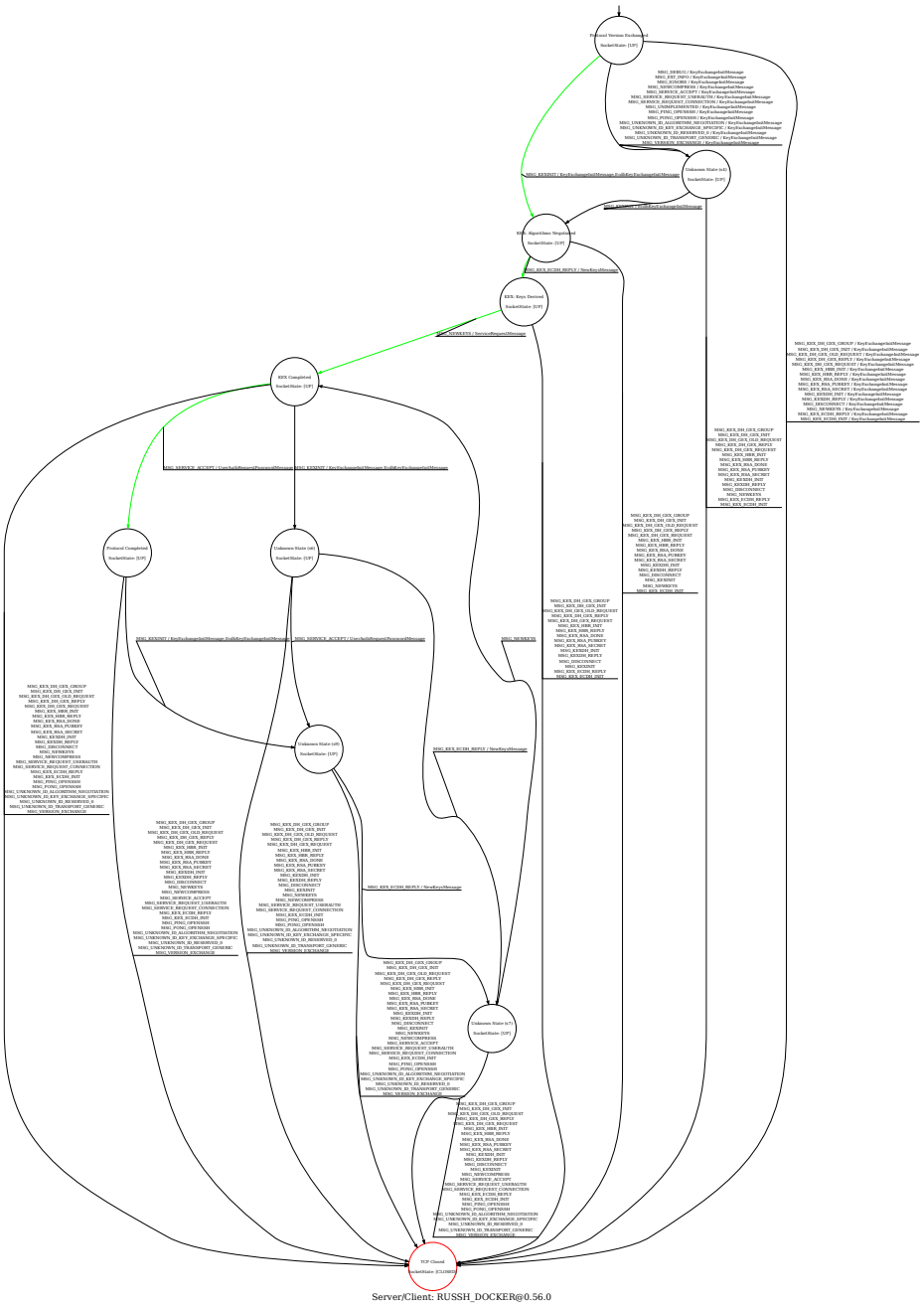


Figure A.1. Example state machine learned for RuSSH in version 0.56.0. Additional details about the state machine have been omitted for readability.

A.3 Additional Examples of State Machine Differentials

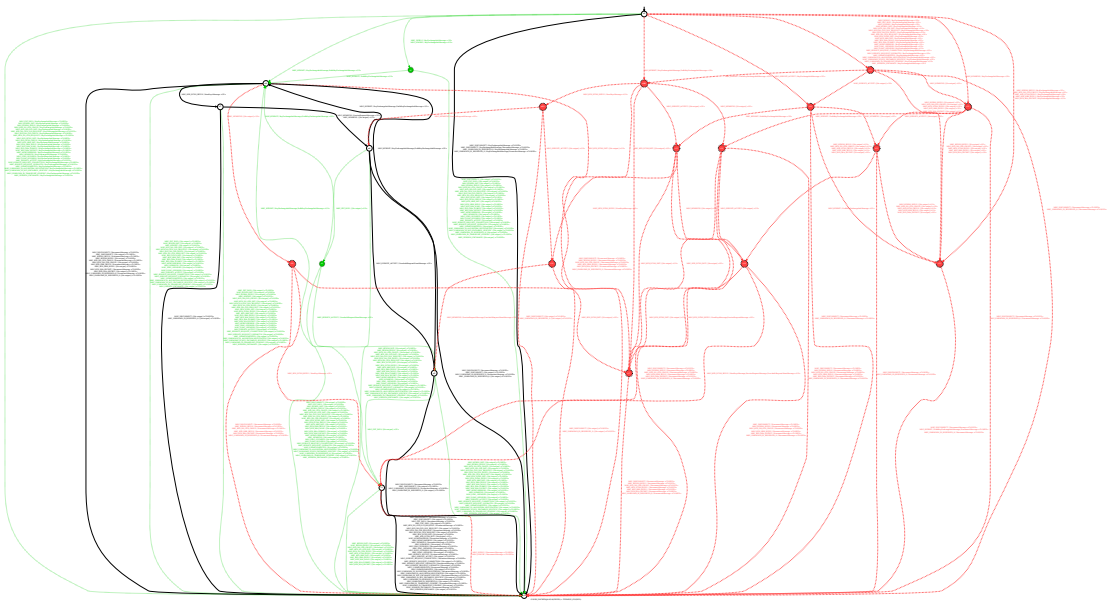


Figure A.2. A truly unusable differential between GoSSH and Termius. There are happy flow states $s10$ and $s11$ in Termius that are not matched.

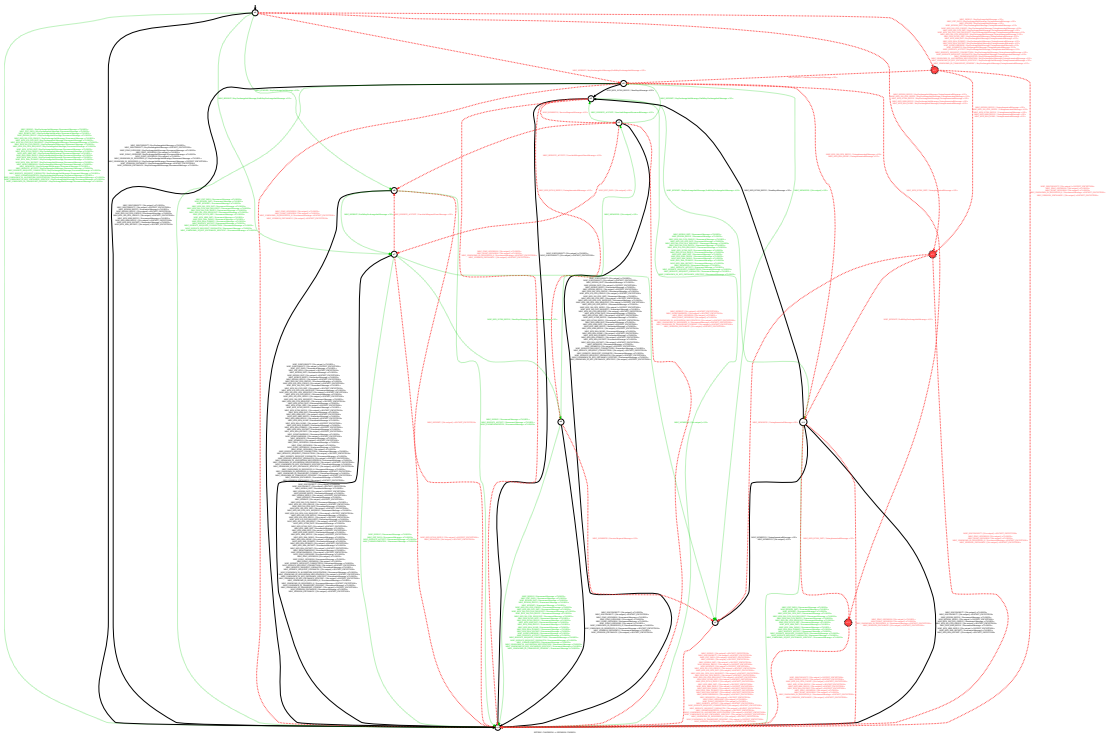


Figure A.3. A usable differential between Bitvise (`ADDED`) and OpenSSH (`REMOVED`), according to our heuristic, but it is actually a false positive. The happy flow states were matched with states that were not on the happy flow.

A.4 Extended Implementation Similarity

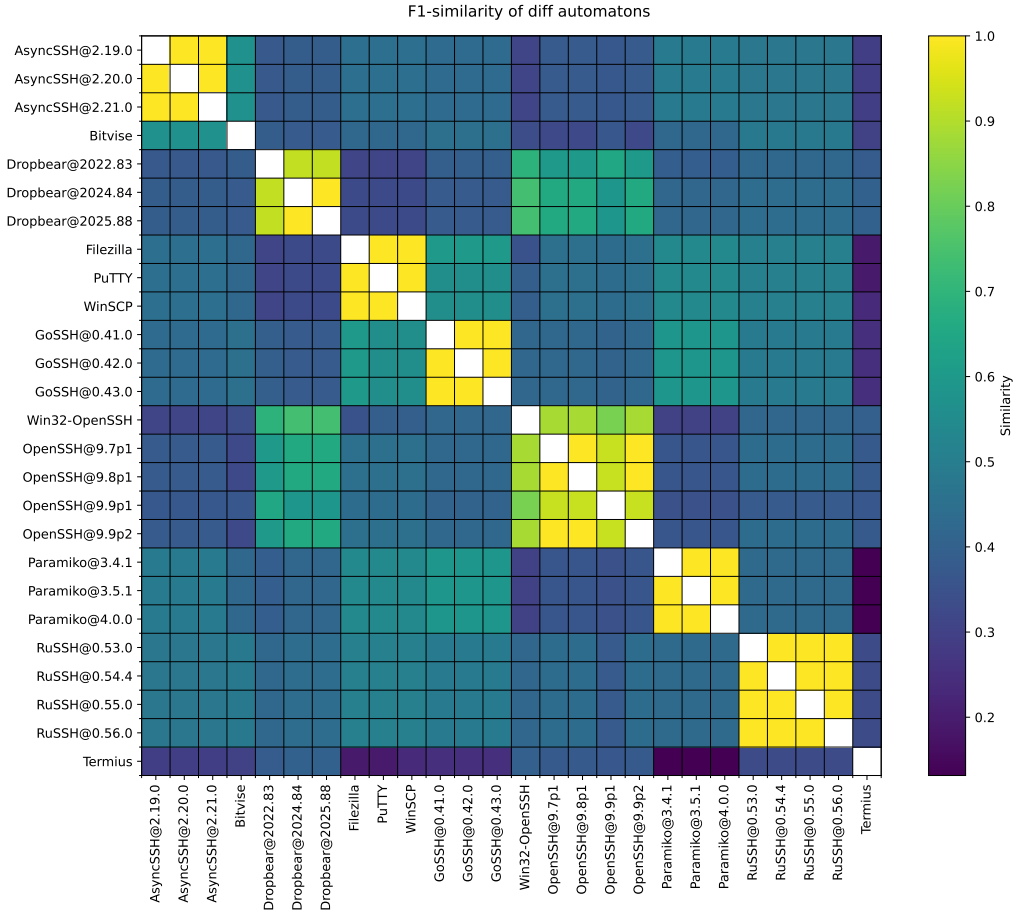


Figure A.4. Extended similarity matrix, which also includes different versions of the same software. Overall, all unrelated implementations have low similarity scores < 0.75 given this similarity metric. Related implementations receive higher scores > 0.8 and small differences are properly captured in the score. We find no significant unexpected similarities, where we did not assume them.