Research Project (PRe)

Specialization : Information Technology

Academic Year : 2024/2025

A Fingerprinting Platform for SSH

Non confidential Report

Author :
Mohamed Mziou

Promotion :
2026

Tutor at ENSTA Paris :
Françoise Levy-dit-Vehel

Tutor at Télécom SudParis :
Olivier Levillain

Internship from 10/06/2025 **to** 29/08/2025

Hosting organization : Département RST (Réseaux et Services de Télécommunications) Télécom SudParis
Address : 19 Place Marguerite Perey, 91120 Palaiseau

Abstract

SSH is a pervasive protocol used on our modern systems. It is thus interesting to study its deployment in the wild. One of the ways to identify and better understand the ecosystem is to work on fingerprinting tools. Fingerprinting tools can be categorized depending on the elements they rely on to identify a given implementation.

On one hand, it is clear that published information can be tampered with to mimic another stack, and that configurations are subject to changes from one system to another, which makes these elements not necessarily very robust. On the other hand, state-machine-based fingerprinting and other behavior-based approaches are still relatively new, and establishing the fingerprints can be costly.

This internship aims at designing and contributing to a fingerprinting platform for SSH to compare existing tools and assess their separation power, as well as their robustness against configuration changes.

Résumé

Le protocole SSH est omniprésent dans les systèmes modernes, ce qui rend son étude cruciale. Une approche pour mieux comprendre son écosystème consiste à utiliser des outils de *fingerprinting*, qui peuvent être classés selon les éléments qu'ils exploitent pour identifier une implémentation donnée.

Toutefois, les informations publiques peuvent être falsifiées, et les configurations varient d'un système à l'autre, rendant ces éléments peu fiables. D'autre part, les méthodes basées sur les machines à états et autres approches comportementales, encore récentes, peuvent être coûteuses.

Ce stage a pour objectif de concevoir et contribuer à une plateforme de fingerprinting TLS, pour comparer les outils existants et évaluer leur capacité de distinction ainsi que leur robustesse face aux modifications de configuration.

Acknowledgments

I would like to express my sincere gratitude to Mr. Olivier Levillain for his invaluable guidance throughout the entire internship, without which much of the progress made would not have been possible. I also warmly thank his entire team for their welcome and support during my time there. I am also grateful to Mrs. Françoise Levy-dit-Vehel for kindly accepting to be my tutor. Finally, I would like to thank everyone who contributed, directly or indirectly, to the success of this internship.

Contents

List of Figures 5

Introduction 6

1 Problem Statement 7
1.1 Secure Shell (SSH) . . . 7
1.2 SSH Fingerprinting . . . 8

2 SSH Stacks Deployment 9
2.1 Docker Environment . . . 9
2.2 Dockerfile Standardization . . . 10
2.2.1 Build arguments . . . 10
2.2.2 Cross-Version Compatibility . . . 11
2.3 SSH Test Bed[1] . . . 11
2.3.1 Overview . . . 11
2.3.2 Validation and Testing . . . 12

3 Existing Fingerprinting Methods 14
3.1 Published information . . . 14
3.1.1 SSH Identification String . . . 14
3.1.2 Limitations as a Fingerprint . . . 15
3.2 Configuration-Based Fingerprinting . . . 17
3.2.1 Method Overview . . . 17
3.2.2 *HASSH*[2] . . . 21
3.2.3 Supported Algorithm Analysis . . . 23
3.2.4 Limitations . . . 26

4 State-machine-based fingerprinting 28
4.1 Conceptual Overview . . . 28
4.1.1 State-Machine Inference in TLS . . . 28
4.1.2 Extending the Approach to SSH . . . 29
4.2 Crafting Payloads . . . 30

4.2.1 SSH Binary Packet Protocol 30
4.2.2 SSH_MSG_KEXINIT 30
4.2.3 SS_MSG_DISCONNECT 30
4.2.4 SSH_MSG_SERVICE_REQUEST 31
4.2.5 SSH_MSG_UNIMPLEMENTED 31
4.2.6 SSH_MSG_NEWKEYS 31
4.2.7 SSH_MSG_USERAUTH_SUCCESS 32
4.2.8 SSH_MSG_USERAUTH_FAILURE 32
4.2.9 SSH_MSG_USERAUTH_BANNER 32
4.2.10 SSH_MSG_USERAUTH_REQUEST 33
4.3 Experimental Results 34

Conclusion **36**

Bibliography **37**

Appendix **39**

List of Figures

1.1 SSH Handshake Process 7

2.1 Docker Logo 9
2.2 Example tags from the OpenSSH Portable repository. 10
2.3 Overview of the SSH test bed. 12
2.4 Example validation record of OpenSSH stack versions. 13
2.5 Wireshark view of the OpenSSH `V_10_0_P2` packet capture. 13

3.1 Demonstration of banner spoofing in practice. 16
3.2 Demonstration of banner spoofing in practice. 16
3.3 Basic nmap scan output. 18
3.4 basic SSH enumeration with Nmap. 18
3.5 SSH algorithm enumeration with Nmap. 19
3.6 Nmap version detection on a spoofed banner. 20
3.7 Nmap version detection on a spoofed banner. 20
3.8 the specific components used to compose hassh and hasshServer[3]. 22
3.9 Excerpt from the json file[1]. 23
3.10 Excerpt from the reference file[1]. 24
3.11 Comparison graph for Key Exchange Algorithms[1]. 25
3.12 Timeline graph for WolfSSH's MAC algorithms support[1]. 26
3.13 `OpenSSH_7.6`'s new fingerprint after the changes. 26
3.14 `libssh_0.8.3`'s fingerprint. 27
3.15 the spoofed fingerprint. 27

4.1 CVE-2018-10933[4]. 29

Introduction

Secure Shell (SSH) is a cryptographic protocol for secure remote login and encrypted network services over an insecure network[5]. It is widely deployed in modern computing environments ranging from Unix-based systems such as Linux and MacOS to Microsoft Windows and embedded devices. Beyond its primary role in secure shell access, SSH also enables secure file transfers, encrypted tunneling, and other services, and serves as the foundation for tools like SCP and SFTP.

During this research internship, I aim to investigate SSH deployments in real-world systems by focusing on fingerprinting techniques to identify specific implementations, which are often considered unreliable. To explore these techniques, I built numerous Docker containers replicating different SSH stacks, with the purpose of running controlled experiments on configuration changes, protocol behaviors, and implementation-specific quirks.

Originally conceived as a study of TLS fingerprinting, this project has evolved to focus on SSH, whose simpler structure, broader availability of test targets, and lower setup complexity make it better suited for rapid experimentation and iterative development. Accordingly, the study will concentrate on server-side SSH stacks, as opposed to client-side implementations, since servers provide a simpler black-box target for controlled experiments and fingerprinting is more relevant when applied to servers in the wild.

The work will test different SSH fingerprinting tools, comparing their accuracy, ability to tell implementations apart, and resilience to configuration changes. It will require knowledge of network protocols, cryptography, and state-machine behavior, as well as skills in packet analysis and automation scripting.

Chapter 1

Problem Statement

1.1 Secure Shell (SSH)

Secure Shell (SSH) is a protocol that allows secure remote login and other secure network services over an insecure network. It works in three main parts[5]:

- **The Transport Layer Protocol**[6] provides a secure connection with server authentication and (optional) compression.
- **The User Authentication Protocol**[7] verifies the client's identity to the server.
- **The Connection Protocol**[8] splits the encrypted tunnel into multiple logical channels to accomplish various tasks.

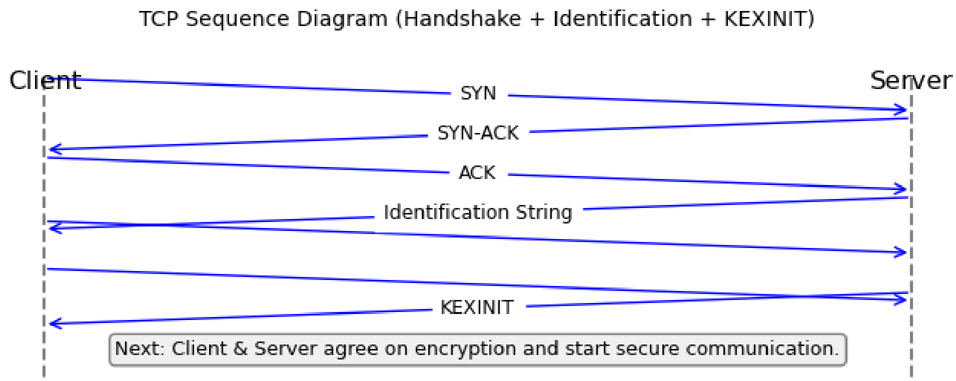


Figure 1.1: SSH Handshake Process

All communication, including authentication, commands, and file transfers, is protected by encryption and integrity checks to defend against network attacks, though the initial exchange of identification strings and protocol banners is sent in plain text once the connection is established.

1.2 SSH Fingerprinting

Fingerprinting in cybersecurity is a way to gather detailed information about a system or network. By looking at things like software versions, configurations, and protocols, you can create a digital fingerprint that uniquely identifies a system, similar to how physical fingerprints distinguish people.

There are many examples of fingerprinting for both unencrypted and encrypted protocols. However, one might be surprised to notice that very few open-source fingerprinting methods exist for SSH, despite it being one of the most widely used encrypted protocols and a critical component of the internet.

Profiling SSH implementations matters because even minor differences between versions, unknown implementations, or unusual configurations can create security vulnerabilities that attackers might exploit.

Fingerprinting tools can be categorized based on the layers they rely on to identify a given implementation:

- **Published information layer**: mainly the identification string shared during the initial handshake, which may include the protocol version and implementation details.
- **Configuration layer**, including cryptographic capabilities, supported algorithms, or protocol features.
- **Behavioral layer**, which comes from observable differences in how state machines behave across different software stacks.

During this project, the aim will be to test the first two layers, identify their limitations, and attempt a proof of concept for the third layer.

Chapter 2

SSH Stacks Deployment

2.1 Docker Environment

Docker is an open platform for developing, shipping, and running applications. It allows you to separate applications from the underlying infrastructure[9]. At its core, Docker provides a way to package and run applications in isolated environments called containers. Containers are lightweight, self-contained, and include everything needed to run an application, making them portable across environments and there's no need to rely on what's installed on the host.

Figure 2.1: Docker Logo

In this context, we require a large sample of SSH implementations to study their behaviors under different configurations and environments. Docker containers are ideal for this purpose, as they allow us to quickly deploy and manage multiple, isolated SSH stacks. Each container can replicate a specific SSH implementation or configuration, ensuring reproducibility and enabling systematic experiments. This approach provides a flexible and controlled environment to explore a wide variety of SSH deployments without impacting production systems.

2.2 Dockerfile Standardization

2.2.1 Build arguments

One of the main challenges in this step was the sheer number of images we would be building across many versions of multiple SSH implementations. Managing this complexity required an approach that minimized duplication while still allowing flexibility. To achieve this, the Dockerfiles were designed to accept **build arguments**, which are parameters defined within the Dockerfile that can be overridden externally during the build phase. This made it possible to script and automate the creation of a large number of images without having to maintain separate Dockerfiles for each version.
The example below shows how to pass a build argument to an image build:

```
$ docker build --build-arg COMMIT=$COMMIT -t openssh-$COMMIT_LOWER .
```

Here, `COMMIT` refers to the specific version tag of the SSH implementation to be built, pulled directly from the project's Git tags. This way, we can iterate over multiple tags at a time. Note that `COMMIT_LOWER` is an environment variable derived from `COMMIT` and converted to lowercase, as Docker image names cannot contain uppercase characters. It is used solely for syntax compliance and is not otherwise significant.

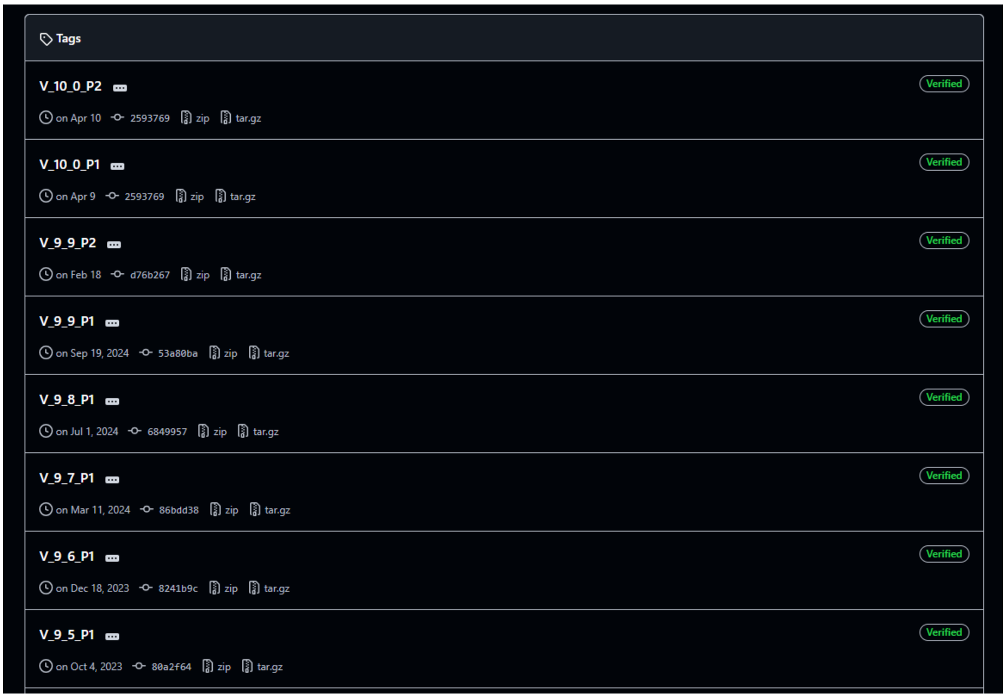


Figure 2.2: Example tags from the OpenSSH Portable repository.

In the Dockerfile, the argument is used to select the tag:

```
# Build argument for OpenSSH version (commit or tag)
ARG COMMIT=V_10_0_P2
# Clone OpenSSH repository and checkout the specified version
RUN git clone https://github.com/openssh/openssh-portable --depth=1 -b
    ↪ ${COMMIT} . && \
    autoreconf -i
```

For the complete Dockerfile template used in this example, see Appendix.1. Other implementations may require additional parameters. For instance, WolfSSH uses two build arguments, one for the WolfSSH version and another for WolfSSL, which is required as a dependency, as not all versions are mutually compatible.

2.2.2 Cross-Version Compatibility

Another difficulty with this approach lies in writing a single Dockerfile template that remains functional across multiple versions. Older versions often depend on deprecated libraries or outdated build tools, which may not integrate cleanly with modern base images or package managers.

Ensuring that these version-specific requirements could be handled within one generalized Dockerfile required careful structuring, conditional installation steps, and occasional workarounds to maintain compatibility. This standardized but flexible design ultimately made it possible to automate the image build process at scale while still accommodating the quirks of each SSH implementation and version.

For example, I was able to organize OpenSSH into three different Dockerfiles, each covering its own range of versions. As the versions got older, they became less compatible with modern build environments. While newer implementations were relatively straightforward to build, older ones required applying patches to remain compatible, with the earliest releases being the most challenging due to their outdated build systems, which necessitated performing the entire installation process manually within the Dockerfile.

2.3 SSH Test Bed[1]

2.3.1 Overview

The result of these efforts is a collection of **180** SSH server stacks, each providing client access to a shell. These stacks were built using 9 Dockerfiles across 5 different SSH implementations: **OpenSSH**, **WolfSSH**, **Dropbear**, **libssh**, and **AsyncSSH**. The coverage of versions and implementations is summarized in the table below where each row represents a version range handled by a single Dockerfile:

Implementation	Versions	Release Timeframe	# of Stacks
OpenSSH	apt-get version (9.2p1)	Feb 2023	1
	V_10_0_P2 to V_7_9_P1	Apr 2025 to Oct 2018	25
	V_7_8_P1 to V_6_3_P1	Aug 2018 to Sep 2013	18
	V_6_2_P2 to V_4_4_P1	May 2013 to Sep 2006	20
Dropbear	apt-get version (v2022.83)	Nov 2022	1
	v2025.88 to v0.44	May 2025 to Jan 2005	48
WolfSSH	v1.4.20-stable to v1.4.13-stable	Feb 2025 to Apr 2023	8
libssh	0.11.2 to 0.8.3	Jun 2025 to Sep 2018	25
AsyncSSH	v2.21.0 to v2.0.0	May 2025 to Oct 2019	34

Figure 2.3: Overview of the SSH test bed.

Note: The "apt-get version" stacks use whichever version is currently available via apt in Debian Bookworm, which at the time was the latest stable release of Debian.

The selection of implementations was guided by both relevance and diversity. OpenSSH, Dropbear, and libssh are among the most widely deployed server-side SSH stacks, while WolfSSH and AsyncSSH represent alternative, lightweight, or feature-rich implementations that broaden the spectrum of behaviors covered. This mix ensures that the test bed captures both mainstream usage and edge cases, making it suitable for comprehensive analysis.

These standardized images can be reused for testing vulnerabilities or conducting controlled experiments. They have been collected into a single repository titled **SSH Test Bed**, inspired by similar work done on TLS stacks conducted years earlier by my predecessors at Telecom SudParis[10]. This prior work will be referenced later in Chapter 4, as it is important for context and methodology.

2.3.2 Validation and Testing

The next step in validating the SSH test bed was to ensure that each Docker stack functioned correctly under its default configuration. This was achieved by building and running each stack, then establishing a connection and executing a simple command inside the shell. Specifically, I opted to simply request the SSH implementation's software version, which served as a minimal but reliable check to confirm that the build and runtime processes were successful, while also storing the output separately for later verification and record-keeping.

The entire validation process is automated using a shell script (see the OpenSSH example in Appendix.2). Establishing the SSH connection itself, including handling password prompts and the host key verification prompt, is handled by an **expect** script (also included in Appendix.3). To facilitate later analysis, all network traffic generated during these interactions is captured with **tcpdump** and stored in a

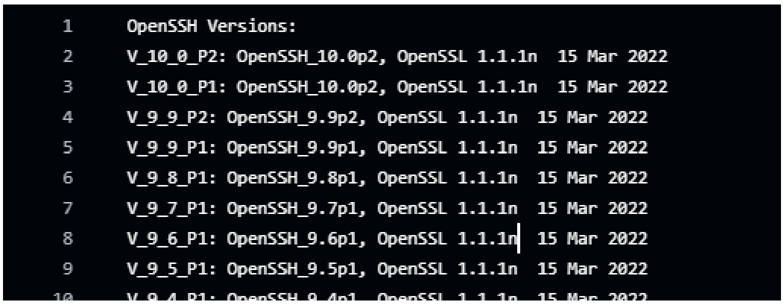


Figure 2.4: Example validation record of OpenSSH stack versions.

`.pcap` file. This approach allows us to study a single exchange in isolation, eliminates variability across runs, and provides a reproducible dataset for fingerprinting and protocol analysis.
During testing, each stack was run with Docker's port mapping feature:

```
docker run --rm -d -p <host_port>:<container_port> <image_name>
```

The `--rm` flag ensures that containers are automatically removed after stopping, and `-d` runs them in detached mode. The SSH service inside each container runs on its default port but it is accessible from the host via port 2222 through Docker's port mapping. Most stacks used the default container-side port 22, which is officially assigned for SSH by IANA when used over TCP/IP[6], though some implementations preferred different ports, such as 11111 for WolfSSH or 8022 for AsyncSSH. This consistent host mapping ensured uniform access for automated testing while keeping the implementation-specific default ports inside the container.

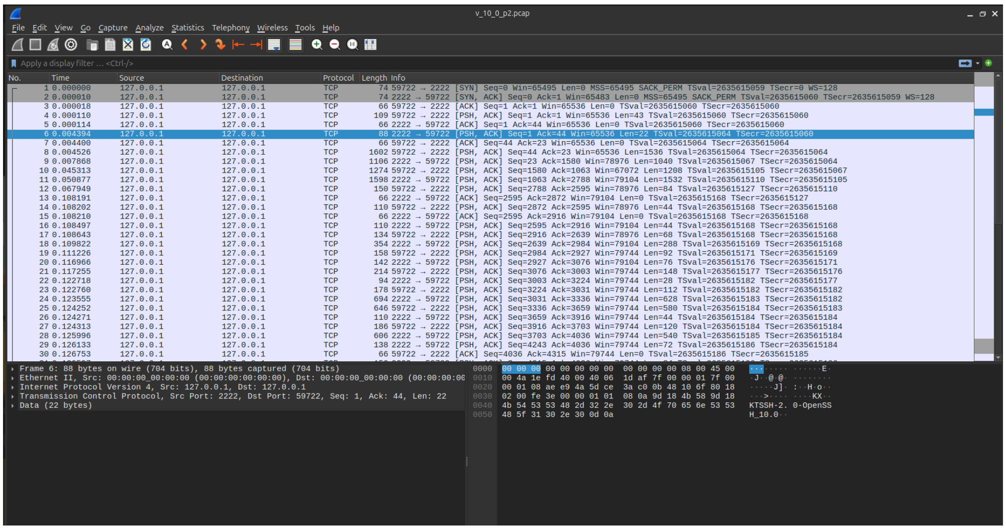


Figure 2.5: Wireshark view of the OpenSSH `V_10_0_P2` packet capture.

Chapter 3

Existing Fingerprinting Methods

3.1 Published information

3.1.1 SSH Identification String

As defined in the SSH Transport Layer Protocol specification[6], both the client and the server are required to send an *identification string* immediately after the connection is established. This string follows the general format:

```
SSH-protoversion-softwareversion [<SP> comments] <CR><LF>
```

Where `protoversion` is generally set to `2.0`, as this is the version of the protocol currently standardized and the one exclusively used throughout this project. The `softwareversion` specifies the particular SSH implementation in use, while the optional `comments` field (enclosed in brackets in the above specification) may contain additional information that could assist in diagnosing user problems. The separator <SP> corresponds to the ASCII 32 space character, which is used to delimit the `softwareversion` from the optional `comments`. The identification string must be terminated by a single Carriage Return (<CR>, ASCII 13) followed by a single Line Feed (<LF>, ASCII 10), and its total length is limited to 255 characters.

For example, a typical identification string might be:

```
SSH-2.0-libssh_0.11.2<CR><LF>
```

and an identification string including the optional comment field could be:

```
SSH-2.0-OpenSSH_9.6p1<SP>Ubuntu-3ubuntu13.11<CR><LF>
```

The identification string serves two important roles. First, it ensures protocol compatibility between client and server, as older undocumented versions of SSH may deviate from the expected formatting. Second, it is incorporated into the Diffie-Hellman key exchange, thereby binding the session's cryptographic setup to the precise software version string exchanged at connection time.

In addition, it acts as the delimiter that finalizes the connection setup phase and signals the transition into the cryptographic handshake, as the specification permits servers to send arbitrary pre-identification lines prior to the version string but mandates that once the identification string is exchanged, key exchange begins immediately.

3.1.2 Limitations as a Fingerprint

While it may seem intuitive to use the identification string for fingerprinting, it is immediately clear that it only represents what the stack **claims to be**, rather than **what it actually is**. Moreover, it is a simple plaintext string that can be read and transmitted with minimal effort.

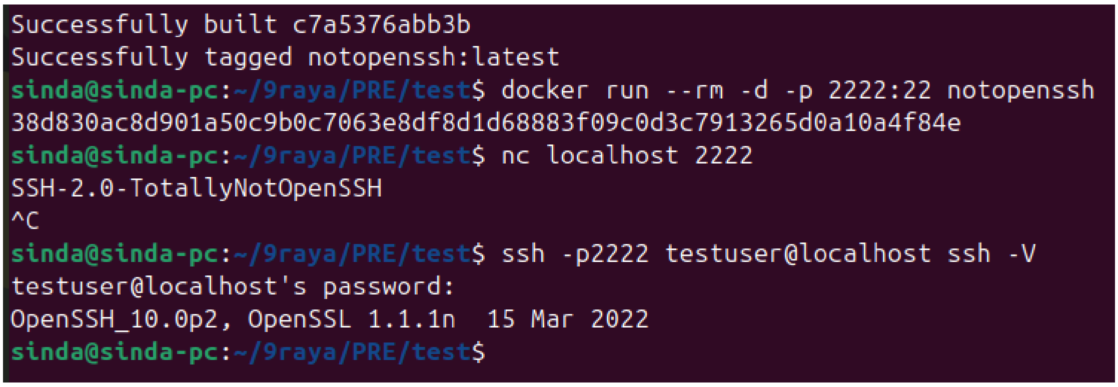
It is trivial to modify a stack so that it claims to be a different implementation or even a completely made-up custom version without changing any underlying behavior. For example, in OpenSSH, the logic responsible for sending the identification string is handled within the file `kex.c` of the source repository:

```
/* Prepare and send our banner */
sshbuf_reset(our_version);
if (version_addendum != NULL && *version_addendum == '\0')
        version_addendum = NULL;
if ((r = sshbuf_putf(our_version, "SSH-%d.%d-%s%s%s\r\n",
    PROTOCOL_MAJOR_2, PROTOCOL_MINOR_2, SSH_VERSION,
    version_addendum == NULL ? "" : " ",
    version_addendum == NULL ? "" : version_addendum)) != 0) {
        oerrno = errno;
        error_fr(r, "sshbuf_putf");
        goto out;
}
```

By altering this snippet, it is possible to change the advertised identification string to an arbitrary value. As a playful demonstration, we configured it to send the placeholder string `SSH-2.0-TotallyNotOpenSSH<CR><LF>` instead.

```
sshbuf_reset(our_version);
if (version_addendum != NULL && *version_addendum == '\0')
        version_addendum = NULL;
if ((r = sshbuf_putf(our_version, "SSH-2.0-TotallyNotOpenSSH\r\n
    ↪ ")) != 0) {
        oerrno = errno;
        error_fr(r, "sshbuf_putf");
        goto out;
}
```

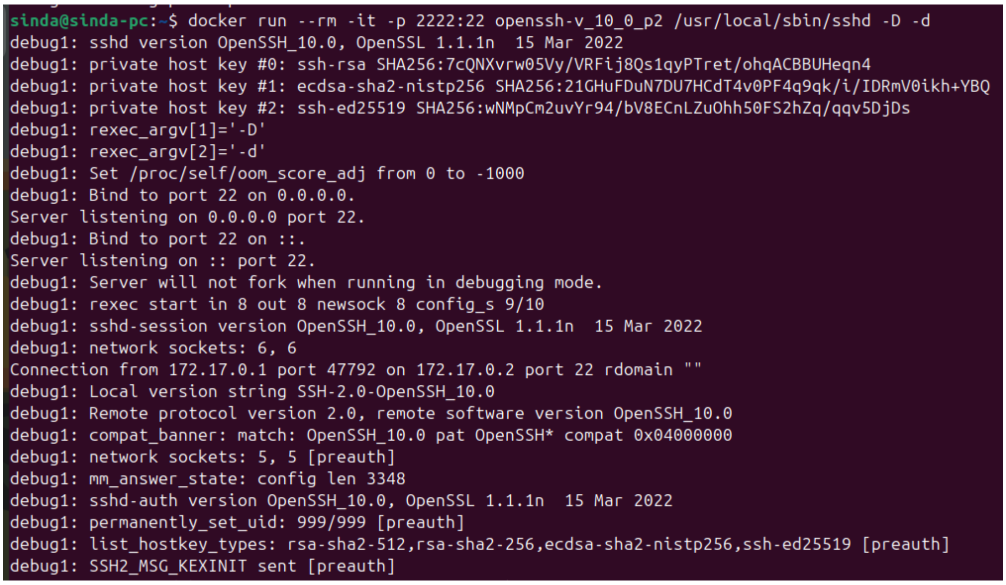
As we can see in the screenshot below, the stack sends the modified banner, which is captured by `nc` as the very first line upon connection. This demonstrates how the banner is the initial message any client receives, confirming that the advertised version string has been altered. However, the implementation is demonstrably still `OpenSSH_10.0p2`, as seen when directly asking inside the shell which version of SSH is currently running.



```
Successfully built c7a5376abb3b
Successfully tagged notopenssh:latest
sinda@sinda-pc:~/9raya/PRE/test$ docker run --rm -d -p 2222:22 notopenssh
38d830ac8d901a50c9b0c7063e8df8d1d68883f09c0d3c7913265d0a10a4f84e
sinda@sinda-pc:~/9raya/PRE/test$ nc localhost 2222
SSH-2.0-TotallyNotOpenSSH
^C
sinda@sinda-pc:~/9raya/PRE/test$ ssh -p2222 testuser@localhost ssh -V
testuser@localhost's password:
OpenSSH_10.0p2, OpenSSL 1.1.1n  15 Mar 2022
sinda@sinda-pc:~/9raya/PRE/test$
```

Figure 3.1: Demonstration of banner spoofing in practice.

Additionally, a non-SSH party can mimic the protocol by sending the correct payloads at the right times (an idea that will be fundamental for understanding Chapter 4) and thereby also claim to be an implementation it is not. We can, for example, send `OpenSSH_10.0p2`'s ID string via a small Python script (Appendix.4).



```
sinda@sinda-pc:~$ docker run --rm -it -p 2222:22 openssh-v_10_0_p2 /usr/local/sbin/sshd -D -d
debug1: sshd version OpenSSH_10.0, OpenSSL 1.1.1n  15 Mar 2022
debug1: private host key #0: ssh-rsa SHA256:7cQNXvrw05Vy/VRFij8Qs1qyPTret/ohqACBBUHeqn4
debug1: private host key #1: ecdsa-sha2-nistp256 SHA256:21GHuFDuN7DU7HCdT4v0PF4q9qk/i/IDRmV0ikh+YBQ
debug1: private host key #2: ssh-ed25519 SHA256:wNMpCm2uvYr94/bV8ECnLZuOhh50FS2hZq/qqv5DjDs
debug1: rexec_argv[1]='-D'
debug1: rexec_argv[2]='-d'
debug1: Set /proc/self/oom_score_adj from 0 to -1000
debug1: Bind to port 22 on 0.0.0.0.
Server listening on 0.0.0.0 port 22.
debug1: Bind to port 22 on ::.
Server listening on :: port 22.
debug1: Server will not fork when running in debugging mode.
debug1: rexec start in 8 out 8 newsock 8 config_s 9/10
debug1: sshd-session version OpenSSH_10.0, OpenSSL 1.1.1n  15 Mar 2022
debug1: network sockets: 6, 6
Connection from 172.17.0.1 port 47792 on 172.17.0.2 port 22 rdomain ""
debug1: Local version string SSH-2.0-OpenSSH_10.0
debug1: Remote protocol version 2.0, remote software version OpenSSH_10.0
debug1: compat_banner: match: OpenSSH_10.0 pat OpenSSH* compat 0x04000000
debug1: network sockets: 5, 5 [preauth]
debug1: mm_answer_state: config len 3348
debug1: sshd-auth version OpenSSH_10.0, OpenSSL 1.1.1n  15 Mar 2022
debug1: permanently_set_uid: 999/999 [preauth]
debug1: list_hostkey_types: rsa-sha2-512,rsa-sha2-256,ecdsa-sha2-nistp256,ssh-ed25519 [preauth]
debug1: SSH2_MSG_KEXINIT sent [preauth]
```

Figure 3.2: Demonstration of banner spoofing in practice.

As illustrated by the following debug output from the server:

```
debug1: Remote protocol version 2.0, remote software version OpenSSH_10
    ↪ .0
debug1: compat_banner: match: OpenSSH_10.0 pat OpenSSH* compat 0
    ↪ x04000000
```

even though the identification string was sent by a minimal Python script rather than a genuine SSH client, the server interprets it as originating from OpenSSH and begins preparing to interact accordingly.

It thus becomes abundantly clear that the identification string is primarily part of the SSH transport protocol and the key exchange process, and is not a reliable fingerprint for our purposes.

3.2 Configuration-Based Fingerprinting

3.2.1 Method Overview

Configuration-based fingerprinting involves identifying servers or clients by examining the specific details of their protocol implementations, such as supported cryptographic algorithms, key exchange methods, and other protocol features. This approach is particularly straightforward with SSH, since both the client and server are required to announce their supported configurations during the initial handshake.

Despite SSH's widespread adoption as a critical encrypted protocol, very few open-source tools exist that can efficiently and systematically fingerprint its configurations. Most fingerprinting efforts have focused on other protocols.

One straightforward approach to configuration-based fingerprinting is to observe the key exchange packets during the SSH handshake, as these packets contain information about the algorithms and features supported by both the client and server.

Another widely used method is enumeration using Nmap[11]. Enumeration involves actively probing a target network or host to gather information about available services, open ports, and software versions. When applied to SSH, Nmap can identify hosts with open SSH ports, determine the server software in use, and detect potential vulnerabilities or misconfigurations.

Beyond simple service enumeration, Nmap supports targeted enumeration and fingerprinting of SSH servers. Its dedicated scripts can extract software details, highlight weak configurations, and even detect servers concealed behind firewalls, turning basic scans into a practical tool for assessing SSH security. Basic enumeration can be initiated with a simple scan command:

```
nmap [target IP address]
```

In our case, the target will be set to localhost for demonstration purposes. The resulting output would resemble the following:

```
sinda@sinda-pc:~$ nmap localhost
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-08-17 02:58 CEST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.000044s latency).
Not shown: 997 closed tcp ports (conn-refused)
PORT      STATE SERVICE
111/tcp   open  rpcbind
631/tcp   open  ipp
2222/tcp open  EtherNetIP-1

Nmap done: 1 IP address (1 host up) scanned in 0.02 seconds
```

Figure 3.3: Basic nmap scan output.

As we can see, port 2222/tcp is reported as open on localhost, which makes sense because Docker mapped the container's internal SSH service (port 22) to port 2222 on our host. Nmap labels it as "EtherNetIP-1" because its built-in service database associates port 2222 with the EtherNet/IP industrial protocol rather than SSH, and since the SSH daemon is running on a nonstandard port, Nmap defaults to that label even though the service is in fact SSH.

To resolve this ambiguity and obtain more accurate results, the `-sV` flag can be used to enable version detection, which provides detailed information about the services bound to open ports. A basic SSH enumeration with Nmap can be performed as follows:

```
sinda@sinda-pc:~$ nmap -sV -p 2222 localhost
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-08-17 03:09 CEST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.000054s latency).

PORT     STATE SERVICE VERSION
2222/tcp open  ssh     OpenSSH 10.0 (protocol 2.0)

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 0.12 seconds
```

Figure 3.4: basic SSH enumeration with Nmap.

We can even leverage Nmap's script engine to run specialized scripts that extract detailed information from an SSH server, such as the supported algorithms using `ssh2-enum-algos.nse`.

```
sinda@sinda-pc:~$ nmap -sV --script ssh2-enum-algos.nse -p 2222 localhost
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-08-17 03:14 CEST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.000071s latency).

PORT     STATE SERVICE VERSION
2222/tcp open  ssh     OpenSSH 10.0 (protocol 2.0)
| ssh2-enum-algos:
|   kex_algorithms: (10)
|       mlkem768x25519-sha256
|       sntrup761x25519-sha512
|       sntrup761x25519-sha512@openssh.com
|       curve25519-sha256
|       curve25519-sha256@libssh.org
|       ecdh-sha2-nistp256
|       ecdh-sha2-nistp384
|       ecdh-sha2-nistp521
|       ext-info-s
|       kex-strict-s-v00@openssh.com
|   server_host_key_algorithms: (4)
|       rsa-sha2-512
|       rsa-sha2-256
|       ecdsa-sha2-nistp256
|       ssh-ed25519
|   encryption_algorithms: (6)
|       chacha20-poly1305@openssh.com
|       aes128-gcm@openssh.com
|       aes256-gcm@openssh.com
|       aes128-ctr
|       aes192-ctr
|       aes256-ctr
|   mac_algorithms: (10)
|       umac-64-etm@openssh.com
|       umac-128-etm@openssh.com
|       hmac-sha2-256-etm@openssh.com
|       hmac-sha2-512-etm@openssh.com
|       hmac-sha1-etm@openssh.com
|       umac-64@openssh.com
|       umac-128@openssh.com
|       hmac-sha2-256
|       hmac-sha2-512
|       hmac-sha1
|   compression_algorithms: (2)
|       none
|_      zlib@openssh.com
```

Figure 3.5: SSH algorithm enumeration with Nmap.

Some other information Nmap could provide us includes identifying the operating system of a target using the `-O` flag, which helps uncover OS-specific vulnerabilities, and employing other scripts such as `http-enum.nse` or `smb-enum-users.nse` to enumerate additional services like HTTP or SMB, giving a broader view of the target's exposure.

However, despite appearing to probe the server for detailed information, this type of scan is not a fully reliable fingerprinting method, as it primarily relies on the identificatiton string returned by the target. Initially, it was interesting to observe that Nmap did not recognize the deliberately modified banner `SSH-2.0-TotallyNotOpenSSH<CR><LF>`

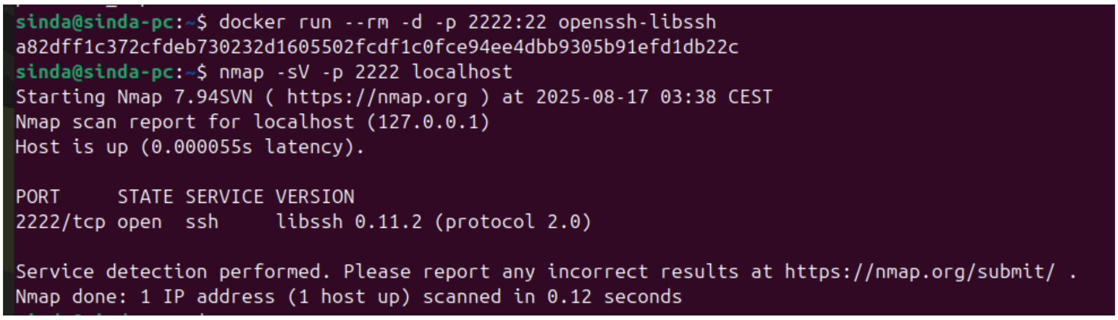

```
sinda@sinda-pc:~$ nmap -sV -p 2222 localhost
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-08-17 03:32 CEST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.000054s latency).

PORT     STATE SERVICE VERSION
2222/tcp open  ssh     (protocol 2.0)
1 service unrecognized despite returning data. If you know the service/version, please submit the following fingerprint at https://nmap.org/cgi-bin/submit.cgi?new-service :
SF-Port2222-TCP:V=7.94SVN%I=7%D=8/17%Time=68A13128%P=x86_64-pc-linux-gnu%r
SF:(NULL,1B,"SSH-2\.0-TotallyNotOpenSSH\r\n");

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 6.12 seconds
```

Figure 3.6: Nmap version detection on a spoofed banner.

Yet, When the experiment was repeated using the legitimate identification string `SSH-2.0-libssh_0.11.2<CR><LF>`, Nmap classified the service as a genuine libssh stack.


```
sinda@sinda-pc:~$ docker run --rm -d -p 2222:22 openssh-libssh
a82dff1c372cfdeb730232d1605502fcdf1c0fce94ee4dbb9305b91efd1db22c
sinda@sinda-pc:~$ nmap -sV -p 2222 localhost
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-08-17 03:38 CEST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.000055s latency).

PORT     STATE SERVICE VERSION
2222/tcp open  ssh     libssh 0.11.2 (protocol 2.0)

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 0.12 seconds
```

Figure 3.7: Nmap version detection on a spoofed banner.

Thus, Nmap should be treated as a scanner that allows us to study the SSH stack, rather than as a bona fide fingerprinting tool which would require a thorough analysis of the server's configuration, with the fingerprint deduced from those observed properties. Enter *HASSH*.

3.2.2 *HASSH*[2]

HASSH is an open-source network fingerprinting method developed by the Detection Cloud team at Salesforce. It allows identification of specific SSH client and server implementations. The resulting fingerprints are represented as standardized text strings, `hassh` for clients and `hasshServer` for servers, making them easy to store, search, and share[3].

HASSH addresses the lack of a framework to verify the authenticity of SSH components by providing a standardized fingerprint of SSH clients and servers. it can notably allow precise identification of software implementations even when multiple clients share an IP, detect clients with unusual or multiple fingerprints that may indicate covert exfiltration, enforce security policies by blocking or alerting on unapproved clients, identify deceptive or honeypot servers, monitor unpatched or unauthorized clients and servers, and detect IoT devices communicating over encrypted channels.

As per the specification[6], after the initial TCP three-way handshake and the exchange of identification strings, the key exchange begins with each side sending name-lists of supported algorithms. This is done using the following packet:

```
byte SSH_MSG_KEXINIT (value 20)
byte[16] cookie (random bytes)
name-list kex_algorithms
name-list server_host_key_algorithms
name-list encryption_algorithms_client_to_server
name-list encryption_algorithms_server_to_client
name-list mac_algorithms_client_to_server
name-list mac_algorithms_server_to_client
name-list compression_algorithms_client_to_server
name-list compression_algorithms_server_to_client
name-list languages_client_to_server
name-list languages_server_to_client
boolean first_kex_packet_follows
uint32 0 (reserved for future extension)
```

Each of the algorithm name-lists must be a comma-separated list of algorithm names, and the `cookie` must be a random value generated by the sender, its purpose being to make it impossible for either side to fully determine the keys and the session identifier. This packet is titled `SSH_MSG_KEXINIT`. This packet is necessary so that the server and client can negotiate which algorithms will be used for the session.

HASSH and HASSHServer fingerprints are MD5 hashes constructed from a specific set of algorithms supported and preferred by various SSH applications.

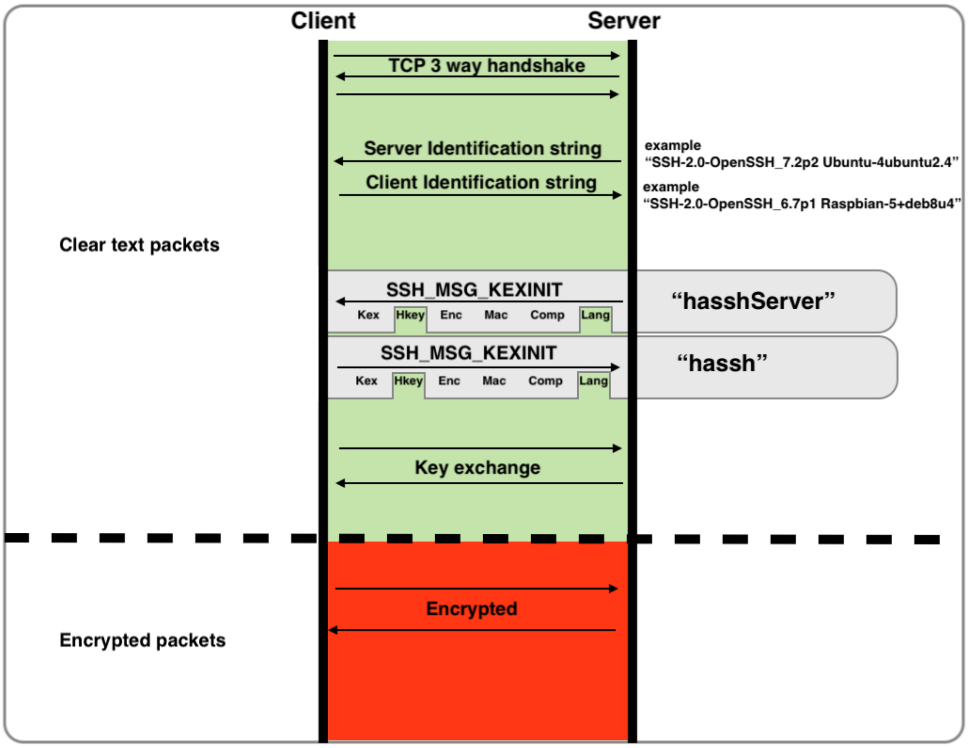


Figure 3.8: the specific components used to compose hassh and hasshServer[3].

The next step is to examine the fingerprints of the 180 containers in our testbed, cataloging the observed SSH fingerprints for the default configurations of these implementations. This analysis will be performed using the traffic previously captured and stored in the `.pcap` files.

Fortunately, there exists a Zeek script, `hassh.zeek`, which by default adds several fields to your `ssh.log` file, Zeek being a powerful open-source network monitoring platform that passively observes traffic and generates detailed logs for analysis. These fields include `hasshVersion`, `hassh` along with `hasshAlgorithms`, `hasshServer` with `hasshServerAlgorithms`, and the client and server host key algorithms, recorded as `cshka` and `sshka` respectively. To use it, simply run:

```
zeek -C -r "$pcapfile" local
```

and then analyze the results in ssh.log, which is made easy if we use zeek-cut.

```
server_list=$(zeek-cut server < "$ssh_log")
hassh_list=$(zeek-cut hasshServer < "$ssh_log")
algos_list=$(zeek-cut hasshServerAlgorithms < "$ssh_log")
sshka_list=$(zeek-cut sshka < "$ssh_log")
```

Following these steps, I was able to compile a complete inventory of all server stacks across multiple formats, the most important being a **JSON file**. This file details each stack, its `hasshServer`, all the algorithms that make up its configuration, and the release date, allowing us to organize them both by implementation and by time of release.

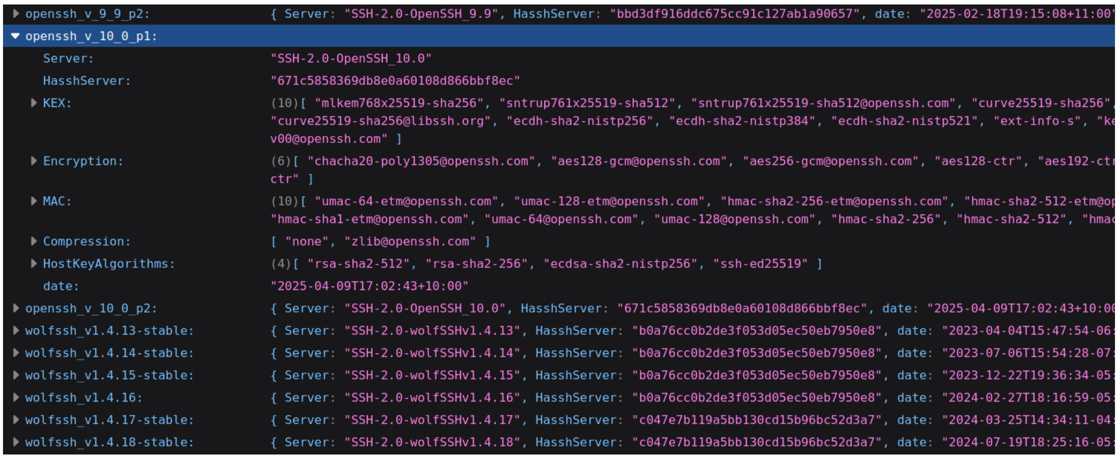


Figure 3.9: Excerpt from the json file[1].

A minor issue one might notice is that versions released close to each other often retain the same fingerprint, which makes sense, as no major configuration changes would typically occur in such a short timeframe. A more challenging question is whether two tangentially different implementations could share a fingerprint. To verify this, one would need to check for overlap in their supported algorithms and adjust the configuration accordingly to see whether the resulting fingerprints coincide, which would reveal a limitation of the method by showing that distinct implementations can appear identical.

3.2.3 Supported Algorithm Analysis

Starting from OpenSSH version `V_6_3_P1`, obtaining the complete list of supported algorithms has become straightforward. This is due to the introduction of the `-Q` option, which was specifically added for this purpose.

```
for t in kex cipher mac compression key; do
    echo $t:;
    ssh -Q $t;
    echo;
done
```

However, for older versions of OpenSSH, obtaining the list of supported algorithms is slightly more complicated.

Going forward, we will explore approaches to identify all supported algorithms for these older versions. This effort could be useful for bibliographic purposes and future research. For the sake of simplicity, however, we will focus primarily on OpenSSH in this step, as it represents the most widely used and complete implementation.

A first step towards this was establishing a point of reference for the algorithms that can be supported by any SSH implementation. To achieve this, I combined three sources of information: my pre-existing results from the earlier HASSH analysis, the results obtained using the `-Q` flag, and the Secure Shell Protocol Parameters IANA specifications[12]. These were unified into a single JSON file containing a comprehensive list of supported algorithms for each type.

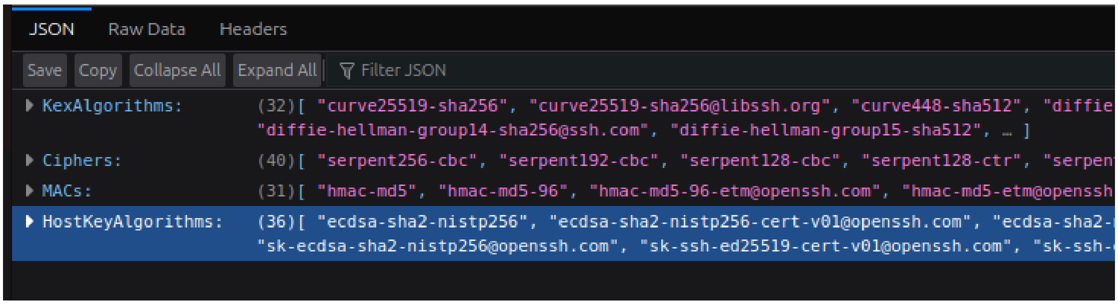


Figure 3.10: Excerpt from the reference file[1].

A more preliminary, rough approach involved using `strings` on the `sshd` binary and piping the output through a series of carefully crafted `grep` commands. These commands were designed to filter the results based on our reference file, extracting only the algorithms of interest. For instance, extracting key exchange algorithms using this approach could be done as follows:

```
strings /usr/local/sbin/sshd |
  tr ',' '\n' |
  grep -Ei 'diffie|curve|ecdh|sntrup|mlkem|x25519|4591761|kexguess|rsa
      ↪ [0-9]+-sha|kex-strict' |
  grep -E '^[a-z0-9\-]+(@[a-z0-9.\-]+)?$' |
  grep -vEi 'hmac|aes|cbc|gcm|ctr|zlib|none|chacha20|arcfour|blowfish|
      ↪ cast128|twofish|umac|ssh-dss|ecdsa|ssh-rsa' |
  tr -d '\r"' | sed 's/^"//;s/"$//' | sort -u ;
```

While this approach produced results consistent with those obtained using the `-Q` option in versions that support it, it is far too brittle to be considered reliable. Simply filtering the contents of a binary is inherently unstable and prone to inconsistencies, making it unsuitable as a definitive method.

The alternative approach that proved effective was to iterate over all algorithms listed in the reference file and attempt to use each one with the SSH implementation.

The SSH response then directly indicates whether a given algorithm is supported.

In this case, we modified our Expect script to call:

```
spawn ssh -p2222 [lindex $argv 1]@[lindex $argv 0] ssh -o [lindex $argv
    ↪ 3]=[lindex $argv 4] betise
```

Here, `argv3` represents the algorithm type and `argv4` the algorithm name. The placeholder `betise` is a bogus, non-existent hostname, ensuring that the test does not accidentally connect to a real host.

We then simply check whether the SSH stack reports an error due to the non-existent hostname or due to an unsupported algorithm, and keep track of the results accordingly.

The final step is to study the evolution of algorithm support over time across different SSH implementations, which can be valuable for bibliographic purposes, and to compare which algorithms OpenSSH has actually supported versus those announced in its default configurations over time. As always, all the results can be found in the SSH test bed repository[1]. For this report, however, we will present two example graphs:

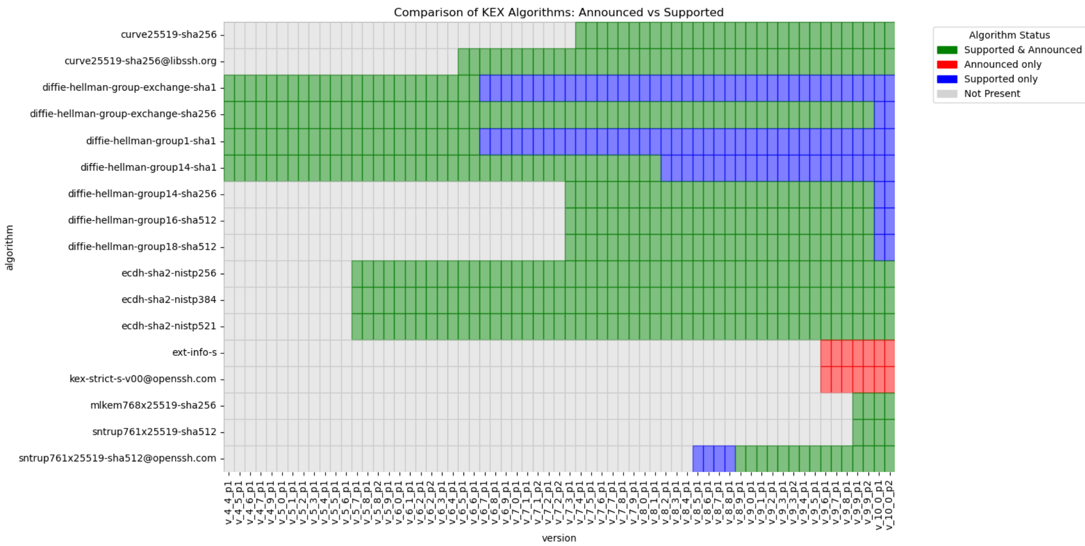


Figure 3.11: Comparison graph for Key Exchange Algorithms[1].

One might notice the unusual presence of two algorithms that are announced but not listed as supported. These are `ext-info-s` and `kex-strict-s-v00@openssh.com`, which are not key exchange algorithms but rather serve as auxiliary or protocol-specific extensions.

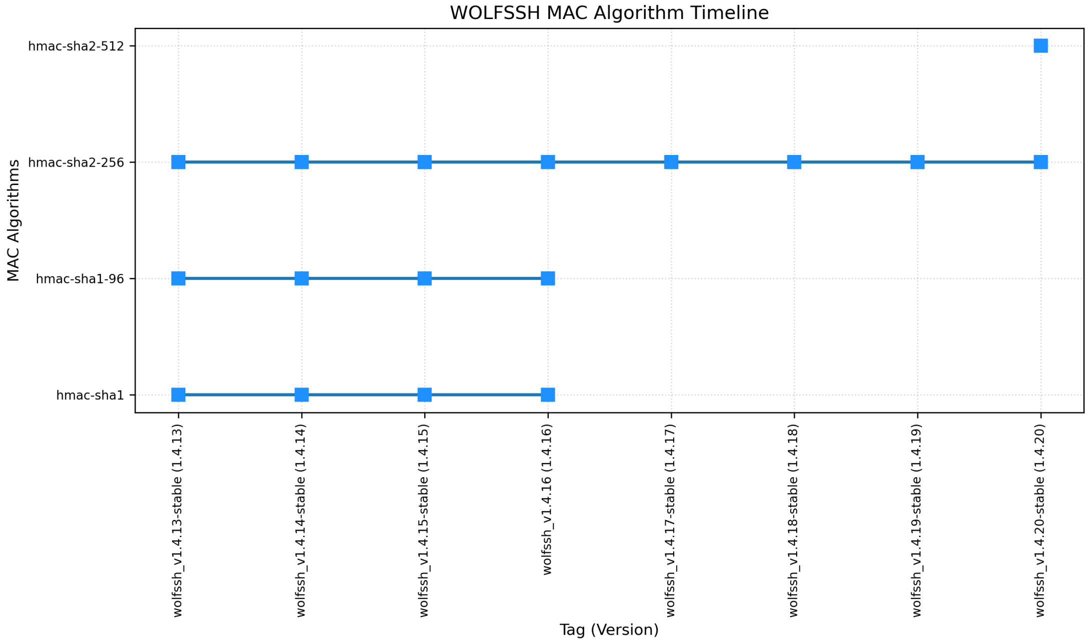
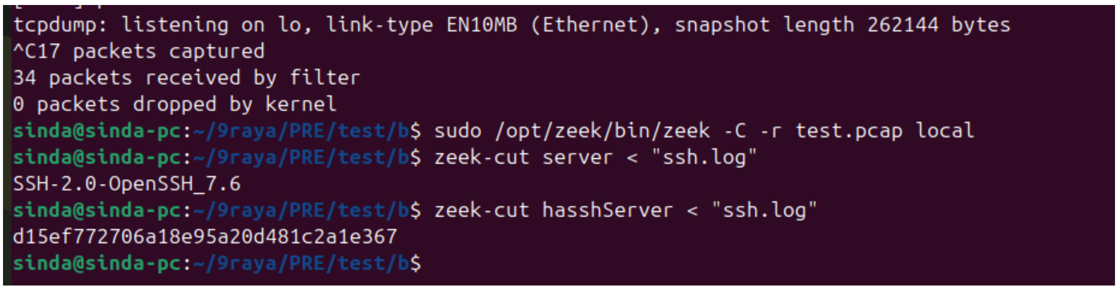


Figure 3.12: Timeline graph for WolfSSH's MAC algorithms support[1].

3.2.4 Limitations

As stated previously, demonstrating the limitation of this method only requires identifying two different implementations that could, in theory, produce the same fingerprint. While not trivial to set up, `OpenSSH_7.6` and `libssh_0.8.3` exhibit such a possibility. By adding `zlib` support and modifying the order in which algorithms are announced, we were able to achieve the desired effect.



```
tcpdump: listening on lo, link-type EN10MB (Ethernet), snapshot length 262144 bytes
^C17 packets captured
34 packets received by filter
0 packets dropped by kernel
sinda@sinda-pc:~/9raya/PRE/test/b$ sudo /opt/zeek/bin/zeek -C -r test.pcap local
sinda@sinda-pc:~/9raya/PRE/test/b$ zeek-cut server < "ssh.log"
SSH-2.0-OpenSSH_7.6
sinda@sinda-pc:~/9raya/PRE/test/b$ zeek-cut hasshServer < "ssh.log"
d15ef772706a18e95a20d481c2a1e367
sinda@sinda-pc:~/9raya/PRE/test/b$
```

Figure 3.13: `OpenSSH_7.6`'s new fingerprint after the changes.

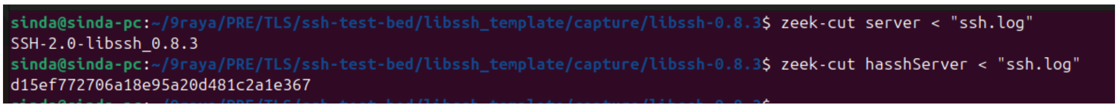


Figure 3.14: `libssh_0.8.3`'s fingerprint.

As an additional exercise, it was also possible to craft and send the appropriate payloads directly over a raw socket, thereby reproducing the same fingerprint, as shown below. The code for this spoof implementation can be found in Appendix.5.

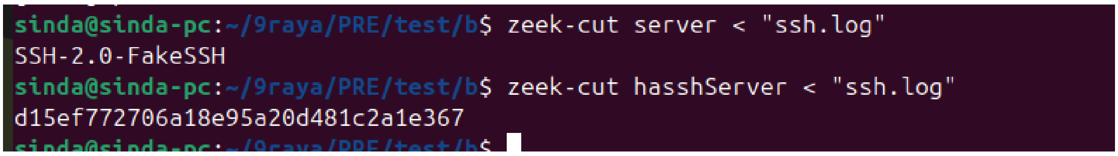


Figure 3.15: the spoofed fingerprint.

So, while not trivial, it is possible to forge configuration-based fingerprints. Although HASSH is well suited for tracking known honeypot configurations and other malicious deployments, it is not a foolproof fingerprinting method capable of immediately identifying the exact implementation and version of a given stack.

Chapter 4

State-machine-based fingerprinting

4.1 Conceptual Overview

4.1.1 State-Machine Inference in TLS

The main inspiration for the approach taken in this work comes from earlier research at Télécom SudParis on applying state machine inference to TLS implementations in order to study how they react when faced with unusual or malformed message sequences[10].

TLS sits at the heart of Internet security, yet the RFC does not provide a reference automaton to guide implementers. Instead, developers must reconstruct the state machine themselves from textual descriptions of messages and their sequencing. As a result, TLS implementations often diverge from one another in subtle ways, and sometimes in ways that open the door to serious vulnerabilities.

To investigate this, the authors rely on an active learning method first introduced by Angluin in the late 1980s. Known as the L* algorithm, it was originally designed to infer deterministic finite automata through systematic queries. Later work extended this method to Mealy machines, which are a better fit for communication protocols since they capture both the input messages and the corresponding outputs. This makes them particularly suitable for TLS, where each incoming message is expected to trigger a specific response.

The methodology treats a TLS stacks as a black box. The learner generates sequences of protocol messages, which a mapper then translates into concrete TLS packets. By sending these sequences and recording how the implementation responds, the learner gradually builds a hypothesis of the hidden state machine. Because the actual state machine is unknown, the usual equivalence queries of L* cannot be answered directly, so they are approximated using various techniques.

Armed with this approach, the authors analyzed more than 400 TLS stacks in both client and server roles. Their experiments reproduced known vulnerabilities, such as authentication bypasses in wolfSSL, and uncovered new ones ranging from further bypasses to denial-of-service loops in multiple implementations. Even when no exploitable flaw was present, the inferred machines showed a wide diversity of behaviors, far removed from the clean and linear "happy path" envisioned by the specification. These behavioral differences are not only a source of potential weaknesses but also a powerful fingerprinting mechanism. Each library and even each version of a library displays its own distinct state machine, which can be identified through carefully chosen message sequences.

4.1.2 Extending the Approach to SSH

Much like TLS, SSH implementations also diverge from one another in subtle but important ways. As a consequence, sending the wrong message at the right time can trigger interesting responses. For instance, CVE-2018-10933[4] demonstrates how, for certain libssh versions, presenting the server with an `USERAUTH_SUCCESS` message in place of the expected `USERAUTH_REQUEST` used to initiate authentication allowed an attacker to authenticate without any credentials.

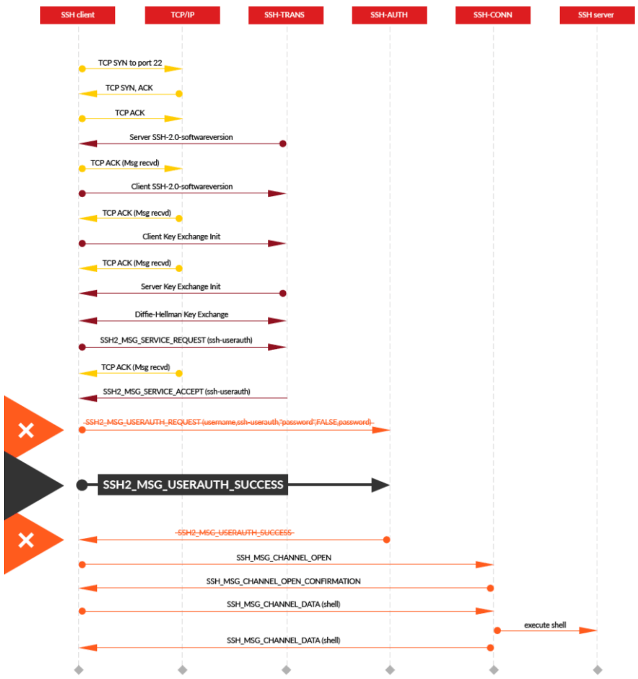


Figure 4.1: CVE-2018-10933[4].

As a result, applying state machine inference to SSH could be an interesting approach.

4.2 Crafting Payloads

4.2.1 SSH Binary Packet Protocol

As per specification[6], Each SSH packet is structured as follows:

```
uint32 packet_length
byte padding_length
byte[n1] payload // n1 = packet_length - padding_length - 1
byte[n2] random_padding // n2 = padding_length
byte[m] mac // Message Authentication Code, m = mac_length
```

`packet_length` specifies the length of the packet in bytes, excluding the `mac` and the `packet_length` field itself. `padding_length` indicates the length of the random padding in bytes. `payload` contains the actual packet contents and is compressed if compression has been negotiated (initially set to "none"). `random_padding` is arbitrary-length padding that brings the combined length of the previously listed packet fields up to a multiple of the cipher block size or 8 (whichever is larger). It must be at least 4 bytes, consist of random values, and can be up to 255 bytes. `mac` is the Message Authentication Code containing the MAC bytes if message authentication has been negotiated (also initially set to "none").

We implement this structure in the `wrap_ssh_packet()` function, as shown in Appendix.6.

Going forward, we will be crafting unencrypted packets and will be mostly focusing on the pre-DHINIT ones for simplicity's sake (i.e., before doing any actual cryptographic operations).

4.2.2 SSH_MSG_KEXINIT

As discussed in Subsection 3.2.2, the RFC 4253[6] specification defines the structure of the KEXINIT message. Using the reference JSON file and its comprehensive list of algorithms, we were able to construct a key exchange payload that should be compatible with nearly any SSH implementation, since it includes a wide range of algorithms, as implemented in the `kexinit_payload()` function shown in Appendix.6.

4.2.3 SS_MSG_DISCONNECT

As per specification[6], a disconnect message is structured as follows:

```
byte SSH_MSG_DISCONNECT (value 1)
uint32 reason code
string description in ISO-10646 UTF-8 encoding
string language tag
```

Reason codes 1–15 have predefined meanings, the range 0x00000010 to 0xFDFFFFFF must be assigned via the IETF consensus process, and 0xFE000000 through 0xFFFFFFFF are reserved for private use. To make our messages easily distinguishable, we use the private-use reason code 0xFEAAAAAA and set the description to "byebye", as implemented in the `disconnect_payload()` function in Appendix.6.

4.2.4 SSH_MSG_SERVICE_REQUEST

As per specification[6], a service request message is structured as follows:

```
byte SSH_MSG_SERVICE_REQUEST (value 5)
string service_name
```

The `service_name` identifies the requested service. Currently, the reserved names are `ssh-userauth` and `ssh-connection`. In our examples, we generate service request payloads using the `servicereq_payload()` function in Appendix.6. We thus created three service request payloads: one for `ssh-userauth`, one for `ssh-connection`, and one custom payload `"quelconque"` to observe how the stack reacts to an unrecognized service name.

4.2.5 SSH_MSG_UNIMPLEMENTED

As per specification[6], an implementation must respond to all unrecognized messages with an `SSH_MSG_UNIMPLEMENTED` message in the order the messages were received. Such messages must otherwise be ignored. Later protocol versions may assign additional meanings to these message types.

The message structure is as follows:

```
byte SSH_MSG_UNIMPLEMENTED (Value 3)
uint32 packet sequence number of rejected message
```

In our examples, we construct an `SSH_MSG_UNIMPLEMENTED` payload using the `unimplemented_payload()` function in Appendix.6, using a packet sequence number of 1. This allows us to simulate a response to an unrecognized message while maintaining compliance with the SSH specification.

4.2.6 SSH_MSG_NEWKEYS

As per specification[6], an `SSH_MSG_NEWKEYS` message is structured as follows:

```
byte SSH_MSG_NEWKEYS (Value 21)
```

This message is sent by each side to signal the end of the key exchange, its purpose is to ensure that a party can respond with a valid `SSH_MSG_DISCONNECT` if any issues occur during the key exchange.

In our examples, we construct this payload using the `newkeys_payload()` function in Appendix.6.

4.2.7 SSH_MSG_USERAUTH_SUCCESS

As per specification[7], a `SSH_MSG_USERAUTH_SUCCESS` message is structured as follows:

```
byte SSH_MSG_USERAUTH_SUCCESS (value 52)
```

This message is sent only when the authentication is fully complete and not after each step in a multi-method authentication sequence.

In our examples, we construct this payload using the `userauthsuccess_payload()` function in Appendix.6.

4.2.8 SSH_MSG_USERAUTH_FAILURE

As per specification[7], a `SSH_MSG_USERAUTH_FAILURE` message is structured as follows:

```
byte SSH_MSG_USERAUTH_FAILURE (value 51)
name-list authentications that can continue
boolean partial success
```

The `authentications that can continue` field is a comma-separated list of authentication method names that may still be attempted, while the `partial success` boolean indicates if any previous authentication steps were partially successful.

In our examples, we construct this payload using the `userauthfail_payload()` function in Appendix.6, with the authentication method set to `"password"` and `partial success` set to false.

4.2.9 SSH_MSG_USERAUTH_BANNER

As per specification[7], a `SSH_MSG_USERAUTH_BANNER` message is structured as follows:

```
byte SSH_MSG_USERAUTH_BANNER (value 53)
string message in ISO-10646 UTF-8 encoding
string language tag
```

The `message` field contains text intended to be displayed to the client user prior to authentication, while the `language tag` specifies the language of the message.

In our examples, we construct this payload using the `userauthbanner_payload()` function in Appendix.6, with the message set to `"Coucou."` and the language tag set to `"fr"`.

It is interesting to note that `SSH_MSG_USERAUTH_BANNER`, `SSH_MSG_USERAUTH_FAILURE`, and `SSH_MSG_USERAUTH_SUCCESS` are all messages typically sent by the server. In our experiments, however, we generate them from the client's perspective. As highlighted by CVE-2018-10933[4], sending these server-intended messages from a client can trigger unexpected or interesting behaviors in SSH implementations, revealing potential vulnerabilities or edge-case handling issues.

4.2.10 SSH_MSG_USERAUTH_REQUEST

As per specification[7], an SSH user authentication request message has the following structure:

```
byte SSH_MSG_USERAUTH_REQUEST
string user name in ISO-10646 UTF-8 encoding [RFC3629]
string service name in US-ASCII
string method name in US-ASCII
... method specific fields
```

In our experiments, we focus on three base authentication methods:

- **Public key authentication**

  ```
  byte SSH_MSG_USERAUTH_REQUEST
  string user name in ISO-10646 UTF-8 encoding [RFC3629]
  string service name in US-ASCII
  string "publickey"
  boolean FALSE
  string public key algorithm name
  string public key blob
  ```

  For our experiments, we captured this payload from a handshake using a WolfSSH client.

- **Password authentication**:

  ```
  byte SSH_MSG_USERAUTH_REQUEST
  string user name
  string service name
  string "password"
  boolean FALSE
  string plaintext password in ISO-10646 UTF-8 encoding [RFC3629]
  ```

  We also captured this payload from a handshake using a WolfSSH client.

- **None authentication**:

  We construct a "none" method user authentication request payload using the `userauthrequest_payload()` function in Appendix.6.

4.3 Experimental Results

All in all, we have ended up with 13 distinct payloads to work with. The next step is to send these payloads toward various SSH stacks in the wild to observe their behavior and responses.

The idea here is a simplified version of the TLS inference learner: we send one sequence after another toward a given stack and track its responses in the form of a tree, which serves as a visual representation of the stack's state transitions, as well as a hash that will serve as our fingerprint. The start node corresponds to the stack's state after exchanging identification strings and receiving the server's banner. From this node, our client decides what to send next, independent of the SSH "happy path."

Each node of this tree represents a new state. We keep track of the transitions as well as the ancestry that led to each state so that states do not overlap with each other, except in the final layer, where different message paths originating from the same node are allowed to converge to the same response.

Our system predefines a number of sequences based on the number of layers we want in our final tree, formed by taking the Cartesian product of the 13 possible messages. Thus, the total number of sequences required is 13^n where n is the depth of the tree.

The internship being still in progress, the results of this part are not yet finalized. However, preliminary observations with $n = 2$ indicate that the sequences produce distinctly different behaviors between a WolfSSH stack and an OpenSSH stack.

```
Hash of the result for openssh-v_10_0_p2 is 93
    ↪ f67978386703a3092e4772430503b9aff4d4b5cd99f1d42f1b4e2029abf360

Hash of the result for wolfssh-v1.4.13-stable is
    ↪ bceabbc0a44474f2d0df9bc05e2a2a2a892843d8aecf336572fe81cc348bb1f6
```

The resulting state machine trees are not included in this report because they are far too large, they have instead been stored in the SSH test bed repository[1].

For now, most of the work left is to continue testing this tool with various SSH stacks and noting the results before drawing any conclusions.

Some design choices might require improvement. The sheer size of the resulting tree makes it difficult to read and to process in most formats, which has forced me to opt for a PDF representation. It might also be worthwhile to explore some optimization to handle multiple sequences simultaneously due to the large number of sequences that need to be processed. For example, in the experiments conducted so far, even building trees of depth 2 generates 169 sequences takes a significant amount of time (approximately 5 minutes 30 seconds to process a single stack). This number will only increase as we attempt to build trees of greater depth. It would also be interesting to include a learner to automate the inference of stack behaviors and potentially optimize the tree construction process..

Further results will be added as addendums as the internship progresses.

Conclusion

To conclude, this internship was an invaluable opportunity to gain a comprehensive understanding of SSH and the intricacies of fingerprinting tools in the context of cybersecurity. I developed practical skills in deploying and managing 180 Docker containers, each running different SSH stacks, which allowed me to observe and manipulate their behavior in a controlled environment. I mastered using scripting languages to automate tasks and analyze data efficiently, and acquired a deep understanding of the SSH protocol, including the structure and flow of its message packets.

I also learned to manipulate system configurations, perform packet analysis using tools such as Wireshark, Zeek, and tcpdump, and evaluate the quality of existing fingerprinting tools. Additionally, I made initial steps toward developing my own behavior-based fingerprinting approaches. Although creating a fully robust fingerprinting platform remains a long-term challenge, this experience provided me with significant insights into both theoretical and practical aspects of SSH security and fingerprinting research, strengthening my skills in cybersecurity.

This internship has not yet concluded, and addendums to this report are to be expected as further results and developments emerge.

Bibliography

[1] MedMzi. MedMzi/ssh-test-bed. https://github.com/MedMzi/ssh-test-bed (GitHub). original-date: 2025-06-12T23:10:48Z.

[2] corelight/hassh. https://github.com/corelight/hassh (GitHub), April 2025. original-date: 2024-01-29T02:08:34Z.

[3] Ben Reardon. Open Sourcing HASSH, September 2018.

[4] LIBSSH Auth Bypass (CVE-2018-10933).

[5] Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Protocol Architecture. RFC 4251, January 2006.

[6] Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, January 2006.

[7] Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Authentication Protocol. RFC 4252, January 2006.

[8] Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Connection Protocol. RFC 4254, January 2006.

[9] What is Docker?, September 2024.

[10] Aina Toky Rasoamanana, Olivier Levillain, and Hervé Debar. Towards a systematic and automatic use of state machine inference to uncover security flaws and fingerprint TLS stacks. In *ESORICS 2022 : 27th European Symposium on Research in Computer Security*, volume 13556 of *Lecture Notes in Computer Science*, pages 637–657, Copenhague, France, September 2022. Springer Nature Switzerland.

[11] Mrinal Prakash. Enumerating SSH with Nmap, April 2023.

[12] Secure Shell (SSH) Protocol Parameters. https://www.iana.org/assignments/ssh-parameters/ssh-parameters.xhtml.

[13] Expect Script SSH Example Tutorial | DigitalOcean.

Appendix

Appendix.1 Dockerfile Template Example for OpenSSH V_10_0_P2 to V_7_9_P1[1]

```
FROM debian:buster AS builder

# Set environment variable to disable interactive prompts
ENV DEBIAN_FRONTEND=noninteractive

# Install dependencies for building OpenSSH
RUN apt-get update && apt-get install -y --no-install-recommends \
    git build-essential zlib1g-dev libssl-dev libpam0g-dev libselinux1-
        ↪ dev ca-certificates \
    autoconf automake && \
    apt-get clean

# Set working directory
WORKDIR /openssh

# Build argument for OpenSSH version (commit or tag)
ARG COMMIT=V_10_0_P2

# Clone OpenSSH repository and checkout the specified version
RUN git clone https://github.com/openssh/openssh-portable --depth=1 -b
    ↪ ${COMMIT} . && \
    autoreconf -i

# Configure and build OpenSSH
RUN ./configure && make && make install

# Minimal runtime environment
FROM debian:buster

# Install runtime dependencies
RUN apt-get update && apt-get install -y --no-install-recommends \
    libssl-dev libpam0g libselinux1 binutils && \
    apt-get clean
```

```
# Fix privilege separation issue
RUN useradd -r -d /var/empty -s /usr/sbin/nologin sshd && \
    mkdir -p /var/empty

# Copy OpenSSH binaries from the builder stage
COPY --from=builder /usr/local /usr/local

# Create SSH host keys
RUN mkdir -p /etc/ssh && ssh-keygen -A

# Create a test user
RUN useradd -m -s /bin/bash testuser && \
    echo "testuser:root" | chpasswd

# Expose SSH port
EXPOSE 22

# Start OpenSSH server in foreground mode
CMD ["/usr/local/sbin/sshd", "-D"]
```

Appendix.2 Docker Automation Script for OpenSSH V_10_0_P2 to V_7_9_P1[1]

```
#!/bin/bash

echo "Please␣enter␣your␣sudo␣password␣to␣begin␣(necessary␣for␣capture)
    ↪ ..."
sudo -v

# List of COMMIT versions
COMMIT_FILE="commits.txt"
mapfile -t COMMITS < $COMMIT_FILE

#COMMITS=(
# "V_10_0_P2"
# "V_10_0_P1"
#)

# SSH login details
HOST="127.0.0.1"
SSH_USER="testuser"
SSH_PASSWORD="root"

# Output file for results
```

```
RESULT_FILE="results.txt"
echo "OpenSSH␣Versions:" > $RESULT_FILE

# Loop through each COMMIT version
for COMMIT in "${COMMITS[@]}"; do
    echo "Processing␣COMMIT:␣$COMMIT"

    # Convert COMMIT to lowercase (avoid errors)
    COMMIT_LOWER=$(echo "$COMMIT" | tr '[:upper:]' '[:lower:]')

    # Build the Docker image with the current COMMIT passed as a build argument
    echo "Building␣Docker␣image␣for␣$COMMIT..."
    docker build --build-arg COMMIT=$COMMIT -t openssh-$COMMIT_LOWER .

    # Remove old host key to avoid issues
    echo "Removing␣old␣host␣key␣for␣$HOST:2222..."
    ssh-keygen -f ~/.ssh/known_hosts -R "[${HOST}]:2222"

    # Run tcpdump in background
    PCAP_FILE="capture/${COMMIT_LOWER}.pcap"
    echo "Starting␣tcpdump,␣saving␣to␣$PCAP_FILE..."
    sudo tcpdump -i lo port 2222 -w "$PCAP_FILE" &
    TCPDUMP_PID=$!

    # Run the container
    echo "Running␣container␣for␣$COMMIT..."
    CONTAINER_ID=$(docker run --rm -d -p 2222:22 openssh-$COMMIT_LOWER)

    # Call login.sh and capture the output
    echo "Calling␣login.sh␣for␣$COMMIT..."
    OUTPUT=$(./login.sh $HOST $SSH_USER $SSH_PASSWORD)

    # Extract the last line of the output
    LAST_LINE=$(echo "$OUTPUT" | tail -n 1)

    # Save the result to the file
    echo "$COMMIT:␣$LAST_LINE" >> $RESULT_FILE

    # Kill the container
    echo "Stopping␣container␣for␣$COMMIT..."
    docker kill $CONTAINER_ID

    # Stop tcpdump
    echo "Stopping␣tcpdump..."
    sleep 2
    sudo kill -SIGINT $TCPDUMP_PID
    wait $TCPDUMP_PID 2>/dev/null
```

```
    echo "Finished␣processing␣$COMMIT."
    echo "----------------------------------"
done

echo "All␣COMMITs␣processed!␣Results␣saved␣to␣$RESULT_FILE."
```

Appendix.3 Expect Script for for OpenSSH `V_10_0_P2` to `V_7_9_P1`[13]

```
#!/usr/bin/expect

#Usage sshsudologin.expect <host> <ssh user> <ssh password> <su user> <
    ↪ su password>

set timeout 60

spawn ssh -p2222 [lindex $argv 1]@[lindex $argv 0] ssh -V

expect "yes/no" {
       send "yes\r"
       expect "*?assword" { send "[lindex␣$argv␣2]\r" }
       } "*?assword" { send "[lindex␣$argv␣2]\r" }

#expect "# " { send "su - [lindex $argv 3]\r" }
#expect ": " { send "[lindex $argv 4]\r" }
#expect "# " { send "ls -ltr\r" }
interact
```

Appendix.4 Identification String Test Script

```
import socket
import time

banner = "SSH-2.0-OpenSSH_10.0\r\n"

tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
tcp_socket.connect(("localhost", 2222))

print(f"[Client]␣Sending␣banner:␣{banner.strip()}")
tcp_socket.sendall(banner.encode())
server_banner = tcp_socket.recv(4096).decode(errors='ignore').strip()
```

```
print(f"[Client]␣Received␣server␣banner:␣{server_banner}")

time.sleep(1)

tcp_socket.close()
print("[Client]␣Connection␣closed.")
```

Appendix.5 Spoofing HASSHServer Fingerprint Script

```
import socket
import time

port = 2222
banner = "SSH-2.0-FakeSSH\r\n"
with open("kexinit_payload.bin", "rb") as f:
    kexinit = f.read()


tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

server_address = ("0.0.0.0", port)
tcp_socket.bind(server_address)
tcp_socket.listen(1)


connection, client = tcp_socket.accept()

print(f"[+]␣Connection␣from␣{client[0]}:{client[1]}")


server_banner = connection.recv(4096).decode(errors='ignore').strip()
print(f"[Client]␣Received␣client␣banner:␣{server_banner}")

print(f"[Client]␣Sending␣banner:␣{banner.strip()}")
connection.sendall(banner.encode())

connection.send(kexinit)
print(f"[Client]␣sending␣KEXINIT␣payload␣({len(kexinit)}␣bytes)")
data = connection.recv(1024)
print("[+]␣KEXINIT␣received␣and␣sent")

data = connection.recv(2048)
print("[+]␣DHINIT␣received")

time.sleep(1)
```

```
connection.close()
```

Appendix.6 Python Script for Crafting Custom SSH Protocol Packets

```
import json
import os
import struct
import random

def encode_ssh_string(s):
        encoded = s.encode()
        return struct.pack(">I", len(encoded)) + encoded

def name_list_to_bytes(name_list):
    # name-list is length(uint32) + comma-separated string bytes
    s = ",".join(name_list)
    return encode_ssh_string(s)

def wrap_ssh_packet(payload: bytes, block_size: int = 8) -> bytes:
    unpadded_len = len(payload) + 5
    padding_length = (block_size - (unpadded_len % block_size)) %
        ↪ block_size
    if padding_length < 4:
        padding_length += block_size

    packet_length = len(payload) + padding_length + 1 # +1 for
        ↪ padding_length field

    # DEBUG INFO
    print("[DEBUG]␣---␣Packet␣Construction␣---")
    print(f"Payload␣length:␣␣␣␣␣␣␣{len(payload)}")
    print(f"Unpadded␣length␣(+1):␣{unpadded_len}")
    print(f"Block␣size:␣␣␣␣␣␣␣␣␣␣␣{block_size}")
    print(f"Padding␣length:␣␣␣␣␣␣␣{padding_length}")
    print(f"Total␣packet␣length:␣␣{packet_length}")
    print(f"Total␣full␣length␣(with␣4-byte␣prefix):␣{packet_length␣+␣4}"
        ↪ )
    print("----------------------")

    packet = struct.pack(">I", packet_length)
    packet += struct.pack("B", padding_length)
    packet += payload
    packet += os.urandom(padding_length)

    # Print hex of final output
```

```
    print("[DEBUG]␣Final␣packet␣(hex):")
    print(packet.hex())

    return packet


def kexinit_payload():
    cookie = os.urandom(16)
    msg_id = 20 # SSH_MSG_KEXINIT

    with open("Reference.json", "r") as f:
        ref = json.load(f)
    compression_algorithms = ["none", "zlib@openssh.com", "zlib"]

    payload = struct.pack("B", msg_id) # message id
    payload += cookie

    payload += name_list_to_bytes(ref["KexAlgorithms"])
    payload += name_list_to_bytes(ref["HostKeyAlgorithms"])
    payload += name_list_to_bytes(ref["Ciphers"]) # client to server
        ↪ encryption algos
    payload += name_list_to_bytes(ref["Ciphers"]) # server to client
        ↪ encryption algos (using same list)
    payload += name_list_to_bytes(ref["MACs"]) # client to server MAC
        ↪ algos
    payload += name_list_to_bytes(ref["MACs"]) # server to client MAC
        ↪ algos
    payload += name_list_to_bytes(compression_algorithms) # client to
        ↪ server compression
    payload += name_list_to_bytes(compression_algorithms) # server to
        ↪ client compression
    payload += name_list_to_bytes([]) # languages client to server (
        ↪ empty)
    payload += name_list_to_bytes([]) # languages server to client (
        ↪ empty)

    payload += struct.pack("B", 0) # first_kex_packet_follows = False
    payload += struct.pack(">I", 0) # reserved uint32 = 0

    payload = wrap_ssh_packet(payload)

    with open("kexinit_payload.bin", "wb") as f:
        f.write(payload)

    print("kexinit_payload.bin␣created.")

def disconnect_payload(DESCRIPTION="byebye", LANGUAGE_TAG="en"):
    msg_id = 1 # SSH_MSG_DISCONNECT
```

```
    REASON_CODE = 0xFEAAAAAA # Private-use code

    # Build the payload
    payload = bytearray()
    payload.append(msg_id)
    payload += struct.pack(">I", REASON_CODE)
    payload += encode_ssh_string(DESCRIPTION)
    payload += encode_ssh_string(LANGUAGE_TAG)

    payload = wrap_ssh_packet(payload)

    # Write to file
    with open("disconnect_payload.bin", "wb") as f:
        f.write(payload)

    print("disconnect_payload.bin␣created.")

def servicereq_payload(service_name):
    msg_id = 5 # SSH_MSG_SERVICE_REQUEST


    payload = bytearray()
    payload.append(msg_id)
    payload += encode_ssh_string(service_name)

    payload = wrap_ssh_packet(payload)

    with open(f"{service_name}_payload.bin", "wb") as f:
        f.write(payload)

    print(f"{service_name}_payload.bin␣created.")

def unimplemented_payload():
    msg_id = 3 # SSH_MSG_UNIMPLEMENTED
    payload = bytearray()
    payload.append(msg_id)
    payload += struct.pack(">I", 1)

    payload = wrap_ssh_packet(payload)

    with open("unimplemented_payload.bin", "wb") as f:
        f.write(payload)

    print("unimplemented_payload.bin␣created.")

def newkeys_payload():
    msg_id = 21 # SSH_MSG_NEWKEYS
    payload = bytearray()
    payload.append(msg_id)
```

```
    payload = wrap_ssh_packet(payload)

    with open("newkeys_payload.bin", "wb") as f:
        f.write(payload)

    print("newkeys_payload.bin␣created.")

def userauthsuccess_payload():
    msg_id = 52 # SSH_MSG_USERAUTH_SUCCESS
    payload = bytearray()
    payload.append(msg_id)

    payload = wrap_ssh_packet(payload)

    with open("userauthsuccess_payload.bin", "wb") as f:
        f.write(payload)

    print("userauthsuccess_payload.bin␣created.")

def userauthfail_payload():
    msg_id = 51 # SSH_MSG_USERAUTH_FAILURE
    payload = bytearray()
    payload.append(msg_id)
    payload += encode_ssh_string("password")
    payload += struct.pack("B", 0) # partial success

    payload = wrap_ssh_packet(payload)

    with open("userauthfail_payload.bin", "wb") as f:
        f.write(payload)

    print("userauthfail_payload.bin␣created.")

def userauthbanner_payload(message="Coucou.", language_tag="fr"):
    msg_id = 53 # SSH_MSG_USERAUTH_BANNER
    payload = bytearray()
    payload.append(msg_id)
    payload += encode_ssh_string(message) # banner message
    payload += encode_ssh_string(language_tag) # language tag

    payload = wrap_ssh_packet(payload)

    with open("userauthbanner_payload.bin", "wb") as f:
        f.write(payload)

    print("userauthbanner_payload.bin␣created.")

def userauthrequest_payload():
```

```
    msg_id = 50 # SSH_MSG_USERAUTH_REQUEST
    username = "testuser"

    payload = bytearray()
    payload.append(msg_id)

    payload += encode_ssh_string(username)
    payload += encode_ssh_string("ssh-connection")
    payload += encode_ssh_string("none")

    payload = wrap_ssh_packet(payload)

    with open("userauthrequest_none_payload.bin", "wb") as f:
        f.write(payload)

    print("userauthrequest_none_payload.bin␣created.")


if __name__ == "__main__":
    #kexinit_payload()
    #disconnect_payload()
    #servicereq_payload("ssh-userauth")
    #servicereq_payload("ssh-connection")
    #servicereq_payload("servicereqQuelconque")
    #unimplemented_payload()
    #newkeys_payload()
    #userauthsuccess_payload()
    #userauthfail_payload()
    #userauthbanner_payload()
    userauthrequest_payload()
```