**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF COMPUTER SYSTEMS**
**ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ**


# FINGERPRINTING AND IDENTIFICATION OF TLS CONNECTIONS

**ROZPOZNÁVÁNÍ A IDENTIFIKACE TLS SPOJENÍ**

**BACHELOR'S THESIS**
**BAKALÁŘSKÁ PRÁCE**


**AUTHOR** **LUKÁŠ HEJCMAN**
**AUTOR PRÁCE**

**SUPERVISOR** **Ing. LUKÁŠ KEKELY, Ph.D.**
**VEDOUCÍ PRÁCE**



**BRNO 2021**

**Brno University of Technology**
Faculty of Information Technology

Department of Computer Systems (DCSY) Academic year 2020/2021

# Bachelor's Thesis Specification

Student: **Hejcman Lukáš**
Programme: Information Technology
Title: **Fingerprinting and Identification of TLS Connections**
Category: Networking

Assignment:

1. Prostudujte protokol TLS a zaměřte se na informace posílané v tzv. hello zprávách posílaných na začátku spojení.
2. Seznamte se s veřejně dostupnými nástroji a databázemi, které se zabývají analýzou TLS pro identifikaci zařízení a komunikujících aplikací (např. Mercury, JA3 DB) a s open source exportérem síťových toků ipfixprobe.
3. Navrhněte prototyp softwarového modulu, který na základě informací o TLS spojení odhadne pravděpodobný typ zařízení a komunikující aplikaci.
4. Vytvořte navržený prototyp jako modul do open source systému NEMEA.
5. Experimentálně otestujte vytvořený modul pomocí vlastních datových sad a následně i pomocí dat z reálného provozu, která dodá vedoucí práce.
6. Zhodnoťte dosažené výsledky a diskutujte možnosti pokračování projektu.

Recommended literature:

- Dle pokynů vedoucího.

Requirements for the first semester:

- Splnění bodů 1 až 3 zadání.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Kekely Lukáš, Ing., Ph.D.**
Head of Department: Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work: November 1, 2020
Submission deadline: May 12, 2021
Approval date: October 30, 2020

# Abstract

TLS is the most popular encryption protocol used on the internet today. It aims to provide high levels of security and privacy for inter-device communication. However, it presents a challenge from a network monitoring and administration standpoint, as it is not possible to analyse the communication encrypted with TLS at a large scale with existing methods based on deep packet inspection. Analysing encrypted communication can help administrators to detect malicious activity on their networks, and can help them identify potential security threats. In this work, I present a method that allows us to leverage the advantages of two TLS fingerprinting methods, JA3 and Cisco Mercury, to determine the operating system and processes of clients on a computer network. The proposed method is able to achieve comparable or better results than the existing Mercury approach for selected datasets whilst providing more analysis opportunities than JA3. A software implementation of the proposed fingerprinting approach is created as an analysis module for the NEMEA framework.



# Abstrakt

TLS je dnes nejpopulárnější šifrovací protokol používaný na internetu. Jeho cílem je poskytnout vysokou úroveň zabezpečení a soukromí pro komunikaci mezi zařízeními. Představuje však výzvu z hlediska monitorování a správy sítí, protože není možné analyzovat komunikaci šifrovanou pomocí tohoto protokolu ve velkém měřítku, pomocí existujících metod založených na detailní analýze obsahu paketů. Analýza šifrované komunikace může správcům pomoci detekovat škodlivou aktivitu v jejich sítích a také jim může pomoci identifikovat potenciální bezpečnostní hrozby. V této práci představuji metodu, která nám umožňuje využít výhod dvou metod otisků TLS, JA3 a Cisco Mercury, k určení operačního systému a procesů klientů v počítačové síti. Navržená metoda je schopna dosáhnout srovnatelných nebo lepších výsledků v porovnání se stávajícím přístupem Cisco Mercury pro vybrané datové sady a zároveň poskytuje možnosti pro detailnější analýzy klasifikací než JA3. V rámci práce je dále implementován modul pro systém NEMEA, který je schopný analyzovat TLS provoz pomocí nově navrženého přístupu.



# Keywords

TLS, Fingerprint, JA3, Cisco, Mercury, NEMEA, Process, Operating, System, Classification, Module

# Klíčová slova

TLS, Otisk, JA3, Cisco, Mercury, NEMEA, Proces, Operační, Systém, Klasifikace, Modul


# Reference


HEJCMAN, Lukáš. *Fingerprinting and Identification of TLS Connections.* Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Lukáš Kekely, Ph.D.

# Rozšířený abstrakt

Identifikace a analýza síťového provozu je nezbytnou součástí systémů zajišťujících síťovou bezpečnost. Nicméně, tyto systémy jsou většinou založeny na analýze obsahu paketů, která je výpočetně velice náročná. Tento přístup je dále znemožněn použitím bezpečnostních protokolů, jako například TLS, které šifrují obsah komunikace mezi klientem a serverem. Tyto protokoly jsou ale využívány i škodlivým síťovým provozem, je tedy nezbytné správně identifikovat bezpečný a škodlivý provoz a klienty, kteří tento provoz vytvořili.

Z tohoto důvodu vznikl přístup, který využívá nešifrovaných zpráv na začátku TLS komunikace pro vytvoření *otisků* této komunikace. Otisky jsou vytvořeny z parametrů TLS komunikace, které klient nabízí serveru při jejich spojení. Různé kombinace procesů, operačních systémů a kryptografických knihoven vytváří různé kombinace těchto parametrů. Je tedy možné extrapolovat informace o klientovi na základě vytvořeného otisku a databáze otisků různých klientů.

Pro tuto identifikaci jsou používány existující přístupy a databáze otisků. Nejznámější a nejpoužívanější přístup se jmenuje JA3. Otisky vytvořené s tímto přístupem mají konstantní délku a jsou velice rychlé na zpracování. Hlavní problém s algoritmem JA3 jsou volně přístupné databáze otisků, které neobsahují dostatečné množství informací a nejsou strukturované tak, aby umožnily další analýzu výsledků identifikace klientů.

Alternativní metoda pro analýzu otisků TLS provozu, Mercury, byla vytvořena firmou Cisco. Tato metoda umožňuje mnohem detailnější identifikaci klientů, protože otisky generované touto metodou obsahují větší množství informací o paketu, ze kterého pochází. Hlavní výhodou tohoto přístupu je veřejně dostupná databáze, která byla vytvořena sjednocením informací o klientech, jejich procesech a otisků, které tito klienti vytvořili. Bohužel, tato metoda není tak rozšířená jako JA3, protože je mnohem složitější na implementaci a na zpracování ve vysokorychlostních sítích.

Proto jsem vytvořil metodu JA3cury, která kombinuje tyto dva přístupy k vytváření otisků a identifikaci síťových klientů. Tato metoda využívá otisky v JA3 formátu a konvertované databáze založené na Mercury. To umožňuje naší metodě využití v existujících sítích, kde JA3 převládá. JA3cury zároveň otevírá dveře pro nový způsob detailní analýzy klasifikačních výsledků, umožněnou Mercury databází.

Navržená metoda dosahuje stejných nebo vyšších výsledků přesnoti klasifikace procesů a operačních systémů klientů v porovnání s metodou Mercury, na které je založena. Kratší délka JA3 otisků, která obsahuje menší množství identifikačních informací než Mercury otisky, vedla k větší přesnosti identifikace z důvodů větších kolizí otisků v databázi. Využití JA3 otisků rovněž vedlo k zvýšení přesnoti identifikace procesů na síti, která obsahovala jiné poměry procesů, než jsou v Mercury databázi.

Navržený algoritmus je dále implementován jako modul pro systém NEMEA. Tento systém je využíván pro měření a analýzu síťového provozu. Vytvořený modul implementuje několik rozdílných klasifikačních algoritmů a umožňuje analýzu TLS provozu na existujících systémech s minimální časovou investicí administrátorů.

# Fingerprinting and Identification of TLS Connections

## Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Lukáš Kekely Ph.D. The supplementary information was provided by Ing. Tomáš Čejka Ph.D. and Ing. Karel Hynek. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Lukáš Hejcman
May 10, 2021


## Acknowledgements

I would like to thank my supervisor Lukáš Kekely for his feedback. Furthermore, I am grateful to Tomáš Čejka and Karel Hynek for their support and advice. Lastly, I would like to extend my thanks to Alena Žárská, Andrej Lukačovič, David Beneš, and Zdena Tropková for supplying me with captured network traffic for my experiments.

# Contents


**1 Introduction** **3**

**2 Security protocols and TLS** **5**
2.1 Cryptography . . . . . 5
2.2 Security protocols . . . . . 7
2.3 The Transport Layer Protocol . . . . . 8

**3 Fingerprinting and TLS** **13**
3.1 Fingerprints and fingerprinting . . . . . 13
3.2 JA3 . . . . . 14
3.3 Mercury . . . . . 16

**4 JA3cury and its classification algorithms** **21**
4.1 The JA3cury approach . . . . . 21
4.1.1 Process name normalisation . . . . . 22
4.2 Classification algorithms . . . . . 25
4.2.1 Operating system classification . . . . . 25
4.2.2 Process and category classification . . . . . 26

**5 NEMEA module implementation** **31**
5.1 NEMEA . . . . . 31
5.2 The JA3cury experimental module . . . . . 32
5.3 The JA3cury production module . . . . . 35
5.4 Future work . . . . . 39

**6 Classification results overview** **41**
6.1 Datasets . . . . . 41
6.2 Process detection . . . . . 43
6.3 Category detection . . . . . 44
6.4 Operating system detection . . . . . 45
6.5 Conclusion regarding detection algorithms . . . . . 46

**7 Conclusion** **47**

**Bibliography** **48**

**A JA3cury usage example** **52**

# List of Figures


2.1 Symmetric and asymmetric encryption schemes. . . . . 7
2.2 The TLS stack. . . . . 8
2.3 The structure of a TLS Record. . . . . 9
2.4 An overview of a TLS Handshake. . . . . 10
2.5 Structure of a TLS Client Hello. . . . . 11

3.1 A graphical illustration of fingerprint generation. . . . . 13
3.2 A common fingerprinting scheme. . . . . 14
3.3 The suggested format of a JA3 database entry. . . . . 16
3.4 Cisco Mercury network and endpoint fusion. . . . . 17
3.5 The Mercury database structure. . . . . 19

4.1 Mercury to JA3 fingerprint conversion. . . . . 22
4.2 Comparison of fingerprint lengths across formats. . . . . 23
4.3 An example of the operating system tree. . . . . 25
4.4 New dictionary for a client. . . . . 28
4.5 Example dictionary for a client after classification. . . . . 28

5.1 A basic overview of the NEMEA system structure. . . . . 32
5.2 The structure of the experimental JA3cury module. . . . . 34
5.3 An overview of the production JA3cury module. . . . . 36
5.4 Visualisation of the module concurrency. . . . . 40

6.1 Detection results for all processes. . . . . 43
6.2 Detection results for the top 5 processes. . . . . 44
6.3 Detection results for all categories. . . . . 44
6.4 Detection results for the top 5 categories. . . . . 45
6.5 Detection results for the operating systems. . . . . 46

A.1 Example directory structure. . . . . 52

# Chapter 1

# Introduction

Since the dawn of time, humans have developed methods to allow themselves to communicate with each other without having to worry about eavesdroppers intercepting or deciphering their communication. One of the oldest examples of encryption was found in the tomb of Khnumhotep II, who ruled ancient Egypt in around 1900 B.C. [13] The encryption was based on symbol replacement, and required a cipher key to decrypt. Since the ancient times, encryption has steadily developed to the point where directly decrypting the message is not necessary to infer the contents of the communication.

For example, before the first world war, the French military intelligence built a network of radio stations on the western front intended for intercepting radio traffic, based on the work of a Dutch cryptanalyst Auguste Kerckhoff [28]. Subsequently, when the Germans crossed the frontier during the first world war, the French intelligence was able to intercept German radio traffic and devise an algorithm which used the signal intensity, call-signs, volume of traffic, among other factors to identify German military units and distinguish between fast moving cavalry and slow moving armoured units. This allowed them to gain a tactical advantage over their enemy in battle, and proactively react to new threats. This approach can be applied to our modern communication standards as well.

Modern computer networks are a tangle of different devices, protocols, and applications, all trying to communicate with each other. Naturally, these networks also contain a large amount of malicious clients and traffic. This malicious traffic can be responsible for data breaches, denial of service attacks, degradation in the quality of service, and other security vulnerabilities. In 2018 the US Council of Economic Advisers estimated the total cost of malicious cyber activity to the US economy at between $57 billion and $106 billion [42].

Furthermore, according to the Google Transparency Report, the amount of encrypted traffic (both malicious and benign) on the web has increased from around 48% in 2014 to 95% in 2020 [26]. Thus, to adequately analyse traffic passing through modern computer networks, we must tackle the issue of analysing encrypted traffic at a high enough throughput. Whilst it is possible to use deep packet inspection and brute force methods to decrypt encrypted communication and analyse its contents, this approach introduces many problems. Even if we ignore its impact on network throughput and the required computational performance, network traffic decryption presents an ethical challenge, as it is a violation of user privacy and security.

This provides the basis for new approaches to network monitoring and traffic analysis. Much like the French military intelligence during the first world war, network administrators today set up specialised software and hardware on their networks to capture, analyse, and observe long term trends and patterns in the communication across their network. This

long term data is then used to discover unusual traffic on the network, and react to a possible new threat or a security vulnerability.

One of the methods for analysing network traffic is fingerprinting. Using this method, a set of notable information is extracted from selected network packets and compared against a database containing possible identifications of the user or process that generated the packet. This approach is favoured in high throughput networks, because it is much less computationally intensive than deep packet inspection [38] or decrypting encrypted network traffic, whilst providing accurate results [14]. Furthermore, fingerprinting is more respecting of user privacy and security than comparable methods, such as deep packet inspection which accesses the contents of the communication.

Fingerprinting is an area of rapid development. In general, fingerprinting schemes depend on specific network protocols, such as TLS, which are also constantly developing. This means that fingerprinting schemes must be updated regularly to keep up with the developments of network standards, so they can adequately represent the real state of network traffic. Moreover, the updates to cryptographic and operating system libraries can also have some impact on the fingerprintability of network traffic, and it is thus important to keep our fingerprinting approaches up to date with current developments.

The purpose of this work is to develop an algorithm which is capable of predicting the operating systems and the processes generating encrypted traffic on the network using a fingerprinting approach. This approach is based on existing fingerprinting schemes; Cisco Mercury and JA3. The proposed method, JA3cury combines the convenience and widespread adoption of JA3 with the analysis possibilities presented by Cisco Mercury.

The theoretical basis of TLS communication is presented in chapter 2. Comparable algorithms are discussed in chapter 3, with a novel solution presented in chapter 4. Next, the implementation of two modules for the Network Measurement and Analysis (NEMEA) system is described in chapter 5, which shows the working of the proposed approach in practice. The results of these modules using the proposed approach are discussed in detail in chapter 6. Lastly, the thesis is concluded in chapter 7.

# Chapter 2

# Security protocols and TLS

## 2.1 Cryptography

Cryptography is the study of how to alter a message so that someone intercepting it cannot read it without the appropriate algorithm and key [24]. This is done by converting a freely readable piece of data or text called plaintext into ciphertext, known as the process of encryption. This ciphertext is in a format that cannot be interpreted without decrypting its contents. A general cryptographic scheme can be defined by the following pair of formulas:

$$C = E(P, k_\alpha) \tag{2.1}$$

$$P = D(C, k_\beta) \tag{2.2}$$

Here, $C$ is the cipher-text, $P$ is the plain-text, $E$ is the encryption algorithm, $D$ is the decryption algorithm, and $k_\alpha$ and $k_\beta$ are the private and public keys respectively. It is possible that $k_\alpha = k_\beta$, which is the case in symmetric encryption. Kessler defined the following five goals for encryption [24]:

1. **Privacy/confidentiality**
   The privacy/confidentiality goal aims to ensure that only the target recipients are able to read the contents of the message.

2. **Authentication**
   Authentication aims to provide a mechanism to the communicating parties which they can use to ensure the identity of the other communicating parties.

3. **Integrity**
   Ensuring the integrity of a message proves that it was not altered in transit, and that its contents are identical to the message which was originally sent.

4. **Non-repudiation**
   Non-repudiation refers to the mechanism by which one can prove that the received message originated with the advertised sender.

5. **Key exchange**
   A method bych which two parties exchange cryptographic keys.

Based on the overall approach to message encryption, cryptography can be split into three main types; symmetric encryption, asymmetric encryption, and hashing. A graphical overview of symmetric and asymmetric encryption based on the formulas 2.1 and 2.2 can be seen in figure 2.1. These types of encryption define only the overall concept regarding the process of encryption; each of them can be further split into many different encryption algorithms. However, for the purpose of this work, it is sufficient to explain the high level overview of approaches to encryption in modern computing.

### Symmetric encryption

Symmetric encryption has been the predominant cryptographic method for most of history. It is sometimes also referred to as private-key cryptography, because it utilises one key for both encrypting the plaintext and deciphering the ciphertext. It is thus important to keep this key private, hence the name. Using the formulas we defined before, symmetric encryption can be characterised by the relationship $k_\alpha = k_\beta$.

The main disadvantage with symmetric encryption is the management of private keys. Primarily, both the sender and the receiver must have the same key to communicate using symmetric encryption. In most cases, encryption is used to communicate over an insecure channel, which cannot be used for sharing the encryption key, as this would compromise all further communication. Furthermore, when a user communicates with two or more distinct parties, each communication should use a separate private key for encryption. However, this means that the user must keep track of at least one private key for each communication, which can be highly demanding, and exposes a large attack target.

However, symmetric encryption presents a major advantage; it is over 1000 times faster than comparable asymmetric encryption algorithms due to its larger mathematical simplicity [43]. This results in lower processing requirements, lower latency computation, and lower power consumption. This is especially important for high throughput networks, low power embedded devices, and streaming services [34].

### Asymmetric encryption

Asymmetric encryption, also called public key cryptography, differs from symmetric encryption by using two separate keys for encryption and decryption; called the public and private keys respectively. Asymmetric encryption has been called the largest leap in cryptography in the last 400 years [24]. It was first introduced by Whitfield Diffie and Martin Hellman as the Diffie-Hellman key exchange in November of 1976 [20].

The main advantage of asymmetric encryption is that it solves the problem of sharing encryption keys because each client in the communication has a public and private key. The public key can be be freely shared, as it allows only encrypting the messages. This means that a secure encrypted communication channel can be established over an unsecure medium without having a risky private key exchange.

Furthermore, when a private key from a client is compromised, only one side of the communication can be read by the attacker. Since only the incoming messages can be decrypted using the recipients private key, an attacker cannot read sent messages.

### Hashing

Hashing is a one way type of encryption, and is sometimes listed as a type of symmetric encryption. However, hashing algorithms do not have a key; instead, they compute a

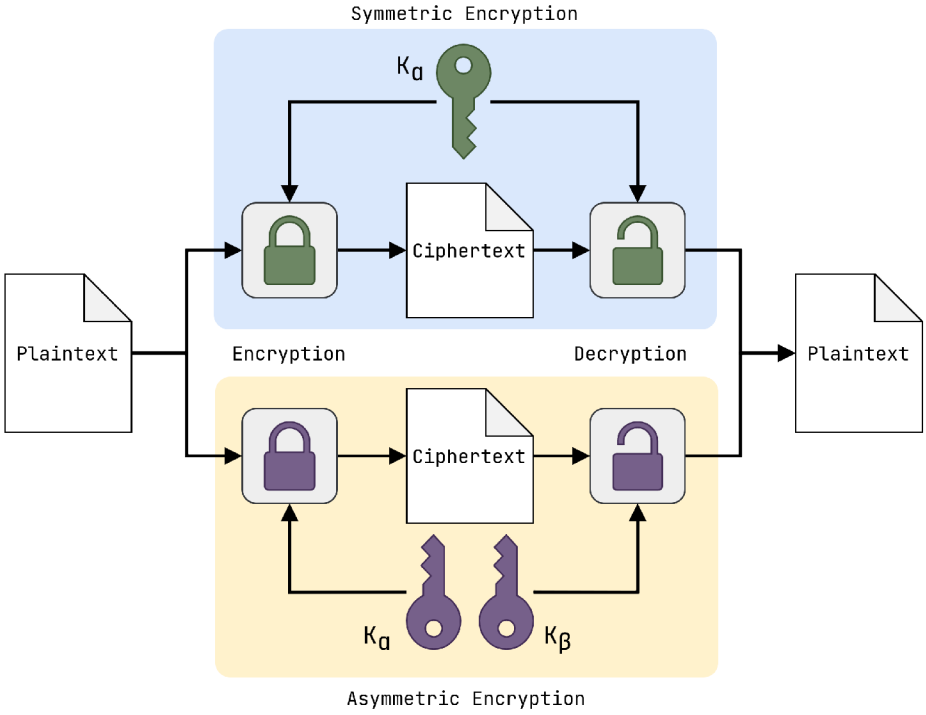


Figure 2.1: Symmetric and asymmetric encryption schemes.

fixed length hash, or message digest, from a piece of plaintext. Hashing can be usually encountered in the context of checksums; messages, which are used for checking the integrity of files, or the presence of tampering.

Similarly to symmetric and asymmetric encryption, there are numerous hashing algorithms. A hashing algorithm is said to be broken if it is possible to reverse it, and obtain the plaintext from the hash. Furthermore, an undesirable characteristic of hashing functions is collisions; when more than one piece of plaintext leads to the same output hash.

## 2.2 Security protocols

Security protocols, sometimes also referred to as cryptographic protocols, are a set of rules that define the standard of communication between two or more entities, with the goal of providing different security functions.

A well chosen encryption algorithm is one of the most important elements of a security protocol, as it determines the security of the protocol, and indirectly affects the use cases in which the security protocol will be deployed. Different security protocols incorporate different encryption algorithms based on their use-case and perceived goals; for example, a security protocol which is designed to be used in low latency situations will probably use a weaker encryption algorithm than a security protocol designed for maximum security and confidentiality [10].

Security protocols can be split into three main categories based on their goal:

**Subject authentication protocol**

The purpose of these protocols is to unequivocally prove the identity of either one or more entities in the system. For example, this is necessary to prevent an attacker from

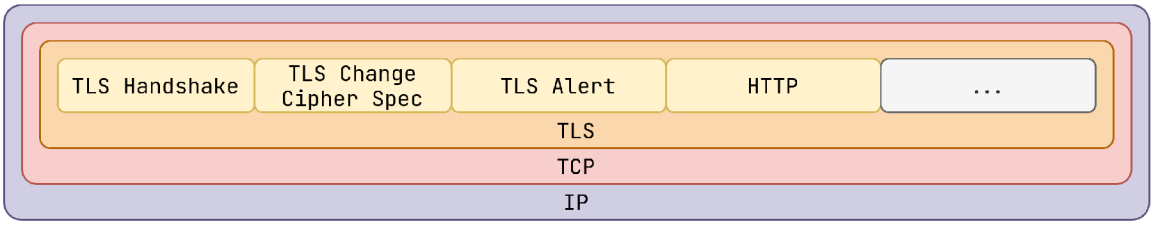


Figure 2.2: The TLS stack.

hijacking the communication. This is a process where a malicious attacker pretends to be the rightful destination party of the communication, and thus receives the private messages.

**Key distribution protocol**
The aim of this type of security protocol is to distribute encryption keys between entities in the system securely to allow them to further communicate using encrypted messages. If a malicious third party gained access to these keys, they could easily decrypt the encrypted messages and eavesdrop on the communication.

**A combination of the previous two**
This type of security protocol contains a combination of the aforementioned two groups, providing both subject authentication and key distribution. An example of this type of protocol is TLS.

## 2.3 The Transport Layer Protocol

Historically, there have been many security protocols designed for high throughput communication between endpoints on the internet. Nowadays, the most prominently used protocol is Transport Layer Security (TLS). It is based on the now deprecated Secure Sockets Layer (SSL) 3.0 protocol released in 2011 [23], and its latest version 1.3 described by RFC 8446 [21] was released in 2018.

The main goal of TLS is to provide cryptographic security with a relative efficiency; thus, TLS is used in a wide range of applications, including email clients, instant messaging, and others. The most prominent usage of TLS on the world wide web today is as the protocol providing the security layer in the HTTPS protocol. According to Google, the use of HTTPS has increased from around 48% of all internet traffic in 2014 to 95% in 2020 [26]. This makes TLS the most widely used encryption protocol today, and thus the subject of much interest from a network security standpoint.

The TLS standard specifies multiple different protocols which are responsible for different parts of the secure communication between the client and server. Due to the versatility of TLS, it is important to differentiate between different tasks that can be accomplished by creating protocols intended for those tasks.

TLS works by exchanging „records“, the overall structure of which is defined by the TLS Record protocol. This protocol serves as the basis for further TLS communication, such as the Handshake or Alert protocols [35]. A diagram of this stack can be seen in figure 2.2. However, due to the complexity of TLS and its underlying protocols, only the TLS Record and Handshake protocols will be expanded upon, as other underlying TLS protocols are beyond the scope of this work.

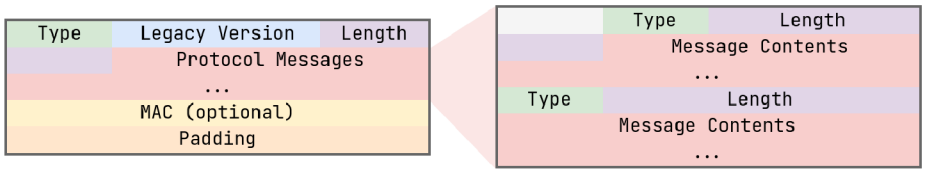


Figure 2.3: The structure of a TLS Record.

When a message is supposed to be sent using the TLS Record protocol, it is broken down into smaller, manageable fragments, each of which is then compressed and encapsulated within the TLS record and sent. Much like lower level protocols, the TLS record protocol can be thought of as an envelope used to carry higher level TLS protocols. The TLS Record protocol also supports protecting the records against corruption with integrity protection [39] or by encrypting its contents. For performance reasons, encrypting TLS records is usually done using symmetric encryption. It is then the task of the receiving party to decompress, decrypt, and assemble the received data. A structure of the TLS record can be seen in figure 2.3.

Before a client and server can securely communicate using the TLS protocol, they must first establish a secure channel of communication. Because each client and server support or desire to use different combinations of parameters on which to base further secure communication, the TLS Handshake protocol is used to negotiate these parameters between the server and client in the „handshake“ phase of a TLS communication.

During this phase, all communication between the client and the server is not encrypted and is transferred as plaintext using the TLS Record protocol. A simplified overview of the handshake protocol can be seen in figure 2.4. The handshake part of a TLS communication is quite complex, and mostly above the scope of this work. It can be split into four flights [35], an overview of which is explained bellow:

**Flight 1**

A client contacts the server using a Client Hello, requesting a TLS connection.

**Flight 2**

During this flight, the server responds to the client with 2 to 5 messages. These are the following:

1. **Server Hello** - A response to the Client Hello.
2. **Certificate** - The server authentication.
3. **Server Key Exchange** - This message contains the parameters which can be used to generate the private key.
4. **Certificate Request** - A request for the client certificate.
5. **Server Hello Done** - Finishing the Server Hello.

**Flight 3**

During this flight, the client responds with 2 to 5 messages to the server:

1. **Certificate** - Sent if the server requested it.
2. **Certificate Verify** - Signed verification of the certificate.

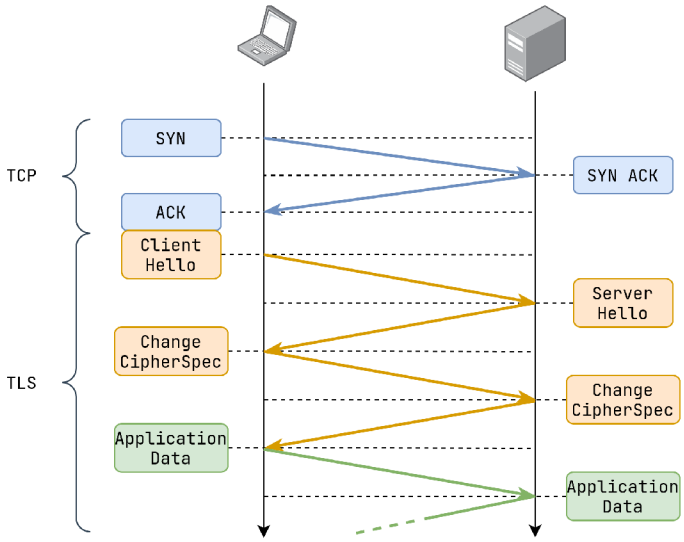


Figure 2.4: An overview of a TLS Handshake.

3. **Client Key Exchange** - The main part of the third flight, contains the parameters which can be used to generate the private key.
4. **Change Cipher Spec** - Request for changing the parameters chosen by the server.
5. **Finished** - The first cryptographically protected message.

**Flight 4**

The final flight is made up of two messages sent from the server to the client:

1. **Change Cipher Spec** - Response to a request for parameter change.
2. **Finished** - This message is also protected.

As can be seen, the server and client use asymmetric encryption to agree on common parameters, which are then used to generate a common private key between the two parties, so that symmetric encryption can be used instead. This is done for performance reasons, as the handshake occurs only at the beginning of the communication, and all further messages are sent using the less resource intensive symmetric encryption.

The most important part of the handshake for us is the Client Hello message, as it is the easiest part of the handshake to use for client identification.

### Client Hello

As mentioned previously, when a client wishes to communicate with a server using TLS, the client sends a Client Hello message. This Client Hello contains the default parameters regarding the subsequent communication that the client offers to the server. The definition of the Client Hello message as defined by RFC 8446 [21] can be seen in figure 2.5.

This struct contains the parameters offered by the client. Out of these, the most notable ones are the `CipherSuite` and `Extension` fields, as they are the parts of the client hello which are most dissimilar between different clients. These are then extensively used during fingerprint generation (see chapter 3).

```
struct {
    ProtocolVersion legacy_version = 0x0303; /* TLS v1.2 */
    Random random;
    opaque legacy_session_id<0..32>;
    CipherSuite cipher_suites<2..2^16-2>;
    opaque legacy_compression_methods<1..2^8-1>;
    Extension extensions<8..2^16-1>;
} ClientHello;
```

Figure 2.5: Structure of a TLS Client Hello.

### GREASE Values

The TLS GREASE extension („Generate Random Extensions And Sustain Extensibility“) was first published by David Benjamin in January 2016 and is now defined by RFC 8701 [19]. It is a mechanism to prevent extensibility failures in the TLS ecosystem. The standard reserves a set of TLS protocol values that may be advertised to ensure peers correctly handle unknown values.

Based on the implementation specification in the TLS standard, a client or server must ignore unknown TLS extensions in order to allow new capabilities to be introduced into the TLS ecosystem, without compromising interoperability between devices [21]. The set of GREASE values in hexadecimal format can be seen below:

```
[0a0a, 1a1a, 2a2a, 3a3a, 4a4a, 5a5a, 6a6a, 7a7a,
 8a8a, 9a9a, aaaa, baba, caca, dada, eaea, fafa]
```

RFC 8701 further suggests that TLS implementations which do not correctly implement randomly selected GREASE values can be more prone to fingerprinting. Furthermore, it is important to note that different fingerprinting schemes can have different approaches to dealing with GREASE values in a Client Hello (see sections 3.2 and 3.3).

# Chapter 3

# Fingerprinting and TLS

## 3.1 Fingerprints and fingerprinting

Much like in the physical world, a fingerprint in the context of information theory is a set of information which can be used to uniquely identify its host; it can relate to software, hardware, or both. Fingerprints are usually constructed to capture different attributes or features from the original information set. This means that the number of features that are used to construct the fingerprint directly contribute to the uniqueness of the fingerprint [31]. A graphical illustration of a fingerprint for a common network packet can be seen in figure 3.1. Fingerprinting is the action of collecting and analysing fingerprints, and correlating them with existing sets of data to identify the probable origin of the fingerprint.

Wagner [45] defined an entire fingerprint taxonomy, and a common fingerprinting scheme, which is made up of the following three actors:

**Distributor**
: A distributor is the authorised supplier of fingerprintable objects.

**User**
: A user is an individual or agency who is authorised to gain access to objects that the `distributor` supplies. In many cases the `users` will be aware that certain objects have been fingerprinted.

**Opponent**
: Opponent is the party in the communication which fingerprints the objects and usually has access to a fingerprint knowledge base.

A diagram of this system working with a TLS Client Hello and Server Hello messages can be seen in figure 3.2. It is important to note that in the context of this work, each

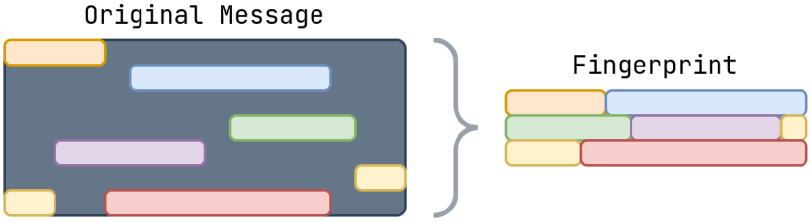


Figure 3.1: A graphical illustration of fingerprint generation.

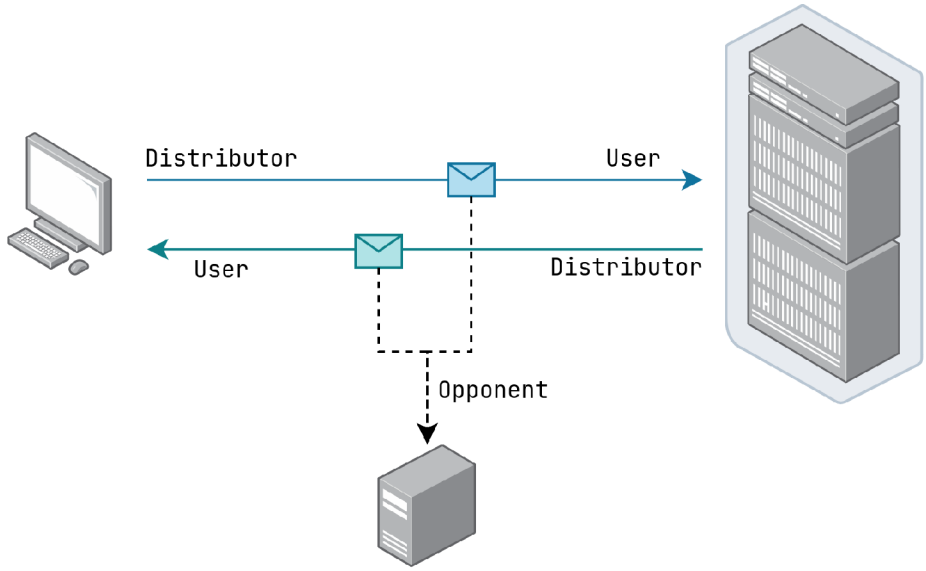


Figure 3.2: A common fingerprinting scheme.

`distributor` and `user` are taken to be any processes that generate or respond to a TLS handshake, rather than a whole network client. Whilst performing fingerprinting on a whole client can be useful, process detection tends to be more widespread in practice.

Furthermore, this fingerprinting scheme makes no assumptions about the intentions of its parties. It is equally valid to assume that the `user` has malicious intentions and that the `opponent` aims to maintain a high level of security in the network by identifying malicious connections.

Fingerprinting can also be split into two categories; active and passive [2]. Active fingerprinting requires interaction with the client to generate a meaningful fingerprint, and is usually done when a passive approach would not yield the required number of attributes to generate a fingerprint with the required entropy. However, this approach can be quite resource intensive, as it puts further strain on network infrastructure and requires processing both from the client and server side [32]. This is why the passive approach is preferred in high availability environments.

In the specific case of TLS, the fingerprinting is based on extracting information from the unencrypted part of the TLS Handshake, as introduced in section 2.3. Since both the client and the server send an unencrypted message, it is possible to create a fingerprint for both. However, most fingerprinting schemes are only interesting in identifying the client. This is because client identification is usually more valuable, as they can have a larger impact on overall network security and performance. This is why they mainly focus on the Client Hello message.

Based on the definition of a fingerprint in section 3.1, the Client Hello message is approached as a set of attributes, which can be used to generate a fingerprint.

## 3.2 JA3

JA3 fingerprinting was developed at Salesforce by John Althouse, Jeff Atkinson, and Josh Atkins. It is based on the previous work by Lee Brotherson and his FingerprinTLS pro-

gram [3]. JA3 is currently the most widely adopted TLS fingerprinting scheme, and it has seen wide industry support in network monitoring software [29, 40, 41]. JA3 is the preferred fingerprinting method in high speed networks due to its predictable fingerprint format and constant fingerprint size.

JA3 works by extracting the following fields from the Client Hello message:

- TLS version
- Supported cipher suites
- TLS extension headers
- Elliptic curves
- Elliptic curve point formats

A graphical illustration to how this extraction works could be seen in figure 3.1. The fingerprint is then generated by concatenating these extracted fields in their decimal representation into a string separated by comas, to generate the following:

`Version,Ciphers,Extensions,EllipticCurves,EllipticCurvePointFormats`

The TLS standard does not require that all these fields are present in the Client Hello message [21]. When either of these fields is missing, it is replaced with an empty string. For example, the following is a valid JA3 fingerprint string with missing `EllipticCurves` and `EllipticCurvePointFormats` fields:

`769,4-5-47-51-5 0-10-22-19-9-21-18-3-8-20-17-255, , ,`

The JA3 fingerprinting standard also hashes the strings with the MD5 hashing algorithm. This makes the strings portable, and easily shareable between different software implementations. For this particular JA3 fingerprint string, the hash would be

`b677934e592ece9e09805bf36cd68d8a`

An important feature of the JA3 approach is the fact that it completely ignores the GREASE values as described at the end of section 2.3. This is done to reduce the number of fingerprints for a single client. If GREASE values were not ignored, and a TLS client included one or more GREASE values in the Client Hello, each permutation of the Client Hello that a client could generate would create a new fingerprint. This would lead to a single process being represented many times in the database. This is compounded by the fact that JA3 utilises MD5 hashing which disallows the use of comparing string distances to compare between fingerprints; a single character change would lead to a completely different fingerprint.

A fingerprinting scheme is as useful as the available databases mapping fingerprints to client identifications. However, this is the main fallback of the JA3 fingerprinting method; its available datasets. The suggested format for a JA3 database entry can be seen in figure 3.3. However, there is currently no specification for the format of the `identification` string; this results in fingerprints which cannot be easily analysed further, and must be taken at face value. As an example, take the following three fingerprint `identification` strings from a publicly available JA3 database:

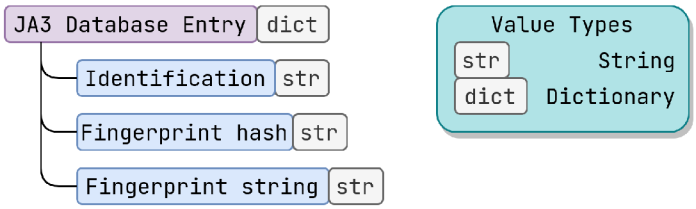


Figure 3.3: The suggested format of a JA3 database entry.

- `BurpSuiteFree(Tested:1.7.03 on Windows 10),eclipse,JavaApplicationStub`
- `SCANNER: wordpress wp-login Firefox/40.1`
- `Chrome/59.0.3071.115 Win10`

As can be seen, the fingerprints are not in a format that is conducive to further analysis. The operating system of the `distributor` is either not present, or formatted inconsistently. Each fingerprint then contains either one or more processes, but without the ability to distinguish between the more and less probable ones.

Furthermore, it has been demonstrated that JA3 fingerprints are highly dependant on the family and version of the operating system when the same process is fingerprinted across different network clients [33]. This further discredits existing JA3 databases, as they need to contain many duplicate fingerprints for each process to capture all the possible operating systems that can be associated with the process. However, since fingerprinting schemes are usually interested in outputting a single identification for a process, the different operating systems are usually lost during the identification.

## 3.3 Mercury

Mercury fingerprinting is an up-and-coming fingerprint standard created by Blake Anderson and David McGrew at Cisco. It was originally developed as part of the Cisco Joy project [4], which is a software package for capturing and analysing network flow data, meant for network research, forensics, and security monitoring.

However, Mercury fingerprinting was spun off into a project of its own, called Cisco Mercury. This new project contains its own set of software and databases for analysing network traffic using their new fingerprint format [7].

The Mercury fingerprint format is very different from the JA3 fingerprint format, in that it contains more detailed information about the Client Hello message. The overall fingerprint format is shown below:

```
(version)(cipher suites)((extension0)(extension1)...)
```

The Mercury fingerprint format leaves the contents of the fingerprint in hexadecimal format. As can be seen, the Mercury fingerprint contains many of the same pieces of information as the JA3 fingerprint, but in a different format. They both contain the `version` and `cipher suites` parameters from the Client Hello; however, they differ in regards to the extensions, where Mercury contains a much more detailed representation of the packet. Different from the JA3 fingerprint, the `extension` field in a Mercury fingerprint contains a nested list of all extensions found in the Client Hello. Furthermore, the following extensions include the extension contents:

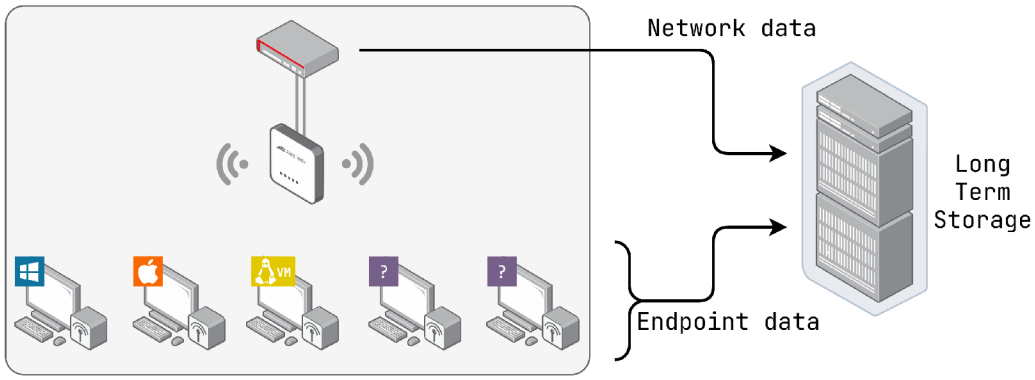


Figure 3.4: Cisco Mercury network and endpoint fusion.

- Supported groups
- Elliptic curve point formats
- Status request
- Signature algorithms
- Application layer protocol negotiation
- Supported versions
- PSK key exchange algorithms

Extensions in the above list would be formatted as `(XXXXYYYYZ...Z)`, where `XXXX` is the extension code, `YYYY` is the length of the extension contents, and `ZZZZ` is the content of the extension. If an extension is not in the above list, and only its code is stored in the fingerprint, it would be in the format `(XXXX)`. An example of a valid Mercury fingerprint can be seen below:

`(0301)(c00ac009c014c0130035002f000a)((0000)(000a00080006001d00170018)(000b00020100)(0023)(0017)(ff01))`

The Mercury approach normalises all GREASE values to the value `0a0a`, but it doesn't change the location of the GREASE extension within the extension block in the Client Hello. This leads to an overall decrease in duplicate fingerprints per client compared to recognising all GREASE extension values; however, it does not eliminate all duplicates.

The TLS standard specifies that „the extensions MAY appear in any order“ [21], which can result in duplicate fingerprints due to the location of the GREASE extensions in the Client Hello. For example, an extension `0a0aXXXX0a0a` where `XXXX` is any extension content would be recognised as a different extension by Mercury than `0a0a0a0aXXXX`, even though they contain the same extension data in regards to creating a TLS handshake.

A great differentiator between the JA3 and Mercury approach is the creation and formatting of the fingerprint databases. According to the creators of Mercury, other fingerprint databases are slow to update and lack real-world contextual data [8]. This is why they created a new approach to building fingerprint databases, a diagram of which can be seen in figure 3.4. Because the authors of Mercury had full access to the network on which the

Mercury database was created, they were able to use the combination of detailed endpoint data and traffic analysis to create a much more comprehensive database with more in depth information.

However, because the private version of the database contained sensitive information about the authors' network, the publicly available Mercury database only contains a subset of the information present in the full private database [5]. The authors of Mercury fingerprinting suggest to create specialised knowledge bases for each network that Mercury will be deployed on to obtain the full accuracy of the results they achieved [6]; however, this is impractical for larger networks, or for networks where the administrators don't have control over the endpoint devices.

A diagram of the overall structure of a single entry in the Mercury database can be seen in figure 3.5. In the diagram, the keys in the Mercury database are presented in a human readable format with the type of data they contain appended to the key. Contrast this to the JA3 database format shown in figure 3.3. The plethora of information present in the Mercury database is its main advantage compared to the approach presented by JA3.

As can be seen in the diagram, the database structure is quite complex, and contains many different types of information. Most importantly, it doesn't contain simple $1:1$ relationships between the keys and values in the database. For example, a single fingerprint string contains information about $1:N$ fingerprints. At the time of writing, the most process identifications that a single entry in the database contained was 9.

Due to the larger complexity of the Mercury database, it is fitting to describe the different parts of the database entry in larger detail. A description of selected fields can be seen below.

**Autonomous Systems**

RFC 1930 defines an Autonomous System as a „connected group of one or more IP prefixes run by one or more network operators which has a single and clearly defined routing policy". Autonomous Systems are uniquely identified by their number, referred to as the ASN. These numbers are present in the Mercury database along with the frequency of hits for each process. These can then be used with the destination IP of the Client Hello packet to increase the detection rate of a process.

**Hostname Domains**

The hostname domain fields in the database contain the second level domain names (such as `google.com`, `cisco.com`, ...) to further increase the precision of the expected destination information of the process.

**Application Port**

The Mercury database contains a mapping of port numbers to common application level protocols [5]; for example, 443 → HTTPS, 993/995 → email, etc. However, the public version of the database contains only two values for the port keys in the database; `https` and `unknown`. This information value is quite limited, and doesn't present much leeway for further analysis.

**Operating Systems**

As you can see in figure 3.5, the keys for operating systems are split into three levels; the `Family`, the `Name`, and the `Build`. These values then again contain the frequency with which this particular key was recognised. This is one of the fields which benefited the most from the fusion of endpoint and network data, as shown in figure 3.4, as no other public database contains similar amount of detailed information.

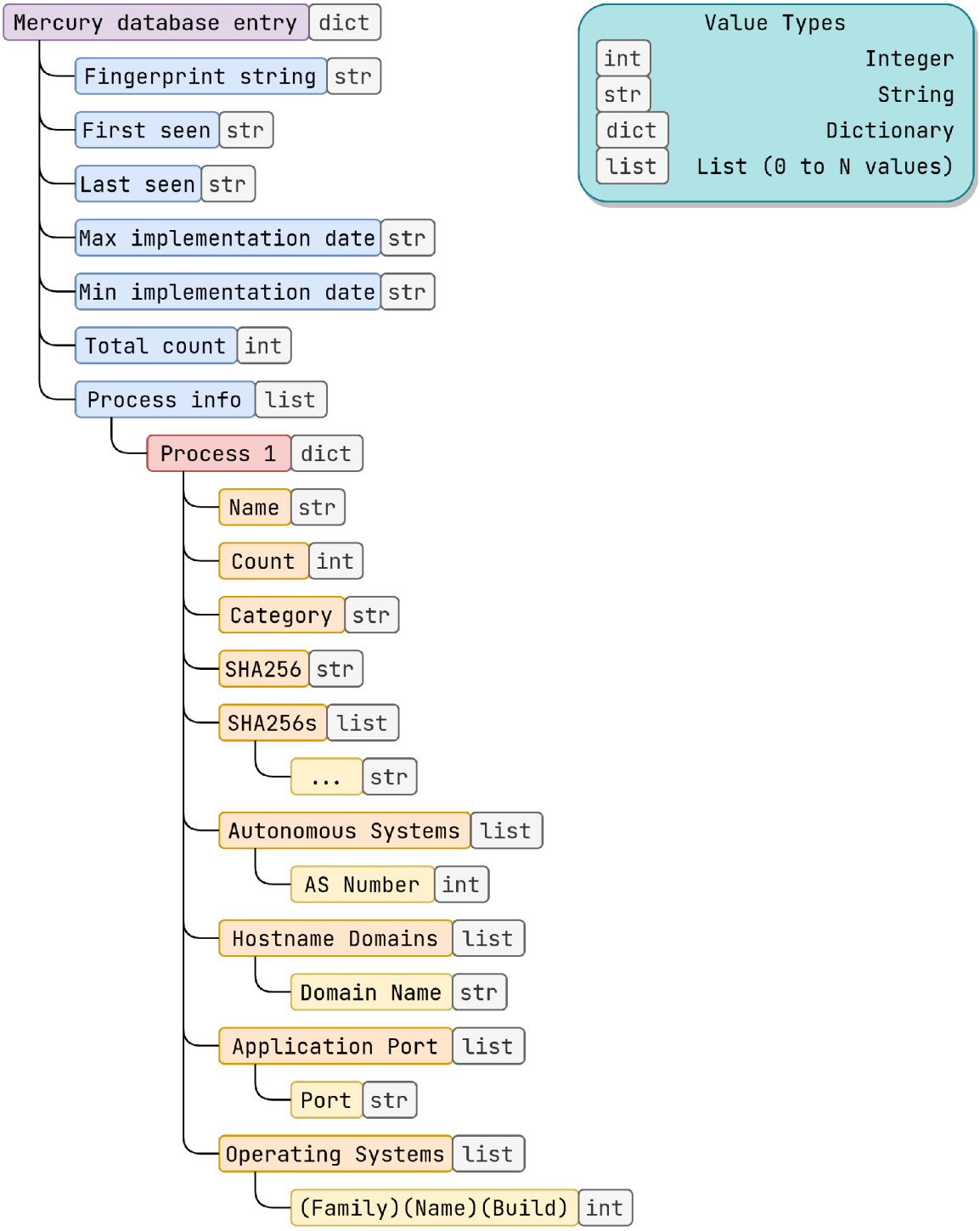


Figure 3.5: The Mercury database structure.

# Chapter 4

# JA3cury and its classification algorithms

To leverage the high adoption of JA3 fingerprinting whilst allowing for deep analysis options offered by Mercury, I have created a new fingerprinting approach called JA3cury. JA3cury is the combination of the JA3 fingerprint format with the detailed and content rich Mercury database. The motivation for creating JA3cury is to bring the detection accuracy and analysis possibilities offered by Cisco Mercury to the mainstream, where JA3 is mostly used.

Current network monitoring approaches usually rely on probes or flow exporters, which are highly specialised and optimised pieces of software for working with high throughput traffic [22]. Furthermore, probes for network traffic analysis are sometimes used in combination with specialised hardware, which is expensive to develop and maintain. However, these probes in many places already have support for JA3. Whilst exchanging current JA3 detection algorithms for JA3cury may not be a simple 'plug-and-play', the detection modules are usually separated from these high speed probes, and their development is easier and cheaper. For these reasons, the output JA3 fingerprints make sense to be used with this new approach.

One of the areas of JA3 fingerprinting which is improved using this new approach is JA3 fingerprint collisions. Due to the large combinations of applications, their versions, and the underlying operating systems that are possible, Kotzias et al. discovered using a dataset created by capturing network traffic over 3 years that JA3 fingerprints have a 7.3% collision rate [30].

This presents a problem for classical JA3 databases, as it is very difficult to distinguish between two fingerprints in the database that were detected. However, the proposed approach allows the same JA3 fingerprints to be identified differently due to the contextual data. Furthermore, due to the introduction of collisions into the database during the conversion from Mercury to JA3 representation (see section 4.1), I have built the ability to cope with fingerprint collisions into my detection algorithms to increase accuracy (see section 4.2.2).

## 4.1 The JA3cury approach

Converting a fingerprint from a Mercury format to JA3 is a destructive conversion: some information is lost along the way. This is due to the fact that Mercury fingerprints contain

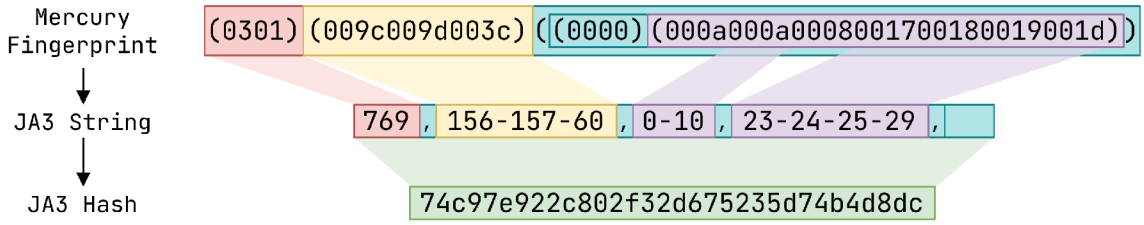


Figure 4.1: Mercury to JA3 fingerprint conversion.

a larger amount of information than JA3 fingerprints. A diagram of Mercury to JA3 conversion can be seen in figure 4.1.

The fact that some fingerprint information is lost during the conversion leads to collisions in the fingerprints. As of the time of writing, the public Mercury database contains 9060 unique fingerprints, but after converting the database to use JA3 fingerprints, it contains only 7146 unique fingerprints. That is a net loss of fingerprint count of 21.1%. Initially, I was worried that this would drastically decrease the accuracy of the proposed method. However, the fingerprints that were lost during the conversion represent a miniscule proportion of the network traffic. This is supported by Laperdrix et al., who discovered that around 1.72% of fingerprints are responsible for 21.75% of communications [31].

Nevertheless, I investigated the overall length of fingerprints in the Mercury and JA3 versions of the database. The results of this investigation can be seen in figure 4.2. By default, the Mercury representation has a larger amount of formatting characters that don't increase the informational value of the fingerprint. This is why three different sets of fingerprints were compared; the original Mercury fingerprints without any formatting characters, a JA3 hex representation, which contained a concatenated string of all the values extracted from the Mercury fingerprint without further formatting, and then JA3 fingerprints, where the hex values were converted to decimal representation.

As can be seen from the graph, the Mercury fingerprints (blue) are much longer overall, and more evenly spread out across the graph. The JA3 hex values (red) are more grouped together, mainly due to the removal of extension contents from the Mercury fingerprints. Lastly, when the hexadecimal JA3 values are converted to decimal representation (green), the fingerprints decrease in length further despite containing the same amount of information. This is due to values such as $0017_{16}$ being represented as $23_{10}$.

Overall, the takeaway is that even though the JA3 representation reduces the fingerprint size and thus looses identifying information, it is not as severe as expected. Furthermore, the grouping of fingerprints into larger chunks rather than spreading them all out and the increased number of collisions might prove useful due to the ability to average out anomalies during the detection process.

### 4.1.1 Process name normalisation

One of the characteristics of the Mercury database is the precision with which it is able to identify different process names. However, this precision is unnecessary for real world detection; for example, it is improbable that minor version change of a program introduces large changes that would require for it to be detected as a separate process. Due to this, a part of the JA3cury approach is to modify the existing process names in the Mercury database.

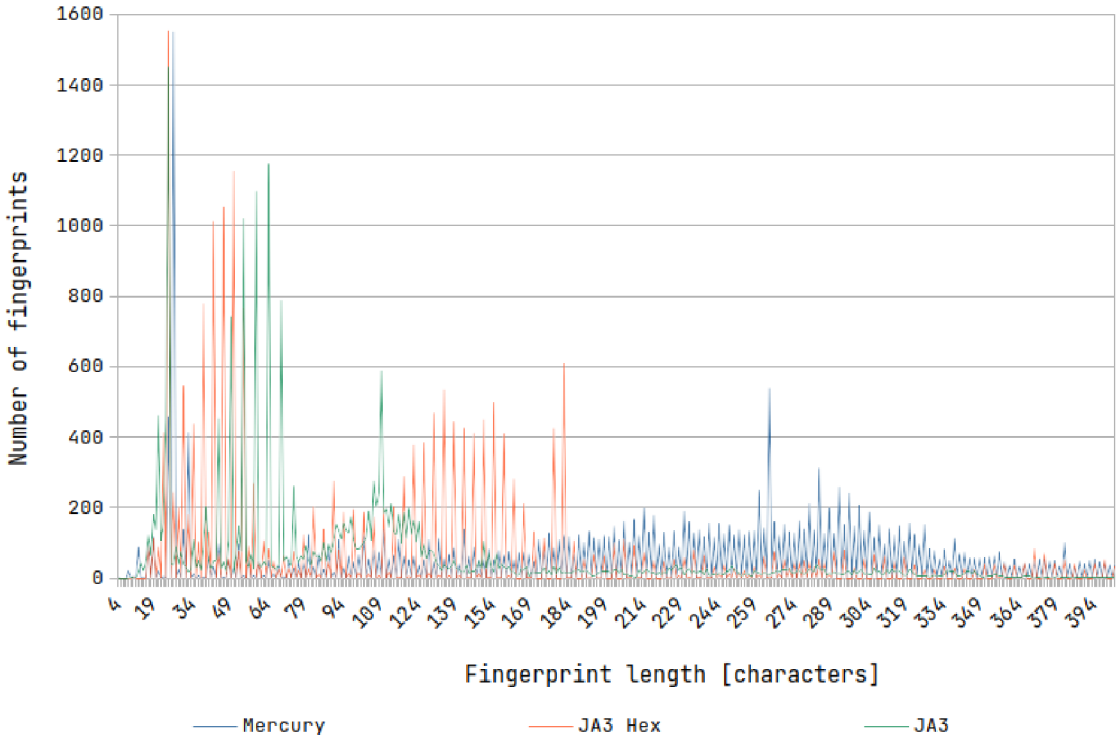


Figure 4.2: Comparison of fingerprint lengths across formats.

### Tumour removal

After parsing the database for process names, the following process suffixes („tumours“) have been identified:

```
.exe, x86_64, i386, 64, 32
```

These suffixes add very little informational value to the process name, as the architecture of the system is either of low importance, or can be deduced from the detected operating system. For example, without removing these suffixes, the processes `PyCharm` and `PyCharm64` would be identified as separate processes. However, it is beneficial for the sake of detection accuracy to detect both processes as `PyCharm`, and thus increase the overall probability of identifying this process.

### Removing capitalisation

Another modification that was done to the Mercury database was the removal of capitalisation, and the conversion of all process names to strictly lowercase. This was done much like in section Tumour removal to decrease the number of similar processes being recognised separately. A concrete example can be the processes `Chrome` and `chrome`, which are the same process but were originally detected separately.

### Defining equivalence classes

During the testing phase of JA3cury development, it was noticed that many processes are still detected separately even though they had all their tumours removed and were converted

to lowercase. The most common reason for this was the version number being mentioned in the process names. In some cases, this could be thought of as a valid distinction between processes (for example `Firefox-86` and `Firefox-87`), but for most cases it presented an unwanted obstacle to process identification. An unwanted example could be the `terraform` process, where the following processes names were detected separately:

```
terraform-provider-aws_v2.31.0_x4
terraform-provider-aws_v2.33.0_x4
terraform-provider-aws_v2.46.0_x4
terraform-provider-aws_v2.49.0_x4
terraform-provider-aws_v2.50.0_x4
terraform-provider-aws_v2.54.0_x4
terraform-provider-aws_v2.59.0_x4
```

As can be seen, even after removing the tumours, the processes can still be perceived as having abnormalities that separate them from similar processes. However, I decided that it is impractical to define tumours manually to achieve the correct detection of all process names, as some tumours could collide with process names, and lead to unwanted renaming. Furthermore, a large number of tumours would be hard to track, and it would be unclear which tumours are defined for which processes.

To solve this problem, I used a clustering algorithm to experimentally group together similar process names. Based on these groups, I decided which equivalence classes should be defined, and which clusters were spurious and contained unrelated processes.

The `DBSCAN` clustering algorithm was used with a custom metric to define the equivalence classes between processes. `DBSCAN` is a density-based clustering algorithm introduced in 1996 [36] which has received the 2014 SIGKDD Test of Time Award [9].

To measure the similarity between two process names I experimented with Hamming [27] and Levenshtein [44] string distances, whilst varying parameters of the `DBSCAN` algorithm. Hamming and Levenshtein distances are well known metrics for measuring the difference between two strings.

The key part of this approach was to correctly define the equivalence classes from the generated clusters. Due to the nature of process names in the database, my clustering approach grouped together process names that are very similar, but that are separate in reality; such as `node` and `code`. This is why I didn't use the direct results of the clustering algorithm to generate the classes, but hand-picked sensible results. I have chosen to define the following equivalence classes:

```
chrome, firefox, terraform, python,
outlook, edge, java, qemu, skype
```

These equivalence classes are compared against after the process name modifications defined in sections Tumour removal and Removing capitalisation. When a process contains a substring which is present as an equivalence class, the whole process name is replaced with the equivalence class. Some examples of equivalence class substitution can be seen below:

```
[chrome, google chrome, google chrome helper]    →  chrome
[microsoftedge, microsoftedgecp, microsoftedgesh] →  edge
[outlook, hxoutlook]                             →  outlook
[skype, skypeapp]                                →  skype
```

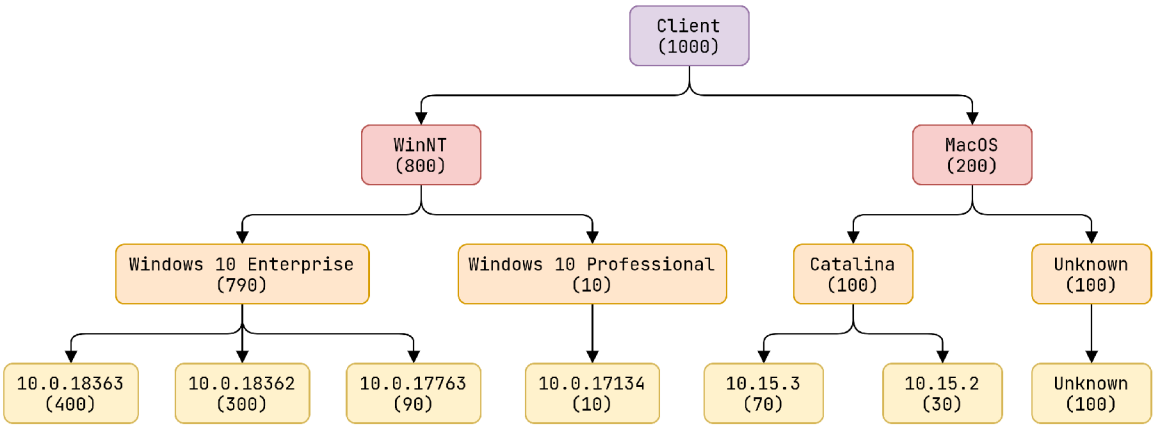


Figure 4.3: An example of the operating system tree.

## 4.2 Classification algorithms

Thanks to the depth and complexity of information present in the Mercury database, I was able to create a number of traffic classification algorithms, where each classifier took advantage of different information in the database.

Whilst the implementation details regarding my approach will be discussed in detail in chapter 5, it is important to note that the proposed algorithms were able to identify multiple clients in a single network capture using the source IP addresses. When talking about a network client in the upcoming sections, a single source IP is meant.

### 4.2.1 Operating system classification

As mention in section 3.3, the operating system items in the database are in the `key:value` format. However, the operating system `key` can be split into three distinct parts; the operating system `Family`, `Name`, and its `Build`. As such, my approach was built around detecting each of these three values separately. I believe that, for example, knowing the family of the operating system is much more valuable than knowing its build number. For this reason, a new library, `OSTree`, was designed, which is able to choose the most probable operating system for a number of input values.

The library is centred around a tree structure with depth 4, where each level in the tree represents another part of the operating system designation. Each node thus contains its name, and the number of times it was inserted. An example tree can be seen in figure 4.3.

To give a concrete example, the tree in the diagram could have been created by adding the following keys and their values:

| Key | Value |
|---|---|
| `(WinNT)(Windows 10 Enterprise)(10.0.18363)` | 400 |
| `(WinNT)(Windows 10 Enterprise)(10.0.18362)` | 300 |
| `(WinNT)(Windows 10 Enterprise)(10.0.17763)` | 90 |
| `(WinNT)(Windows 10 Professional)(10.0.17134)` | 10 |
| `(MacOS)(Catalina)(10.15.3)` | 70 |
| `(MacOS)(Catalina)(10.15.2)` | 30 |
| `(MacOS)(Unknown)(Unknown)` | 100 |

Furthermore, as can be seen from the diagram, the children of each node are sorted left to right in a descending order. It is thus possible to simply take all the leftmost children from the root to obtain the most probable operating system based on the inserted children. Furthermore, each node contains the frequency which equals to the sum of frequencies of its immediate children.

Similarly to the input, the library returns the operating system identification in three levels; the `Family`, `Name`, and the `Build`. Furthermore, the library includes the possibility of including a „trust threshold“. For example, if we say we want to be 80% sure of the operating system identification, the value `(WinNT)(Unknown)(Unknown)` would be returned for the example tree. Similarly, a trust threshold value of for example 60% would result in `(WinNT)(Windows 10 Enterprise)(Unknown)`. If the threshold value is too large to identify any operating system, all the return values are `Unknown`. By default, no trust threshold is set.

An example of initialisation, children appending, and returning the classification for an `OSTree` can be seen below:

```
tree = OSTree('client123')
tree.add_child(['WinNT', 'Windows 10 Professional', '10.0.17134'], 100)
tree.get_most_probable_children()
```

### 4.2.2 Process and category classification

As previously explored in section 3.3, each entry in the Mercury database contains $1 : N$ processes, and each process contains the process name and its category.

This gives us the opportunity to detect the category and process names separately. Detecting the category of the process in the database separately could be useful when combining this fingerprinting approach with existing traffic classification methods based on network flow monitoring. Because these methods are usually capable of detecting the type of traffic accurately, the results of category detection from fingerprinting could be used to enhance these methods. This gives my approach an advantage, as it can be used by itself to detect the processes of network clients, or it could be used in tandem with network flow monitoring to enhance the accuracy of those methods.

I have implemented 7 different classifying algorithms which can be used for identifying TLS connections. These algorithms differ in how they score the process values in the database, and which information from the database and the Client Hello context they take into account. The simplest approach can be seen in algorithm 1.

As can be seen, the algorithm has two required inputs; the fingerprint database and a Client Hello packet, and it outputs a dictionary containing an instance of `OSTree`, a dictionary of categories, and a dictionary of processes. An example of an empty classification dictionary and a filled classification dictionary can be seen in figure 4.4.

A detailed overview of the algorithm 1 can be seen below:

**Lines 1-5** The creation of an empty classification dictionary for this Client Hello.

**Line 6** The fingerprint is generated from the Client Hello.

**Line 7-9** All the fingerprints in the supplied database that are identical to the given fingerprint are taken into account.

**Line 10** A loop through all the processes in the fingerprint entry.

**Algorithm 1:** A basic algorithm to classify a single Client Hello

**Input:** The Client Hello
**Output:** Dictionary of process and category names
**Data:** Fingerprint database

```
1  classification ← dict {
2      OS ← OSTree();
3      Categories ← {};
4      Processes ← {};
5  };
6  fingerprint ← generate_fgpt(Client Hello);
7  foreach entry in database do
8      if entry ≠ fingerprint then
9          continue
10     foreach process in processes do
11         foreach os in os_info do
12             add score(os) to classification;
13         if process not in classification then
14             add process to classification with score 0;
15         increment classification for process with score(entry);
16         if category not in classification then
17             add category to classification with score 0;
18         increment classification for category with score(entry);
19 return classification
```

```
d = {
    "os": OSTree(),
    "processes": {},
    "categories": {}
}
```

Figure 4.4: New dictionary for a client.

```
d = {
    "os": ...,
    "processes": {
        "chrome": 100,
        "firefox": 20
    },
    "categories": {
        "communication": 12,
        "security": 3
    }
}
```

Figure 4.5: Example dictionary for a client after classification.

**Lines 11-12** This is where the scoring of all the operating systems for each process takes place. Note the use of the function `score`, which is used to generate the score of the operating system that will be used upon the insertion of the operating system entry into the `OSTree`. This score function could extract the frequency from the database, modify this value, or insert a custom value.

**Lines 13-15 and 16-18** These two sets of loops are identical for the process names and classifications. However, they are computed separately due to the reason outlined previously. Here, we loop over all the process and category entries in the fingerprint respectively. Furthermore, note the use of the `score` function, which is used in the same way as for operating system classification.

**Line 19** Finally, the created classification dictionary is returned.

As mentioned previously, I have created 7 classification algorithms which take into account different parts of the `Client Hello` message, use different combinations of contextual information about the packet, and approach the scoring of processes and categories differently. An overview of the classifiers can be seen below.

### Simple

This is the most basic approach, which I used to provide a baseline against which I could compare my subsequent classifiers. The scoring for this method was done by taking the frequency values straight from the database for each process only. I felt it wasn't appropriate to modify these scores for the baseline algorithm. This approach didn't take into account any contextual information about the packet.

### Unique

This approach was similar to the above mentioned `Simple` approach. However, I classified only unique fingerprints in each captured traffic file, without checking for further fingerprint collisions. This approach was developed to measure the impact of fingerprint collisions by converting the database from Mercury to JA3 representation.

### Simple Total Count

This approach is different from the `Simple` algorithm in its scoring approach. This method took into account the `Total count` field present for each fingerprint which can be seen in figure 3.5.

Whilst this field is separate from the process list and notes how many times a certain fingerprint was encountered, this approach was developed to check whether the most common processes can have their classification influenced. Laperdrix et al. discovered that around 1.72% of fingerprints are responsible for 21.75% of communications [31], and this was an attempt to favour the classification of these processes.

### Simple ASN

This approach is also based on the `Simple` algorithm, but it takes into account the destination autonomous system of the Client Hello packet. The destination autonomous system was generated from the destination IP of the packet. However, a possible error was introduced, as the autonomous system database that I used is more recent than the autonomous system information in the Mercury database.

When a process was found to contain the relevant autonomous system, the score for that autonomous system was added to the score for the entire process. However, when a process was found *not* to contain the required autonomous system, it was not discarded, and its score remained unchanged.

### No Frequency

This approach is different from the previous algorithms in its scoring method; it disregards the frequencies in the database, and instead generates its own score. A point is awarded to the process, category, and the operating system for each hit they receive.

This approach was created to measure the effects of the scores in the database by comparing the classification results with the `Simple` algorithm.

### No Frequency ASN

This algorithm was created as a combination of the `Simple ASN` and `No Frequency` classifiers. As such, it uses its own scoring algorithm, and it also takes into account the destination autonomous system of the `Client Hello` packet.

### No Frequency All

This algorithm takes advantage of all the information present in the Mercury database. Whilst it is the most resource demanding algorithm, I expected its results to be the most accurate. Specifically, it takes into account the destination autonomous system, the destination domain name, and the source port from which the packet was sent. However, since the public Mercury database contains barely any information about the source ports of the processes, this last item is thought to be of little impact. Similarly to previous two algorithms, this approach uses a custom scoring system.

# Chapter 5

# NEMEA module implementation

During the development of JA3cury fingerprinting, it was essential to test different approaches and detection algorithms on the same network traffic and to compare their results. Furthermore, comparing the achieved results to a stable baseline was important to correctly evaluate the achieved results. That is why I developed two versions of the NEMEA module, an experimental version, and a production version. The experimental version of the module was developed first, and some of its parts were carried over to the production version.

Both modules were created using the Python programming language in version 3.6. Python was selected for its compatibility across Unix operating systems, the existence of supporting NEMEA libraries written in Python, my familiarity with the language, and its fast prototyping ability. Furthermore, the official `pmercury` used by my modules is also written in Python. During implementation, code quality analysers and linting tools such as `Pylint` were extensively used to ensure a maintainable and high quality code base.

## 5.1 NEMEA

NEMEA (Network Measurement and Analysis) is a system for the analysis of network traffic developed by Čejka et al. for CESNET, a developer and operator of national e-infrastructure for science, research, development and education in the Czech Republic [15]. The NEMEA system has been used as the basis for a number of papers [18], and is deployed on the backend infrastructure of CESNET [11].

The NEMEA system is designed with very strong modularity in mind. The system is split into 3 important main parts [17]; the framework, the supervisor, and the modules. An overview of an example NEMEA system can be seen in figure 5.1.

The modules can be seen as the building blocks of a running NEMEA instance. Each running instance of the NEMEA system is made up of 1 to $N$ modules. These are the parts of the NEMEA system which are responsible for the processing of network traffic and any related computation, such as anomaly detection. These modules create a flexible pipeline, where different modules are chained behind one another, and process the received data in a different way.

Each module runs in a separate process and performs a task with the received data. Furthermore, each module has 1 to $N$ interfaces shown in blue, which can be used for receiving and sending network traffic. The modules can be split into **detectors** (orange) which are responsible for detecting specific traffic, and **general modules** (yellow) which perform other tasks with the incoming traffic.

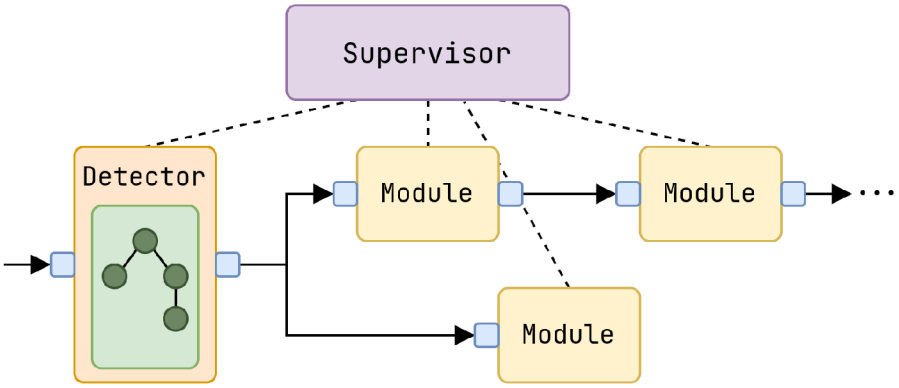


Figure 5.1: A basic overview of the NEMEA system structure.

The NEMEA framework provides common libraries which implement features shared among different NEMEA modules [12]. The three most important libraries used by all the NEMEA modules are the following:

**TRAP Interface**
The TRAP (Traffic Analysis Platform) interface shown in the diagram in blue is the interface which is used for communication between modules. The underlying `libtrap` library implements features such as buffering and abstracts these features to the modules, which are presented with a simple to use interface.

**UniRec data format**
UniRec stands for Unified Record. It is a unified data format used by the `libtrap` system for communication between modules. It was designed to be flexible, memory efficient, and to allow fast processing speeds. It supports many different data formats, including strings, signed and unsigned integers of different lengths, IP addresses, and so forth.

**Common library**
This library is shown in the diagram in green. It implements common data structures and algorithms which are frequently used by the different modules.

Lastly, the supervisor acts as a central management tool which is responsible for controlling, managing, and running the different NEMEA modules. The supervisor is capable of accepting different configurations, and setting up the NEMEA system accordingly. All the presented NEMEA parts were utilised during the development of the JA3cury modules.

## 5.2 The JA3cury experimental module

The goal of the experimental module was to provide a way for comparing different classification approaches against each other, whilst ensuring annotation of the generated data. For this reason, the modularity and performance of the module was sacrificed in order to allow quick and easy extensibility.

This experimental module was also used for comparisons of the original Mercury fingerprints and the proposed JA3cury approach. For this reason, I was unable to use existing NEMEA modules, such as IPFIXProbe [16], to convert from the captured PCAP files

overviewed in section 6.1 to `trapcap` files supported by `libtrap`. Thus, I wrote my own module for converting the captured PCAP files to the `trapcap` format called the Ingestion Engine. The overall structure of the experimental module can be seen in figure 5.2.

### Ingestion Engine

The goal of the ingestion engine was to convert captured PCAP files into a `trapcap` data format using the `UniRec` specifier. The ingestion engine used the `pyshark` library for parsing the contents of the packets. Whilst this library is quite slow compared to other available libraries for packet parsing [37], it presents the user with a high level interface which simplifies the development of the module. Since the performance of the module was not essential for experimental use, the fast prototyping made available to me using this library was preferred to high speed processing of data using libraries with complex implementation requirements. The following data was extracted from the Client Hello packets:

- Source IP address
- Destination IP address
- Source port number
- Destination port number
- Raw handshake contents

This raw information was used to construct the Mercury and JA3 fingerprints. Since the Mercury fingerprint contains more information than a JA3 fingerprint for the same Client Hello, it was constructed first with the help of the `TLS` utility from the `pmercury` library, which is part of the official Mercury GitHub repository. The fingerprint was then converted to the JA3cury format using a custom function.

After the fingerprints were constructed, the ingestion engine module outputted all the relevant information about the Client Hello messages in the following UniRec specified format:

```
timestamp, src_ip, dst_ip, src_port, dst_port, ja3, mercury, domainname
```

The converted data was then passed to the JA3cury module in this format.

### JA3cury module

The job of this central module was to aggregate all the information passed to it from the Ingestion Engine through the TRAP interface, and to pass this information on to the different classification engines. Each classification engine is created with its own instance of the fingerprint database. Whilst this is very memory consuming, it allows for different approaches regarding the formatting of the database, such as using different fingerprint formats. It is then the job of the classifier to correctly analyse the data received from the JA3cury module and output a classification for the fingerprint and its contextual information.

Adding new analysers to this module is very simple. The module contains a dictionary of all the classifiers, and its format is then directly transferred to the output JSON file for each client. For example, a format of this dictionary can be seen below:

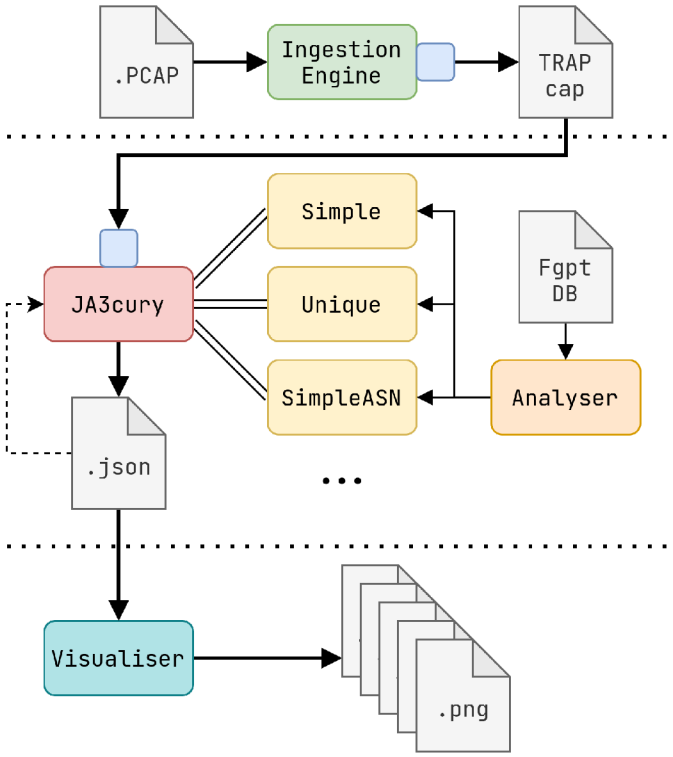


Figure 5.2: The structure of the experimental JA3cury module.

```
fmt = {
    simple:  {
        mercury:  SimpleClassifier(mercury_db),
        ja3cury:  SimpleClassifier(ja3cury_db)
    }
    no_frequency:  {
        mercury:  NoFrequencyClassifier(mercury_db),
        ja3cury:  NoFrequencyClassifier(ja3cury_db)
    }
}
```

If the dictionary above was used to configure the JA3cury module, it would pass the fingerprinted traffic to 4 different classifier objects, created by instancing two different classifier classes, `SimpleClassifier` and `NoFrequencyClassifier`. Each of them would be instantiated with two different fingerprint databases. This shows the flexibility of the experimental module, as it is possible to very simply configure it and extend it with further classifiers and databases.

Lastly, this module has support for loading already existing JSON output files created by previous runs of the module. To compare between the accuracy of different classification algorithms, the JSON file contains process and category annotation explained in section 6.1. Because the output file is overwritten every time, the module loads the annotations from an existing JSON file to preserve them between runs. Because the annotation is done by hand, it would be very time intensive to repeat the process of annotation every time, and would negate all the other benefits of this experimental module. The final output of this module

for a TRAPCAP file containing multiple clients and using the previously introduced format dictionary would be the following:

```
{
    client1: {
        simple: {
            mercury: {
                os: OSTree(),
                processes: {
                    process1: 1000,
                    process2: 500,
                    ...
                },
                categories: ...
            },
            ja3cury: {
                ...
            }
        },
        ...
        annotations: {
            processes: {
                process1: True,
                process2: False,
                ...
            },
            categories: ...
        },
    ...
}
```

**Visualiser**

This module was used for generating graphs for visual representation of the results. It loads the JSON files generated by the experimental JA3cury module and uses the `matplotlib` library to generate graphs of the results. This module is also able to load annotation information from the JSON file, and use this information for generating coloured graphs. Whilst this module doesn't include any further processing of the results, it is very helpful to visually compare the different approaches and analysers.

## 5.3 The JA3cury production module

The goal of the production version of the JA3cury module was to take tested parts of the experimental module and implement them together with higher regard for code quality and performance, and with the intent for production usage of the module. As you can see from the diagram in figure 5.3, its overall structure is much simpler and is optimised for cleaner and more maintainable code.

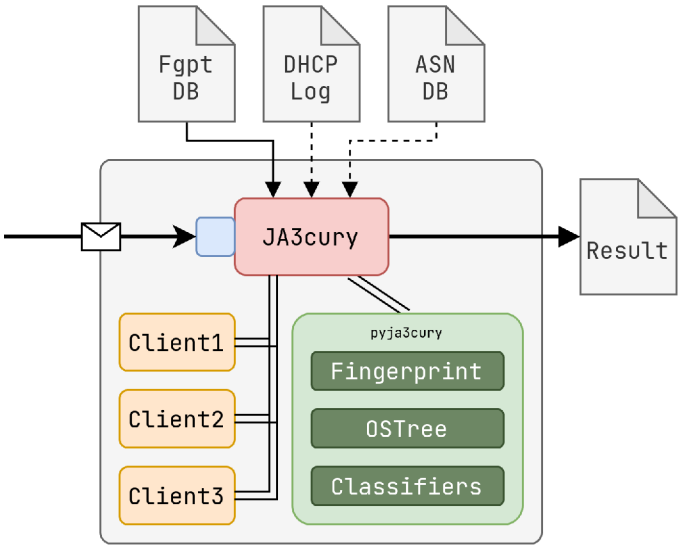


Figure 5.3: An overview of the production JA3cury module.

The main parts of this module are the JA3cury supervisor, the client threads, and the `pyja3cury` library. This module doesn't include supporting software such as the `Ingestion Engine` as it is meant to plug directly into the output of the IPFIXProbe module.

### pyja3cury

A general purpose library was created, which contains modified code from the experimental module. Namely, it contains

- The `OSTree` implementation discussed in section 4.2.1.
- The implementation of the classification algorithms discussed in section 4.2.2.
- The `mercury2ja3` conversion module described in section 4.1.
- The portable implementation of a class representing a Fingerprint.

The library is currently structured in a format which would allow easy publishing to `PyPI`, the Python Package Index. This would allow easy sharing of this library, and giving more users access to the basic implementation of this new fingerprinting approach.

### JA3cury supervisor

Much like the NEMEA system, the production module contains a main supervisor, which administers the different client threads created for client identification. This central supervisor is responsible for managing the fingerprint and autonomous system databases, client threads, and outputting of the results.

The communication between the supervisor and the client threads is done using an input and output queue. When a client hello is received by the supervisor, it uses an internal `Fingerprint` structure to encapsulate the relevant data extracted from the incoming interface and pushes this object into the `client_queue`. This queue is checked by all running

threads, and when the source IP address of the fingerprint matches the IP address of the client, the thread removes the fingerprint from the queue and processes it.

The JA3cury module also supports the input of a DHCP log file containing a mapping of local IP addresses to client names using the `-dhcp` parameter. When the module creates a client for an IP address present in the log file, it names it accordingly. When a local IP address is not present in the log file on the other hand, the module generates a sensible and human readable name for the network client. An example generated client name could be `Client_192.168.0.105`.

The module was designed to make its usage as simple as possible. In the initial stages of development, a separate conversion script was necessary to convert Mercury databases to the JA3cury format. However, in the production version of the module, it has support for automatic conversion of Mercury databases into the JA3cury format using the `pyja3cury` library, and formatting the process names in the database. This means that it is possible to simply pass existing Mercury databases to the module using the `-mercury` parameter. A full overview of the parameters can be seen in appendix A.

The algorithm for the supervisor can be seen in algorithm 2 and described below:

**Lines 1-3** Here, the JA3cury supervisor prepares all the necessary databases it will use during the classification process. Firstly, it prepares the fingerprint database. This is done by normalising all the process names as explained in section 4.1.1. The database is also converted to use JA3 format of fingerprints. Secondly, the DHCP log file is parsed to extract client information, which will be used for naming the detected clients. Lastly, a set of monitored client IP addresses is created, which will hold all the client IP addresses that the supervisor is able to classify. The fingerprint database is immutable to allow quick, read-only access from all client threads. The `known_clients` and `monitored_ips` variables are both used only the supervisor thread.

**Lines 4-5** On these two lines, the communication queues to the clients are created. Both queues are instances of the Python `Queue` class, which is a part of the Python standard library. This class implements the synchronous access of multiple threads to the contents of the queue. These queues are used by all threads to pass information between each other. The `client_queue` is used to pass `Fingerprint` objects to the client threads, and the `results_queue` is used by the supervisor to receive classification information.

**Lines 6-9** During these lines, the JA3cury supervisor tries to receive data on its TRAP interface. If data is received, it is processed, and the module starts receiving again. However, if no data is received by the module, it breaks the cycle, and outputs the classification present in the `results_queue`.

**Lines 10-13** On these lines, the supervisor compares the source IP address present in the received UniRec message against the set `monitored_ips`. If the set doesn't contain the source IP address of the message, a new client thread is created to administer the classification of this network client. This thread is named according to the name present in the DHCP file, or a generated name otherwise. This thread is then executed.

**Lines 14-15** During this phase, the supervisor creates a `Fingerprint` object using the `pyja3cury` library, which is then inserted into the `client_queue`, where it can be accessed by all the running client threads.

**Lines 16-19** In this last phase, the module outputs the classification of all the client threads, which has been inserted into the `results_queue`.

**Algorithm 2:** An algorithm for the JA3cury module supervisor.

**Input:** TRAP Interface data
**Output:** Classification of network clients
**Data:** Fingerprint database
**Data:** DHCP Log
**Data:** Autonomous System Number Database

```
1  db ← prepare_database(Fingerprint Database);
2  known_clients ← parse_dhcp_log(DHCP Log);
3  monitored_ips ← ∅;
4  client_queue ← Queue();
5  result_queue ← Queue();
6  while true do
7      data ← interface.receive();
8      if no data received then
9          break;
10     if data.SrcIP ∉ monitored_ips then
11         monitored_ips += SrcIP;
12         client ← Client(SrcIP, known_clients[SrcIP]);
13         client.run();
14     fgpt ← Fingerprint(data);
15     client_queue.put(fgpt);
16 results = {};
17 foreach r in result_queue do
18     results.append(r);
19 return results;
```

### JA3cury client thread

The fingerprint classification for a single client is all computed within the thread for that client. Each client thread has access to a global immutable fingerprint database which it uses for comparing fingerprint strings with the fingerprint present in the Client Hello it receives through the `client_queue`. A simplified version of the client algorithm can be seen in algorithm 3.

**Lines 1-4** Here, the client checks if their lifetime is up. When the client thread existed as long as its lifetime, it outputs its classification to the global `results_queue` and finishes its runtime.

**Lines 5-6** During these two lines, the client thread checks whether the top Fingerprint object in the global `client_queue` has the same source IP as assigned to this thread. If yes, it removes the object from the queue.

**Algorithm 3:** An algorithm for a client thread in the JA3cury module.

```
   Input: Client Queue
   Output: Classification of this client clients
   Data: Fingerprint database
1  while true do
2      if time_running > lifetime then
3          result_queue.push(Classification);
4          return;
5      if last item in client_queue ≡ client IP then
6          client_hello ← client_queue.pop();
7          foreach fingerprint in database do
8              if fingerprint ≠ client_hello then
9                  continue;
10             classification += classifier(fingerprint ,client_hello);
```

**Lines 7-10** With this Fingerprint removed from the queue, it parses the global immutable database for this fingerprint. When it is found, the classification score is calculated for all the available processes, operating systems, and process categories present in the database. The scoring depends on the classifier that is configured with the module, as different classifiers take different contextual information into account.

After the module is launched, we can visualise its working with a concurrency diagram, shown in figure 5.4. As can be seen from the graph, when the module is started, the databases are prepared and converted to a usable format. The module then creates a thread for each client that is detected in the incoming data processed by the supervisor. Each thread has a lifetime which was specified in the parameter, or which uses the default value of 30 seconds if this parameter was not set.

When a client thread reaches the end of its lifetime, it pushes all its results into the `results_queue`. If data from this client is however still received by the supervisor, the thread is brought back to life. Lastly, the final part of the lifetime of the supervisor is dedicated to formatted output of the detection results.

## 5.4 Future work

The module could be improved in the future by implementing an automatic code testing and verification suite, which would allow faster implementation of new features. It would also ensure the expected functionality of implemented parts of the module.

An optimisation problem I noticed is the implementation of the `client_queue` and the way it is accessed by different clients. If a single source IP creates a large number of Client Hello messages at one time, this number of messages could flood the `client_queue` as all the clients read only the top element of the `client_queue`. This could be solved by the clients parsing the whole queue and removing the first relevant Fingerprint, instead of only checking the first element in the queue.

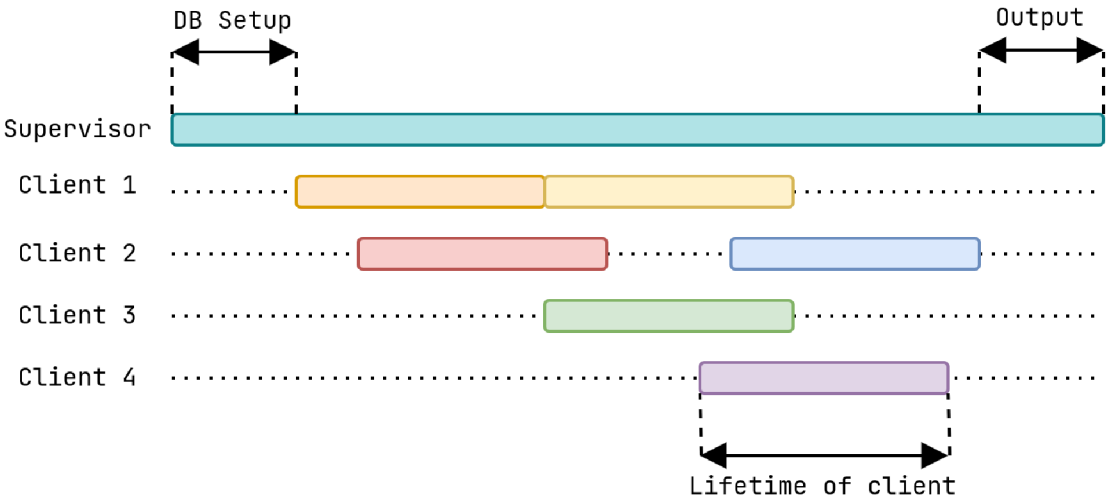


Figure 5.4: Visualisation of the module concurrency.

# Chapter 6

# Classification results overview

To test the proposed JA3cury approach, the experimental version of the JA3cury module presented in chapter 5 was used with datasets presented in section 6.1 to create client classifications. The clients that created the network traffic used for the classification experiments can be seen in table 6.2. This traffic was then parsed using the JA3cury experimental module explained in section 5.2 configured with all the available classifiers outlined in section 4.2.2.

The overall classification was done in two iterations. In the first iteration, all the detected processes generated by the module were taken into account when calculating the accuracy of the approach. In the second iteration, only the top 5 processes with the highest score were taken into account during the accuracy calculation. This is why all the process and category accuracy graphs are split into two groups; the graphs containing the average for all detected processes, and only the top 5 processes. These two iterations were done because it is unlikely that a real world scenario would require the detection of all possible processes running on a device, but it provides an interesting comparison between different methods for edge case scenarios.

To measure the detection accuracy of the JA3cury approach and the developed algorithms, I created a baseline measurement using the official Cisco Mercury tool, `pmercury`. This tool was used with all the captured data files to classify the probable process and category of each client using the original Cisco Mercury fingerprinting approach.

However, `pmercury` was unable to parse Client Hello messages captured from a VPN interface, which excluded all data captured in files `capture` and `tun0_filter_40000` created on my network. I suspect this is related to the `dpkt` Python library which is used by `pmercury` for parsing packets. Furthermore, the tool doesn't include operating system detection, so I was unfortunately unable to provide a baseline using the official Cisco Mercury fingerprinting approach for the operating system detection. Furthermore, to provide a comparison between JA3cury and Mercury, I ran each classifier with JA3 and Mercury fingerprints, and compared the results.

## 6.1 Datasets

For evaluation purposes I created my own dataset of TLS Client Hello packets. The data was captured by members of the Network Monitoring research team at the CESNET Liberouter group. The data usually comes from home environments, with a smaller portion of the traffic being captured in our offices. Altogether, this new dataset was made up of 38,122 `Client`

Table 6.1: Operating system representation in Mercury database and the created dataset

| Operating System | % of scores in Mercury DB | % of packets in my dataset |
|---|---|---|
| Windows | 48.28 | 31.56 |
| Mac | 51.71 | 17.30 |
| GNU/Linux | $1.3 * 10^{-3}$ | 51.14 |

`Hello` packets spanning the three main desktop operating systems (MacOS, Windows, and GNU/Linux). The dataset also contains traffic created by mobile-based operating systems (Android, iOS, WatchOS, ...), which represents about 38% of all captured `Client Hello` packets. Detailed information regarding the dataset can be seen in table 6.2.

The created dataset is fundamentally different from the proportion of operating systems present in the Mercury database. The comparison of proportions of operating systems in the Mercury database and in my dataset can be seen in table 6.1. As you can see, the makeup of the database and my dataset in regards to the operating system families is radically different. The Windows family of operating systems is the most similar, which means I expect the Windows operating system detection accuracy to be the highest. However, whilst the Mercury database barely contains any Linux based operating systems, they are predominant in my datasets. This presents a big challenge to my classification approaches as it drastically decreases the likelihood of classifying the operating system correctly for Linux.

Furthermore, this dataset includes a large amount of traffic generated by mobile devices; however, the Mercury database does not index mobile devices. This discrepancy between operating system representations can be partly explained by the lack of available endpoint information gathering tools targeted at Android and iOS devices; for example, the tool `os_query` is not available for these platforms [1].

I have therefore decided to classify these devices into their most similar desktop operating system families; iOS has been taken to be MacOS, and Android is taken to be Linux. Whilst this could introduce a level of uncertainty into the operating system detection, I felt it was important to include these mobile devices in the dataset, as they make up a significant amount of traffic on modern networks [25].

All the processes that were recognised were annotated by hand. The annotations were done separately for processes and categories, and consisted of a single boolean value for each process and category respectively, which represented whether that particular item was expected or not. For example, a value of `True` would be interpreted as a correctly identified process.

Each author of a capture file attached a protocol with the file, which contained information about the operating system and all the used software on the computer during the capture period. From this software, a list of most probable process names was generated.

For example, when an author listed `Visual Studio Code` in a list of used software, the processes `code`, `chrome`, `chromium`, and `electron` were annotated as correct, as Visual Studio Code is built using the Electron framework, which is based on Chromium. Even though an application could be specifically listed in the database, as is the case with Discord for example, I still annotated related process names as correct due to the tiny frequencies associated with these programs.

Table 6.2: Detailed dataset information.

| Author | File Name | Client Packets | | Client OS | |
|---|---|---|---|---|---|
| | | IP | Packets | Family | Name |
| Lukáš Hejcman | capture | 10.8.0.5 | 3545 | Linux | Fedora 33 |
| | | 10.8.0.8 | 798 | Windows 10 | Professional |
| | | 10.8.0.20 | 4456 | Linux | Android 10 |
| | | 10.8.0.21 | 1000 | Mac OS | IOS 13 |
| | tun0_filter_40000 | 10.8.0.5 | 4105 | Linux | Fedora 33 |
| | | 10.8.0.8 | 4969 | Windows 10 | Professional |
| | | 10.8.0.20 | 6424 | Linux | Android 10 |
| | | 10.8.0.21 | 2617 | Mac OS | IOS 13 |
| | deepthought | 192.168.0.107 | 162 | Linux | Fedora 33 |
| | eth0_filter_500 | 192.168.0.105 | 500 | Linux | Debian 10 |
| | VB_Win10 | 10.0.2.15 | 304 | Windows 10 | Enterprise |
| Karel Hynek | capture | 192.168.88.245 | 1000 | Mac OS | Big Sur |
| | capture2 | 192.168.88.245 | 978 | | |
| Tomáš Čejka | tlshandshakes | 147.32.76.149 | 303 | Linux | Fedora 32 |
| David Beneš | idk | 192.168.1.101 | 3961 | Windows 10 | Enterprise |
| Andrej Lukačovič | capture | 192.168.0.100 | 1000 | Windows 10 | Enterprise |
| | capture3 | 172.21.233.213 | 1000 | | |
| Zdena Tropková | capture | 192.168.0.21 | 1000 | Mac OS | Catalina |

## 6.2 Process detection

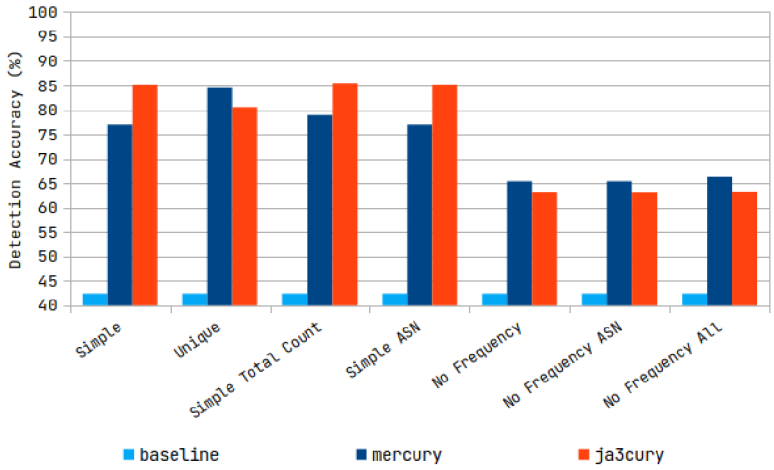


Figure 6.1: Detection results for all processes.

As can be seen from graph 6.1, the most successful classifiers used the original frequencies in the database. Furthermore, the best results were obtained using JA3cury, but Mercury fingerprinting is able to outperform the JA3cury approach in some scenarios, especially when using custom scoring. However, since custom scoring approaches were less accurate in all cases, this looses some relevance.

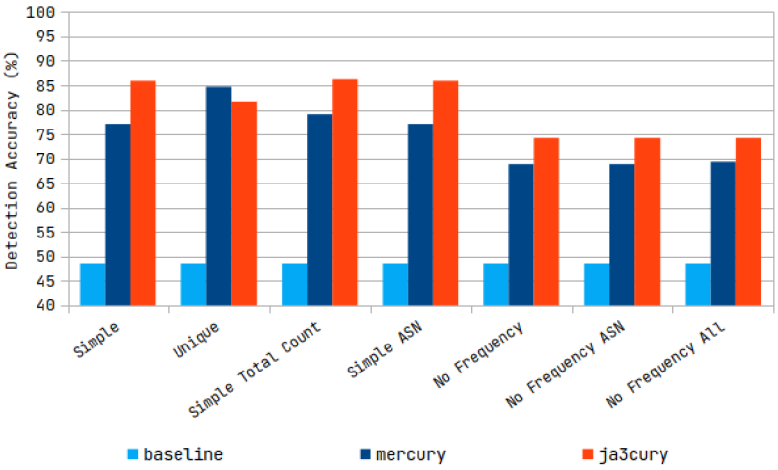


Figure 6.2: Detection results for the top 5 processes.

When regarding only the top 5 processes in graph 6.2, JA3cury is able to outperform Mercury fingerprinting in all but one scenario. Again, a trend of classifiers using database frequencies outperforming custom scoring can be seen.

Overall, the custom scoring schemes results in lower detection accuracy. Furthermore, it is clear that Mercury results are improved when no fingerprint collisions are taken into account, and only unique fingerprints from the capture file are taken into account. Also, JA3cury is able to outperform Mercury regarding the detection of the top 5 processes, but it fares less well when all the detected processes are taken into account. This hints at the ability of Mercury to better cope with very low frequency processes.

## 6.3 Category detection

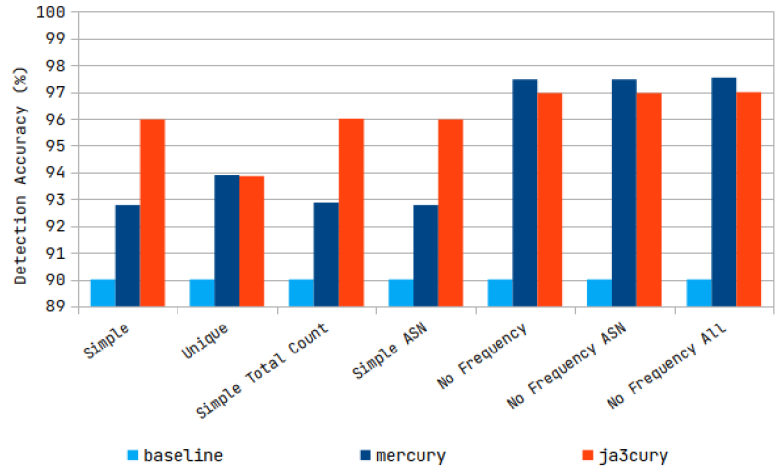


Figure 6.3: Detection results for all categories.

During category detection, we see that JA3cury is more stable regardless of the classifier when all the processes are taken into account in graph 6.3. It vastly outperforms Mercury

fingerprints in all but one algorithm when the frequencies in the Mercury database are used. However, it is beaten by the Mercury fingerprints, which are able to get the best result when using a custom scoring scheme. Furthermore, we see that increasing contextual information has little to no effect on the accuracy of detection.

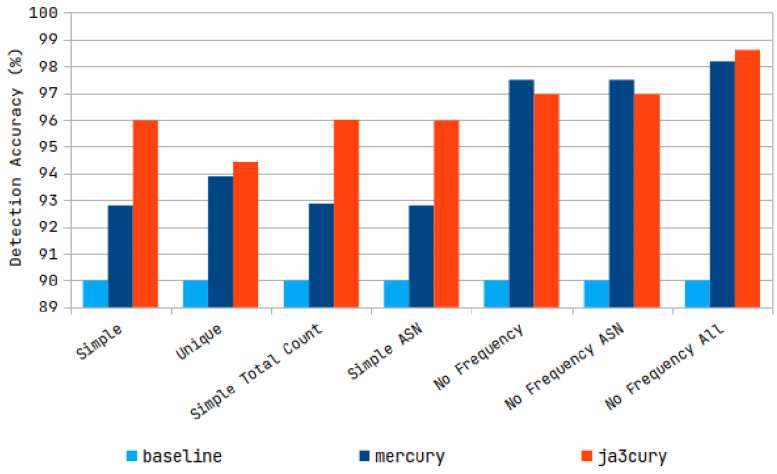


Figure 6.4: Detection results for the top 5 categories.

In graph 6.4, there is roughly a repeat of the previous results, with the exception of the last classifier, where JA3cury outperforms Mercury and reaches the best result overall. However, here the contextual information impacts the detection accuracy somewhat.

Contrary to process detection, category detection fared better when using classifiers with custom frequencies. In both graphs the tendency of JA3cury to stay more stable between tests can be seen, whilst Mercury is very dependant on the scoring approach. Furthermore, the results suggest that Mercury is more dependant on contextual information.

## 6.4 Operating system detection

Graph 6.5 is similar to the graphs seen in the category detection section. JA3cury is able to predict the operating system with a better accuracy for methods using the database frequencies, but is outperformed by Mercury for custom scoring algorithms. However, the number of different operating systems in the dataset was not large enough to definitively come to a conclusion about Mercury and JA3cury operating system detection accuracy.

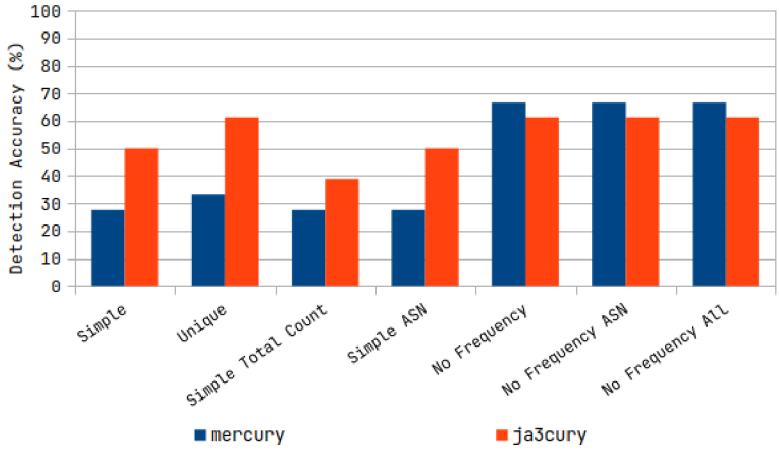


Figure 6.5: Detection results for the operating systems.

## 6.5 Conclusion regarding detection algorithms

For process detection, the results are in line with the predictions made based on the lengths of fingerprints seen in figure 4.2. Whilst Mercury fingerprinting is better suited for precise fingerprints of a larger number of processes, JA3cury is better able to predict a smaller number of processes more accurately. The success of category and process detection clash against each other in regards to the scoring approach. Whilst process detection is more accurate when using the original database frequencies, category detection prefers custom scoring algorithms. However, due to the very high accuracy of category detection regardless, it would be beneficial to sacrifice category accuracy for the comparatively high resultant gains in process detection obtained by using the database frequency values.

Furthermore, JA3cury tends to be less dependant on contextual information than Mercury. This is due to the larger overlap in fingerprints and the inherent collisions between them. However, this did not hamper the overall detection accuracy for JA3cury.

In conclusion, the above experiments showed that the proposed JA3cury fingerprint method is able to match or outperform the Mercury fingerprint approach, despite the shorter length of fingerprints. This is probably due to the ability of the presented method to overcome statistical anomalies created by a database which doesn't represent the network on which the fingerprint approach is used. The JA3cury approach is able to deal with fingerprint collisions, and thus overcomes the shorter fingerprint format. My detection algorithms were also able to outperform all the baseline results generated by the official Mercury tools whilst using both the Mercury and JA3 fingerprint formats.

# Chapter 7

# Conclusion

The goal of this work was to create a new approach to the identification of clients and processes communicating using the TLS protocol on a network using fingerprinting. Another goal was to develop a detection module for the NEMEA framework capable of detecting these network clients. Both of these goal were achieved.

Firstly, a new fingerprinting approach called JA3cury was presented. This approach combines existing fingerprinting approaches, JA3 and Cisco Mercury, to create an approach with high compatibility with existing identification modules, while providing a richer fingerprint database capable of more thorough result analysis.

This new approach achieves a similar or better accuracy to the fingerprinting approaches it is based on. In my experiments, the accuracy of baseline results created using the official tools has been increased from 45% to around 75% for process detection. In specific cases this increase is even up to 86%. Similar increases have been achieved for category and operating system detection, with the highest accuracy of 98.5% for category detection being achieved by the proposed approach.

Furthermore, this new approach is able to maintain this higher accuracy despite using shorter fingerprints than Cisco Mercury, which contain less information about the Client Hello message it was generated from.

These results were achieved using a module for the NEMEA framework, which I have created. This module, and the newly created Python library, provide the implementation of 7 classification algorithms using the newly created fingerprinting approach. I have already published these results at the Excel@FIT conference, and I will be preparing a submission for a regular international conference in the future.

The implemented module also contains a common library for working with the proposed JA3cury fingerprinting approach, which can be published as an open source package to public package repositories, and can be used for implementing further software modules based on the proposed fingerprinting approach.

Future extensions of this thesis can include the further optimisation of the presented software implementation, which would allow the module to be used on high throughput backbone infrastructure. Furthermore, a machine learning approach could be used for the classification of detected network traffic.

# Bibliography


[1] *Android + iOS · Issue #2815 · osquery/osquery*. December 2016. Available at: https://github.com/osquery/osquery/issues/2815.

[2] Adamsky, F., Schiffner, S. and Engel, T. Tracking Without Traces—Fingerprinting in an Era of Individualism and Complexity. In: Antunes, L., Naldi, M., Italiano, G. F., Rannenberg, K. and Drogkaris, P., ed. *Privacy Technologies and Policy*. Cham: Springer International Publishing, October 2020, p. 201–212. Lecture Notes in Computer Science. DOI: 10.1007/978-3-030-55196-4_12. ISBN 978-3-030-55196-4.

[3] Althouse, J., Atkinson, J. and Atkins, J. *Salesforce/ja3*. Salesforce, . june 2017. Original-date: 2017-06-13T22:54:10Z. Available at: https://github.com/salesforce/ja3.

[4] Anderson, B. and McGrew, D. *Using Joy Fingerprinting*. February 2019. Available at: https://github.com/cisco/joy.

[5] Anderson, B. and McGrew, D. Accurate TLS Fingerprinting using Destination Context and Knowledge Bases. *ArXiv:2009.01939 [cs]*. september 2020. arXiv: 2009.01939. Available at: http://arxiv.org/abs/2009.01939.

[6] Anderson, B. and McGrew, D. *Video correspondece in regards to Cisco Cognitive Intelligence and CESNET collaboration.* March 2021.

[7] Anderson, B., McGrew, D., Enright, B., Messenger, L., Weller, A. et al. *Cisco/mercury*. Cisco Systems, . august 2021. Original-date: 2019-08-30T21:58:25Z. Available at: https://github.com/cisco/mercury.

[8] Anderson, B., McGrew, D. and Schomburg, K. *The Generation and Use of TLS Fingerprints*. January 2019.

[9] Association for Computing Machinery's Special Interest Group on Knowledge Discovery and Data Mining. *2014 SIGKDD Test of Time Award*. August 2014. Available at: https://www.kdd.org/News/view/2014-sigkdd-test-of-time-award.

[10] Balfanz, D., Durfee, G., Smetters, D. K. and Grinter, R. E. In search of usable security: five lessons from the field. *IEEE Security Privacy*. september 2004, vol. 2, no. 5, p. 19–24. DOI: 10.1109/MSP.2004.71. ISSN 1558-4046. Conference Name: IEEE Security Privacy.

[11] Bartoš, V., Čeleda, P., Kreuzwieser, T., Puš, V., Velan, P. et al. *Pilot Deployment of Metering Points at CESNET Border Links*. Dec 2012. 16 p. Available at: https://www.cesnet.cz/wp-content/uploads/2013/03/metering_points.pdf.

[12] Bartoš, V., Žádník, M. and Čejka, T. *Nemea: Framework for stream-wise analysis of network traffic.* Dec 2013. 19 p. Available at: https://www.cesnet.cz/wp-content/uploads/2014/02/trapnemea.pdf.

[13] Binance Academy. *History of Cryptography.* January 2021. Available at: https://academy.binance.com/en/articles/history-of-cryptography.

[14] Bujlow, T., Carela Español, V. and Barlet Ros, P. Comparison of Deep Packet Inspection (DPI) Tools for Traffic Classification. june 2013. Publisher: Universitat Politècnica de Catalunya. Available at: https://vbn.aau.dk/en/publications/comparison-of-deep-packet-inspection-dpi-tools-for-traffic-classi.

[15] Cejka, T., Bartos, V., Svepes, M., Rosa, Z. and Kubatova, H. NEMEA: A framework for network traffic analysis. In: *2016 12th International Conference on Network and Service Management (CNSM).* IEEE, Oct 2016, p. 195–201. DOI: 10.1109/CNSM.2016.7818417. Available at: http://ieeexplore.ieee.org/document/7818417/.

[16] CESNET. *CESNET/ipfixprobe.* CESNET, Dec 2020. IPFIXProbe. Available at: https://github.com/CESNET/ipfixprobe.

[17] CESNET. *CESNET/Nemea.* CESNET, Oct 2020. Available at: https://github.com/CESNET/Nemea.

[18] CESNET. *NEMEA - A system for network traffic analysis and anomaly detection.* Jan 2021. Available at: https://nemea.liberouter.org/.

[19] David Benjamin. *RFC 8701 - Applying Generate Random Extensions And Sustain Extensibility (GREASE) to TLS Extensibility.* January 2020. Available at: https://datatracker.ietf.org/doc/rfc8701/.

[20] Diffie, W. and Hellman, M. New directions in cryptography. *IEEE Transactions on Information Theory.* november 1976, vol. 22, no. 6, p. 644–654. DOI: 10.1109/TIT.1976.1055638. ISSN 1557-9654. Conference Name: IEEE Transactions on Information Theory.

[21] Eric Rescorla. *RFC 8446 - The Transport Layer Security (TLS) Protocol Version 1.3.* August 2018. Available at: https://datatracker.ietf.org/doc/rfc8446/.

[22] Flowmon. *ADS 11.2 – More than ordinary blacklists.* Available at: https://www.flowmon.com/en/blog/ads-11-2-more-than-ordinary-blacklists.

[23] Freier, A., Karlton, P. and Kocher, P. *RFC 6101 - The Secure Sockets Layer (SSL) Protocol Version 3.0.* RFC6101. RFC Editor, august 2011. RFC6101 p. Available at: https://www.rfc-editor.org/info/rfc6101.

[24] Gary C. Kessler. *An Overview of Cryptography.* Gary C. Kessler, . 1998. Available at: https://www.garykessler.net/library/crypto.html.

[25] GlobalStats. *Desktop vs Mobile vs Tablet Market Share Worldwide.* 2021. Available at: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide/.

[26] Google Transparency Report. *HTTPS encryption on the web – Google Transparency Report*. 2021. Available at: https://transparencyreport.google.com/https/overview.

[27] Hamming, R. W. Error detecting and error correcting codes. *The Bell System Technical Journal*. april 1950, vol. 29, no. 2, p. 147–160. DOI: 10.1002/j.1538-7305.1950.tb00463.x. ISSN 0005-8580. Conference Name: The Bell System Technical Journal.

[28] Kahn, D. *The codebreakers; the story of secret writing*. Rev Subth ed. Scribner, december 1996. ISBN 0-684-83130-9. Available at: https://archive.org/details/codebreakerssto00kahn/mode/2up?q=Kerckhoffs.

[29] Kane, W. A., Vlach, T. and Luks, R. *Encrypted Traffic Analysis*. Flowmon, . 2019.

[30] Kotzias, P., Razaghpanah, A., Amann, J., Paterson, K. G., Vallina Rodriguez, N. et al. Coming of Age: A Longitudinal Study of TLS Deployment. In: *Proceedings of the Internet Measurement Conference 2018*. New York, NY, USA: Association for Computing Machinery, October 2018, p. 415–428. IMC '18. DOI: 10.1145/3278532.3278568. ISBN 978-1-4503-5619-0. Available at: https://doi.org/10.1145/3278532.3278568.

[31] Laperdrix, P., Bielova, N., Baudry, B. and Avoine, G. Browser Fingerprinting: A Survey. *ACM Transactions on the Web*. april 2020, vol. 14, no. 2, p. 8:1–8:33. DOI: 10.1145/3386040. ISSN 1559-1131. Available at: https://doi.org/10.1145/3386040.

[32] Li, J., Zhang, S., Lu, Y. and Yan, J. Real-Time P2P Traffic Identification. In: *IEEE GLOBECOM 2008 - 2008 IEEE Global Telecommunications Conference*. November 2008, p. 1–5. DOI: 10.1109/GLOCOM.2008.ECP.475. ISSN: 1930-529X.

[33] Matoušek, P., Burgetová, I., Ryšavý, O. and Victor, M. On Reliability of JA3 Hashes for Fingerprinting Mobile Applications. In: Goel, S., Gladyshev, P., Johnson, D., Pourzandi, M. and Majumdar, S., ed. *Digital Forensics and Cyber Crime*. Cham: Springer International Publishing, 2021, p. 1–22. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. DOI: 10.1007/978-3-030-68734-2_1. ISBN 978-3-030-68734-2.

[34] N. Ruangchaijatupon and P. Krishnamurthy. *Encryption and Power Consumption in Wireless LANs*. 2001. Available at: https://scholar.googleusercontent.com/scholar?q=cache:mMtU0yWOM2QJ:scholar.google.com/+allintitle:+Encryption+and+Power+Consumption+in+Wireless+LANs-N&hl=en&as_sdt=0,31.

[35] Rolf Oppliger. *SSL and TLS - Theory and Practice*. 2nd Editionth ed. Artech House, march 2016. Information Security and Privacy Series. ISBN 978-1-60807-998-8.

[36] Schubert, E., Sander, J., Ester, M., Kriegel, H. P. and Xu, X. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Transactions on Database Systems*. july 2017, vol. 42, no. 3, p. 19:1–19:21. DOI: 10.1145/3068335. ISSN 0362-5915. Available at: https://doi.org/10.1145/3068335.

[37] Shaw, J. *PyPCAPKit 0.15.5 documentation.* 2018. Available at: https://pypcapkit.jarryshaw.me/en/latest/.

[38] SonicWall Support. *DPI-SSL - HTTPS Traffic is very slow.* October 2020. Available at: https://www.sonicwall.com/support/knowledge-base/dpi-ssl-https-traffic-is-very-slow/200117053527912.

[39] Stephen A. Thomas. *SSL and TLS Essentials - Securing the Web.* 1st ed. Wiley Computer Publishing, february 2000. ISBN 978-0-471-38354-3.

[40] Suricata. *6.17. JA3 Keywords — Suricata 6.0.1 documentation.* 2019. Available at: https://suricata.readthedocs.io/en/suricata-6.0.1/rules/ja3-keywords.html.

[41] Telesoft. *JA3 Fingerprinting: Encrypted Thread Detection.* March 2020.

[42] The Council of Economic Advisers. *The Cost of Malicious Cyber Activity to the U.S. Economy.* Executive Office of the President of the United States, february 2018. Available at: https://www.hsdl.org/?view&did=808776.

[43] Thomas Hardjono and Lakshminath R. Dondeti. *Security In Wireless LANS And MANS.* Artech House, 2005. Artech House computer security series. ISBN 978-1-58053-756-8.

[44] Vladimir I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics - Doklady.* february 1966, vol. 10, no. 8, p. 4.

[45] Wagner, N. R. Fingerprinting. In: *1983 IEEE Symposium on Security and Privacy.* Oakland, CA, USA: IEEE, April 1983, p. 18–18. DOI: 10.1109/SP.1983.10018. ISBN 0-8186-0467-0. ISSN: 1540-7993.

# Appendix A

# JA3cury usage example

The module supports the following parameters:

**`-output`** The JSON file which the output should be written to.

**`-lifetime`** The lifetime of each client thread in seconds.

**`-treshold`** The trust threshold of the `OSTree` classification. The default value is 0, which means the whole OS detection will be outputted.

**`-ifcspec`** The specification of the `TRAP` interface.

**`-classifier`** The classifier which should be used.

**`-asndb`** The database of autonomous system numbers.

**`-dhcp`** The DHCP file containing the mapping of IPv4 addresses to client names.

If we consider the example directory structure shown in figure A.1, the module could be run with the following parameters:

```
./ja3cury -ifcspec „u:ja3cury" -mercury fgpt_db -dhcp dhcp.log
 -asndb asndb -classifier „simple" -lifetime 30 -threshold 40
```

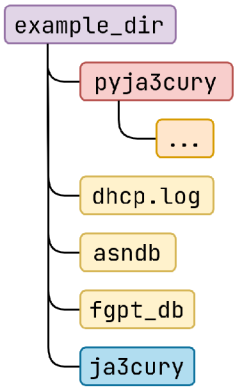


Figure A.1: Example directory structure.