Technical Report: PC Browser and Android Applications Fingerprinting

Malombe Victor

Faculty of Information Technology, Strathmore University - Kenya

Supervisor: Ing. Petr Matousek, Ph.D., M.A.
Faculty of Information Technology
Brno University of Technology

July 2020

Table of Contents

List of Figures iii
List of Tables iv
Preamble 1
Chapter 1: Introduction 2
Definitions 3
Chapter 2: Literature Review 4
Analysis of Encrypted Network Traffic 4
Client Identification and Fingerprinting 4
Transport Layer Security 5
Structure of Web Browser Communication 6
Structure of Android Applications Communication 8
Chapter 3: JA3 Fingerprinting for Web Browsers 9
Motivation 9
Preliminaries 9
TLS library 10
Random Values in TLS 10
JA3 Algorithm 12
Dataset 13
Web Browsers 13
Android Applications 15
JA3 Extension & Implementation 16
Application Architecture 16
Sequence Diagram 20
Chapter 4: Results 22
Web Browsers 22
Firefox 22
Google Chrome & Opera 24
Android Apps Fingerprinting 29
Lumen Analysis 29
Conclusion 30
References 31

List of Figures

Figure 1: Browser Architecture 7
Figure 2: Computing JA3 Hash 9
Figure 3 ClassifyJA3 System Architecture:..... 17
Figure 4: DNS Records Extraction Commands..... 18
Figure 5: TSHARK Fields and Filter..... 19
Figure 6: Sequence Diagram..... 20
Figure 7: Firefox Fingerprints 22
Figure 8: Database Fingerprint Entries 23
Figure 9: Firefox Identification, Kali Linux 23
Figure 10: Firefox Identification, Mac OS 23
Figure 11: Firefox Identification, Windows 23
Figure 12: Wireshark GREASE Fields 24
Figure 13: Google Chrome Ciphersuite Values..... 25
Figure 14: GREASE Values..... 26
Figure 15: Filtered Fields 26
Figure 16: Chrome / Opera Identification 29

List of Tables

Table 1: Popular Web Browsers 6
Table 2: JA3 hashes with and without GREASE values 11
Table 3: Browser Communication Dataset 14
Table 4: Mobile Application Versions 15
Table 5: Google Chrome Fingerprints 27
Table 6: Opera Browser Fingerprints 27
Table 7: Firefox Fingerprints 28
Table 8: Lumen Monitor Flows 30

Preamble

This technical report summarizes results of the research focused on identification of PC browsers and Android applications from network communication. I review several fingerprinting methods and highlight their advantages and disadvantages. I will then delve on web browser fingerprinting by examining the values from TLS handshake, HTTP headers and DNS traffic. I discuss the reliability and stability of this multi-OS profiling of web browsers.

Chapter 3 focuses on JA3, a specific method for creating TLS fingerprints in an easy to produce and shareable way. Based on previous research, I conduct various experiments using datasets obtained from network communication to observe the reliability of this approach in profiling web browsers and Android applications.

This work was supported by Erasmus+ programme at Brno University of Technology in Czech Republic. The author acknowledges and appreciates Ing. Petr Matousek, Ph.D., M.A. for overall guidance in conducting the research, and Strathmore University (Kenya) for granting this chance.

Chapter 1: Introduction

Transport Layer Security (TLS) provides security in the form of encryption to all manner of network connections, including browsers and mobile applications. Those using TLS operate under the assumption that although an eavesdropper can easily observe the existence of their session, its source and destination IP addresses, that the content itself is secure and unreadable without access to cryptographic keying material at one or both ends of the connection. However, using TLS Fingerprinting, it is easy to quickly and passively determine which client is being used, and then to apply this information from both the attacker and the defender perspectives.

Lee Brotherston has discovered that during an SSL handshake, most client user agents will initiate a TLS handshake request in a unique way. This includes Web Browsers in different operating systems such as Linux, Mac OS and Windows, and Android Mobile Applications. The fingerprint relies on data from ClientHello messages in the SSL handshake. Inspired by Lee's work, a team from Salesfoce (John B. Althouse, Jeff Atkinson & Josh Atkins) created JA3, a standard for creating SSL client fingerprints in an easy to produce and shareable way.

The proposed method does not require active interaction with the browser or a mobile application. The communication snapshot can be obtained by simple passive data capturing using, e.g., Wireshark. The approach assumes that each browser / mobile app differs in versions, settings, and implementations that keep traces in network traffic. By profiling, relevant data from the captured network traffic is extracted and used to build a profile of the device. Captured communication usually includes both user-initiated communication, e.g., web browsing, and system/application-initiated communication, e.g., connectivity tests, regular updates, service synchronization, etc. Both types of communication are valuable sources of features for building a device profile.

In this project, a structure of web browser traffic is described. The researcher shows what protocols can be exploited to obtain fingerprinting data. Using the combination of different identification techniques, a browser & mobile app profile that can be used for identification of a given web browser or mobile app in the network traffic is created.

Definitions

In this part, two important terms related this work are defined: Profiling and fingerprinting.

Profiling is the process of "discovering" correlations between data in databases that can be used to identify and represent a human or nonhuman subject (individual or group), and/or the application of profiles (sets of correlated data) to individuate and represent a subject or to identify a member of a group or category. Data mining technology is generally considered as a means by which relevant patterns are discovered and profiles are generated from larger quantities of data (WP6, 2008).

Fingerprinting is a method for collecting publicly available information called attributes or features about a remote computing device for the purpose of identification. The data forms a digital fingerprint of the remote device. Fingerprints can be used to fully or partially identify individual users or devices. Active fingerprinting requests a specific fingerprinting data from a remote device using querying, e.g., obtaining web browser parameters or network settings. Passive fingerprinting relies on data obtained by monitoring the communication of a remote device without interfering to it.

Chapter 2: Literature Review

Analysis of Encrypted Network Traffic

The rising popularity of encrypted network traffic is a double-edged sword. On the one hand, it provides secure data transmission, protects against eavesdropping, and improves the trustworthiness of communicating hosts. On the other hand, it complicates the legitimate monitoring of network traffic, including traffic classification and host identification. Nowadays, we can monitor, identify, and classify plain-text network traffic, such as HTTP, but it is hard to analyze encrypted communication (Husák. et al., 2016).

Researchers have identified the options of establishing SSL/TLS communication and options are used in real traffic. They use methods from a survey by Velan et al., (2015). as the basis and a real network data to identify these options. Then, they found which of the options are varying the most and if the variability of these options indicates different traffic patterns, e.g., different communicating partners or type of traffic.

Client Identification and Fingerprinting

The client identification and browser fingerprinting contribute largely to network security and detection of malicious activities, e.g., by outdated system identification or unusual behavior detection. Identification and fingerprinting are moreover useful for commercial purposes (targeting ads, price discrimination, assessing financial credibility), network accounting, and client behavior monitoring (Bujlow et al., 2015).

Some researchers have used HTTP User-Agent string to identify clients, however this method is not reliable. There is a constant risk that the User-Agent string is forged. For instance, illegitimate web crawlers and bots typically spoof the User-Agent string as to be mistaken for legitimate ones such as Googlebot (Zeifman, 2012). Another reason why HTTP UA strings cannot be trusted is because since a long time ago web browsers have been including identifiers of each other to their User-Agent to resolve compatibility issues with certain web pages.

One of the viable approaches researched in the past for the identification of mobile devices based on the captured network communication is fingerprinting based on mobile device hardware. This method evaluates physical characteristics of the device: the image sensor, frequency response of the speaker-microphone system, an accuracy of the accelerometer, clock skew of GPS, touch screen misalignment, etc (Bojinov, et al., 2014). By this approach, we can identify a group of devices that have the same or similar hardware. Obtaining such fingerprint requires active communication with the device which is usually provided via a specifically tailored application that extracts all necessary data from the device. Passive network monitoring cannot easily obtain hardware features.

Transport Layer Security

Transport Layer Security (TLS) refers to cryptographic protocols designed to provide communications security over a computer network (Dierks & Rescorla, 2008). Prior to entering initiating encrypted communications, TLS needs to create a handshake between the client and server which is then used to select the best mutually acceptable cryptographic ciphers, compression systems, hashing algorithms, etc. This is conducted in the clear, because the method of cryptography to use has yet to be determined.

Using TLS Fingerprinting, it is easy to quickly and passively determine which client is being used, and then to apply this information from both the attacker and the defender perspectives (Lemon, 2015). A TLS connection will always begin with a Client Hello packet which announces to the server end of the connection the capabilities of the client, presented in preference order. The server will send back a similar packet, a “server hello” describing the server capabilities in preference order. By comparing the two packets, the client and server can determine the optimal cipher suites, compression algorithms, etc. to use per their preferences.

By capturing the elements of the Client Hello packet which remain static from session to session for each client, it is possible to build a fingerprint to recognise a client on subsequent sessions. The fields captured are TLS version, record TLS version, cipher suites, compression options, and a list of extensions. Additionally, data is captured from three specific extensions (if available): signature algorithms, elliptic curves and elliptic curve point format. The use of this combined data is not only

reliable in terms of remaining static for any particular client, but offers greater granularity than assessing cipher suites alone, which has a substantially larger quantity of fingerprint collisions.

Structure of Web Browser Communication

There are a lot of web browsers available in the market. The following are the most common web browser available today:

Table 1: Popular Web Browsers

Browser	**Vendor**
Internet Explorer	Microsoft
Google Chrome	Google
Mozilla Firefox	Mozilla
Netscape Navigator	Netscape Communications Corp.
Opera	Opera Software
Safari	Apple
Sea Monkey	Mozilla Foundation
K-meleon	K-meleon

The most basic component that all web browsers must exhibit are the following: **Controller/dispatcher** that works as a control unit in the CPU. It takes input from the keyboard or mouse, interprets it and makes other services to work based on input it receives. **Interpreter** receives the information from the controller and executes the instruction line by line. Some interpreters are mandatory while some are optional, for example, HTML interpreter program is mandatory and java interpreter is optional. **Client Program** describes the specific protocol that will be used to access a service. The most used client programs are HTTP, SMTP, FTP, NNTP, and POP.

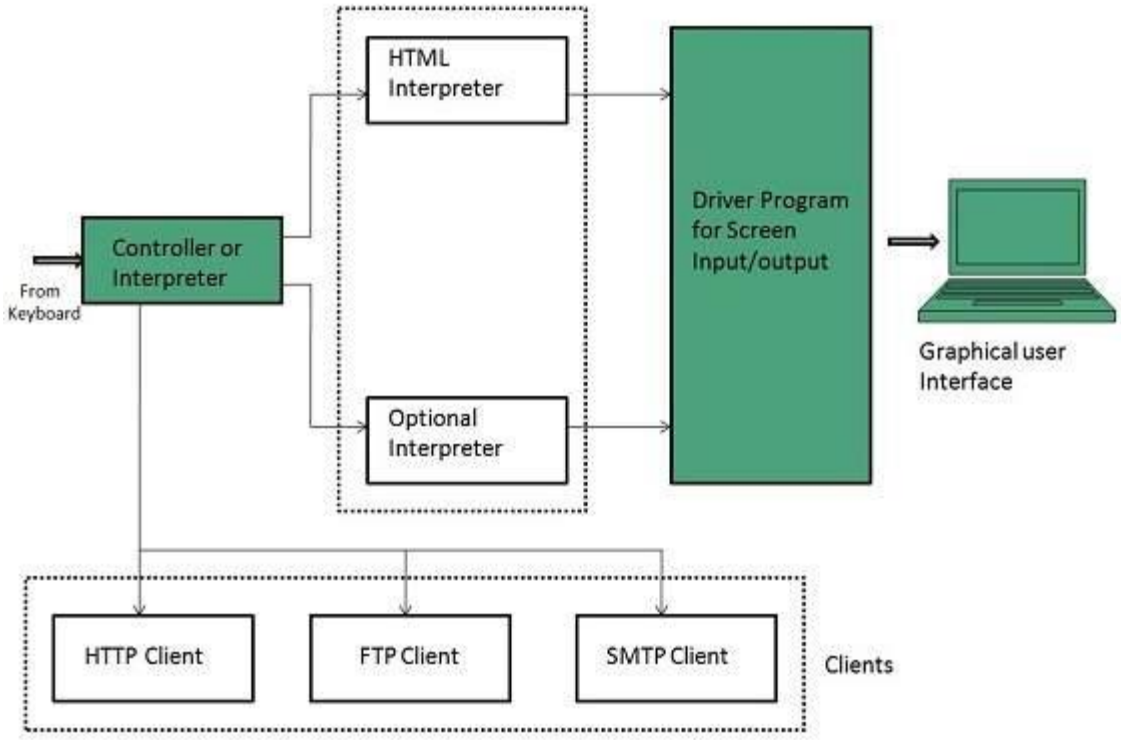


Figure 1: Browser Architecture

Web browsers communicate with web servers using the HyperText Transfer Protocol (HTTP). Web servers wait for client request messages, process them when they arrive, and reply to the web browser with an HTTP Response message. The response contains an HTTP Response status code indicating whether or not the request succeeded (e.g. "200 OK" for success, "404 Not Found" if the resource cannot be found, "403 Forbidden" if the user isn't authorised to see the resource, etc). The body of a successful response to a GET request would contain the requested resource.

When an HTML page is returned it is rendered by the web browser. As part of processing the browser may discover links to other resources (e.g. an HTML page usually references JavaScript and CSS pages), and will send separate HTTP Requests to download these files.

HTTPS (HyperText Transfer Protocol Secure) is an encrypted version of the HTTP protocol. It uses SSL or TLS to encrypt all communication between a client and a server. This secure connection allows clients to safely exchange sensitive data with a server, such as when performing banking activities or online shopping.

Structure of Android Applications Communication

Most network-connected Android apps use HTTP to send and receive data. The Android platform includes the *HttpsURLConnection* client, which supports TLS, streaming uploads and downloads, configurable timeouts, IPv6, and connection pooling. On devices running Android 9 and lower, the platform DNS resolver supports only A and AAAA records, which allow looking up the IP addresses associated with a name, but does not support any other record types. On devices running Android 10 and higher, there is native support for specialized DNS lookups using both cleartext lookups and DNS-over-TLS mode. The *DnsResolver* API provides generic, asynchronous resolution, enabling you to look up SRV, NAPTR, and other record types. Note that parsing the response is left to the app to perform.

Network operations are introduced on a separate thread to avoid creating an unresponsive user interface (UI). Since the *NetworkFragment* runs on the UI thread by default, it uses an *AsyncTask* to run the network operations on a background thread. This Fragment is considered headless because it does not reference any UI elements. Instead, it is only used to encapsulate logic and handle lifecycle events, leaving the host Activity to update the UI. Android applications use *HttpsUrlConnection* to fetch data. The application takes the given URL and use it to perform an **HTTP GET** request. Once a connection has been established, the application retrieves the data as an *InputStream*, which is then decoded and converted into a target data type such as image data.

Chapter 3: JA3 Fingerprinting for Web Browsers

Motivation

The primary concept for fingerprinting TLS clients came from Lee Brotherston's 2015 research (Lemon, 2015) where he discovered that it is possible to build a fingerprint to recognise a particular client on subsequent sessions by capturing the elements of the Client Hello packet which remain static from session to session for each client. This concept was implemented and expanded by John Althouse, Jeff Atkinson and Josh Atkins (2019) by creating an open source tool named JA3.

This research aimed to extend the JA3 work by making it possible to easily identify the type of web browser based on network communication. Three additional SSL handshake fields are introduced to make the data more informative. This includes time since reference or first frame, source IP address, and destination IP address.

Preliminaries

Most of TLS fingerprinting methods use the first packet sent by the client: Client Hello. The Client Hello contains an imprint of TLS configuration of the client application that depends on the used TLS library and operating system. In this paper we study JA3 fingerprint that is computed as MD5 hash from five TLS handshake fields: TLS Handshake version, Cipher suites, Extensions, Supported Groups (former Elliptic Curve), and Elliptic Curve point format, see Figure 2. Some TLS fingerprinting implementations use different TLS fields, e.g., Kotzias et al. (2018) omit the TLS version.

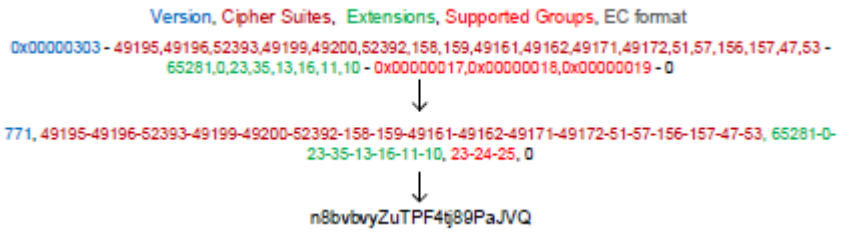


Figure 2: Computing JA3 Hash

TLS library

TLS fingerprint of an application depends on the TLS library that was used during implementation. There are plenty of TLS libraries available to developers, e.g., GnuTLS, Oracle JSSE, BSD LibreSSL, OpenSSL, or Mozilla NSS. When two applications are implemented using the same TLS library, it may happen that their TLS fingerprints are the same. TLS fingerprints can change with a new version of the application, version of the TLS library, or the operating system. Table 4.2 shows JA3 hashes for common web browsers: Mozilla Firefox v.73, Chrome v.80, and Opera v.66 under four operating systems: Linux Ubuntu, Windows 10, Kali Linux and Mac OS.

We can see that Firefox has four unique JA3 fingerprints. Two of them are present in all tested operating systems. In the case of Chrome and Opera, one JA3 fingerprint value corresponds to both browsers under all operating systems. These browsers were possibly compiled with the same TLS library.

	Firefox				Chrome				Opera			
JA3 hash	Ubuntu	Win	Kali	MacOS	Ubuntu	Win	Kali	MacOS	Ubuntu	Win	Kali	MacOS
0e6f3c8f2b18f3011f1d6cbbdcfcbd65						x				x		
1344ed2e9d7d8e3e84e6ab655047ba32	x	x	x	x								
1f3c530fc35e41300422550c3c980e85							x	x	x	x	x	x
4863015f73b8332cf91cfa3a14a4893d		x										
5a291b49748c50adf1da70f8142d4cc4					x				x			
756094f51da8214018fbfba93211d59f	x	x	x	x								
a839cfeed30d55439b09de5f1b47fa3a					x	x	x	x	x	x	x	x
d889531a0389787425d5638caf6d84b3					x	x	x	x	x	x	x	
d90d517f72e9b8af9a8c1e2fe1fb2da8	x			x								

JA3 hashes of Common Web browsers

This experiment proves that TLS fingerprints change with the version and operating system. More observations related to JA3 fingerprinting of web browsers are written in **Section 4.3**. Similar experiment over a larger dataset is also mentioned by Razaghpanah, et al., (2017).

Random Values in TLS

In 2016, Google started to Generate Random Extensions and Sustain Extensibility (GREASE) values to TLS. This technique was adopted by IETF in January 2020 as RFC 8701 [5]. GREASE

values are randomly generated numbers of cipher suites, extensions and supported groups present in TLS Hello packets. They prevent extensibility failures in the TLS ecosystem. During TLS handshake, the responding side must ignore unknown values. Peers that do not ignore unknown values fail to inter-operate which means a bug in implementation. Therefore, RFC 8701 adds GREASE values as a part of the list of cipher suites, extensions and supported groups to detect invalid implementations.

When experimenting with Opera browser under Win 10 we noticed that the browser generates 155 unique JA3 fingerprints out of 207 TLS handshakes. By excluding GREASE values, the number of unique JA3 fingerprints decreased to four. The high number of JA3 fingerprints was caused by random GREASE values in TLS handshakes. Table 4.3 shows six JA3 fingerprints of Opera browsers under Ubuntu with all extracted TLS values (the upper six lines). The last six lines present TLS values without GREASE values. The brown values in the upper table represent GREASE values as defined in RFC 8701. When ignoring these values, the last four lines in the upper table would have the same JA3 hash.

Table 2: JA3 hashes with and without GREASE values

List of Cipher Suites	List of Extensions	Supported Group	JA3 hash
49199-49200-49195-49196-52392-52393-49171-49161-49172-49162-156-157-47-53-49170-10	13172-0-5-10-11-13-65281-16-18	29-23-24-25	839868ad711dc55bde0d37a87f14740d
49199-49200-49195-49196-52392-52393-49171-49161-49172-49162-156-157-47-53-49170-10	13172-0-5-10-11-13-65281-16-18	29-23-24-25	839868ad711dc55bde0d37a87f14740d
56026-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10	60138-0-23-65281-10-11-35-16-5-13-18-51-45-43-27-19018-21	35466-29-23-24	ee972d7d47ec01a9cb9b04efb7346e32
60138-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10	39578-0-23-65281-10-11-35-16-5-13-18-51-45-43-27-56026-21	23130-29-23-24	cb4415a180704432d2e3f70f8dca5783
31354-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10	47802-0-23-65281-10-11-35-16-5-13-18-51-45-43-27-51914	43690-29-23-24	74a57a5f55ce2c9fa637b1f4567308b4
14906-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10	31354-0-23-65281-10-11-35-16-5-13-18-51-45-43-27-43690	56026-29-23-24	a10f93ffdc89d383db0f4437a0530569
49199-49200-49195-49196-52392-52393-49171-49161-49172-49162-156-157-47-53-49170-10	13172-0-5-10-11-13-16-18	29-23-24-25	5a291b49748c50adf1da70f8142d4cc4
49199-49200-49195-49196-52392-52393-49171-49161-49172-49162-156-157-47-53-49170-10	13172-0-5-10-11-13-16-18	29-23-24-25	5a291b49748c50adf1da70f8142d4cc4
4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10	0-23-10-11-35-16-5-13-18-51-45-43-27	29-23-24	a839cfeed30d55439b09de5f1b47fa3a
4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10	0-23-10-11-35-16-5-13-18-51-45-43-27	29-23-24	a839cfeed30d55439b09de5f1b47fa3a
4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10	0-23-10-11-35-16-5-13-18-51-45-43-27	29-23-24	a839cfeed30d55439b09de5f1b47fa3a
4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10	0-23-10-11-35-16-5-13-18-51-45-43-27	29-23-24	a839cfeed30d55439b09de5f1b47fa3a

In addition to GREASE values, it is also good to omit extension value 65281 from TLS fingerprinting. This value represents a renegotiation option in TLS handshake (Rescorla, et al., 2010), see red numbers in the list of extensions. The last option that can be ignored is the TLS Client Hello Padding Extension defined by RFC 7685 (Postel, 1980). The padding extension (value 21, depicted by green value in the table) is added by a client to make sure that the packet is of a desired size.

In order to keep JA3 fingerprints stable, it is necessary to eliminate above mentioned values. Most JA3 implementations usually exclude GREASE values from TLS fingerprinting.

JA3 Algorithm

The JA3 method is used to gather the decimal values of the bytes for the following fields in the Client Hello packet:

1. **TLSVersion**: The version of the TLS protocol by which the client wishes to communicate during this session. This SHOULD be the latest (highest valued) version supported by the client. For example, 00000303.
2. **Ciphers**: This is a list of the cryptographic options supported by the client, with the client's first preference first. If the session_id field is not empty (implying a session resumption request), this vector MUST include at least the cipher_suite from that session. For example, 4865-4867-4866-49195-49199-52393-52392-49196-49200-49162-49161-49171-49172-51-57-47-53-10.
3. **Extensions**: Clients MAY request extended functionality from servers by sending data in the extensions field. For example, 0-23-65281-10-11-35-16-5-51-43-13-45-28-21.
4. **EllipticCurves** and **EllipticCurvePointFormats**: These allow negotiating the use of specific curves and point formats (e.g., compressed vs. uncompressed, respectively) during a handshake starting a new session. These extensions are especially relevant for constrained clients that may only support a limited number of curves or point formats. The client enumerates the curves it supports, and the point formats it can parse by including the appropriate extensions in its ClientHello message. The server similarly enumerates the point formats it can parse by including an extension in its ServerHello message. For example, 0000001d 00000017 00000018 00000019 00000100 00000101 0

The hexadecimal values in tls-version and elliptic-curves are converted to decimal. It then concatenates those values together in order, using a “,” to delimit each field and a “-” to delimit each value in each field. The resulting string looks like below:

771,4865-4867-4866-49195-49199-52393-52392-49196-49200-49162-49161-49171-49172-51-57-47-53-10,0-23-65281-10-11-35-16-5-51-43-13-45-28-21,29-23-24-25-256-257,0

This string is then hashed using md5, and a fixed length hash value is generated. The above string produces the hash below:

B20b44b18b853ef29ab773e921b03422

This is the JA3 fingerprint for one client-hello packet. If a similar browser is used it will have the same client-hello message, leading to a positive identification through matching the hashes. This technique is what is referred to as fingerprinting.

Dataset

Web Browsers

The data used in this project consisted of captured network communication (PCAP) files from different browsers in various operating systems. While capturing the traffic, specific domains / URLs were accessed in all browsers to ensure that communication can be reliably filtered by analysing DNS records. These include:

1. superuser.com/questions/247127/what-is-and-in-linux/247131
2. linuxsig.org/files/bash_scripting.html
3. strathmore.edu
4. vutbr.cz/en
5. facebook.com
6. adobe.com
7. amazon.com
8. bitbucket.org/dashboard/overview
9. forums.kali.org
10. offensive-security.com

The packets relating to these URLs were identified by examining DNS records. Corresponding IP addresses were gathered and used to filter tls.hello packets. These IP addresses include:

198.57.179.99 147.229.2.82 147.229.2.90 157.240.30.35 184.51.8.147 2.18.68.206 52.30.88.10 34.253.101.66 34.249.165.210 52.48.145.94 34.255.235.176 54.171.79.102 52.19.22.175 34.248.108.242 52.45.117.194 52.205.228.94 34.232.204.33 34.232.6.198 52.5.59.12 54.174.68.95 52.7.248.149 52.7.114.31 52.46.135.211 52.46.141.49 54.239.26.255 99.86.240.33 99.86.241.241 18.205.93.1 18.205.93.2 18.205.93.0 192.124.249.12 192.124.249.5

The table 2 describes the captured data in detail:

Table 3: Browser Communication Dataset

			All IP Addresses			Filtered IP Addresses				
Browser	**Operating System**	**Broswe Version**	**Packets #**	**tls.hello Packets**	**Unique Fingerprints**	**Packets #**	**tls.hello Packets**	**Unique FInger prints**	**File Size (Bytes)**	**Timestamp**
Google Chrome	Kali Linux	80.0.3987.106	12013	179	174	4730	33	31	18379776	Feb 17 21:19
Google Chrome	Mac OS	80.0.3987.106	26914	155	149	4431	16	16	20612464	Feb 18 13:15
Google Chrome	Ubuntu	80.0.3987.132	5835	76	73	260	6	6	3368680	Mar 11 14:57
Google Chrome	Windows	80.0.3987.106	19686	180	17	10739	41	11	19693784	Feb 17 15:50
Mozilla Firefox	Kali Linux	68.2.0esr	9864	164	3	3307	36	3	12880012	Feb 18 14:46
Mozilla Firefox	Mac OS	73	20217	194	5	4634	29	3	13399464	Feb 18 13:15
Mozilla Firefox	Ubuntu	73.0.1	20474	183	4	6584	20	3	14374332	Mar 11 15:14
Mozilla Firefox	Windows	70.0.2	18014	220	4	8516	42	4	16870532	Feb 17 15:50
Opera	Kali Linux	66.0.3515.72	12558	191	186	2968	30	28	17763452	Feb 17 20:43
Opera	Mac OS	66.0.3515.72	28300	198	192	8016	27	27	20853084	Feb 18 13:15
Opera	Ubuntu	67.0.3575.53	22053	204	199	6184	18	18	16335228	Mar 11 15:01
Opera	Windows	66.0.3515.95	8445	117	8	5315	22	4	7969388	Feb 17 15:50
Opera	Windows	67.0.3575.53	20344	207	200	5991	25	23	17217192	Mar 5 14:54

Android Applications

Capture Files

The last dataset was focused on variety of mobile apps installed on a real device Tecno J8 with Android 5.1. Dataset includes following apps: BoomPlay Music, Chrome Browser, Equa Bank app, Facebook app, Gmail app, Google calender, KB klic, Messenger, Mobilni Banka app, NextBike, Telegram, TikTok, WhatsApp and Youtube app. Each app was running five times on the restarted device so that captured communication corresponds to a typical usage. 5,308 TLS handshakes were extracted from the captured traffic. The above-mentioned datasets were used for experiments with JA3 and JA3S fingerprints.

Lumen Monitoring App

To capture mobile apps traffic, the researcher evaluated tool Lumen that was created by Int. computer Science Institute in University of California, Berkeley and IMDEA Networks Institute, Madrid, Spain. Lumen App is available through Google store1. Lumen is a tool that helps you to keep user personal data under control and obtain network traffic logs. It analyses the app's traffic to identify personal information leaks and the organizations collecting such sensitive data.

For monitoring, 25 mobile applications on different versions were used, see table 3. During experiments we noticed that 88.7% traffic was transmitted by HTTPs, 4,1% was XMPP, 3,1% was HTTP, and 4.1% other traffic. Communication of 8.951 connections to 288 unique IP addresses was analysed, occupying around 811 MB storage size.

Table 4: Mobile Application Versions

App	Version (Exp1)	Version (Exp2)	Version (Exp3)
FMWhatsApp 2	2.19.291	2.20.123	-
Gmail	2020.04.12.307915656.release	-	2020.04.26.310266462.release
Google Play Store	19.7.12-all [0] [PR] 305919187	20.0.15-all [0] [PR] 309479531	20.1.17-all [0] [PR] 310643216
PHX Browser	4.7.2.2330	4.8.3.2355	4.9.2.2370
Chrome	81.0.4044.117	81.0.4044.138	83.0.4103.83
TikTok	15.5.5	16.0.42	16.2.4
nextbike	v4.7.4	-	v4.8.1

Google Backup Transport	5.1-1743759		
Google	11.6.8.21.arm64		
Maps	10.40.1		
YouTube	15.17.38	15.19.34	15.20.33
Messenger	261.0.0.21.119	261.0.0.23.120	265.0.0.24.107
Equa bank	16.1.0	16.2.0	-
Boomplay	5.7.5	-	
Duolingo	4.61.0		4.65.1
Weather	1		
KB Klic	2.6.1	-	-
Mobilni banka	6.3.2	-	-
Telegram	6.1.1	-	-
Facebook	267.1.0.46.120	270.1.0.66.127	271.0.0.55.109

JA3 Extension & Implementation

The tool consists of a shell script that processes PCAP files to compute JA3 fingerprints of known web browsers. Tshark commands are used to extract the relevant fields from the Client Hello packets. Unix string manipulation commands parse the fields to prepare for fingerprint generation / hashing. Computed fingerprints are saved into CSV files so that unknown PCAP files can be compared.

Application Architecture

The shell script implementation includes various steps intended to extend the JA3 functionality by being able to identify the web browsers used in a communication. Various command-based tools are used to read and analyse PCAP files in order to reveal the browser identity. Figure 1 illustrates the various components of the tool.

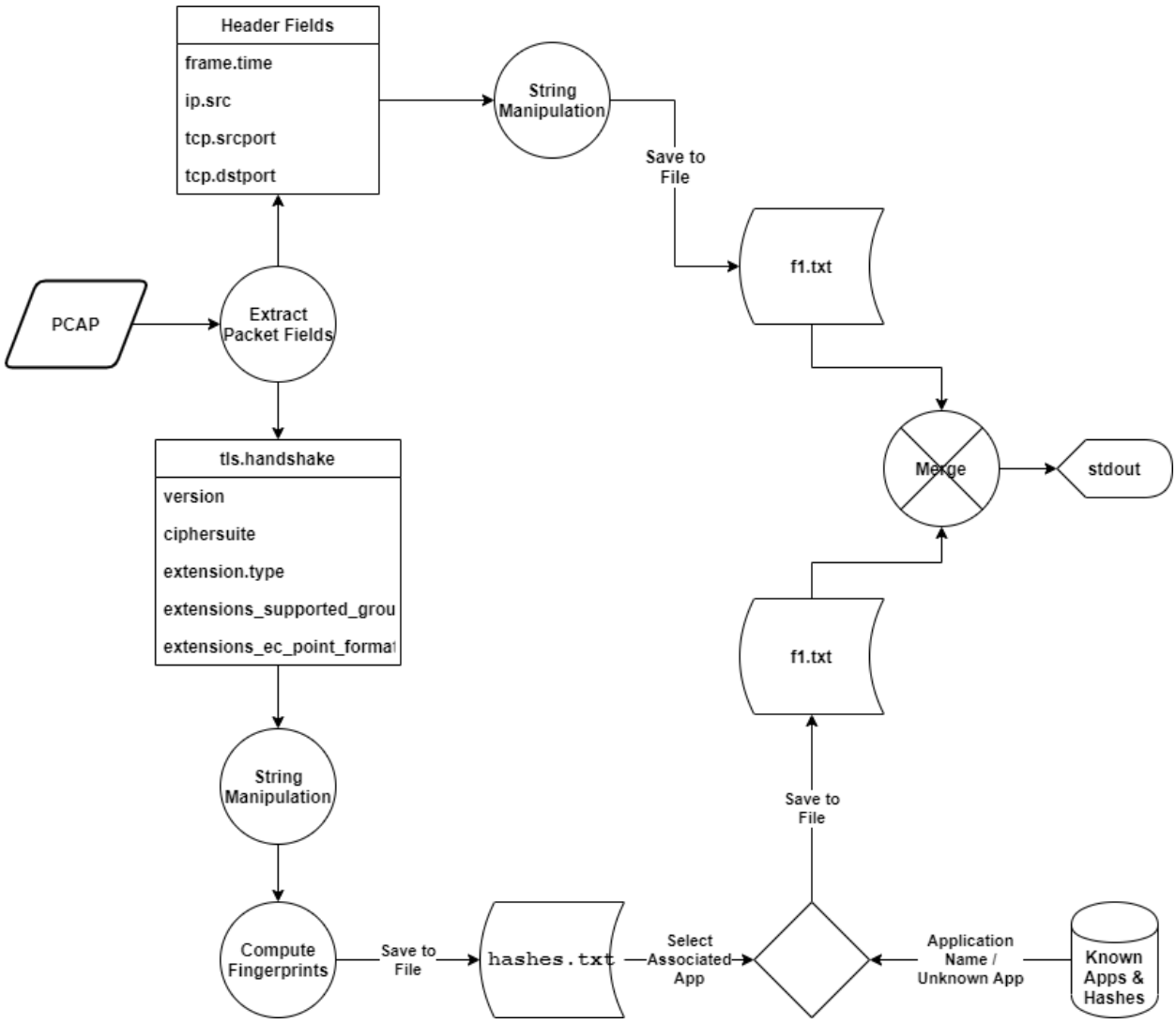


Figure 3 ClassifyJA3 System Architecture:

The first step uses TSHARK to extract the following packet fields from a PCAP file:

A. frame.time - This is the frame arrival time
B. ip.src - This is the Internet Protocol Version 4 address source
C. tcp.srcport - This is the Transmission Control Protocol Source Port
D. tcp.dstport - This is the Transmission Control Protocol Destination Port

In order to fingerprint only packets to known destinations, packets representing noise from other apps should not be examined. A full packet capture includes traffic to / from many destinations, including Operating System, background applications and other running apps communicating with

remote services. Since this project is only focused with browser TLS fingerprinting, traffic from / by other applications / services should be eliminated. However, browser traffic includes communication by browser plugins, advertisements, and other remote services not explicitly initiated by the user. This should also be eliminated to remain with communication to destinations initiated by the user. This will help at ensuring the fingerprints are clean and possible to identify across different operating systems or browser versions. This also has a secondary benefit of minimising the size of the dataset to be analysed, hence increasing the tool efficiency.

This is achieved by filtering traffic based on known DNS destinations. “A” records are analysed and IP addresses matched with known domain names. TLS Client Hello packets to the identified destinations are extracted and fingerprints generated. The steps below were followed to achieve this:

Step 1: Identify a set of URLs to use and run them in a browser. Capture and save traffic using Wireshark. Use Bulk URL Opener browser plugin to load multiple URLs at once.

Step 2: Extract DNS A records and DNS response names from the PCAP files and save the results in CSV files. Combine these in a single CSV file. The commands in figure 4 achieve these:

```
${TSHARK} -r "${INFILE}" -T fields -e dns.a > dns_a.csv
${TSHARK} -r "${INFILE}" -T fields -e dns.resp.name > dns_resp_name.csv
paste -d "," dns_a.csv dns_resp_name.csv | sort | uniq > dnsAllRcsUniq.csv
```

Figure 4: DNS Records Extraction Commands

Step 3: Search for the DNS response names for the domains identified, and match with corresponding IP addresses. Extract the IP addresses and use them inside the TSHARK filter, as shown in figure 5:

```
${TSHARK} -r "${INFILE}" -T fields \
    -e ip.dst \
    -e tls.handshake.version \
    -e tls.handshake.ciphersuite \
    -e tls.handshake.extension.type \
    -e tls.handshake.extensions_supported_group \
    -e tls.handshake.extensions_ec_point_format \
    -R "tls.handshake.type==1 and ip.addr in {198.57.179.99 147.229.2.82 147.229.2.90
        157.240.30.35 184.51.8.147 2.18.68.206 52.30.88.10 34.253.101.66 34.249.165.210
        52.48.145.94 34.255.235.176 54.171.79.102 52.19.22.175 34.248.108.242 52.45.117.194
        52.205.228.94 34.232.204.33 34.232.6.198    52.5.59.12 54.174.68.95 52.7.248.149
        52.7.114.31 52.46.135.211 52.46.141.49 54.239.26.255 99.86.240.33 99.86.241.241
        18.205.93.1 18.205.93.2 18.205.93.0 192.124.249.12 192.124.249.5}" -2
```

Figure 5: TSHARK Fields and Filter

The extracted fields are then processed using awk tool and saved into a text file f1.txt for subsequent merging with browser name information.

The next major step is to calculate JA3 fingerprints, hashed from the following packet fields:

A. tls.handshake.version - This is the Transport Layer Security version,
B. tls.handshake.ciphersuite - This is the supported Transport Layer Security version cipher suite,
C. tls.handshake.extension.type - This is the Transport Layer Security extension type,
D. tls.handshake.extensions_supported_group - This is the Transport Layer Security supported group, and
E. tls.handshake.extensions_ec_point_format - This is the Transport Layer Security elliptic curve point format

These packet fields are extracted from Client Hello messages that are not encrypted. Research shows that different applications generate these fields in unique ways. In order to generate a JA3 equivalent fingerprint, the fields are processed using string manipulation tools such as awk and sed. This processing includes replacing all commas with dashes and converting hexadecimal fields to decimals. Each line is then md5'd and output of fixed length hashes saved into a text file, hashes.txt. The main reason behind MD5 usage is to ensure the fingerprints are short. Also, given the limited data set, hash collisions are not a concern here.

The generated fingerprints (hashes.txt) are then compared with static verified fingerprints in the database, see appendix 1. For each fingerprint in hashes.txt, an SQL query checks the database for matching records and returns the application name (such as Firefox, Chrome, Opera,...), otherwise it returns *UnknownApp* value and saves the output to a text file, f2.txt.

The last step combines the contents of f1.txt and f2.txt to show which we browsers were used to generate each packet.

Sequence Diagram

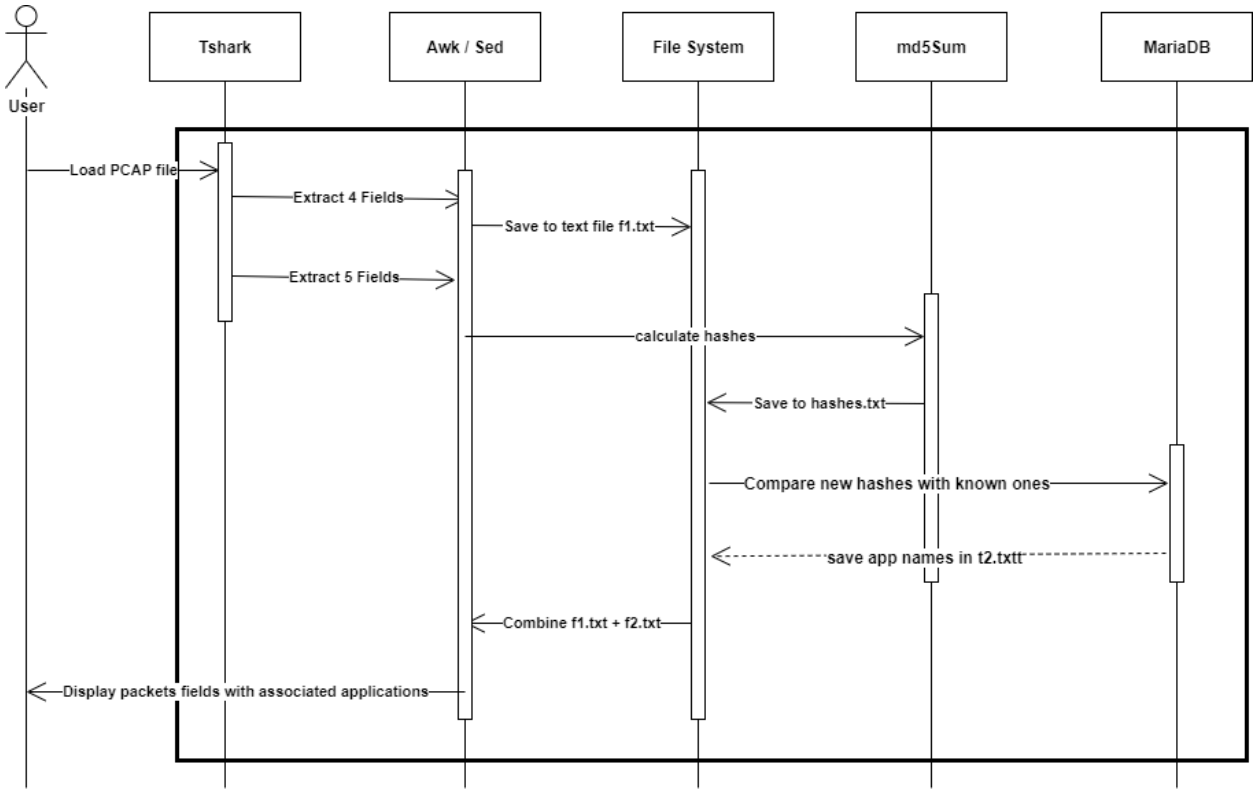


Figure 6: Sequence Diagram

Load PCAP file: The user runs the shell script with a PCAP file as a parameter, which is then loaded into a bash variable $1. If the file cannot be read (due to various reasons such as inexistence), the script returns an error and variable $1 is not initialised.

Extract 4 fields: TShark is a network protocol analyzer that is used by the script to read packets from a previously saved capture file (PCAP). Tshark extracts the following fields from the capture file: frame.time, ip.src, tcp.srcport, and tcp.dstport.

Save to text file f1.txt: Awk and sed are Unix utilities that parse and transform text on an input stream (a file or input from a pipeline). The script passes data from Tshark through a pipe, which is formated and saved into a text file.

Extract 5 fields: Tshark extracts the following fields from the capture file: frame.time, ip.src, tcp.srcport, and tcp.dstport.

Calculate hashes: MD5 message-digest algorithm is to hash the lines, producing a 128-bit hash value.

Save to hashes.txt: The set of 128-bit hash values are saved to a text file for further processing.

Compare new hashes with known ones: An SQL query is used to match each fingerprint to a pre-calculated set and fetch the corresponding application name. If a match is not found, the query returns the “Unknown App” string.

Save app names in t2.txt: The query results above are saved in a text file.

Combine f1.txt + f2.txt: f1.txt contains four packet fields from the communication, with each row corresponding to a row of application name in f2.txt.

Display packet fields with associated applications: A combination of f1 and f2 text files shows which web browser was used in the communication.

Chapter 4: Results

Web Browsers

Firefox

Browser identification was done for Firefox communication across four operating systems i.e. Kali Linux, Mac OS, Ubuntu and Windows. The browser version was different across the three operating systems.

JA3 fingerprints were generated from known PCAP files for Firefox. To ensure that noise is not fingerprinted, DNS analysis was conducted whereby packets in the PCAP file were filtered based on the destination IP addresses of the domains entered during the traffic capture. DNS A records were matched with corresponding DNS response names in order to identify the destination IP addresses of selected domains. These were identified and all client hello packets with such destination IPs were fingerprinted. Unique fingerprints across the four operating systems were identified. As at now, only Firefox had matching fingerprints, as shown in figure 7.

firefox-klinux-68_2_0esr	firefox-mac-73_0	firefox-windows-70_0_2	firefox-ubuntu-73_0_1
334da95730484a993c6063e36bc90a47	334da95730484a993c6063e36bc90a47	00ad5210a79e1f541e5649c7809e2fcc	334da95730484a993c6063e36bc90a47
b20b44b18b853ef29ab773e921b03422	b20b44b18b853ef29ab773e921b03422	334da95730484a993c6063e36bc90a47	b20b44b18b853ef29ab773e921b03422
d470a3fa301d80227bc5650c75567d25	d470a3fa301d80227bc5650c75567d25	b20b44b18b853ef29ab773e921b03422	d470a3fa301d80227bc5650c75567d25
		f557b8e5f8ed6c8bcd8bfb8efb300a94	

Figure 7: Firefox Fingerprints

The matching fingerprint records were inserted into a database table, which will be queried every time a new PCAP needs to identify the browsers used in communication. Figure 8 shows the database table.

```
MariaDB [fingerprints]> SELECT * FROM ja3;
+--------+---------+------------+-----------+----------------------------------+
| ja3_id | app     | version    | os        | fingerprint                      |
+--------+---------+------------+-----------+----------------------------------+
|      1 | Firefox | Kali Linux | 68_2_0esr | 334da95730484a993c6063e36bc90a47 |
|      2 | Firefox | Kali Linux | 68_2_0esr | b20b44b18b853ef29ab773e921b03422 |
|      3 | Firefox | Mac OS     | 73_0      | 334da95730484a993c6063e36bc90a47 |
|      4 | Firefox | Mac OS     | 73_0      | b20b44b18b853ef29ab773e921b03422 |
|      5 | Firefox | Windows    | 70_0_2    | 334da95730484a993c6063e36bc90a47 |
|      6 | Firefox | Windows    | 70_0_2    | b20b44b18b853ef29ab773e921b03422 |
|      7 | Firefox | Ubuntu     | 73_0_1    | 334da95730484a993c6063e36bc90a47 |
|      8 | Firefox | Ubuntu     | 73_0_;    | b20b44b18b853ef29ab773e921b03422 |
+--------+---------+------------+-----------+----------------------------------+
8 rows in set (0.001 sec)
```

Figure 8: Database Fingerprint Entries

Firefox was identified in the communication from the three operating systems, as shown in figures 8, 9 and 10.

```
Feb17,202014:24:12.400466043EST 192.168.10.163  60208   443     Firefox
Feb17,202014:24:12.403280439EST 192.168.10.163  56270   443     Firefox
Feb17,202014:24:12.436651668EST 192.168.10.163  33074   443     Firefox
Feb17,202014:24:12.438070787EST 192.168.10.163  56272   443     UnknownApp
Feb17,202014:24:12.484240455EST 192.168.10.163  53360   443     Firefox
Feb17,202014:24:12.611227391EST 192.168.10.163  58028   443     Firefox
```

Figure 9: Firefox Identification, Kali Linux

```
Feb18,202006:12:12.265326000EST 147.229.183.34  60142   443     Firefox
Feb18,202006:12:12.312673000EST 147.229.183.34  60143   443     Firefox
Feb18,202006:12:12.685234000EST 147.229.183.34  60144   443     Firefox
Feb18,202006:12:12.689932000EST 147.229.183.34  60149   443     Firefox
Feb18,202006:12:12.691948000EST 147.229.183.34  60148   443     Firefox
```

Figure 10: Firefox Identification, Mac OS

```
Feb17,202008:29:03.036663000EST 10.10.10.114    50135   443     UnknownApp
Feb17,202008:29:03.060409000EST 10.10.10.114    50143   443     Firefox
Feb17,202008:29:03.060999000EST 10.10.10.114    50139   443     UnknownApp
Feb17,202008:29:03.061375000EST 10.10.10.114    50144   443     Firefox
Feb17,202008:29:03.063279000EST 10.10.10.114    50147   443     Firefox
Feb17,202008:29:03.064063000EST 10.10.10.114    50146   443     Firefox
```

Figure 11: Firefox Identification, Windows

Google Chrome & Opera

Browser identification was done for Google Chrome & Opera communication across three operating systems i.e. Kali Linux, Mac OS and Windows. The browser version was different across Kali Linux, Ubuntu, Mac OS and Windows operating systems. Unique fingerprints could not be identified across these browsers using the implemented JA3 method.

This led to deep examination of Client Hello fields used in JA3 fingerprinting and identifying the differences in comparison to Firefox. Google Chrome PCAP files were analysed using Wireshark. It was noted that the tls.handshake.ciphersuite field was different. It contains Cipher Suite: Reserved (**GREASE**) (0x9a9a) which is not in Firefox. GREASE, which is an acronym for "Generate Random Extensions And Sustain Extensibility", is a new mechanism designed by Google for TLS to catch incompatibility issues. GREASE was added to Chrome in Version 55. Figure 12 shows this:

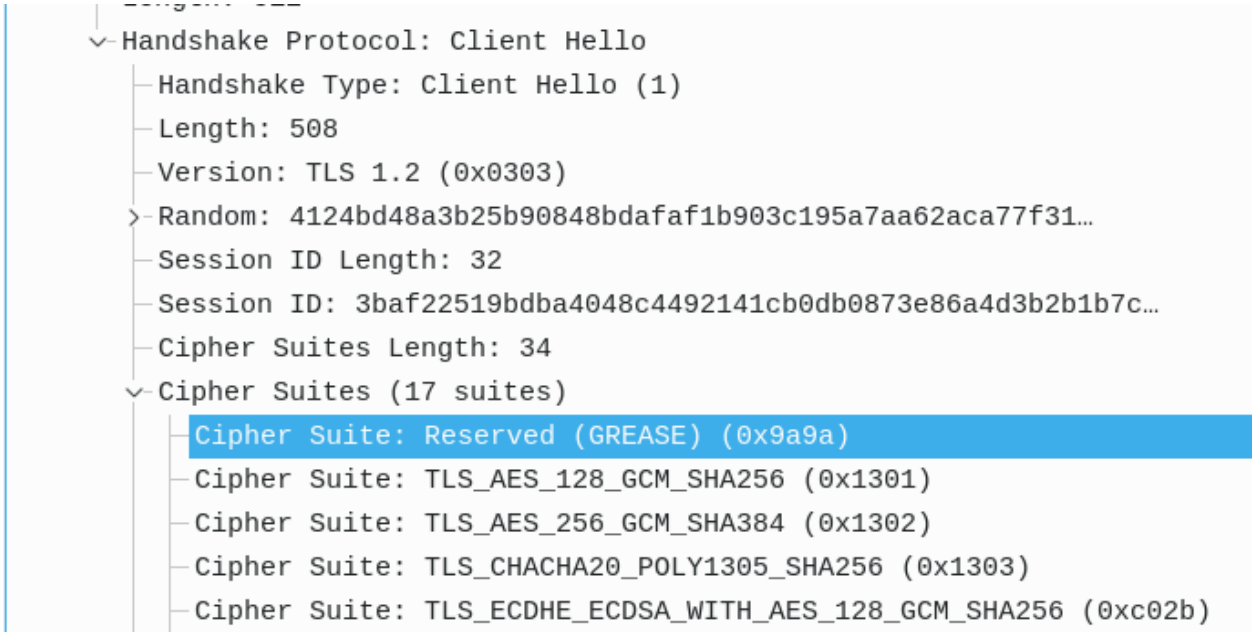


Figure 12: Wireshark GREASE Fields

The values change randomly across packets as shown in the first part of strings in figure 13:

chrome-klinux-ciphersuite
10794-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10
14906-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10
19018-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10
23130-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10
2570-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10
27242-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10
31354-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10
35466-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10
39578-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10
43690-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10
47802-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10
51914-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10
56026-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10
60138-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10
64250-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10
6682-4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53-10

Figure 13: Google Chrome Ciphersuite Values

GREASE values were also seen in tls.handshake.extension.type and tls.handshake.extensions_supported_group Client Hello fields.

It has been proved that the hexadecimal numbers in GREASE are random and change every time a page is refreshed. This explains the instability of fingerprints that we observed across different browser sessions and operating systems. The results were similar for Opera browser because it is built on the Chromium and Blink engine just like Chrome.

Modified Technique for Chrome / Opera Fingerprinting

It is still important to be able to fingerprint Chrome / Opera browsers. Because GREASE has been found to introduce random values, its occurrences in the Client Hello messages will be eliminated in the respective fields and fingerprints generated without it. The screenshot below (figure 14) illustrates the fields to be removed, so that dataset resembles that of Firefox as indicated:

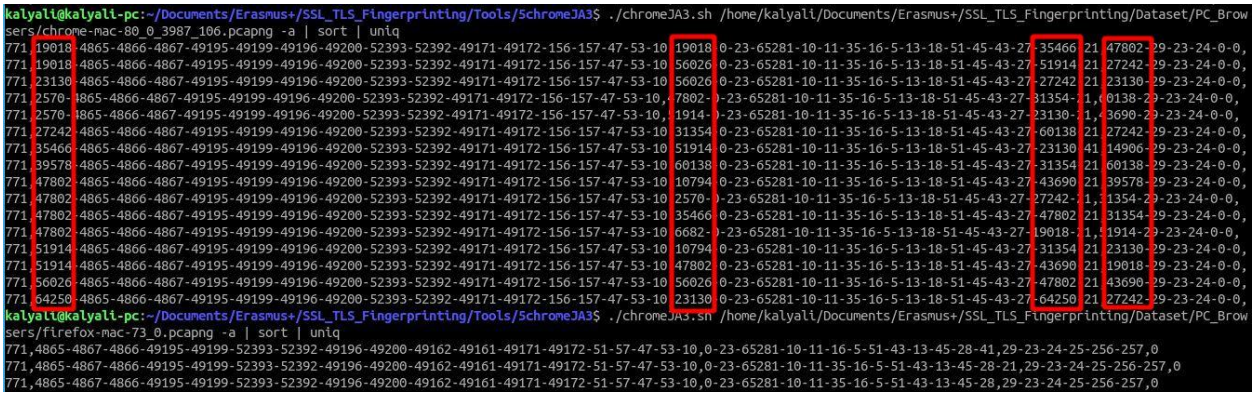


Figure 14: GREASE Values

Tshark was used to extract the five Client Hello fields. These were processed, manipulated and saved as comma separated values. GREASE related fields were removed and finally md5 hash values calculated for each record. Figure 15 shows two sets of records; one with full fields and the other without GREASE fields:

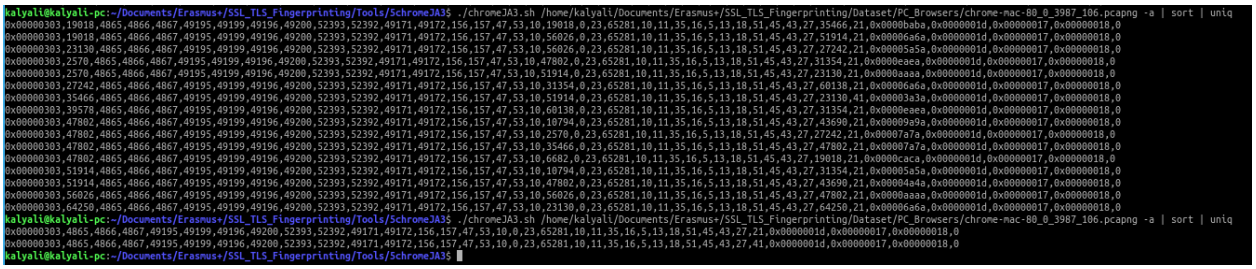


Figure 15: Filtered Fields

A significant decrease in unique records indicate that GREASE values are quite random, and are different for communications with the same host. Elimination of these values gives a more consistent flow, which increases the chances of effective fingerprinting.

Note: Fields with hexadecimal values were not converted to decimal, as required in JA3 method.

This process was done for Google Chrome PCAP files from Windows, Mac OS, Kali Linux and Ubuntu operating Systems. Initial tests show that one fingerprint (9ff0023372e249c161e03a71055216ca) is unique for Google Chrome across all the operating systems under review, see table 3:

Table 5: Google Chrome Fingerprints

chrome-klinux-80_0_3987_106	**chrome-mac-80_0_3987_106**	**chrome-windows-80_0_3987_106**	**chrome-ubuntu-80_0_3987_132**
7ea9c5678db69497cac5ea5efedbbcf3	7ea9c5678db69497cac5ea5efedbbcf3	29928ea197b2dd37dbdc3144040d3bb9	9ff0023372e249c161e03a71055216ca
8551ff8dc5d6c8379e28cf8ecfdbf7e9	9ff0023372e249c161e03a71055216ca	664c79a566a55e42851b76c6e245915b	
9ff0023372e249c161e03a71055216ca		9ff0023372e249c161e03a71055216ca	

Opera Browser:

PCAP files from Opera browser belonging to Windows, Mac OS, Kali Linux and Ubuntu operating Systems were analysed and indicated a common unique fingerprint (9ff0023372e249c161e03a71055216ca) across the four operating systems as indicated in the table 4:

Table 6: Opera Browser Fingerprints

opera-klinux-66_0_3515_72	**opera-mac-66_0_3515_72**	**opera-ubuntu-67_0_3575_53**	**opera-windows-66_0_3515_95**
3534a4d8fcabf5fee54ed010c34b45a0	7ea9c5678db69497cac5ea5efedbbcf3	29928ea197b2dd37dbdc3144040d3bb9	664c79a566a55e42851b76c6e245915b
7ea9c5678db69497cac5ea5efedbbcf3	9ff0023372e249c161e03a71055216ca	6a7f738f44c5ad879841dbe0f688fb66	9ff0023372e249c161e03a71055216ca
9ff0023372e249c161e03a71055216ca	eeb35a05bfa15e7b7dc92f84e3cc3fd7	7ea9c5678db69497cac5ea5efedbbcf3	
		9ff0023372e249c161e03a71055216ca	

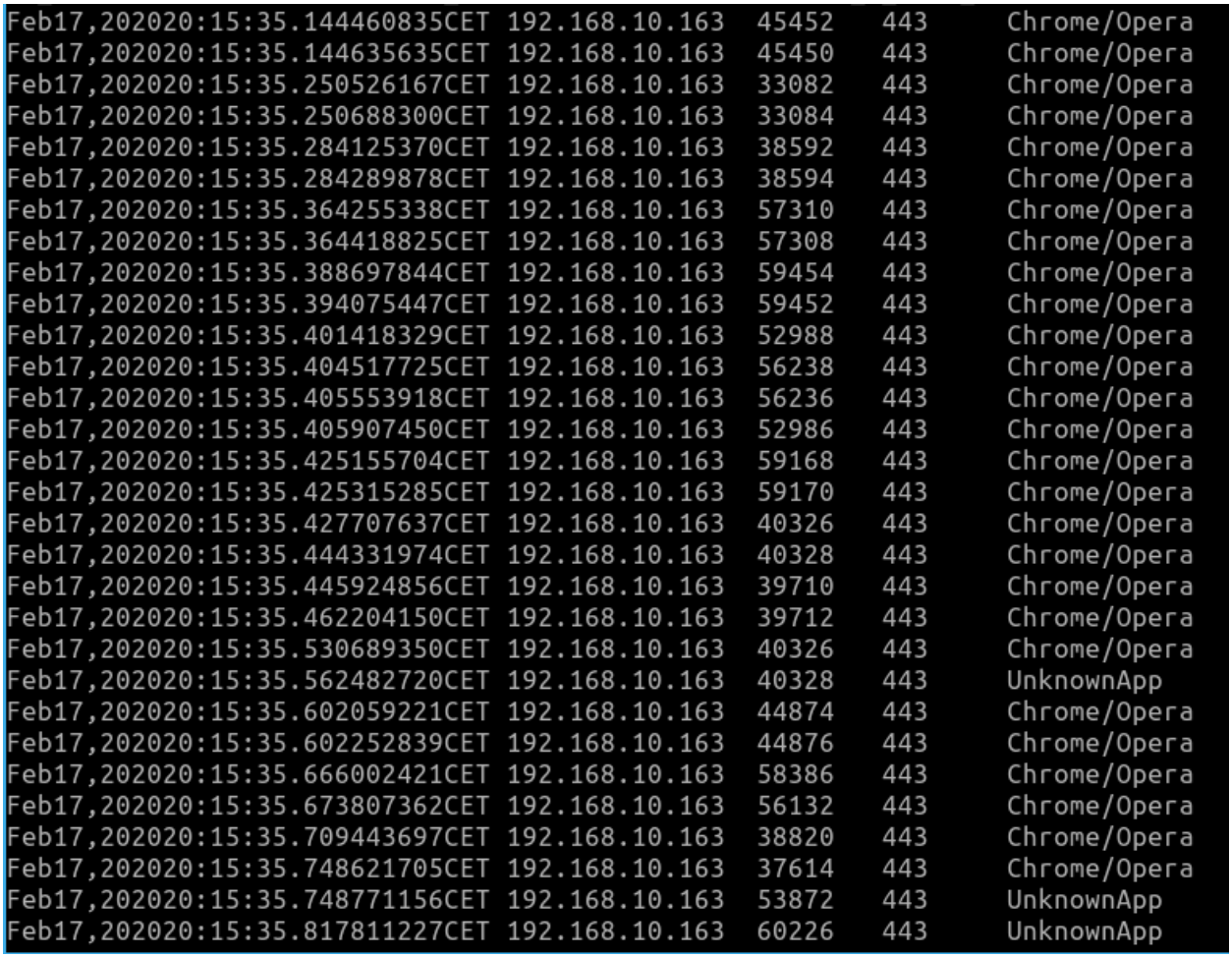
The similarity between Google Chrome and Opera browser is because they share the same engine, Chromium and Blink engine.

To validate the tool for Chrome / Opera fingerprinting, PCAP files for Firefox were used and different fingerprints generated as indicated in the table 5:

Table 7: Firefox Fingerprints

firefox-klinux-68_2_0esr	**firefox-ubuntu-73_0_1**	**firefox-mac-73_0**	**firefox-windows-70_0_2**
21890d87fa7da98f2b6cda22df895bcb	21890d87fa7da98f2b6cda22df895bcb	21890d87fa7da98f2b6cda22df895bcb	4411f0b337f1c2708cb8d98f58b9a447
74f477829a69ba89ff7942171e4f6f54	74f477829a69ba89ff7942171e4f6f54	74f477829a69ba89ff7942171e4f6f54	4b186ca6dfb44d2a496773fdc2b6944a
942292e4839c47998b8c32b97e45694c	942292e4839c47998b8c32b97e45694c	942292e4839c47998b8c32b97e45694c	74f477829a69ba89ff7942171e4f6f54
			942292e4839c47998b8c32b97e45694c

Since Chrome / Opera fingerprints are indistinguishable at the moment, I have merged their names in the database app field. Running the script with x option (./chromeJA3.sh capture.pcapng -x) on a Chrome PCAP generates a results as illustrated in figure 16:



```
Feb17,202020:15:35.144460835CET 192.168.10.163  45452   443     Chrome/Opera
Feb17,202020:15:35.144635635CET 192.168.10.163  45450   443     Chrome/Opera
Feb17,202020:15:35.250526167CET 192.168.10.163  33082   443     Chrome/Opera
Feb17,202020:15:35.250688300CET 192.168.10.163  33084   443     Chrome/Opera
Feb17,202020:15:35.284125370CET 192.168.10.163  38592   443     Chrome/Opera
Feb17,202020:15:35.284289878CET 192.168.10.163  38594   443     Chrome/Opera
Feb17,202020:15:35.364255338CET 192.168.10.163  57310   443     Chrome/Opera
Feb17,202020:15:35.364418825CET 192.168.10.163  57308   443     Chrome/Opera
Feb17,202020:15:35.388697844CET 192.168.10.163  59454   443     Chrome/Opera
Feb17,202020:15:35.394075447CET 192.168.10.163  59452   443     Chrome/Opera
Feb17,202020:15:35.401418329CET 192.168.10.163  52988   443     Chrome/Opera
Feb17,202020:15:35.404517725CET 192.168.10.163  56238   443     Chrome/Opera
Feb17,202020:15:35.405553918CET 192.168.10.163  56236   443     Chrome/Opera
Feb17,202020:15:35.405907450CET 192.168.10.163  52986   443     Chrome/Opera
Feb17,202020:15:35.425155704CET 192.168.10.163  59168   443     Chrome/Opera
Feb17,202020:15:35.425315285CET 192.168.10.163  59170   443     Chrome/Opera
Feb17,202020:15:35.427707637CET 192.168.10.163  40326   443     Chrome/Opera
Feb17,202020:15:35.444331974CET 192.168.10.163  40328   443     Chrome/Opera
Feb17,202020:15:35.445924856CET 192.168.10.163  39710   443     Chrome/Opera
Feb17,202020:15:35.462204150CET 192.168.10.163  39712   443     Chrome/Opera
Feb17,202020:15:35.530689350CET 192.168.10.163  40326   443     Chrome/Opera
Feb17,202020:15:35.562482720CET 192.168.10.163  40328   443     UnknownApp
Feb17,202020:15:35.602059221CET 192.168.10.163  44874   443     Chrome/Opera
Feb17,202020:15:35.602252839CET 192.168.10.163  44876   443     Chrome/Opera
Feb17,202020:15:35.666002421CET 192.168.10.163  58386   443     Chrome/Opera
Feb17,202020:15:35.673807362CET 192.168.10.163  56132   443     Chrome/Opera
Feb17,202020:15:35.709443697CET 192.168.10.163  38820   443     Chrome/Opera
Feb17,202020:15:35.748621705CET 192.168.10.163  37614   443     Chrome/Opera
Feb17,202020:15:35.748771156CET 192.168.10.163  53872   443     UnknownApp
Feb17,202020:15:35.817811227CET 192.168.10.163  60226   443     UnknownApp
```

Figure 16: Chrome / Opera Identification

Android Apps Fingerprinting

Lumen Analysis

Table 7 indicates a subset of the results obtained by the Lumen App. It lists all detected privacy leaks with their description and mobile apps that were involved in these leaks.

Table 8: Lumen Monitor Flows

Lumen Privacy Monitor (v 2.2.2)						
Device model TECNO-J8, Android version 5.1						
App	**Version**	**Domains)**	**Sub Domains**	**App Traffic %**	**Service Popularity**	**Also Used By**
Weather	1	accu-weather.com	alamo.accu-weather.com	100	Very Low	PHX Browser
Equa bank	16.1.0	equa.cz	acs.equa.cz	6.3	Very Low	
		equamobile.cz	ma.equamobile.cz	90.9	Very Low	
		firebaseremoteconfig.googleapis.com	firebaseremoteconfig.google.apis.com	0.2	Very Low	Shazam, Downloads, Play Store, Boomplay
		predk.us	predk.us	2.5	Very Low	
KB Klíč	2.6.1	kb.cz	login.kb.cz	68.5	Very Low	
		mojebanka.cz	wa.mojebanka.cz	30.1	Very Low	
		update.googleapis.com	update.googleapis.com	1.2	Very Low	Chrome
Telegram	6.1.1	appcenter.ms	in.appcenter.ms	100	Very Low	

Conclusion

Preliminary results have indicated that Firefox web browsers can be accurately identified across different operating systems. This is because of its unique fingerprint in the Client Hello TLS message. The application was modified in order to fingerprint Google Chrome and Opera browsers across the four operating systems.

Finally, the browser version does not matter regarding fingerprint generation. The tests were done using different versions of browsers across the four operating systems, and similar fingerprints were identified.

The tool consists of a shell script that processes pcap files to compute JA3 fingerprints of known web browsers. Tshark commands are used to extract the relevant fields from the Client Hello packets. Unix string manipulation commands parse the fields to prepare for fingerprint generation / hashing. Computed fingerprints are saved into CSV files so that unknown pcap files can be compared.

References

Bojinov, H., Michalevsky, Y., Nakibly, G., & Boneh, D. (2014). Mobile device identification via sensor fingerprinting. arXiv preprint arXiv:1408.1416.

Bujlow, T., Carela-Español, V., Solé-Pareta, J., & Barlet-Ros, P. (2015). Web tracking: Mechanisms, implications, and defenses. arXiv preprint arXiv:1507.07872.

Dierks, T., & Rescorla, E. (2008). The transport layer security (TLS) protocol version 1.2.

Husák, M., Čermák, M., Jirsík, T. et al. HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting. EURASIP J. on Info. Security 2016, 6 (2016). https://doi.org/10.1186/s13635-016-0030-7

John Althouse, Jeff Atkinson and Josh Atkins (2019): TLS Fingerprinting with JA3 and JA3S. Retrieved from https://engineering.salesforce.com/tls-fingerprinting-with-ja3-and-ja3s-247362855967 on 4 March 2020.

Kotzias, Platon, et al. "Coming of age: A longitudinal study of tls deployment." Proceedings of the Internet Measurement Conference 2018.

Lemon, S. (2015). TLS Fingerprinting Smarter Defending & Stealthier Attacking. Online], Sep, 25.

Postel, J. (1980). RFC0768: User Datagram Protocol.

Razaghpanah, A., Niaki, A. A., Vallina-Rodriguez, N., Sundaresan, S., Amann, J., & Gill, P. (2017, November). Studying TLS usage in Android apps. In Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies (pp. 350-362).

Rescorla, E., Ray, M., Dispensa, S., & Oskov, N. (2010). Transport layer security (TLS) renegotiation indication extension (Vol. 43, p. 44). RFC 5746 (Proposed Standard).

Zeifman, I,. (2012). Was that really a Google bot crawling my site? Retrieved from https://www.incapsula.com/blog/was-that-really-a-google-bot-crawling-my-site.html on 2 July 2020.