University of Pisa

Department of Computer Science

Master Degree in Computer Science

A Semi-automatic Method for Mobile Application Network Traffic Classification

Supervisor:

Prof. Luca Deri

Candidate:

Francesca Funaioli

Academic Year 2024-2025

Abstract

Network traffic classification is the task of identifying and categorising traffic generated by a monitored network into a number of classes, depending on applications, protocols or services appearing in the observed communications. This type of analysis is relevant in order to better understand the behaviour of the connected devices, to provide the agreed-upon level of QoS, and to detect potential malicious traffic or attacks. Current challenges on the subject matter involve the ability to recognise encrypted network traffic, as well as NAT and VPN communications. This work proposes an approach to the classification of mobile application traffic into a set of clusters, with the goal of identifying the different software installed on the monitored devices and provide information about the network's behaviour. The method consists in an algorithm leveraging notions of similarity to cluster together flows, requiring no machine learning or training phase. The evaluation of the results produced by this method show an accuracy of up to 94.62%, making it a valid alternative to more computationally intensive ML methods.

Contents

Chapter 1

Introduction

1.1 Motivation

Network traffic classification is the task of identifying and categorising traffic generated by a monitored network into a number of classes, depending on applications, protocols or services appearing in the observed communications. Traffic classification has gained relevance in recent years, as the number of devices connected to the network increases.

There are many reasons to strive for a better understanding of the traffic flows of a given network. Some of the areas of interest for such an analysis are (1) the control over QoS (Quality of Service), (2) the need to detect malicious behaviours and to block specific applications or content, and (3) lawful interception of illegal traffic. (1) Traffic classification is relevant to ISPs (Internet Service Providers) in order for them to be able to make available to the services' users the expected level of QoS. In this case, the process would involve analysing the network traffic, gaining knowledge of the behaviour of the users and giving priority to some of the provided services, to the detriment of others. For instance, this analysis allows ISPs to provide low latency to real-time communication applications, and high bandwidth to media streaming services. (2) In addition, having a clear understanding of the monitored network is fundamental in order to be alerted of changes in the traffic behaviour, to be aware of possible threats, and to recognize the presence of malicious traffic. Furthermore, analysing the different behaviours taking place on a given network, in terms of applications and protocols utilised, and amount of traffic generated, is essential in order to gain deep knowledge of the expected and allowed traffic. This information, regarding what kind of traffic each device

typically generates, allows the network manager to notice and be alerted of different and unexpected users' behaviours that deviate from what was previously observed. Moreover, if the monitored network requires additional security (for example in an industrial automation environment), then the network manager, seeking a more fine-grained control over the behaviour of connected devices, can restrict the number of applications running on them and understand when this restriction has been violated. (3) During recent years, regulating cybersecurity strategies and techniques from a legal standpoint has become a growing concern due to the increasing exposure to cyber threats and attacks. In August 2016, the European Commission introduced the NIS1 Directive [1], which had the goal of increasing cybersecurity of NISs (Network and Information Systems). This regulation required EU member states to update their national laws as to include the definition of new response strategies to deal with cybersecurity breaches and minimise the risk of attacks. The NIS1 Directive was repealed by the NIS2 Directive [2], which came into force in January 2024. NIS2 introduced new obligations for member states to take adequate measures for what concerns risk-management, incident notification and reporting, information sharing and cybersecurity education [3]. While its main obligations will be applied from December 2027, The Cyber Resilience Act [4] came into force in December 2024, defining stricter cybersecurity requirements for manufacturers of software and hardware products, as well as requiring them to provide security updates for the entire lifecycle of their products. As a consequence, cybersecurity will need to be taken into account at every stage of the production chain, and customers will also be able to identify in a more straightforward way whether a product actually has proper cybersecurity. Summarising, traffic classification is one of the tools available when managing the provided QoS, monitoring the network behaviour, and noticing malicious traffic.

There are different techniques for gaining an understanding of a network's behaviour and categorise its traffic into classes. Some of the possible approaches are, from the simplest, least resource-intensive, to the most complex approach: the use of the packets' port number, DPI (Deep Packet Inspection) techniques, ML (Machine Learning) algorithms, and deep learning models.

This work will focus on the classification of mobile application traffic from a

given packet capture file. The method involves parsing the packet capture exploiting a tool from nDPI and then grouping together the resulting flows according to their attributes' values.

1.2 Outline

The thesis is organised into different chapters.

Chapter 1 exposed the motivation for implementing traffic classification methods and tools by showing some of the domains that could benefit from it, and, at the same time, it introduced some of the main techniques that can be applied to this subject.

Chapter 2 will present the main approaches adopted for the task of traffic classification, along with their main characteristics, while also listing the current state of the art technologies concerning the discussed topic.

Chapter 3 will exhibit the implementation details, describing the input and output files, analysing the steps of the proposed algorithm, and presenting the employed tools, as well as the metrics used for the evaluation of the results.

Chapter 4 will, in addition to accounting for the data collection phase, report the results obtained by the proposed methodology when applied to mobile network traffic and analyse them in light of the proposed evaluation metrics.

Chapter 5 will draw the conclusions on this work, by summing up the main outcomes, and by proposing improvements to the described algorithm that can be implemented in future versions.

Chapter 2

Background

The OSI (Open Systems Interconnection) reference model [5] describes the components of a communication system as having a layered architecture. In particular, the seven abstraction layers defined are physical, data link, network, transport, session, presentation, and application. Traffic classification usually focuses on layers 3 to 7, assigning 0 to the lowest layer and 7 to the highest.

2.1 Current Methodologies

The following Subsections will each present a technique that can be used to perform network traffic classification.

2.1.1 Port Analysis

One of the easiest ways to categorise traffic is to look at port numbers. This involves analysing L4 packets, which mainly use TCP (Transmission Control Protocol), UDP (User Datagram Protocol) or QUIC. Transport layer packets' port numbers are divided into 3 groups [6]: well known, or system ports (0-1023), registered, or user ports (1024-49151), and dynamic, or private/ephemeral ports (49152-65535). Ports in the system and user ranges are assigned by IANA (Internet Assigned Numbers Authority) according to their purpose. For instance, HTTP (HyperText Transfer Protocol) uses port 80, HTTPS (HTTP Secure) uses port 443, SSH (Secure SHell) uses port 22, SMTP (Simple Mail Transfer Protocol) uses port 25 [7]. Information on the port number can be used to make assumptions on the purpose of the traffic, but there are no guarantees that the advertised service will actually correspond to the assigned one. In fact, applications can randomise port numbers or use a port

number that is assigned to another protocol in order to disguise themselves and avoid detection.

2.1.2 Deep Packet Inspection

A more thorough approach to traffic classification is DPI, which involves the analysis of packets' header and protocol fields, as well as the payload. Such an analysis is useful to monitor the network and evaluate its performance, but it also provides information to detect malware and attacks, as well as performing content filtering. The information gathered through DPI analysis allows to intervene on the observed packets, by dropping, logging, blocking, or re-routing them.

As showed by Bujlow et al. [8], there are many DPI tools, both proprietary and open-source, which can be applied at different levels of the OSI stack, achieving different performances in the accuracy of the classification. The authors study the results obtained when applying 6 DPI tools to the task of classifying at first normal traffic, then truncated packets traffic, and finally truncated flows traffic. The best performing tools for the tasks of classifying application-layer protocols, web services and application protocols (proprietary or not), are, according to Bujlow et al. [8], PACE, nDPI and Libprotoident.

Çelebi et al. [9] identify three different categories of DPI methodologies: DPI related techniques, acceleration techniques for DPI, and DPI for encrypted traffic.

The first group consists of pattern matching and protocol decoding:

- Pattern matching is the act of comparing a given sequence against a pattern in order to check for matches; it can be implemented by comparing hash values of data to the selected pattern, by using probabilistic data structures (such as Bloom filters) to perform membership testing for the pattern, by adopting heuristic algorithms to improve the matching through skipping some of the payload characters, and finally by making use of automaton-based techniques implementing a regular expression to match against.

- Protocol decoding involves re-establishing sessions, based on the captured packets, and verifying that the transferred data matches to the expected data, in terms of syntax, semantics, and protocol conformity.

The second group is made up of hardware and software-based methods for DPI acceleration. The aim of this techniques is to match the speed of the network, that is the amount of incoming packets to analyse per unit of time, without needing to drop them. In order to speed up the processing of the network packets, as far as it concerns the hardware, pattern matching in DPI applications exploits ASICs (Application-Specific Integrated Circuits), FPGAs (Field-Programmable Gate Arrays), and GPUs (Graphics Processing Units). Regarding software, most solutions focus on exploiting parallelism on multicore processors by splitting the pattern matching tasks into subtasks and assigning each unit of work to different cores.

The last group consists of man-in-the-middle, access control and trusted hardware approaches. The proposed techniques have the goal of decrypting network traffic, in order to make it possible to analyse the packets' content, thus allowing DPI precedures.

Dealing with encrypted traffic is gaining relevevance as the amount of network traffic using the TLS (Transport Layer Security) protocol increases. As shown in Figures 2.1 and 2.2, reporting the percentage of pages accessed over HTTPS respectively on Firefox and Chrome browsers, the global amount of encrypted traffic is well over 80%. Encrypted traffic hinders many activities on the monitored network, such as anomaly detection and analysis, by hiding relevant information including HTTP header fields and in general application-layer metadata [10].

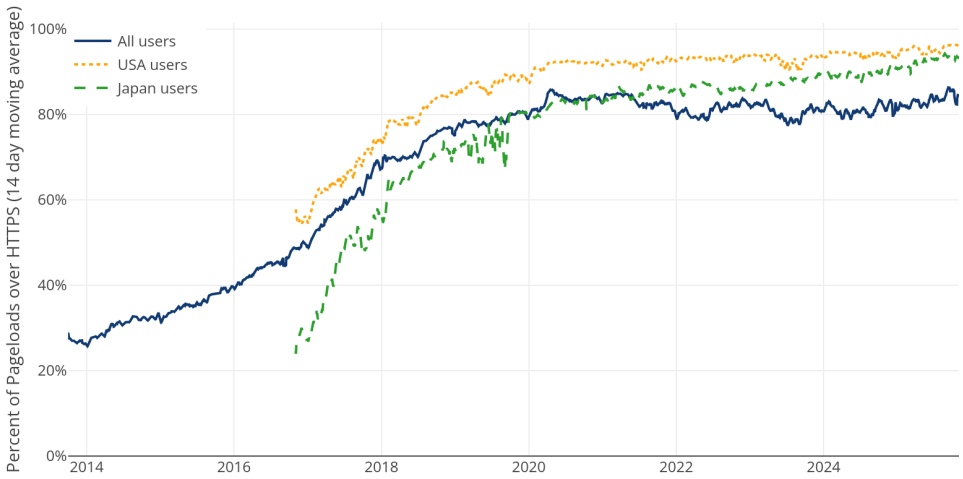

Figure 2.1: Percentage of Web Pages Loaded by Firefox Using HTTPS
Source: `https://letsencrypt.org/stats/`, accessed: 04-11-2025

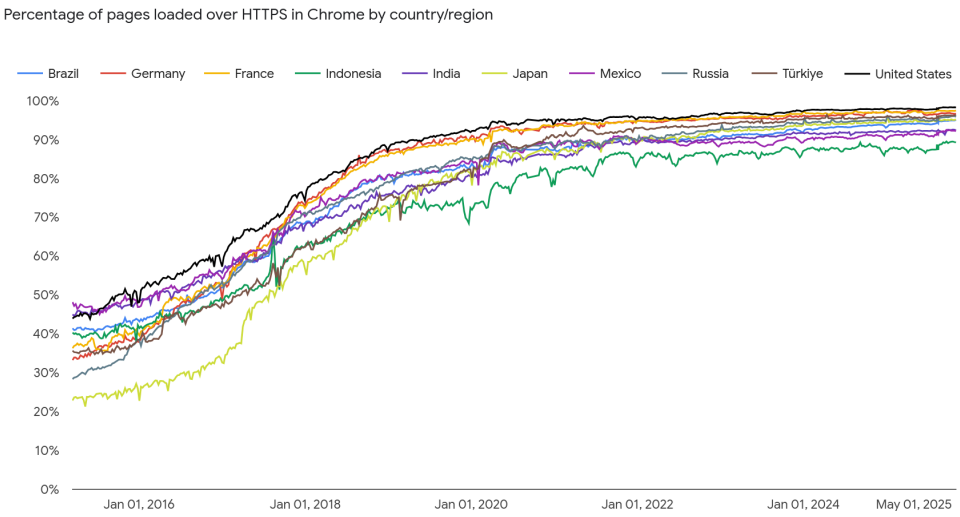

Figure 2.2: Percentage of Web Pages Loaded by Chrome Using HTTPS
Source: `https://transparencyreport.google.com/https/overview`, accessed: 04-11-2025

2.1.3 Machine Learning

ML is another technique that can be applied to the classification of network traffic. The task can be understood as either classification or clustering of the given data. In fact, in case the groups in which the data is to be divided are already known, the task is a classification one, consisting in assigning the given traffic to the most fitting class among those provided to the model. Otherwise, if the number and the names of the groupings is not known in advance, and one merely wants to find similarities among the network traffic, the task is a clustering one, meaning that the final result will be the creation of subsets of data that are believed to be produced by the same application.

A ML task can be supervised or unsupervised. In the first case, the model is provided with a training dataset consisting of labeled data. If network traffic is the domain of the ML model, then the labeling of data consists in assigning it to the right class, be it application, protocol, or web service. This can be a cumbersome and time consuming job to perform. On the other hand, when taking into consideration unsupervised learning, the model is not provided with any kind of ground knowledge, which allows to forgo the labeling phase, needed for the training of the model.

Additionally, Jacobs et al. [11] point out that network operators are wary of

ML when applied to network traffic recognition and to network security. They cite the problem of black-box understandability, noting that model interpretability is an essential quality and one of the main reasons for scepticism. They state that trusting the ML black-box model is comparable to accepting to surrender control to it. Furthermore, the lack of publicly available labeled datasets, along with the fact that the available ones are either synthetic or have been generated in a small test environment, makes it hard to properly train a ML model. This scarcity is due to the fact that making network data publicly accessible raises privacy and security concerns. The same authors, Jacobs et al. [11], present the TRUSTEE framework, which is able to provide a decision-tree explanation, given the training dataset as input.

A possible solution to privacy concerns is the use of payload-independent information, such as port numbers, packet length and inter-arrival time, and first and last occurrences of a flow in the analysed traffic. This approach protects the privacy of the users as the packets' payload is not examined. ML can exploit statistical classification methods in order to extract ground truth from payload-independent information.

Cunha et al. [12] define two categories of statistical methods: parametric and non-parametric. Parametric methods make specific assumptions on the distribution probability of the given data, in order to deduce which probability distribution better represents it, and then assign to its parameters the values that more suitably correspond to the data. The most relevant benefits of such an approach are the small amount of data that is required to estimate the selected distribution parameters, and the shorter time necessary for the training phase. While this is true, on the other hand, a significant drawback is the fact that the model is more vulnerable to outliers and to the initial assumptions on the chosen distribution. In the same work [12], the authors include in the group of parametric methods linear SVMs (Support Vector Machines), Euclidean distance, Pearson correlation, and Jensen-Shannon divergence. On the contrary, non-parametric methods make no assumption on the distribution probability of data, meaning that there is no fixed number of parameters to assign values to. The advantages of this grouping of methods are, for instance, its independence from a specific distribution, allowing

it to better represent more complex behaviours and the small impact of outliers. Non-parametric approaches, though, have the problem of demanding more time and more data in order to perform the training step when compared to parametric methods. The authors list in the second category non-linear SVMs, Bhattacharyya distance, Hellinger Distance, Kullback-Leibler divergence, Wootters distance, and Kolmogorov-Smirnov and Chi-square tests.

2.1.4 Deep Learning

Another approach to the network classification problem is DL, a branch of ML specifically focusing on neural networks. Because of the layered quality of neural networks, DL can better catch complex patterns in the analysed data, with respect to general ML. Additionally, DL algorithms do not need the traditional step of feature engineering, exploiting feature learning instead. Feature engineering is the process of transforming the given raw data into a more suitable set of inputs in order to improve the effectiveness of the model. This can be done by either cleaning, combining, transforming, or selecting some of the already present features, and it has the objective of choosing the attributes better representing the given domain, so as to obtain a more precise model. Feature learning refers, instead, to the ability to automatically identify the most useful representation of the feature to use. Removing the need for hand-crafted feature engineering has a positive impact on DL algorithms, compared with non-DL ML approaches, while also allowing to more adherently model complex features.

Iliyasu et al. [13] provide a classification of the most used DL models, citing FNNs (Feed-forward Neural Networks), CNNs (Convolutional Neural Networks), RNNs (Recurrent Neural Networks), and GANs (Generative Adversarial Networks). Furthermore, Iliyasu et al. [13] also note that, while the learning task can be either supervised, unsupervised, or semi-supervised, the majority of the techniques they analysed made use of either supervised or semi-supervised learning. The main concern regarding the use of DL methods, according to the authors, is the fact that they require large labeled datasets, whose acquirement gets more complex in the case of encrypted traffic.

Kalwar and Bhatti [14], in their analysis of DL applied to the task of network traffic classification for IoT (Internet of Things) devices, mention, among the disadvantages of such approaches: the large amount of time and computational resources needed for the training of the model, as well as the need for comprehensive labeled datasets, which, as already mentioned, can be hard to acquire, and finally the crucial need for explainability and accountability, as was the case with the more general ML approaches. The authors also list, among the most common DL methods for network traffic classification: CNNs, RNNs, and FNNs.

In conclusion, ML, and more specifically DL methodologies, are the most commonly used for the task of network traffic classification. Regardless of this trend, these models present some drawbacks, such as the necessity of a large amount of data for training, the quality of which will impact the end result and the model's accuracy and performance. As a consequence, labeling needs to be regarded as an important activity when approaching the task of network traffic classification: even if it is time consuming and computationally intensive, the labeling has to be correct in order to provide good results in the trained models. Moreover, because of the frequent changes in network traffic, ML models need to be retrained frequently to adapt to the new behaviours.

2.2 Fingerprinting

DPI traffic classification and recognition is based on the matching of packets' payload with already known signatures or fingerprints. A fingerprint is a compact representation of an arbitrarily large data item generated by a function mapping the given input value into its unique fixed-size representation (an example of well known such functions are cryptographic hash functions). In the field of network traffic analysis, building a large database of signatures is relevant as it allows to quickly recognise the observed traffic in terms of OS (Operating System), applications, devices or users. For instance, considering the task of OS fingerprinting, one can examine the TCP/IP stack, and specifically the values of: initial packet size, TTL (Time To Live), window size, maximum segment size, and TCP flags. These parameters' values are not fixed by the implementation, which

means that different OSs, but also different versions of the same OS, can be set apart by inspecting these attributes.

Fingerprinting can be performed either actively or passively. Active fingerprinting refers to the act of sending packets to the chosen device with the objective of analysing its response. Because of the consequent injection of traffic into the network, this approach has the drawback of rendering the prober visible to the other devices. An example of active fingerprinting is port scan, which consists in probing a device for open ports with the goal of finding out: what the implemented services are, as well as information regarding the device itself, like its OS. As noted by Pittman [15], port scanning can be used as a tool for malicious purposes, but it is also exploited in network managing tasks: in the first case it can reveal vulnerabilities to exploit, while, in the latter, it is used for validating configurations, mapping exposed endpoints and their services, and troubleshooting. The same work [15], performs an analysis of some of the most used tools for port scanning, namely Nmap [16], ZMap [17], and MASSCAN [18]. Passive fingerprinting is concerned, instead, with observing the network traffic without generating new packets. Because of this, it can be employed to examine offline data, that is traffic that has been previously captured and can now only be observed. Given its non-intrusive qualities, passive fingerprinting is employed in the context of long-lasting network monitoring, when gathering information without the observed target's knowledge. As a consequence of its passive nature, though, this approach offers a less detailed result, with respect to the corresponding active technique, and it can be hindered by noise in the monitored traffic.

Deri and Nardi [19] proposed a passive TCP/IP fingerprint for Wireshark, as there currently is no standardized one; its format is the following:

```
<TCP Flags>_<TTL>_<TCP Win>_SHA256(<Options Fingerprint>)
```

Applied to the task of OS and device recognition, this fingerprinting strategy allows to deduce interesting information about the observed devices. Specifically, the proposed TCP fingerprint highlights the differences among the most common OSs in terms of TTL, options' order, and potential duplication, thus offering enough information to infer the OS of embedded devices. Additionally, the same work [19],

points out that the same OS and the same device can each have different values of TCP fingerprint, depending respectively on the version of the OS, and on the performed activity.

The Joy package was developed by Perricone et al. [20] with the goal of analysing network data both online and from an already captured .pcap file. The tool can be used for different purposes, ranging from network monitoring and administration, to vulnerability detection; it exploits flows, expressed as a 5-tuple consisting of source and destination IP address, source and destination port, and protocol number. The Joy package was subsequently improved, by Perricone et al. [21], with the addition of TLS fingerprinting, in order to better function as a malware detection tool, as well as to enhance its OS and application identification capabilities. The Joy signature consists of the hash of Client Hello parameters: it is structured in different sections, containing, respectively, the metadata information of the TLS features, the description of TLS cipher suites and extensions, and the lists of associated processes and OSs that were observed having the given fingerprint.

The Mercury package, developed by McGrew et al. [22], and direct successor of Joy, extends Joy's capabilities in terms of capturing network traffic and analysing it in order to extract interesting metadata and export it in .json format. Mercury implements fingerprinting for TLS, DTLS, SSH, HTTP, TCP, and other protocols. This methodology proposes as output a string representation consisting of relevant protocol parameters, and a hashed representation computed using SHA256. For the TCP/IP protocol, Mercury extracts from SYN packets relevant parameters. The resulting fingerprint format is the one showed below:

```
"tcp/" (IP_Version)(IP_ID)(IP_TTL)(TCP_Window)((TCP_Option)*)
```

The TLS fingerprint obtains information from the Client Hello, and it is defined in the following way:

```
"tls/" (TLS_Version) (TLS_Ciphersuite) ((TLS_Extension)*)
```

Worthy of note is the fact that, because some TLS clients randomize the extension list, computing the fingerprint without sorting the proposed extensions would result in obtaining different fingerprint values for clients having the same extension

list. In order to avoid this eventuality, Mercury's developers have chosen to provide three different versions of TLS fingerprint that differ only in the reordering of the extension list, thus providing a way to counteract randomization.

If the goal of fingerprinting is the detection of attacks and malicious traffic, then the signature database can be used to check for, and reveal, incoming intrusion attempts and to better understand the attacker's path by detecting intrusion patterns. This knowledge can lead to the discovery of new protection mechanisms and useful signatures to detect malicious network activity.

In 2017, Saleforce proposed the JA3 fingerprint [23], a technique that can be used for generating fingerprints based on SSL/TLS traffic. In particular, the fingerprint is created by analysing the data exchanged during the TLS negotiation step, namely the TLS version, the list of cipher suites, the list of extensions, elliptic curves, and elliptic curves formats. The purpose of JA3 is to detect malicious traffic and allow to separate it from legitimate traffic, filtering potential threats. In May 2025, JA3 was deprecated in favour of JA4+ [24], which introduced multiple protocols other than TLS. Created in order to improve on JA3, JA4 is the JA4+ fingerprint dedicated to the TLS protocol Client Hello. The following is an example of JA4 fingerprint presented by Althouse [24]:

```
JA4 = t13d1516h2_acb858a92679_e5627efa2ab1
```

The above signature can be split into three different parts, respectively JA4_a, JA4_b, and JA4_c, by the presence of the underscore character. The meaning of each one of the fingerprint section's is as follows:

- JA4_a contains general information from the TLS Client Hello, such as, in order:
  - Protocol Identifier, indicating the use of TLS over TCP (`t`), QUIC (`q`) or DTLS (`d`);
  - TLS Version, as an example "`12`" for TLS1.2, and "`13`" for TLS1.3;
  - SNI (Server Name Indication) Presence, indicating whether a SNI is specified (`d`), or not (`i`);

  - Number of Cipher Suites, indicating the total number of cipher suites provided by the Client Hello;
  - Number of Extensions, indicating the total number of extensions provided by the Client Hello;
  - ALPN (Application-Layer Protocol Negotiation) Values, corresponding to the first and the last characters of first ALPN extension value; for instance, "`h2`" for HTTP/2, "`h1`" for HTTP/1, "`dt`" for DNs-over-TLS, "`00`" no ALPN present.

- JA4_b is the sha256 hash of the list of cipher hexadecimal codes, sorted in hexadecimal order, truncated after the twelfth character;

- JA4_c is the sha256 hash of the combination of the list of extension hexadecimal codes sorted in hexadecimal order with the list of signature algorithms, truncated after the twelfth character.

Since the major purpose of JA4+ is network security, the framework's main concern is to accurately map and recognise cyber threats and malware. This goal is achieved by checking fingerprints for an exact match, as opposed to performing a similarity analysis among different services sharing some values in the cipher suites and extensions+signature algorithms lists. Table 2.1 shows some of the fingerprints reported by Althouse [24].

In conclusion, fingerprinting is a valuable tool for network analysis: it can be employed both to examine the traffic in terms of connected devices, their OS, and the applications running on them, and to detect cyber threats and attacks. Additionally, signatures can be obtained even from encrypted traffic by means of extracting information from the TLS packets' Client Hello.

2.3 State of the Art

The present Section aims at providing a comprehensive review of the research currently conducted on the topic of network traffic classification. The majority of the cited works will have a component of ML, as it is, at present, one of the most employed methods for classification and clustering projects.

Application	JA4+ Fingerprints
Chrome	JA4=t13d1518h2_8daaf6152771_e5627efa2ab1 (TCP) JA4=q13d0310h3_55b375c5d22e_cd85d2d88918 (QUIC)
IcedID Malware Dropper	JA4H=ge11cn020000_9ed1ff1f7b03_cd8dafe26982
IcedID Malware	JA4=t13d201100_2b729b4bf6f3_9e7b989ebec8 JA4S=t120300_c030_5e2616a54c73
Sliver Malware	JA4=t13d190900_9dc949149365_97f8aa674fd9 JA4S=t130200_1301_a56c5b993250 JA4X=000000000000_4f24da86fad6_bf0f0589fc03 JA4X-000000000000_7c32fa18c13e_bf0f0589fc03
Cobalt Strike	JA4H=ge11cn060000_4e59edc1297a_4da5efaf0cbd JA4X=2166164053c1_2166164053c1_30d204a01551
SoftEther VPN	JA4=t13d880900_fcb5b95cb75a_b0d3b4ac2a14 (client) JA4S=t130200_1302_a56c5b993250 JA4X=d55f458d5a6c_d55f458d5a6c_0fc8c171b6ae
Evilginx	JA4=t13d191000_9dc949149365_e7c285222651
Reverse SSH Shells	JA4SSH=c76s76_c71s59_c0s70

Table 2.1: Some examples of JA+ fingerprints of real-world applications and malwares.

Source: `https://blog.foxio.io/ja4+-network-fingerprinting`, accessed: 07-11-2025

Azab et al. [25] select recent literature on the subject in order to show the main techniques of network traffic classification, examining in particular port-based analysis, DPI, supervised and unsupervised ML, and DL. Specifically, DPI is noted to be more precise than port-based classification, with the drawback of requiring more computational resources to access, inspect, and store the analysed packets, as well as being hindered by encryption, which does not allow to inspect the packets' payload. In the same work [25], ML solutions are distinguished among full-flow or sub-flow monitoring, and detection of previously unknown traffic. The first two approaches focus, respectively, on analysing the entirety of the flow from the first to the last packet, and the first few packets or bytes of a communication. The third method evaluated is concerned with the detection of applications of which the model had no previous knowledge. The authors conclude that there is no single best solution to the network traffic classification task, in terms of classification accuracy, computational resources needed, classification speed, and untrained detection.

Wang et al. [26] focus on classification of network traffic with the purpose of providing a clear understanding of the users' platforms to ISPs, in particular to

video streaming providers. In order to identify flows belonging to four streaming services, namely YouTube, Netflix, Amazon Prime Video, and Disney+, three machine learning methods are used to recognise user platform, device type and software agent: random forest, MLP (MultiLayer Perceptron), and KNN (K-Nearest Neighbours); the first model is shown to be the best performing one. The attributes analysed to perform the classification are extracted from TCP/IP packets' header and from TLS handshake parameters, allowing to reach high accuracy levels (above 90%).

Nader and Bou-Harb [27] explore the task of identifying IoT devices behind a NAT (Network Address Translation), by applying: two unsupervised clustering algorithms, specifically K-Means and DBSCAN (Density-Based Spatial Clustering of Applications with Noise), and two semi-supervised method, a first one based on autoencoders and logistic regression, and a second based on Bernoulli RBM (Restricted Boltzmann Machine) and logistic regression. The authors show that, while clustering algorithms perform poorly, the method employing autoencoders and logistic regression achieves an accuracy of 75%, and the one utilising Bernoulli RBM and logistic regression reaches an accuracy of 98% in recognising IoT devices behind a NAT.

Chen and Wang [28] propose a DL model, MPAF (Multi-Phase Attribute Fingerprint), for the classification of TLS traffic. The authors identify three main phases in the encrypted communication between a client application and the corresponding server: the query for the requested domain name, the handshake phase of the TLS protocol, with the negotiation of the encryption suite to adopt, and the actual exchange of encrypted data. The proposed model executes three steps of classification, one for each phase: the first one is performed on the extracted DNS (Domain Name Server) records, the second on significant raw bytes from the handshake phase metadata, and the third on the length of TLS messages. The final evaluation of the model proposed in [28] shows high classification accuracy (above 96%) when exploiting a single dataset, while displaying a drop in accuracy (even reaching 70%) if utilised in a cross-dataset scenario.

Rezaei et al. [29] propose a DL model with the purpose of identifying mobile applications, that can be applied to encrypted traffic. Specifically, the authors' goal

is to provide early prediction by only observing the first few packets of a flow. The proposed methodology employs a CNN model that is able to identify 80 different applications with an accuracy of 94%, as well as a CNN+LSTM (Long Short-Term Memory) approach to classify ambiguous flows not clearly belonging to any mobile application.

Bortolameotti et al. [30] define an approach, called FlowPrint, for performing semi-supervised fingerprinting of mobile application from encrypted network traffic. The FlowPrint method requires no previous knowledge of the traffic, as the analysed data is divided into clusters on the basis of TCP/IP and TLS parameters, and then application fingerprints are extracted by observing temporal correlations among flows appearing in the same cluster. Evaluating the performance of FlowPrint, the authors achieve an accuracy of 89.2%.

Towhid and Shahriar [31] present a self-supervised approach to the classification of encrypted network traffic that requires a smaller amount of labeled data when compared to similar methodologies. The evaluation is performed in comparison with other similar baseline methods and the proposed model is able to achieve a $\sim$3% increase in accuracy.

Wang et al. [32] employ a federate learning approach, consisting of: home gateways performing a first DPI-based traffic labeling, a semi-supervised convolutional autoencoder method classifying unlabebled traffic, and an XAI-based model for explainability of the results. Edge nodes label data by parsing DNS requests, while, at the same time, training each one a local model and participating in the federated global model. The accuracy achieved is high (up to 98%) and model explainability is able to show the contribution of each feature to the obtained classification, displaying the download/upload flow ratio and packet length as the most influential features.

Huoh et al. [33] propose a model based on GNNs (Graph Neural Networks) and compare it against CNNs and RNNS-based reference models in the task of classifying encrypted network traffic. Specifically, network data, in the form of bidirectional flows, is translated into a graph representation and given as input to the GNN model. The described approached is tested with different input formats and training configurations and it succeeds in outperforming the two reference

models, achieving an overall accuracy of 97.56%.

Okonkwo et al. [34] define a classification model based in GNNs, in which bidirectional flows, consisting of the first ten packets of a session, are converted into ten-nodes graphs. The evaluation phase, conducted on two different datasets, shows an accuracy of up to 97%. Later, the same authors, Okonkwo et al. [35], present an approach to network traffic classification that models flows into graphs, which allows to analyse traffic both at packet-level and at flow-level granularity. In particular, packets are represented as nodes, and edges are determined by their similarities. The graph representation is, then, employed in a GNN-based classifier, reaching accuracy levels above 96%.

Telikani et al. [36] propose a cost-sensitive learning method to be used in tandem with a DL algorithms. In particular, a SAE (Stacked AutoEncoder) and a CNN are each coupled with the cost-sensitive method proposed in [36] and tasked with network traffic classification, in the specific case of unbalanced encrypted traffic. The classification benefits from the cost-sensitive approach, which penalises misclassification errors, reaching an average accuracy of above 98%.

Izadi et al. [37] define a methodology combining DL with data fusion techniques applied to the case of encrypted network traffic classification, with the addition of discerning VPN (Virtual Private Network) and non-VPN traffic. Data fusion is employed to remedy the lack of large labeled datasets that can be used for DL, as well as to increase accuracy and reduce classification errors. Coupling data fusion with three DL models, namely CNN, DBN (Deep Belief Network), and MLP, allows to achieve an accuracy of 97%.

Zheng et al. [38], by proposing MTT (Multi-Task Transformer), focus on a way to increase efficiency in terms of computation costs, storage, and processing time. MTT architecture consists of an embedding module, a multiple-encoders module, and a classifier module. The evaluation process, conducted against a one-dimensional and a bi-dimensional CNN, reveals that MTT reaches an accuracy of 99.37%, outperforming the two reference CNN models.

Shamsimukhametov et al. [39] focus on network traffic classification applied to encrypted data, given the universal adoption of TLS in current communications. The authors propose an algorithm leveraging SNI information obtained from

the TLS handshake Client Hello, and compare the results against a CNN-based algorithm. The results show that the accuracy of the CNN-based algorithm are strongly influenced by the labeling of input data, and that the SNI-based classifier achieves better accuracy, while being more computationally efficient, than a NN-based approach. Additionally, in the same work [39], it is noted that the addition of a new class, or the modification of an existing one, requires effort in updating the classification algorithm, but that retraining and upgrading the CNN model would require an equal or even greater amount of work.

Aksoy et al. [40] present a methodology for OS recognition that employs ML and genetic algorithms. In particular, genetic algorithms are applied for the purpose of feature subset selection in order to identify features to exploit in the classification step. The classifier, detailed in the same work [40], has a layered architecture: a first layer assigns a OS family (Linux, Mac, Windows) to each packet, which is then passed to one of three second-layer classifiers depending on the assigned OS in order to identify the OS version.

Hagos et al. [41] introduce a methodology for performing passive OS fingerprinting exploiting the TCP variant as an input feature for the OS classification process. The TCP variant is first identified by passively observing the network traffic and performing a classification step; the resulting information is employed in different ML and DL algorithms, achieving higher accuracy in predicting the correct OS, with respect to the same ML and DL algorithms executed without the TCP variant knowledge.

Anderson and McGrew [42] define a technique for OS fingerprinting that allows to identify the version of the OS using: the TCP/IP header parameters, the HTTP User-Agent, and the values from the TLS handshake. The listed data is then used to train random forest models, achieving varying levels of accuracy, depending on the initial data type being TCP/IP, TLS, HTTP, or a combination of the three.

Lastovicka et al. [43] develop a passive fingerprinting method that exploits a combination of HTTP User-Agent, TCP/IP parameters and OS-specific domains, contacted, for instance, to perform connectivity checks or system updates. The final classification is given with a majority vote among the three methods, and, in case they all disagree, a specific hierarchical order is imposed among the three.

Applying this methodology to real network traffic allows to correctly identify the observed OS with an accuracy of above 80%, with the User-Agent method being the most accurate (above 91%).

Husák et al. [44] define a fingerprint based on the analysis of the SSL/TLS handshake, with the objective of deriving the HTTPS User-Agent of the device initiating the communication. This is achieved by linking observed cipher suites to user agents in a dictionary-like structure that is then used to classify live traffic. Inspecting live traffic, the authors were then able to infer the User-Agent by looking at the list of cipher suites contained in the TLS handshake. This methodology allows to estimate the correct User-Agent with an accuracy of 99.6%, while only relying on unencrypted data.

Bai et al. [45] propose the P40f tool that can be run on programmable switches hardware. Taking inspiration from the fingerprinting methodology implemented in p0f by Zalewski [46], P40f implements a parser for TCP options, employing p0f signatures as rules to deduce the OS of the device that generated the observed packet. P40 also allows to take action on the packets, thus enabling the implementation of different policies on the switches it runs on: `drop_pkt`, `drop_ip`, and `redirect`.

Wondracek et al. [47] discuss the possibility of using protocol reverse engineering in order to identify the format of an unknown protocol and to produce a grammar that can then be used to parse different messages of the same type. The automatic approach proposed by the authors leverages dynamic data tainting to track which parts of the message are inspected by the application receiving the unknown message protocol. Messages are inspected both individually and together with other messages produced using the same unknown protocol in order to identify optional protocol fields. This method efficacy is confirmed in the same work [47] as it is successfully applied to real-world protocols and it is able to extract format specification from different types of messages, ranging from DNS to SMB (Server Message Block).

In conclusion, current literature on the topic of network traffic classification focuses mainly on ML and DL methods applied in the case of encrypted or VPN traffic. The most employed traffic features are TCP/IP metadata and TLS handshake parameters, in particular the Client-Hello. Additionally, there is also an interest in

performing OS fingerprinting, again exploiting TCP/IP header data, as well as TLS cipher suites list, and HTTP User-Agent.

Chapter 3

Implementation

The method proposed in this work has the objective of identifying the applications running on monitored mobile devices. The recognition procedure is performed by creating clusters exploiting the metadata contained in the analysed packets and assigning each cluster to an application. The algorithm developed for this purpose was implemented in Python, due to the great availability of libraries and the succinct nature of the programming language.

The development of the algorithm started from a parser of .json files implemented by Deri [48], which focused on grouping together flows on the basis of TCP parameters and of the value of JA4 fingerprint. In particular, the idea of the parser was to create a fingerprint obtained as a concatenation of the TCP fingerprint [19] already mentioned and the value of JA4 corresponding to the flow. The parser also allowed to exclude the first part of JA4, that is the JA4_a, as it was the least informative of three parts, while also enabling to perform the analysis either on the hostname or on the domain name of the flows. The result of the execution of the parser on a .json input file is:

- a list of the hostnames corresponding to each fingerprint;
- a list of (source) IP addresses corresponding to each fingerprint.

The general structure of the algorithm proposed in this work involves: reading the input file and obtaining information on the flows, storing the information for reproducibility, creating the clusters, and producing evaluation metrics values.

3.1 Input Data and `ndpiReader`

The input data employed for this analysis is in the form of a .json file containing flow information extracted from a packet capture (.pcapng) file. The .json file in question is obtained by running the `ndpiReader` tool from the nDPI package by Deri et al. [49], which, given a .pcapng file in input, performs a DPI analysis on the traffic, groups it into flows, and produces a file containing dictionary-like structures, one for each identified flow: each line of the `ndpiReader` output contains information on one of the flows in the form of key:value pairs. The flows are characterised by many attributes, for instance relating to the amount of data sent and received in the established communication, to the information about TLS parameters, and, in case of TCP flows, to the TCP flags. The relevant flow metadata for the algorithm proposed in this work is the following:

- `first_seen` and `last_seen`, timestamps expressed in seconds and represented in the epoch time format[1] describing the first and the last occurrences of the flow in the analysed .pcapng file;
- `duration`, total life span of the flow, expressed in seconds, computed as `last_seen−first_seen`;
- `src_ip` and `dest_ip`, the IP addresses, respectively, of the device initiating the communication and of the one receiving the communication;
- `src_port` and `dest_port`, the port number, respectively, used by the device initiating the communication and by the one receiving the communication;
- `proto`, the transport layer protocol, for instance TCP or UDP;
- `tcp_fingerprint`, the signature proposed by Deri and Nardi [19] and discussed in Section 2.2;
- `hostname` and `domainame`, contacted host as extracted from the SNI in the TLS Client Hello;

[1] Unix time, or epoch time, is a time representation that counts number of seconds elapsed from 00:00:00 UTC on 1 January 1970.

- `proto_by_ip`, protocol associated to the flow by the `ndpiReader` DPI analysis, based on the contacted IP address;
- `ja4`, `ja4_raw`, and `ja4_raw_uns`, discussed below.

  The `ja4` attribute is the signature mentioned in Section 2.2, proposed by Althouse [24]; it refers to the already analysed fingerprint, consisting of three parts: `ja4_a`, `ja4_b`, and `ja4_c`. The last two parts are computed by sorting the cipher suites and the extensions respectively and then calculating the SHA256 hash function.

  In order to study the network traffic classification problem in a more efficient way, this work proposes an alternative fingerprint: `ja4_raw` consists in representing the same data contained in the standard `ja4` signature, with the difference that the cipher suites list and the extensions list are expressed in their raw form, not applying any hashing function. An example of such a signature is the following:

  ```
  ja4_raw = t13d1516h2_002f,0035,009c,009d,1301,1302,1303,
  c013,c014,c02b,c02c,c02f,c030,cca8,cca9_0005,000a,000b,
  000d,0012,0017,001b,0023,002b,002d,0033,44cd,fe0d,ff01_
  0403,0804,0401,0503,0805,0501,0806,0601
  ```

  The fingerprint can be split into four different parts by the presence of the underscore character. The first part is left unchanged with relation to the original `ja4` fingerprint and corresponds to `ja4_a`; the second represents the list of sorted, unhashed cipher suites, corresponding to `ja4_b`; the third and fourth parts are respectively the sorted list of extensions and the sorted list of signature algorithms, corresponding to the whole of `ja4_c`.

  Additionally, for the purpose of better differentiating the analysed applications, yet another version of fingerprint was defined: `ja4_raw_uns`. Indeed, some of the studied applications, having the same value of `ja4`, were revealed to have different lists of cipher suites, extensions, and signature algorithms, thus allowing for a more fine-grained method for clustering. By discriminating application traffic on the basis of the actual lists of cipher suites,

extensions, and signature algorithms, it can be noticed that some applications use the same cipher suites/extensions/signature algorithms, but in a slightly different order, or that some of the items are repeated. For this reason, the `ja4_raw_uns` fingerprint represents the lists of cipher suites, extensions, and signature algorithms without any reordering or hash function application. An example of this signature is the one below:

```
ja4_raw_uns = t13d1516h2_1301,1302,1303,c02b,c02f,c02c,
c030,cca9,cca8,c013,c014,009c,009d,002f,0035_fe0d,ff01,
0005,000a,000b,0023,002b,44cd,000d,0017,002d,0012,0033,
      001b_0403,0804,0401,0503,0805,0501,0806,0601
```

In a similar way to the `ja4_raw` fingerprint, `ja4_raw_uns` consists of four parts, defined by the position of the underscore character. The first part corresponds to `ja4_a`, the second to `ja4_b`, and the last two to `ja4_c`.

The input file containing the described attributes is parsed by the algorithm to retrieve the desired information. Regardless of the above characterisation, not all of these attributes were employed by the proposed algorithm for the purpose of clustering the data, as testing revealed that the provided information was not enough to finely differentiate network traffic.

3.2 External Libraries and Algorithms

The clustering process proposed in this method as a solution to the network traffic classification problem makes use of specific tools to calculate distances and similarities among the flows' attributes in order to determine which of them belong to the same cluster, and thus were generated by the same application.

The first and simplest one of these tools is the method used to check if two `first_seen` timestamps are close in terms of time proximity. First of all, the flows are sorted in chronological order by observing the value of the `first_seen` attribute. Then, a list of pairwise distances is computed in the following way: for each `first_seen` value f_i appearing in the chronologically sorted list, the

algorithm computes the differences with the preceding and the succeeding values of `first_seen`, respectively $f_i - f_{i-1}$ and $f_{i+1} - f_i$. The result is a list of differences describing the elapsed time among the beginning of each analysed flows. The differences list is then parsed using a variable length window, whose size can be passed as an input parameter to the Python code, and compared against a threshold. Three different versions have been examined in order to find the one threshold value that allows to group together flows belonging to the same application and only those flows:

- the first method uses the arithmetic mean of the flows appearing in the current window as threshold and considers two flows as belonging to the same application if their pairwise distance is lesser or equal to the mean;
- the second approach evaluates time proximity by considering the difference between the mean and the standard deviation of the flows in the window as threshold;
- the third and final method uses the difference between the arithmetic mean and the standard deviations of the flows currently in the window as threshold.

The latter approach yields the smaller threshold of the three, and it is the one that will be used in the final tests as it allows to group flows together, without creating clusters containing traffic belonging to different applications.

One of the attributes used to cluster flows together is the domain name. Some of the observed applications employ more than one domain name. This is the case, for instance, of TikTok, which employs a plethora of domain names; some examples of this behaviour are: `tiktokv.eu`, `tiktokv.com`, `tiktokcdn.com`, `tiktokcdn-us.com`, and `tiktokcdn-eu.com`. The solution employed for clustering together flows of applications displaying different domain names makes use of the Python `difflib` library. Specifically, the string similarity function used belongs to the `SequenceMatcher` class [50]. The applied method is `SequenceMatcher(isjunk=None, a='', b='', autojunk=True).ratio()`, implemented as follows. The idea is to find the longest matching subsequences among the two input sequences `a` and `b` by parsing them, possibly excluding some junk characters, and creating a list of

matching subsequences. The list consists of triples composed of: the two indexes of the matching starting point, respectively in `a` and in `b`, and the length of the matching subsequence. The method then computes the sum of the subsequences lengths and returns a value belonging to the $[0, 1]$ interval by calculating the ratio as follows: the sum of lengths is doubled and divided by the sum of the lengths of the input sequences `a` and `b`. The similarity value returned in output is checked against a threshold in order to assign flows with similar domain names to the same cluster. The threshold value has been chosen by running the proposed algorithm on different dataset, and its value has been fixed at 0.85.

3.3 Proposed Method

The algorithm used for the clustering of network traffic is structured into four main parts. The first one, detailed in Subsection 3.3.1, consists in parsing the input file, retrieving information on each flow by establishing a connection with the contacted hostname, and finally filtering out some of the flows that are not relevant for the objective. Subsection 3.3.2 describes the second part, that involves storing the data obtained in the previous step in order to allow for reproducibility in later runs. The third part is discussed in Subsection 3.3.3 and it details the method used for clustering the network flows. Finally, Subsection 3.3.4 explains the evaluation metrics computed by the proposed method as a final step. Figure 3.1 shows a summary of the main steps of the algorithm.

3.3.1 Parsing the Input File and Retrieving Flow Information

The first part of the algorithm consists in reading the input file and extracting the relevant information. This step involves parsing each line of the .json file, extracting the metadata described in Section 3.1, as well as obtaining additional information by contacting some of the hosts appearing in the analysed traffic. Auxiliary information is also necessary for the clustering process, namely: the hash of the TLS certificate, the list of DNS names, the list of nameservers, and the hash of the favicon.

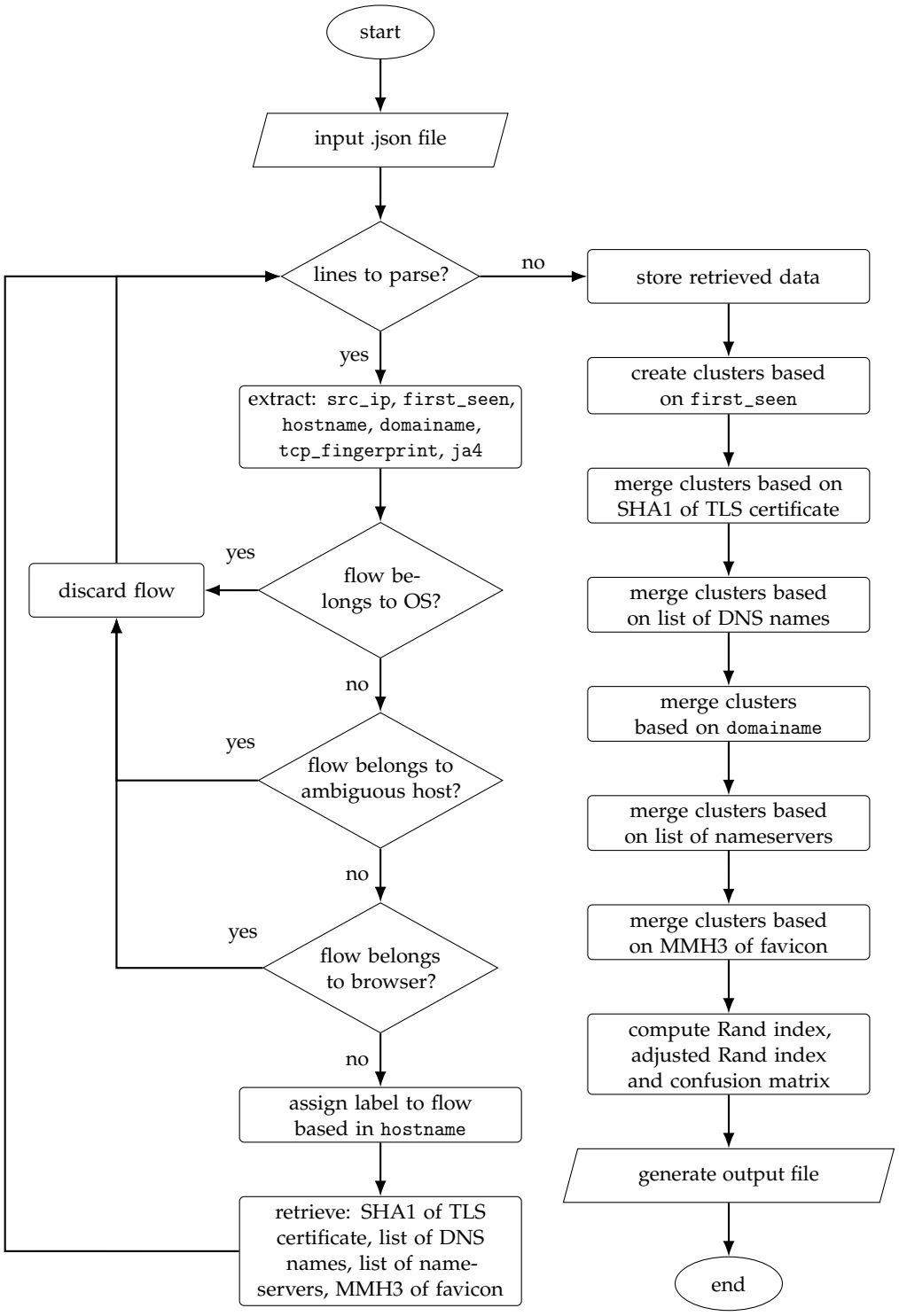

Figure 3.1: Flowchart of the method proposed in this work. The larger loop describes the parsing of the input file, along with the filtering out of flows to discard. The longer sequence of actions on the rightmost column delineates the clustering steps.

The hash of the TLS certificate, provided by the server as a form of certified identification, is obtained by establishing a connection with the desired hostname, getting its certificate, producing the SHA1 digest and converting it to its hexadecimal representation. The following is the result obtained when contacting the hostname `instagram.com`:

`F9:6C:35:7D:37:C5:57:18:12:85:EC:6A:3B:78:85:06:AD:F2:EE:3C`

The list of DNS names consists of the domain names associated with the considered hostname. It can happen that more than one hostname corresponds to the same domain name. A CNAME (Canonical NAME) record is an instance in which different hostnames are mapped to the same domain name, the canonical name. An application having different DNS names is WhatsApp, which uses the following list of DNS names: `*.whatsapp.net`, `*.cdn.whatsapp.net`, `*.snr.whatsapp.net`, `*.whatsapp.com`, `*.wa.me`, `whatsapp.com`, and `whatsapp.net`;

The list of nameservers consists of the servers designated with the task of resolving DNS queries. Some of the analysed applications show, associated to different hostnames, the same nameservers responsible for responding to DNS names requests. As an example, both the hostnames `z-m-gateway.facebook.com` and `scontent-mxp1-1.xx.fbcdn.net` are associated to the `facebook.com` nameserver.

The favicon is a small icon stored in a file having the .ICO extension. The icon file is retrieved from the desired hostname and the hash is computed locally using the algorithm proposed by Kieran M [51], which consists of encoding the icon at first in base 64, then decoding it in UTF-8 string format and computing MMH3 (MurmurHash3) function to get the final digest.

Not all the flows present in the given input file are considered for the final clustering task. In case a flow is generated either by the OS of the monitored device, and not by an application, or by a web browser, then it is removed, as it is not relevant for the performed analysis. In order to notice that a flow is produced by the OS, the proposed algorithm is provided with a list of hostnames that are to be excluded from the flows to analyse. The listed hostnames correspond to system processes, such as checking for new push notifications to display, implementing vocal assistant services (`geller-pa.googleapis.com` [52]).

Some examples of such flows, belonging either to Android or to iOS devices, are `android.googleapis.com`, `courier.push.apple.com`, `dls.di.atlas.samsung.com`, and `sdkconfig.ad.intl.xiaomi.com`. On the other hand, flows that are generated by a web browser are more difficult to recognise: it is not easy discerning a legitimate application, connecting to one of its associated hostnames, from a browser communicating with the same hostname. The solution proposed in this work employs the `ja4` fingerprint by `althouse_ja4` as follows. Different web browser mobile applications have been observed and their `ja4` signature stored in a key:value structure, which is then used when running the code to match against the provided flows. Every flow found having the same `ja4` as one of the observed web browser is removed and not considered for the following analysis.

In addition to flows generated by the device's OS or by a web browser, another category of traffic is discarded: flows belonging to ambiguous services. Among this group of discarded flows, one can find mostly third-party services employed by mobile application to get an insight on the behaviour of their users. For instance, in this category appear services for displaying advertisements and recommendations (`adsrvr.org`, `criteo.com`, `outbrain.com`, etc.), providing marketing analytics (`appsflyer.com`, `apptentive.com`, `singular.net`, `usbla.net`, etc.), supplying software tools (`fastly.net`, `firebaseio.com`, `instabug.com`, etc.), managing privacy and security (`iubenda.com`, `onetrust.io`, etc.), offering location services (`radar.io`). Furthermore, it was not possible to identify some of the contacted hostnames, which were excluded as, for instance, the corresponding domain names where inactive or for sale. Such flows could not be associated to any actual mobile application and were, thus, filtered out. Many of the hostnames listed above were identified by employing the information provided by the resources made available by Netify [53].

Subsequently, this first step of the proposed algorithm performs an inspection for captivity check connections. A captivity check is the action of verifying whether a device can freely browse the network or if, when trying to access the Internet, it is presented with a captive portal. A captive portal is a landing page, often displayed the first time a device connects to a new network, which may ask users to log in, or to accept an EULA (End-User Licence Agreement), in or-

der to be able to browse the Internet. Some of the hostnames used to check for captive portals are: `captive.apple.com`, `connectivitycheck.gstatic.com`, `msftconnecttest.com`, `detectportal.firefox.com`, `nmcheck.gnome.org`, and `connect.rom.miui.com`. Captive portal information is produced as output by the method described in this work, but it is not currently being used in the clustering process of flows: it is excluded given that it is generated by the device's OS.

The last step of this first part consists in assigning labels to the flows. The labels are needed in order to automatically verify the final results, as well as for computing the evaluation metrics on the produced output. Labeling is not needed for the actual clustering phase, as this method does not employ any training step. In order to carry out the labeling process, each flow's hostname is checked against a dictionary data structure of hand-selected host and domain names, containing the key:value association of host/domain name and corresponding application. In case the selected flow is generated by one of the applications present in the dictionary, it is assigned as label that application name. Alternatively, in the event that the analysed hostname has never been observed before (and, hence, it does not appear in the dictionary), the `proto_by_ip` attribute, provided by `ndpiReader`, is employed instead. This concludes the initial processing of the input data.

3.3.2 Storing Obtained Data for Later Runs

The second part of the proposed algorithm consists in storing the data obtained in the first step in order to be able to reproduce the test at a later time. Information acquired by establishing a connection to a host is slow to obtain as it requires to create a new connection for each flow appearing in the input file, as well as to wait for the host to respond with the requested information. Some of the hosts take a few seconds to send back the information, while other completely fail to respond, slowing down the whole process. The proposed work employs a technique for minimising the hosts to contact that involves storing in some dictionary-like data structures the following information: the already contacted hostnames for which the requested information was obtained successfully, the list of hostnames that did not respond to the requests (either due to errors or timeouts), the list of domain names successfully contacted when their respective hostname was not available,

and finally the list of domain names that could not be reached. This methodology allows to check whether the required information has already been retrieved when analysing a previous flow, or if it needs to be obtained by contacting the host, and, in this case, if the host is reachable or not. Moreover, there were instances in which contacting an hostname resulted in an error, but connecting to the corresponding domain name allowed to retrieve the required information. The described data structures make it possible to establish a new connection only when necessary, never trying to obtain the same information twice and never contacting twice an host proven to be unreachable.

In addition to being useful for the reasons mentioned above, storing the data related to DNS name, nameservers and TLS certificates, is relevant in order to be able to run the test on the same input file at a later date. It may happen that, repeating the same test some time later than the date in which the packets were captured, one finds different results. This is due to the fact that information like DNS names and nameservers can change overtime, thus altering the results of the proposed algorithm runs performed within some time of each other. Furthermore, companies offering online services contacted by the monitored application may decide to remove some of the information required by this work. This is the case, for instance, of the fields associated with the domain name registration information. One of the parameters initially considered to perform the clustering task was the domain name registrant organization. The registrant organization is the entity who registers the domain name, entering a contract with a registrar (the entity responsible for the domain name registration service): at this moment the registrant agrees to the terms and conditions proposed by the registrar, as well as accepting to provide accurate and updated information to be displayed in registries, such as those accessed by WHOIS. For the purpose of this work, registration was checked initially using the WHOIS protocol and, in the end, employing its successor RDAP (Registration Data Access Protocol). It was noticed that many companies choose not to make the registrant organization available, and instead opt to display a text notifying the information is not provided to the public. The lookup tool provided by ICANN [54], implemented using RDAP, has been used to check that, at the moment of writing, even well known domain names such as `instagram.com` or

`tiktok.com` return the following text: "The RDAP server redacted the value". Due to the mixed availability of information regarding the domain name registrant, it was decided to discard it as a parameter for the clustering process detailed in this work, as the data was not reliable.

3.3.3 Performing the Clustering Step

The third part of the algorithm contains the actual clustering process. The main idea is to, at first, create a partition of the set of given flows, consisting in very small groupings, and, subsequently, in the case they meet specific conditions, to merge the obtained groups.

The first subsets are created by taking into consideration the `first_seen` values. As mentioned in Section 3.2, the method consists in chronologically sorting the traffic in input, creating a sliding window over the data and clustering together the flows having high time proximity. In order to evaluate whether two flows belong to the same cluster three possible threshold values are considered: the arithmetic mean, the difference between the arithmetic mean and the standard deviation, and the difference between the arithmetic mean and double the standard deviation. The value of the threshold can be passed in input to the Python code, thus making it possible to try different values and check which one yields the best results. In spite of that, the default value was chosen to be the latter, as, being the smallest of the three, it led to clusters with low enough cardinality to be containing only one application name and never mix two different applications in the same cluster. This choice can be ascribed to the fact that the proposed algorithm produces clusters by merging previously identified groups, and never by splitting an already created cluster. This implies that assigning flows belonging to different applications to the same cluster, at any time during the execution process, would certainly result in an error in the produced output, as the mistake could not be remedied in the following steps of the procedure, given a cluster is never separated into two smaller ones.

After having created some initial clusterings by observing the values of the `first_seen` attribute, the produced groups are merged together according to the values of some of the previously described metadata. The merging phase is

performed in an iterative fashion, as the employed merging algorithm carries out almost the same steps for each attribute. In particular, the attributes used in the merging phase are the SHA1 of the TLS certificate, the list of DNS names, the domain name, the list of nameservers and the MMH3 of the favicon. The merging algorithm performs the following operations. For each one of the attributes listed above, it scans the current clusters in pairs checking for matches that indicate that two groups of flows belong to the same application. At the end of the process, all clusters marked for merging are joined into one single cluster.

The only difference in how the procedure is executed on the different attributes consists in the definition of the matching indicating that two clusters must be merged into one. In case the considered attribute is the domain name, then two clusters are deemed to be merged if the output of the similarity score presented in Section 3.2 is greater or equal to 0.85. The value of this threshold has been carefully identified during the algorithm testing phase. A similarity of 0.85 allows to group together flows having domain names, for instance `tiktokcdn-eu.com` and `tiktokcdn-us.com`, while still differentiating among apparently similar domain names belonging to different applications. Instead, when checking for matches the remaining attributes, namely the list of SHA1 of the TLS certificates, the list of DNS names, the list of nameservers, and the MMH3 of the favicon, two clusters are enqueued for the merging process in case the intersection among the currently observed attribute list is not empty. This means that there is at least a common value among the two clusters when comparing the lists those described above.

In the end, attributes like `ja4` or `tcp_fingerprint` were not utilized for performing the clustering step. It was found out, in fact, that many different applications share the `ja4` fingerprint, thus making it a non-unique attribute for identifying specific applications. This is due to the fact that `ja4` looks for precise matches in the parameters exchanged during the TLS protocol handshake, which mainly depend on the TLS library adopted by the application, and have little connection with the actual application generating the traffic. The `tcp_fingerprint` value, instead, is more of an OS fingerprint, more suitable for correctly identifying the device, or a device family, than for implementing an application fingerprint. Still, `tcp_fingerprint` appears in the output returned by the presented algorithm, asso-

ciating the detected devices each with their values of `tcp_fingerprint`, as a way to monitor their behaviour. It typically happens that a device has at most two different values of `tcp_fingerprint`, and noticing a device appearing with a large number of distinct values for the attributes can arise suspicions, for instance, to the use of a hotspot mechanism.

3.3.4 Computing Evaluation Metrics

The final part of the algorithm consists in returning the clustered flows and computing different evaluation metrics in order to better understand the correctness of the obtained results. The clusters identified by the proposed method are compared to the groups that can be deduced from the labeling assigned to the flows in Subsection 3.3.1. As already mentioned, the labels are only employed in this final step so as to gain an understanding of the quality of the returned clusters in terms of similarity to the labels.

Because the task consists in clustering network traffic, and not in classifying it into predefined classes, it is not possible to know from the start how many clusters will be identified. Additionally, common evaluation metrics used for clustering methods inadequately apply to the proposed algorithm, as they are based on a notion of distance among the items to classify. Such metrics are, for instance, the Silhouette Score and the Davies-Bouldin Index, which rely on a definition of distance that ineffectively applies to the network traffic flow representation employed in this algorithm. For these reasons, the chosen evaluation metrics are Rand index and adjusted Rand index.

The first was described for the first time by Rand [55] and it can be used to measure the similarity of two clusterings in terms of agreements and disagreements between the two. In particular, in [55], Rand index is defined as follows. Given a set $X = \{X_1, \ldots, X_N\}$ of items and two different clusterings of the items in X, namely $Y = \{Y_1, \ldots, Y_{K_1}\}$ and $Y' = \{Y'_1, \ldots, Y'_{K_2}\}$, then the similarity measure $c(Y, Y')$ between Y and Y' is computed as:

$$c(Y, Y') = \sum_{i<j}^{N} \gamma_{ij} \Big/ \binom{N}{2},$$

where the term γ_{ij} is computed in the following way:

$$\gamma_{ij} = \begin{cases} 1, & \text{if } \exists k, k' \text{ such that both } X_i \text{ and } X_j \text{ are assigned} \\ & \text{to both } Y_k \text{ and } Y'_{k'} \\ 1, & \text{if } \exists k, k' \text{ such that } X_i \text{ is assigned to both } Y_k \text{ and } Y'_{k'}, \\ & \text{while } X_j \text{ is assigned to neither} Y_k \text{ nor } Y'_{k'} \\ 0, & \text{otherwise} \end{cases}$$

In other words, the Rand index $c(Y, Y')$ can be seen as the ratio between the agreements among the pair of clusterings Y and Y' and the total number of pairs. Specifically, the numerator consists of the number of pairs that have been put in the same cluster by both Y and Y', and by the number of pairs that have been put in different clusters by both Y and Y'. The denominator accounts instead for all the possible pairs, meaning the agreements already discussed, as well as all the occurrences of pairs appearing in the same cluster in Y and in different clusters in Y', and vice versa. Rand index assumes values in the interval $[0, 1]$, reaching 0 when there are no agreements (for instance Y contains only a single cluster, while Y' is composed of singletons), and 1 in case Y and Y' represent the same clustering structure.

Rand index measures the similarity between two clusterings in terms of items assigned to the same clusters or to different clusters in the two partitioning. By employing the method proposed in [55], it may happen, though, that two random clusterings manage to obtain a high value of the index by chance. For this reason, Hubert and Arabie [56] introduced an adjusted version of Rand index taking into consideration random clusterings. Given two clusterings, $X = \{X_1, \ldots, X_r\}$ and $Y = \{Y_1, \ldots, Y_s\}$, the adjusted Rand index is defined in the following way:

$$\frac{\sum_{ij} \binom{n_{ij}}{2} - \left[\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}\right] \Big/ \binom{n}{2}}{\frac{1}{2}\left[\sum_i \binom{a_i}{2} + \sum_j \binom{b_j}{2}\right] - \left[\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}\right] \Big/ \binom{n}{2}}$$

The meaning of n_{ij}, a_i, and b_j, can be understood by looking at the contingency

table represented below. The inner elements of the table, namely n_{ij}, represent the number of items of belonging both to the cluster X_i and to the cluster Y_j, that is to the intersection $X_i \cap Y_j$. The elements in the last row and in the last column of the table are the sums of the values on the same column and row respectively: the cardinality of each one of the Y clusters is displayed in the last row, while the cardinality of each one of the X clusters is displayed in the last column.

$X \backslash Y$	Y_1	Y_2	$\cdots$	Y_s	sums
X_1	n_{11}	n_{12}	$\cdots$	n_{1s}	a_1
X_2	n_{21}	n_{22}	$\cdots$	n_{2s}	a_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
X_r	n_{r1}	n_{r2}	$\cdots$	n_{rs}	a_r
sums	b_1	b_2	$\cdots$	b_s	

The value of the adjusted Rand index is bounded above by 1, which is the case of equality in the analysed clusterings, but it can also assume negative values. The adjusted Rand index is 0 whenever the clustering has no better similarity than that expected of two random partitions. As a consequence, a value of 1 indicates a perfect match between the compared clusters, 0 indicates that the match is the same as in the case of random clusterings, and negative values represent a match even worse than random.

A final evaluation on the clustering results is obtained by constructing a confusion matrix [57]. Usually, confusion matrices are used to evaluate classification methods, not clustering ones. This is due to the fact that a confusion matrix can be seen as a graphical representation of the four categories: true positive, false positive, true negative, and false negative. A binary classifier can only yield two possible results when assigning a class to an item, given that there are only two possible classes: the presence of the distinguishing feature or its absence. As a consequence, true positive and true negative correspond to the case in which the classifiers assigns the correct class to the item; a false positive happens when the classifier mistakenly indicates the presence of the feature; a false negative occurs in case the classifier wrongly reports the absence of the feature. The resulting

confusion matrix is a 2×2 square matrix having as rows the two assignments to classes predicted by the classifier, and as columns the actual classification. The notion can then be extended from binary classification to the case of multi-class classifiers by simply extending the matrix to have the same numbers of columns and rows as the total number of classes, indicating on the rows and column labels of the matrix the labels of the classes.

This approach ineffectively applies to evaluate a clustering process, because, even if the number of expected classes is known, it cannot be taken for granted that the number of resulting clusters will correspond to it. The algorithm proposed in this work can return as output a different number of clusters, depending, for instance, on the different threshold parameters previously described. The number of expected classes, instead, depends on the labels assigned in the first phase of the algorithm (Subsection 3.3.1) and it is fixed. For this reason, a slightly modified version of the confusion matrix is employed: the number of rows corresponds to the number of clusters identified by the proposed algorithm, while the number of columns is given by the number of labels assigned to the traffic. The inner elements of the matrix, then, represent the number of items appearing in the intersection of each cluster with each label-grouping. The resulting matrix is no longer square, being it (# clusters) $\times$ (# labels). The graphical representation of the matrix is useful in gaining a better understanding of the correctness of the clustering process: the ideal output would be to have a sparse matrix having a single non-zero element per row and per column, as this would imply a perfect clustering, completely adhering to the expected outcome represented by the labels.

In conclusion, three different evaluation metrics are employed to inspect the results obtained. Rand index and adjusted Rand index are metrics commonly used to evaluate clustering methods: values of the indexes closer to 1 are associated with a high similarity of the clusters in output to the ground truth. The modified confusion matrix, instead, is exploited in order to offer a graphical representation of the clustering results.

As a final note, the process detailed in the current Section is the one corresponding to an "online" run, implying that the information used for clustering is retrieved by establishing a communication with the host appearing in the currently analysed

flow. Given that, as already explained, the process of fetching this information is the most time consuming and it may not be repeated with the same results at a later time, the proposed method offers the possibility of storing the retrieved data for future runs. As a consequence, the algorithm can be executed in a similar way to the one specified above, by passing as input, in addition to the .json file containing the flows, the data file consisting of the associations among hostnames and corresponding retrieved data. The only change in the execution process, therefore, is that data is fetched seamlessly from the provided file, making the process faster, as well as repeatable.

3.4 Output Data

The format of the output file returned by the algorithm described above is a .txt file, not conforming to any specific formatting. The unconstrained nature of the output format is due to the fact that, at the current stage of the development of the proposed algorithm, there are no tools that employ its results for some analysis. The text file given in output contains the following information:

- the list of flows employing a `ja4` fingerprint previously observed on browser traffic;
- the clusters outputted at every step of cluster creation and of cluster merging, for a better understanding of the weight and the effect of each attribute in the merging process;
- the list of flows appearing in the given input file sorted in chronological order, along with some of the attributes' values;
- the list of flows grouped together according to the cluster they belong to;
- the values of Rand index and adjusted Rand index respectively;
- the confusion matrix computed on the clustering in output.

3.5 On the Exclusion of ML Methods

The method described at length in the previous Section (3.3) only employs basic techniques for the computation of similarities among the flows, making no use of machine learning algorithms. This has been a choice taken at the beginning of the research process into the topic. As it can be noticed from Section 2.3, the current methodologies on the topic of network traffic classification have been thoroughly explored and the majority of them, especially the most recent ones, employ ML methods. On the other hand, simple non-ML techniques, as the one proposed by [39], exploiting the analysis of the SNI attribute, are competitive and can even surpass ML methods in terms of accuracy, proving that ML is not the only way to solve network traffic classification tasks efficiently.

The approach proposed in this work defines a simple algorithm, that only employs notions of distance and similarity to cluster together flows according to the mobile application they belong to. The clustering process does not require any previous training to build a model and it is computationally efficient, as it only checks the values of some predetermined attributes of the given flows. The lightweight quality of the proposed approach, coupled with the good level of accuracy achieved (shown in Section 4.2), make this algorithm a viable candidate for the classification of network traffic, showing that methods other than ML can be used with comparable results.

In general, on the matter of network traffic classification, one of the most time-consuming and cumbersome tasks is the labeling of data, as it is difficult to acquire large labeled datasets and, in the case of collecting one's own data, to apply the correct label to it is not a trivial task. The creation of a labeled dataset is a difficult step both for ML models and for the algorithm described in this work, but it is necessary in the first case in order to train the model and in the second to compute the evaluation metrics. In the proposed methodology, labeling is handcrafted, as described in Subsection 3.3.1, which allows for precision and fine-grained control over the labels' definition. The major drawback of manually associating data with its label is the amount of time required by the process. The effort is warranted by the acquisition of a deeper knowledge of the reason why flows are labeled a

certain way. Additionally, the proposed algorithm does not require labeled data to perform the clustering of flows, as labels are only needed for the final computation of the evaluation metrics. For this reason, the algorithm can be run, and clustering can be performed, even on previously unknown data, meaning data not having a label yet, as the only operations of the process requiring the presence of labels are the metrics showing the correctness of the clustering in output. The fact that the proposed algorithm does not need labels to create the clusters is an advantage when comparing it to ML models requiring ground truth, and potential retraining on unknown data, in order to perform the clustering.

In summary, the reasons for undertaking the development of a traffic classification strategy that does not involve ML are: a more lightweight model, a deeper understanding of the clustering-relevant features and an independence from data labeling, as far as the clustering process goes.

Chapter 4

Results

The current Chapter will discuss the methodology employed to capture traffic as well as the results obtained by the algorithm on the collected data. As mentioned before, the algorithm proposed in this work is not computationally intensive, and it can be executed on a laptop. The majority of the execution time is spent retrieving data from the hosts present in the given input file. For this reason, the algorithm can run for a longer time in case the input file is lengthy and contains many unreachable hostnames, as opposed to the running time required when executing the offline version of the algorithm.

4.1 Data Collection

The input data has been obtained from various mobile devices, running different versions of either Android or iOS. A laptop having a wired (Ethernet) Internet connection was used to create a hotspot to which mobile devices could connect. Each device has been connected to the laptop, which, running the Wireshark [58] tool, captured the traffic and encoded it in .pcapng format. Table 4.1 shows the mobile devices used to conduct the tests, as well as their OS and the amount of captured traffic in terms of number of flows. The data has been captured both while the mobile devices were idle and while they were in use. Specifically, in the latter case, the captured traffic has been generated by the device's owner in order to more closely mimic real-life traffic generated by a normal user. The observed traffic contains, for instance, several applications, including social networks, mobile operators' apps, video sharing platforms, music and video streaming apps, and online shopping platforms.

Device id	Device model	Device OS	Number of flows
1	Redmi Note 11 Pro 5G	Android 13	2718
2	Redmi Note 13 Pro	Android 15	2477
3	Samsung Galaxy A34 5G	Android 15	807
4	Google Pixel 8	Android 16	1905
5	Apple iPhone	iOS 15.7	298
6	Apple iPhone	iOS 18.1	809

Table 4.1: Mobile devices employed for the evaluation process, along with their OS and the amount of data collected from each in terms of number of flows. The device id is employed in the following Sections to mention each specific device in a clearer way.

4.2 Experimental Results

As mentioned in Section 3.1, the .pcapng files, obtained using Wireshark, were given in input to the `ndpiReader` tool from nDPI, getting as output a .json file for each packet capture. The current Section will analyse the results in terms of the metrics described in Subsection 3.3.4, in particular by examining the confusion matrices computed for each device. Tables 4.2, 4.3, 4.4, 4.5, 4.6, and 4.7 show the confusion matrices, having as rows the labels and as columns the clusters' id, calculated on the results obtained by the proposed algorithm on the devices displayed in Table 4.1. For each one of the considered mobile devices, the corresponding matrix's caption displays the values of Rand index and adjusted Rand index. The confusion matrices shown below display the intersections among the clustering identified by the labels and the clustering obtained as output of the algorithm. The matrices are not square, as it is not necessarily true that the number of clusters found by the proposed method corresponds to the number of labels, but rather the algorithm often distinguishes more clusters than the number of labels. Additionally, it may be noticed that the number of flows per device displayed in Table 4.1 is greater than the sum of non-zero elements of the confusion matrices. This discrepancy is due to the fact that a portion of the flows present in the input file is discarded (Subsection 3.3.1) for belonging either to the OS, to a web browser, or to traffic not ascribable to any known mobile application.

The best-case scenario would be to obtain square confusion matrices having only one non-zero element per row and per column, as this would imply perfect adherence of the clusters identified by the proposed algorithm to the labels, which

represent the ground truth. As a consequence, the case of occurrences of non-square matrices, or of multiple non-zero elements in some of the rows or columns, would highlight an error in the clustering process.

The first case corresponds to the event in which flows belonging to the same application are assigned to different clusters by the proposed algorithm. Given the fact that the method described in this work never splits already created clusters, but only merges them, the occurrence of a non-square confusion matrix underlines the fact that the attributes currently employed for the clustering process are not informative enough to completely identify all of the mobile applications present in the traffic in input.

The second case represents the event in which there is more than one non-zero value per row or per-column. In case these elements appear in the same row, then the analysis done in the case of non-square matrices holds: flows generated by the same applications are placed into different clusters by the proposed algorithm as none of the considered attributes allows to merge them into a single cluster. Otherwise, in case there is more than one non-zero element per column, the cause can be found in the fact that (at least) two clusters were wrongly merged, implying that (at least) one of the attributes used in the merging phase is not specific enough and it is used by more than one application. It can be noticed that there is only one occurrence, specifically in Table 4.5, of a cluster containing more than one application label: it can be observed that cluster 7 contains both the applications Microsoft Teams and Copilot. In this particular case, the two applications are assigned to the same cluster because, during the merging steps, the two corresponding clusters are joined when examined for the domain name attribute. Even though this event results in an error, it is mistake difficult to correct as Copilot, being the generative AI (Artificial Intelligence) chatbot developed by Microsoft, appears in the input file as a set of six flows having `copilot.microsoft.com` as hostname, which shares the domain name with the hostname associated with the Microsoft Teams application, that is `teams.microsoft.com`.

For what concerns the case of flows belonging to the same application being assigned to different clusters, if an application is not correctly grouped into one single cluster in one of the devices, then, in every other device in which it appears,

it also happens that the applications' flows are assigned to different clusters. The only exception to this behaviour is the case of the Spotify application, which in device 6 is contained in a single cluster, but, given that in that device Spotify appears in only one flow, then this is not significant. Table 4.8 displays all of the applications that the proposed algorithm erroneously assigns to more than one cluster. In particular, the Table accounts for the fact that some devices do not exhibit traffic belonging to a particular application by showing the number of flows, as well as the number of clusters to which those flows are assigned to. This event is significant as it highlights the fact that, even if the proposed algorithm is able to correctly identify most applications by assigning them to the same cluster, there are cases in which the attributes chosen for the merging iterations are not informative enough to completely characterise the traffic in input.

In terms of attributes used for clustering, the results obtained from devices 1, 2, and 3 have been examined in order to deduce which of the attributes have the bigger impact on the final output. The three devices have been chosen for this analysis as they are the ones with the largest amount of flows and applications. The clustering of device 1 consists of a total of 565 flows. In the beginning, the creation of the clusters by means of time proximity, employing the `first_seen` attribute, creates small clusters; 506 of the total 565 flows are reassigned to another cluster during the first merging phase, exploiting the SHA1 of the TLS certificate. In the next two steps, utilising respectively the list of DNS names and the domain names, 40 and 178 flows respectively are assigned to a different cluster by merging. In a similar way, device 2's flows, totalling up to 512, are merged into bigger clusters in steps of 441, 29, and 148 flows employing respectively the attributes: SHA1 of TLS certificate, list of DNA names, and domain names. Likewise, the number of device 3's flows, amounting to 490 after the filtering, that are merged together employing the same three attributes, are 451, 20, and 126 respectively. In every one of the three mentioned devices, the amount of flows merged together due to the other attributes is very low, in the range of $[0, 6]$ flows per attribute. As a result, additional attributes, such as the list of nameservers and the MMH3 of the favicon, are only rarely needed to perform the clustering, as the majority of flows have already been grouped together in the previous merging iterations: the

predominant part of grouping together flows in the merging step is performed employing only the three attributes mentioned above. This does not necessarily imply that the less impactful attributes are useless, as they are sometimes needed in order to merge together clusters belonging to the same application that none of the previous merging steps was able to join, and that would thus result in an error in the final output if left as separate clusters.

Rand index and adjusted Rand index can be used, along with the confusion matrices discussed above, to gain an understanding of the correctness of the clusters obtained when running the algorithm, in terms of how closely they match the labels acting as ground truth. As discussed in Subsection 3.3.4, Rand index, even though it assumes values ranging in the interval $[0.9484, 0.9856]$, cannot be taken as the only measure of correctness of the results, as it does not account for random clustering. When considering, instead, the values assumed by adjusted Rand index, it can be noticed that the methodology proposed in this work performs well, having values of $[0.7986, 0.9462]$. The average values of Rand index and adjusted Rand index are 0.9746 and 0.9013 respectively, meaning that device 6, obtaining 0.9484 and 0.7986 in the two metrics respectively, deviates from the behaviour observed in the clusters corresponding to the other devices. Most probably, this inconsistency in performance is due to the fact that the errors committed in device 6 when clustering TikTok and Zalando applications (that have been assigned to two and three clusters respectively) have a greater impact when computing the two evaluation indexes, as the flows belonging to the two applications amount to 42% of the total number of flows of device 6.

In conclusion, the methodology proposed to execute the clustering of mobile application traffic performs well, reaching an adjusted Rand index value of 0.9462, with an average of 0.9013. The errors in classification can be explained in a satisfactory way as either applications that have been developed by the same company, and thus share attributes' values (as is the case of Microsoft Teams and Copilot), or by the lack of more precise attributes to join together clusters of flows generated by the same application (Table 4.8). Additionally, the fact that there is only one occurrence of a cluster mixing together two different applications is a positive aspect, as it follows that the methodology described in this work can

be improved by finding new attributes on which to perform additional merging steps. Finally, it has been noticed that the most useful flow characteristics that can be employed to match traffic belonging to the same application are, in order of importance: the SHA1 of the TLS certificate, the domain names, and the list of DNS names.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**
Pinterest	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TooGoodToGo	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**5**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Whatsapp	0	0	**5**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PayPal	**15**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Vinted	0	0	0	0	**96**	0	0	0	0	0	**3**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Clue	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Letterboxd	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**44**	0	0	0	0	0	0	0	0	0	0	0
Shein	0	0	0	0	0	0	**15**	0	0	**9**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Trenitalia	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**2**
TheFork	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**5**	0	**1**	0	0	0	0	0	0	0	0
TheStorygraph	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**4**	0	0	0
Splitwise	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**	0	0	0	0	0	0	0	0	0	0	0	0	0
Snapchat	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Outlook	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**109**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Goin′	0	0	0	0	0	0	0	0	0	0	**4**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Poste	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**9**	0	0
Facebook	0	**61**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Howbout	0	0	0	0	0	0	0	0	0	0	0	**3**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TikTok	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**8**	0	**1**	0	**77**	**12**	0	0	0	0
CoopVoce	0	0	0	0	0	0	0	0	**11**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Instagram	0	0	0	**6**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
YouTube	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**16**	0	0	0	0	0	0
Spotify	0	0	0	0	0	**2**	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**28**	0	0

Table 4.2: Device 1: Redmi Note 11 Pro 5G (Android 13).
Rand index: 0.9856.
Adjusted Rand index: 0.9291.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
ArubaPEC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0
Trenitalia	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Outlook	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**68**
Instagram	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**7**	0	0	0	0	0	0	0	0	0	0	0	0	0
Poste	0	0	0	0	0	0	0	0	**8**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Ryanair	0	0	0	0	0	0	0	0	0	**9**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Dice	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**	0	**4**	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0
EasyPark	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**12**	0	0	0	0	0	0	0	0	0	0	0
Airbnb	0	0	0	0	0	0	0	0	0	0	**11**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
YouTube	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**6**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Splitwise	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0
PayPal	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**14**	0	0	0	0	0	0	0	0	0	0
Whatsapp	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**6**	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Vinted	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**41**	0	0	0
TheStorygraph	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**6**	0
Spotify	**64**	**3**	0	**17**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
JustEat	0	0	0	0	**2**	**2**	**5**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TikTok	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**13**	**1**	0	0	0	0	0	0	0	0	0	**5**	**18**	0	0	0	0	0	0	0	0
Internazionale	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**46**	0	0	0	0	0	0
Steam	0	0	0	0	0	0	0	0	0	0	0	**4**	**3**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MicrosoftTeams	0	0	0	0	0	0	0	0	0	0	0	0	0	**22**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Facebook	0	0	**79**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RaiPlay	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**7**	0	0
Letterboxd	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	**17**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 4.3: Device 2: Redmi Note 13 Pro (Android 15).
Rand index: 0.9856.
Adjusted Rand index: 0.9092.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
Instagram	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**7**	0	0
TooGoodToGo	0	0	0	**2**	**2**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Whatsapp	0	0	0	0	0	0	0	0	0	0	0	**23**	0	0	0	0	0	0	0	0
PayPal	0	0	0	0	0	0	0	0	0	0	**4**	0	0	0	0	0	0	0	0	0
Trenitalia	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**	0	0	0	0
Outlook	0	0	0	0	0	0	0	0	0	0	0	0	0	**86**	0	0	0	0	0	0
Facebook	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**55**	0	0	0
Firefox	0	0	0	0	0	0	0	0	**3**	0	0	0	0	0	0	0	0	0	0	0
Bandcamp	**17**	0	**87**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Vinted	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**81**	**4**
Ryanair	0	0	0	0	0	**28**	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TikTok	0	0	0	0	0	0	0	0	0	0	0	0	**11**	0	**49**	0	0	0	0	0
Clue	0	0	0	0	0	0	**7**	**2**	0	0	0	0	0	0	0	0	0	0	0	0
CoopVoce	0	0	0	0	0	0	0	0	0	**17**	0	0	0	0	0	0	0	0	0	0
YouTube	0	**2**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 4.4: Device 3: Google Pixel 8 (Android 16).
Rand index: 0.9803.
Adjusted Rand index: 0.9126.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
MicrosoftTeams	0	0	0	0	0	0	0	**40**
Outlook	0	0	0	0	0	0	**70**	0
Spotify	**1**	0	**6**	0	0	0	0	0
Linkedin	0	0	0	0	0	**2**	0	0
Facebook	0	**5**	0	0	0	0	0	0
Whatsapp	0	0	0	**12**	0	0	0	0
UpDay	0	0	0	0	**2**	0	0	0
Copilot	0	0	0	0	0	0	0	**6**

Table 4.5: Device 4: Samsung Galaxy A34 5G (Android 15).
Rand index: 0.9761.
Adjusted Rand index: 0.9462.

	0	**1**	**2**	**3**	**4**	**5**	**6**
YouTube	0	0	0	**4**	0	0	0
Twitter	0	0	**11**	0	0	0	0
Facebook	0	0	0	0	**6**	0	0
Discord	**2**	**1**	0	0	0	0	0
Linkedin	0	0	0	0	0	**6**	**2**

Table 4.6: Device 5: Apple iPhone (iOS 15.7).
Rand index: 0.9718.
Adjusted Rand index: 0.9124.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
TikTok	0	0	0	**12**	0	0	0	**11**	0	0	0	0
MicrosoftTeams	0	0	0	0	0	0	0	0	0	**18**	0	0
Pinterest	0	0	0	0	0	0	0	0	0	0	0	**3**
Facebook	**4**	0	0	0	0	0	0	0	0	0	0	0
Snapchat	0	**5**	0	0	0	0	0	0	0	0	0	0
Subito	0	0	0	0	0	0	**11**	0	0	0	0	0
Spotify	0	0	0	0	0	0	0	0	0	0	**1**	0
Ryanair	0	0	0	0	0	0	0	0	**3**	0	0	0
Zalando	0	0	**7**	0	**2**	**1**	0	0	0	0	0	0

Table 4.7: Device 6: Apple iPhone (iOS 18.1).
Rand index: 0.9484.
Adjusted Rand index: 0.7986.

	Device 1		Device 2		Device 3		Device 4		Device 5		Device 6	
	#flows	**#clusters**	**#flows**	**#clusters**	**#flows**	**#clusters**	**#flows**	**#clusters**	**#flows**	**#clusters**	**#flows**	**#clusters**
Bandcamp	0		0		104	2	0		0		0	
Clue	4	2	0		9	2	0		0		0	
Dice	0		8	3	0		0		0		0	
Discord	0		0		0		0		3	2	0	
TheFork	6		0		0		0		0		0	
JustEat	0		10	4	0		0		0		0	
Letterboxd	45	2	18	2	0		0		0		0	
Linkedin	0		0		0		2	1	8	2	0	
Shein	24	2	0		0		0		0		0	
Spotify	31	3	84	3	0		7	2	0		1	1
Steam	0		7	2	0		0		0		0	
Tiktok	98	4	37	4	60	2	0		0		23	2
TooGoodToGo	6	2	0		4	2	0		0		0	
Vinted	99	2	42	2	85	2	0		0		0	
Zalando	0		0		0		0		0		10	2

Table 4.8: Applications assigned to different clusters by the proposed algorithm. The table shows, for each one of the examined devices, the amount of flows per application and the number of clusters the flows are distributed into.

Chapter 5

Conclusions

This work describes a methodology for the task of network traffic classification, in particular for the identification of mobile applications. The proposed algorithm leverages simple notions of similarity in order to associate flows together, making no use of ML methods. The performance is evaluated in terms of adjusted Rand index and confusion matrices, reaching an average correctness of the clustering of 90.13%.

5.1 Challenges

The implementation of the proposed algorithm has been an incremental process: many attributes have been examined and considered for the merging procedure: some have been discarded, for instance the `ja4` fingerprint, while others have been added as a way to attempt to reduce the errors in the clustering, for example the MMH3 of the favicon.

The goal, throughout the development of the method described in this work, has been to create clusters that most closely resembled the groupings identified by the labels. In order to achieve this objective, priority was given to the identification of features that allowed to merge together groups of flows believed to be produced by the same mobile application. For this reason, it was preferable to have more clusters corresponding to the same label, than it was to have more labels referring to same cluster. In the first case, in fact, one could try to find additional attributes to implement a further merging step on, in order to join together clusters belonging to the same application. In the second case, instead, because the proposed method works in merging steps (and never splits any of the identified clusters), having more

labels pertaining to the same cluster results in an error that cannot be corrected in the subsequent operations of the process. As observed in Section 4.2, the majority of the clustering errors can be ascribed to the first case, and thus can be corrected by identifying more attributes that accurately characterise each application.

The exclusion of flows that are irrelevant to the identification of mobile applications employs the values of `ja4` fingerprint (for web browsers), as well as a list of hostnames that do not directly belong to any application. The values of browsers' `ja4` have been incrementally updated by testing different browsers and devices, but an extensive amount of devices, with different versions of OSs, would be required in order to have a comprehensive inventory of browsers' fingerprints. On the other hand, the list used for filtering out flows includes hostnames related to different platforms, ranging from advertising, to marketing, to user tracking, which can be employed by the applications' developers as a third-party service, thus being shared by multiple mobile applications. As for the `ja4` values, the list of hostnames has been updated as new devices were tested and such hostnames were discovered. In both cases, the records are not as thorough as it would be desirable, as that would require a greater amount of devices and of applications to test. Additionally, some of the examined applications share the value of `ja4` fingerprint with a web browser: this can be justified by the fact that some of the content rendering internally performed by the application exploits the system browser. However, it could also be the case that some of the applications have the same value of `ja4` as a browser, but they differ from a browser in their value of `ja4_raw_uns`. This possibility could be better explored by more carefully analysing the `ja4_raw_uns` of applications and browsers.

5.2 Future Works

The proposed methodology manages to reach a good level of accuracy, but it could be improved in the following ways.

Surely, in order to achieve better clustering results, more attributes need to be identified to perform the merging iterative steps. The current version of the algorithm produces good enough clusterings in terms of adjusted Rand index,

but the majority of the shortcomings of the produced output are due to the same application appearing in different output clusters. These mistakes could be fixed by performing additional merging phases on new attributes.

A second refinement in the clustering process could be implemented by monitoring the succession of chronologically sorted flows in order to identify patterns. User tracking, advertisement, and monitoring services' flows are irrelevant on their own, as they can be generated by multiple applications, and thus cannot be unambiguously associated with a specific one. Nevertheless, there could be an observable behaviour in the succession of flows having a close time proximity: it could happen that an application always contacts the same hostnames in the same order, even if some the hostnames refer to third-party services and not to the application's main service. Recognising these patterns would allow to better identify the observed traffic and lead to more accurate results in terms of clustering, as the patterns could be employed as another feature to use in the merging steps of the proposed method.

Additionally, because some mobile applications share the value of `ja4` fingerprint with a browser, they are excluded from the clustering process by the current version of the proposed algorithm. Indeed, a possible enhancement to this work could be brought about by the identification of a different way to recognise a browser, as this would enable the algorithm to avoid filtering out many flows, and thus to perform the clustering step with a much larger set of them. Having more flows to examine could, in fact, lead to a more precise identification of the software running on the monitored devices, as well as balancing the computation of the evaluation metrics, redistributing the weight of the clustering errors in the final output.

A final improvement to this work could be the creation of an automated system, which would take a .pcap, or .pcapng, file in input and return as output the clustering of flows, along with some measure of accuracy. Currently, the clusters in output are not associated with a name, as they are only identified by a number. The automated system could estimate the correct name of the application linked to each of the clusters, in order to provide, with a level of confidence, the user with a list of the applications installed into the monitored device. The process of identifying each

one of the clusters with a name could be performed, for instance, by prompting a LLM (Large Language Model) with the lists of hostnames associated with each one of the clusters in order to receive in output an application name per list.

Bibliography

[1] European Parliament and Council of European Union. Directive (EU) 2016/1148 of 6 July 2016 concerning measures for a high common level of security of network and information systems across the Union. *Official Journal of the European Union*, L194/1, 2016. URL https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016L1148. Accessed: 04-10-2025.

[2] European Parliament and Council of European Union. Directive (EU) 2022/2555 of 14 December 2022 on measures for a high common level of cybersecurity across the Union, amending Regulation (EU) No 910/2014 and Directive (EU) 2018/1972, and repealing Directive (EU) 2016/1148 (NIS2 Directive). *Official Journal of the European Union*, L330/80, 2022. URL https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32022L2555. Accessed: 18-09-2025.

[3] NIS2 Directive: securing network and information systems, 2025. URL https://digital-strategy.ec.europa.eu/en/policies/nis2-directive. Accessed: 12-10-2025.

[4] Cyber Resilience Act, 2025. URL https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act. Accessed: 13-09-2025.

[5] ISO/IEC 7498-1:1994. Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model. Standard, International Organization for Standardization, November 1994. URL https://www.iso.org/standard/20269.html. Accessed: 11-09-2025.

[6] M. Cotton, L. Eggert, J. Touch, M. Westerlund, and S. Cheshire. Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry. RFC 6335, RFC Editor, August 2011. URL https://www.rfc-editor.org/rfc/rfc6335.html. Accessed: 24-10-2025.

[7] Service Name and Transport Protocol Port Number Registry. URL https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml. Accessed: 24-10-2025.

[8] Tomasz Bujlow, Valentín Carela-Español, and Pere Barlet-Ros. Independent comparison of popular DPI tools for traffic classification. *Computer Networks*, 76:75–89, 2015. ISSN 1389-1286. doi:10.1016/j.comnet.2014.11.001.

[9] Merve Çelebi, Alper Özbilen, and Uraz Yavanoğlu. A comprehensive survey on deep packet inspection for advanced network traffic analysis: issues and challenges. *Niğde Ömer Halisdemir Üniversitesi Mühendislik Bilimleri Dergisi*, 12(1):1–29, 2023. doi:10.28948/ngumuh.1184020.

[10] K. Moriarty and A. Morton. Effects of Pervasive Encryption on Operators. RFC 8404, RFC Editor, July 2018. URL `https://dl.acm.org/doi/pdf/10.17487/RFC8404`. Accessed: 04-11-2025.

[11] Arthur S. Jacobs, Roman Beltiukov, Walter Willinger, Ronaldo A. Ferreira, Arpit Gupta, and Lisandro Z. Granville. AI/ML for Network Security: The Emperor has no Clothes. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1537–1551. Association for Computing Machinery, 2022. ISBN 9781450394505. doi:10.1145/3548606.3560609.

[12] Vanice Canuto Cunha, Arturo Zavala Zavala, Damien Magoni, Pedro R. M. Inácio, and Mário M. Freire. A Complete Review on the Application of Statistical Methods for Evaluating Internet Traffic Usage. *IEEE Access*, 10:128433–128455, 2022. doi:10.1109/access.2022.3227073.

[13] Auwal Sani Iliyasu, Ibrahim Abba, Badariyya Sani Iliyasu, and Abubakar Sadiq Muhammad. A review of deep learning techniques for encrypted traffic classification. *Computational Intelligence and Machine Learning*, 4(1):17–25, 2023. URL `https://www.academia.edu/download/106697999/review-deep-learning-techniques.pdf`. Accessed: 05-11-2025.

[14] Jawad Hussain Kalwar and Sania Bhatti. Deep Learning Approaches for Network Traffic Classification in the Internet of Things (IoT): A Survey. 2024. doi:10.48550/arxiv.2402.00920.

[15] Jason Pittman. A Comparative Analysis of Port Scanning Tool Efficacy. March 2023. doi:10.48550/arXiv.2303.11282.

[16] Gordon "Fyodor" Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*, chapter 4. Port Scanning Overview. Nmap Project, 2009. ISBN 9780979958717. URL `https://nmap.org/book/`. Accessed: 06-11-2025.

[17] Zakir Durumeric, Eric Wustrow, and J Alex Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, 2013. ISBN 9781931971034. URL https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/durumeric. Accessed: 06-11-2025.

[18] Robert David Graham. MASSCAN: Mass IP port scanner, 2014. URL https://github.com/robertdavidgraham/masscan. Accessed: 06-11-2025.

[19] Luca Deri and Ivan Nardi. A Deep Dive Into Traffic Fingerprints using Wireshark. SharkFest'24 EUROPE, 2024. URL https://sharkfest.wireshark.org/retrospective/sfeu/sf24eu/. Accessed: 07-11-2025.

[20] Philip Perricone, Bill Hudson, Blake Anderson, Brian Long, and David McGrew. Joy. A package for capturing and analyzing network data features. January 2018. URL https://github.com/cisco/joy/blob/master/doc/using-joy-05.pdf. Accessed: 07-11-2025.

[21] Philip Perricone, Bill Hudson, Blake Anderson, Brian Long, and David McGrew. Joy. A Package for Capturing and Analyzing Network Data Features. TLS Fingerprinting Addendum. February 2019. URL https://github.com/cisco/joy/blob/master/doc/using-joy-fingerprinting-00.pdf. Accessed: 07-11-2025.

[22] David McGrew, Brandon Enright, Blake Anderson, Lucas Messenger, Adam Weller, Andrew Chi, Shekhar Acharya, Anastasiia-Mariia Antonyk, Oleksandr Stepanov, Vigneshwari Viswanathan, and Apoorv Raj. Network Protocol Fingerprinting (NPF): A Flexible System for Identifying Protocol Implementations. URL https://github.com/cisco/mercury/blob/main/doc/npf.md. Accessed: 07-11-2025.

[23] John Althouse, Jeff Atkinson, and Josh Atkins. TLS fingerprinting with JA3 and JA3S, January 2019. URL https://engineering.salesforce.com/tls-fingerprinting-with-ja3-and-ja3s-247362855967/. Accessed: 02-11-2025.

[24] John Althouse. JA4+ Network Fingerprinting, September 2023. URL https://blog.foxio.io/ja4%2B-network-fingerprinting. Accessed: 02-11-2025.

[25] Ahmad Azab, Mahmoud Khasawneh, Saed Alrabaee, Kim-Kwang Raymond Choo, and Maysa Sarsour. Network traffic classification: Techniques, datasets, and challenges. *Digital Communications and Networks*, 10(3):676–692, 2024. ISSN 2352-8648. doi:10.1016/j.dcan.2022.09.009.

[26] Yifan Wang, Minzhao Lyu, and Vijay Sivaraman. Characterizing User Platforms for Video Streaming in Broadband Networks. In *Proceedings of the 2024 ACM on Internet Measurement Conference*, IMC '24, page 563–579, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705922. doi:10.1145/3646547.3688435.

[27] Christelle Nader and Elias Bou-Harb. Revisiting IoT Fingerprinting behind a NAT. In *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 1745–1752, 2021. doi:10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00235.

[28] Yige Chen and Yipeng Wang. MPAF: Encrypted Traffic Classification With Multi-Phase Attribute Fingerprint. *IEEE Transactions on Information Forensics and Security*, 19: 7091–7105, 2024. doi:10.1109/TIFS.2024.3428839.

[29] Shahbaz Rezaei, Bryce Kroencke, and Xin Liu. Large-Scale Mobile App Identification Using Deep Learning. *IEEE Access*, 8:348–362, 2020. doi:10.1109/access.2019.2962018.

[30] Riccardo Bortolameotti, Thijs van Ede, Andrea Continella, Jingjing Ren, Daniel J. Dubois, Martina Lindorfer, David Choffnes, Maarten van Steen, and Andreas Peter. FlowPrint: Semi-Supervised Mobile-App Fingerprinting on Encrypted Network Traffic. *Network and Distributed System Security Symposium (NDSS)*, 27. doi:10.14722/ndss.2020.24412.

[31] Md. Shamim Towhid and Nashid Shahriar. Encrypted Network Traffic Classification using Self-supervised Learning. In *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, pages 366–374, 2022. doi:10.1109/NetSoft54395.2022.9844044.

[32] ZiXuan Wang, ZeYi Li, MengYi Fu, YingChun Ye, and Pan Wang. Network traffic classification based on federated semi-supervised learning. *Journal of Systems Architecture*, 149:103091, 2024. ISSN 1383-7621. doi:10.1016/j.sysarc.2024.103091.

[33] Ting-Li Huoh, Yan Luo, Peilong Li, and Tong Zhang. Flow-Based Encrypted Network Traffic Classification With Graph Neural Networks. *IEEE Transactions on Network and Service Management*, 20(2):1224–1237, 2023. doi:10.1109/TNSM.2022.3227500.

[34] Zulu Okonkwo, Ernest Foo, Zhe Hou, Qinyi Li, and Zahra Jadidi. Encrypted network traffic classification with higher order graph neural network. In Leonie Simpson and Mir Ali Rezazadeh Baee, editors, *Information Security and Privacy*, pages 630–650. Springer Nature Switzerland, 2023. ISBN 978-3-031-35486-1. doi:10.1007/978-3-031-35486-1_27.

[35] Zulu Okonkwo, Ernest Foo, Zhe Hou, Qinyi Li, and Zahra Jadidi. A graph representation framework for encrypted network traffic classification. *Computers and Security*, 148:104134, 2025. ISSN 0167-4048. doi:10.1016/j.cose.2024.104134.

[36] Akbar Telikani, Amir H. Gandomi, Kim-Kwang Raymond Choo, and Jun Shen. A Cost-Sensitive Deep Learning-Based Approach for Network Traffic Classification. *IEEE Transactions on Network and Service Management*, 19(1):661–670, 2022. doi:10.1109/TNSM.2021.3112283.

[37] Saadat Izadi, Mahmood Ahmadi, and Amir Rajabzadeh. Network Traffic Classification Using Deep Learning Networks and Bayesian Data Fusion. *Journal of Network and Systems Management*, 30(2), 2022. ISSN 1573-7705. doi:10.1007/s10922-021-09639-z.

[38] Weiping Zheng, Jianhao Zhong, Qizhi Zhang, and Gansen Zhao. MTT: an efficient model for encrypted network traffic classification using multi-task transformer. *Applied Intelligence*, 52(9):10741–10756, 2022. ISSN 1573-7497. doi:10.1007/s10489-021-03032-8.

[39] D. Shamsimukhametov, M. Liubogoshchev, E. Khorov, and I.F. Akyldiz. Are Neural Networks the Best Way for Encrypted Traffic Classification? In *2021 International Conference Engineering and Telecommunication*, pages 1–5, 2021. doi:10.1109/EnT50460.2021.9681767.

[40] Ahmet Aksoy, Sushil Louis, and Mehmet Hadi Gunes. Operating system fingerprinting via automated network traffic analysis. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 2502–2509, 2017. doi:10.1109/CEC.2017.7969609.

[41] Desta Haileselassie Hagos, Anis Yazidi, Øivind Kure, and Paal E. Engelstad. A Machine-Learning-Based Tool for Passive OS Fingerprinting With TCP Variant as a Novel Feature. *IEEE Internet of Things Journal*, 8(5):3534–3553, 2021. doi:10.1109/JIOT.2020.3024293.

[42] Blake Anderson and David McGrew. Os fingerprinting: New techniques and a study of information gain and obfuscation. In *2017 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9, 2017. doi:10.1109/CNS.2017.8228647.

[43] Martin Lastovicka, Tomas Jirsik, Pavel Celeda, Stanislav Spacek, and Daniel Filakovsky. Passive os fingerprinting methods in the jungle of wireless networks. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9, 2018. doi:10.1109/NOMS.2018.8406262.

[44] Martin Husák, Milan Čermák, Tomáš Jirsík, and Pavel Čeleda. HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting. *EURASIP Journal on Information Security*, 6(1), 2016. ISSN 1687-417X. doi:10.1186/s13635-016-0030-7.

[45] Sherry Bai, Hyojoon Kim, and Jennifer Rexford. Passive OS Fingerprinting on Commodity Switches. In *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, pages 264–268, 2022. doi:10.1109/NetSoft54395.2022.9844109.

[46] Michal Zalewski. p0f v3 (3.09b), 2000-2014. URL `https://lcamtuf.coredump.cx/p0f3/`. Accessed: 14-11-2025.

[47] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Krügel, and Engin Kirda. Automatic Network Protocol Analysis. In ISOC, editor, *NDSS 2008, 15th Annual Network and Distributed System Security Symposium, February, 15-18 2008, San Diego, USA*, San Diego, 2008. URL `https://www.auto.tuwien.ac.at/~chris/research/doc/ndss08_protocol.pdf`.

[48] Luca Deri. Parser for ndpiReader JSON files, 2024. URL `https://github.com/ntop/nDPI/blob/dev/utils/parse_reader_json.py`. Accessed: 15-11-2025.

[49] Luca Deri, Maurizio Martinelli, Tomasz Bujlow, and Alfredo Cardigliano. ndpi: Open-source high-speed deep packet inspection. In *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 617–622. IEEE, 2014. URL `https://luca.ntop.org/nDPI.pdf`. Accessed: 09-11-2025.

[50] difflib — Helpers for computing deltas. SequenceMatcher Objects. URL `https://docs.python.org/3/library/difflib.html#sequencematcher-objects`. Accessed: 10-11-2025.

[51] Kieran M. favicon-hash.kmsec.uk. URL `https://github.com/kmsec-uk/favicon-hash.kmsec.uk`. Accessed: 10-11-2025.

[52] Gerd. HaGeZi DNS Blocklists. URL `https://github.com/hagezi/dns-blocklists/issues/2900`. Accessed: 13-11-2025.

[53] Netify - Network Intelligence. URL `https://www.netify.ai/resources`. Accessed: 10-11-2025.

[54] Registration data lookup tool. URL `https://lookup.icann.org`. Accessed: 11-11-2025.

[55] William M Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association*, 66(336):846–850, 1971. doi:10.2307/2284239.

[56] Lawrence Hubert and Phipps Arabie. Comparing partitions. *Journal of Classification*, 2 (1):193–218, December 1985. ISSN 1432-1343. doi:10.1007/bf01908075.

[57] Juri Opitz. A Closer Look at Classification Evaluation Metrics and a Critical Reflection of Common Evaluation Practice. *Transactions of the Association for Computational Linguistics*, 12:820–836, 2024. ISSN 2307-387X. doi:10.1162/tacl_a_00675.

[58] Wireshark (version 4.4.9). URL `https://www.wireshark.org/`. Accessed: 18-11-2025.