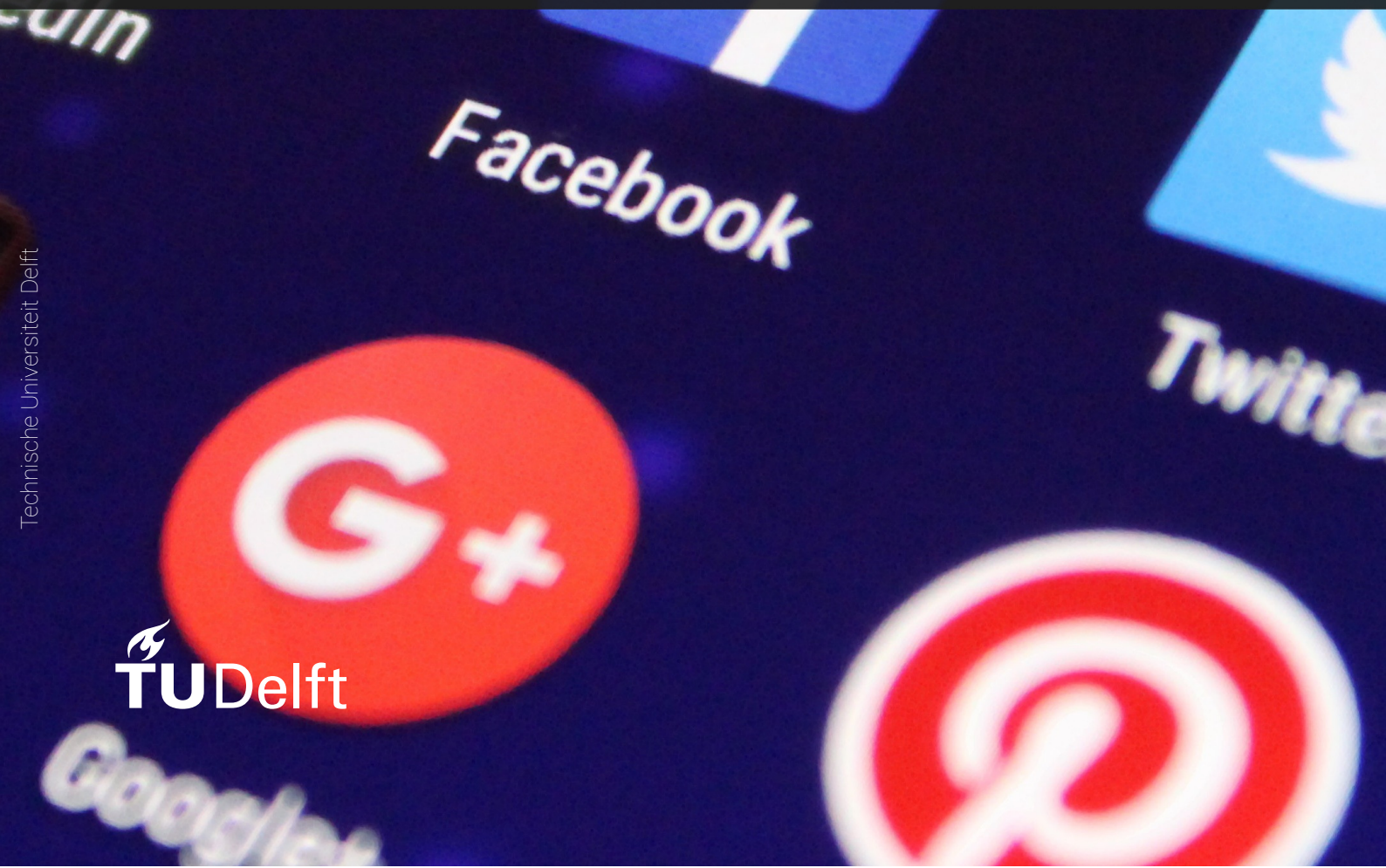
Android App Tracking

Investigating the feasibility of tracking user behavior on mobile phones by analyzing encrypted network traffic

W.K. Meijer



Technische Universiteit Delft

TU Delft

Android app tracking

Investigating the feasibility of tracking user behavior on mobile phones by analyzing encrypted network traffic

by

W.K. Meijer

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on 17 December 2019.

Student number:	4224426	
Project duration:	19 November 2018 – 17 December 2019	
Thesis committee:	Dr. C. Doerr,	TU Delft, supervisor
	Dr. ir. J.C.A. van der Lubbe,	TU Delft, chair
	Dr. F.A. Oliehoek	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

The mobile phone has become an important part of people's lives and which apps are used says a lot about a person. Even though data is encrypted, meta-data of network traffic leaks private information about which apps are being used on mobile devices. Apps can be detected in network traffic using the *network fingerprint* of an app, which shows what a typical connection of the app resembles. In this work, we investigate whether fingerprinting apps is feasible in the real world. We collected automatically generated data from various versions of around 500 apps and real-world data from over 65 unique users. We learn the fingerprints of the apps by training a Random Forest on the collected data. This Random Forest is used to detect app fingerprints in network traffic. We show that it is possible to build a model that can classify a specific subset of apps in network traffic. We also show that it is very hard to build a complete model that can classify all possible apps traffic due to overlapping fingerprints. Updates to apps have a significant effect on the network fingerprint, such that models should be updated every one or two months. We show that by only selecting a subset of apps it is possible to successfully classify network traffic. Various countermeasures against network traffic analysis are investigated. We show that using a VPN is not an effective countermeasure because an effective classifier can be trained on VPN data. We conclude that fingerprinting in the real world is feasible, but only on specific sets of apps.

Preface

When I started this research end of November 2018 I never expected to write so many pages. But to you, the reader, you'll just have to trust me when I say: reading them is worth it. At first, the plan was to finish somewhere at the end of the summer but I am glad that I took the time to work a few extra months on this thesis. In the past months so much more was added, completing the story, and I am happy to present the work that lies before you.

Although my name is on the front page, this work would not have been possible without others. Thanks to dr. Christian Doerr and ir. Harm Griffioen for their daily guidance, fruitful brainstorm sessions, and technical support. Thanks for all the talks we had, about my thesis and the cyber security world in general.

I also want to thank some others who have contributed positively to my day-to-day life at the TU Delft. Thanks to everyone on the 6th floor for studying, drinking coffee, and lunching together. You made working on my thesis every day much more enjoyable. Rico, thank you for having the same pace in your research as I had and thank you for letting me defend earlier than you. Thanks to Sandra, for always trying to improve our lives; by giving us coffee, getting the sun blinds installed, and helping with submitting the correct forms.

And finally, to give credit where credit is due: The icons used in the figures are made by Eucalyp, Dinosoft-Labs, Freepik, Good Ware, Google, Kiranshastry, prettycons, Smashicons, and srip from `www.flaticon.com`.

W.K. Meijer
Delft, December 2019

List of Figures

List of Tables

Contents

1 Introduction

As people started spending more of their time on the Internet, the interest of organizations into what people do there has risen as well. Nowadays, you are what you do online and, more and more, you are what you do on your smartphone. Many companies have built successful business models around the acquisition and sale of vast amounts of user data, either directly or through advertisement services. Privacy has become a commodity [61]. Intelligence and security agencies set up large data centers to analyze vast amounts of Internet data to find or track people of interest and Edward Snowden showed the world that they have been successful in doing so [31]. Criminals have leveraged the intertwinement of the Internet with people's lives for their activities as well. For example by listening in on connections and stealing sensitive information like credit card numbers, but also by gaining access to information that can be used for blackmailing.

Scandals like Cambridge Analytica have put the practice of data collection in the public eye. Lawmakers have responded by passing laws limiting the freedom of companies to collect data from their costumers and increasing the responsibility of those companies to inform their costumers of what data is being collected. Users have become more aware of how they are being tracked online. Many people notice how Google advertisements change based on their browsing behavior and know of trackers like the infamous Facebook pixel and similar analytics services [34]. The widespread usage of browser add-ons that block advertisements and trackers is living proof of this increased awareness.

1.1. Network traffic analysis

While most types of tracking are noticeable, network traffic analysis cannot be detected by users since it does not operate on the user's device. By inspecting the networks which bring data to and from users, organizations can effectively learn what users are doing online. As a user needs a way to send and receive data, this type of analysis cannot be blocked like tracking cookies. Network traffic analysis is a great tool for organizations interested in behavioral information because it can be used for tracking user activity on any device as long as the attacker has access to the network. The interest of organizations in this network traffic analysis was confirmed by revelations from Techcrunch in January 2019 that Facebook and Google pay people (even minors) to install an app that reroutes all network traffic through their servers [12, 13].

Tracking people online by analyzing network traffic used to be trivial because network traffic was sent unencrypted over the wire. Anyone who had control over the wire could read all communications without much effort. Over time, websites and applications started to make more and more use of encrypted connections, which hide the content of the messages that are sent. This forced organizations that want to track users, to leverage other types of methods to track people and is the cause of the rise of analysis purely based on meta-data.

Over the last 10 years, people are increasingly using applications on their mobile phones, instead of a web browser, to do their online business. Information about which apps someone uses can reveal a lot about the person, as shown in research by Seneviratne et al. [52]. In the research, single snapshots of installed apps on mobile phones were taken and that information was used to say something about the traits of the owner of the phone. If a single snapshot can already reveal information about the traits of a user, usage statistics can say vastly more. This is why network traffic analysis poses such a great threat, as it potentially shows when apps are being used without the user even noticing that they are being tracked.

Mobile phone applications, unlike websites, do not necessarily have a single remote point from which data is requested. So where tracking which websites people visit is easy since the destination is clear from the metadata, this is far from trivial for mobile apps. Different parts of the application can request data from different sources, and multiple apps can request data from the same source. This makes it harder to identify a mobile application using metadata as compared to identifying websites, where the website can be identified from the source directly. The rise of Cloud infrastructure and Content Delivery Networks, which make it possible to store data on a dynamic set of servers, have made the analysis even harder as the source of the information changes from time to time.

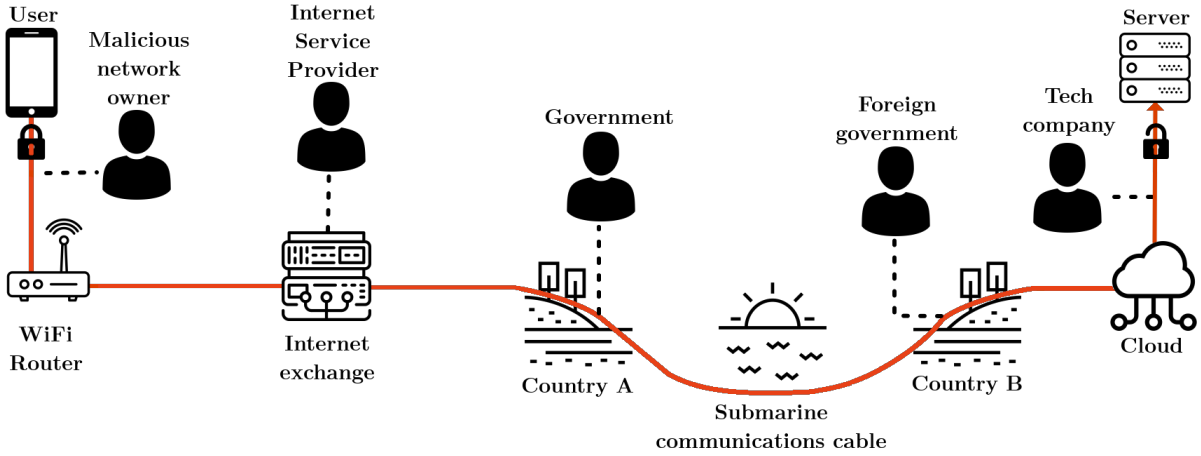


Figure 1.1: A conceptual overview of at which points in a connection various attackers can perform network traffic analysis. A user wants to connect to some app that makes use of a cloud provider to host their content. Various actors have access to the encrypted connection from the phone to the remote server. A malicious network owner could simply copy all network traffic that goes over their WiFi. An ISP has access to all the network traffic coming through their network. The government is able to capture traffic from the large Internet cables that connect countries. And finally, the company hosting the cloud service has access to the network traffic as well. All these actors might have different motivations, but all of them are able to compromise the user's privacy.

1.2. Mobile app fingerprinting

Since mobile apps cannot be identified trivially from meta-data, other methods are required for identifying apps in network traffic. Analyzing statistical information about network traffic has proven to be very effective in the past to obtain a so-called *network fingerprint* of an application or service. A network fingerprint consists of a series of features, like size and arrival time, extracted from the network traffic the application generates. For example, an application that is streaming a video has a very distinct series of network activity every few seconds where the video is being buffered. The advantage of doing this type of fingerprinting on this type of meta-data is that it can be done without inspecting the actual content of the network traffic. It does not require unencrypted network traffic and can thus be applied by anyone that has access to the communication channel over which the data is sent.

Figure 1.1 gives a conceptual overview of an encrypted connection to a remote server and shows at which points this attack can be performed. At each point, a different actor comes into play. A user might be on a malicious WiFi network whose owner copies all network traffic. This actor can be interested in what someone is doing on their phone for blackmailing purposes. While the data is moving further an Internet Service Provider, who can sell personal information to data brokers [43], has access to the data. At the global infrastructure level, governments have access to the large communication cables that connect countries. They can use network traffic analysis to find anyone using apps that are disliked by the government. It is known that governments are actively monitoring these cables. For example, Snowden revealed that the British intelligence agency GCHQ is actively copying data that is being sent over submarine fiber-optic cables [40]. Finally, the tech companies providing hosting services for many of the Internet's applications can also easily access the encrypted network traffic and use it to gain more insight into the interests of people.

All the actors in Figure 1.1 have in common that they have the motivation and the means to perform the attack. The attacker captures encrypted network traffic and extracts meta-data. This meta-data is used as input to a previously trained model that classifies which apps generated the network traffic. Since most apps only generate a lot of traffic when they are used in the foreground, the resulting classification gives an exact

overview of when apps were used. A user never notices that one of these attacks is ongoing since the attack does not disturb the connection itself. This overview shows the potential huge attack surface of using network traffic analysis to fingerprint mobile apps.

If it is indeed possible to successfully build machine learning models for fingerprinting apps in encrypted network traffic, it is very easy to execute attacks that use this technique. While the breach of privacy of mobile phone users sounds bad, it might be hard to get a good idea of how exactly network traffic analysis can be leveraged by an attacker. Therefore, to illustrate this, we give three examples of attacks.

A hacker could be interested in targeting specific apps to exploit while wanting to keep a low profile, to avoid triggering a fix for the exploit. They do not want to try out the exploit on all possible devices without knowing if the targeted app is running on that device. To achieve this, they can perform network traffic analysis to identify which devices in a network run the targeted apps with a high degree of confidence. An individual attacker can probably build a model for just a few apps and collect training data themselves. But if this proves to be too much effort, this type of network traffic analysis or the collection training data has the potential to be sold as a service on the black market.

Data brokers or advertisement companies can be attackers that are interested in building profiles of users. While it is infeasible to reliably classify all apps, they can make a selection of apps with rather unique fingerprints and classify network traffic with those. This will not give a complete overview of the app usage of a user, but for these companies, even a partial profile is a useful asset. The attack is focused on a very large amount of users, individual users can be identified by matching their public IP address to already known other information. To successfully implement this attack not only does the attacker need a lot of training data to build a model for many apps, but they also need access to large amounts of network traffic data. This would require collaboration with large cloud providers or ISPs. This attack could also be attractive to regimes that do extensive surveillance of their citizens.

A company can be interested to see the effect of their policies on the usage of specific apps by their employees. For example, they can build models that can classify social apps that employees use instead of working. Instead of blocking access to these types of apps they want to nudge employees to use their time more effectively. Because a company has complete control over its internal network, they can easily measure how the usage of these apps changes when new policies are implemented and whether there are differences between different parts of the companies.

1.3. Research statement

Previous research has successfully created fingerprints for network traffic from Android apps and trained a machine learning model on this. These models take network data as input and classify the network data as belonging to a specific application based on the fingerprint. Previous research often does not test classifiers on real-world data and experiments are performed on small amounts of data. In these controlled settings it has been shown that network traffic analysis can be successfully applied, with accuracy scores of well over 90%. While previous work claims that the methods can be applied easily in the real world too, the feasibility of applying them in the real world has not been proven yet.

In this thesis, we investigate whether the methods of previous work can indeed be applied in the real world. Our goal is to provide a critical review of the previous work and investigate the circumstances in which the models do and do not work. Furthermore, we take a look at how models could be trained with ease, how often they need to be updated and how much data is required for a good result. We investigate how fingerprints of apps differ from each other to be able to say something about the scalability of the attack. We focus only on Android phones since that platform is most accessible and has a large market share of over 75% [55]. Lastly, we investigate various countermeasures that attempt to make the fingerprints look alike, such as the usage of a VPN service.

A thorough analysis of the reasons behind the results gives a much deeper understanding into which attacks are realistic, the capabilities an attacker would need, and how attacks can be mitigated. This will allow us to answer the following research question:

Is it feasible to track user behavior on the mobile phone platform through analysis of encrypted network traffic in a real-world environment?

The problem with using classifiers to analyze real-world data is that the data originates from a very dynamic set of devices, users, network environments, and app versions. There exist many apps in the world and thus fingerprints might look quite similar between some apps. Investigating the impact of those factors on the

performance of classifiers is, therefore, necessary to get a good idea of the feasibility. Furthermore, the possibility of mitigation and the existence of other methods to obtain information about user behavior need to be investigated to prove the feasibility of this type of network analysis. The following sub-questions should thus be answered before a proper answer to the main research question can be formulated.

- What is the impact of app updates on classifier performance?
- How does the number of apps under consideration affect the classifier?
- How well can a classifier that is trained on automatically generated data, classify real-world data?
- What is the impact of different network environments on classifier performance?
- What are effective countermeasures against user behavior tracking through network traffic analysis?

The main contributions of this research are that 1) we provide methods for deep understanding of why a classifier works on certain apps; 2) we show that the performance of a classifier is very dependent on the set of apps under consideration; 3) contrary to assumptions made in previous work, we show that real-world data classification cannot be used to classify all apps and all network traffic, but is only possible in specific circumstances; 4) we show that commonly known countermeasures against network traffic analysis are not effective.

This thesis is structured as follows. Chapter 2 provides background information about various topics that are relevant for this research. In chapter 3 we discuss literature about network traffic analysis on the mobile platform, this allows us to find research gaps. Chapter 4 discusses the methodology and the setup of the research and corresponding experiments based on findings by previous research. In chapter 5 we present the results of various closed world experiments, focused on investigating the effect that app updates have on the performance of a classifier. Our classification methods are tested on real world data in chapter 6. We investigate the effectiveness of various countermeasures against network traffic analysis in chapter 7. Finally, we conclude the thesis in chapter 8.

2 Background

We do not expect the reader to have all the knowledge required to understand everything written in this thesis. For a few of the subjects that are essential for this research, we provide some background information. In this chapter, we explain how a connection is opened, knowledge that helps understand how we can split the network traffic into flows and bursts and how we can obtain a ground truth for these flows and bursts. We explain how Transport Layer Security (TLS) works, which explains why the classification of traffic is still possible, despite the encryption of the data. We explain how a VPN works to understand how we collected our real-world data, used in chapter 6, and why a VPN might be an attractive countermeasure, as investigated in chapter 7. Finally, we give more information on Random Forests, which is the type of classifier used in all of our experiments.

2.1. Opening a connection

Applications can require data to be sent to or received from a remote server via the Internet. This requires a connection over the Internet from the host device to the remote server. To find the location where the data should be sent to each device that is connected to the Internet is assigned an IP address. To sent data to the remote server, data is sent to the IP address of the remote server. The return address is the IP address of the host device.

The Operating System (OS) of a device handles the incoming and outgoing network traffic. However, when data is simply sent to an IP address the OS does not know for which application the data is meant. As a solution, there is a detailed address called a port. Each device has 65 535 ports per network protocol. The combination of an IP address and a port gives a detailed location to where data should go. The OS keeps track of which application belongs to a port, so when data is received by the OS it can sent it to the correct application.

To open a connection an application opens a so-called socket. A socket is an abstraction of the connection, so the application is not required to deal with building the correct packets for sending data over the wire, this is all handled by the OS. A socket requires a remote IP address and port, so it is clear where the data should go. An application can simply put data into the socket and the data is then sent to the remote server. In the same way, the remote server can send its replies and the application receives these replies via the same socket. To receive these replies the OS needs to know for which application they are meant, so therefore each socket is assigned a local port. This local port together with the host's IP address forms the return address.

An application can either request a specific port or let the OS assign a port to the socket. An application can open as many sockets as required, the limit being the amount of unclaimed ports available on the OS. When an application tries to open a socket using a local port that is already in use, the request is denied by the OS and the application should try to request another port.

We distinguish between long-lived ports and short-lived ports (ephemeral). An example of a long-lived port is a web server that serves a web page to requesting clients. The web server has an open socket on port 80 and listens for any incoming requests for the web page. Because this port is always open a client is always sure where to send the request for the web page.

Short-lived ports are used as the return address for sockets opened by applications. An application requires data and once that data is received the application moves on. Because nothing else will be done with new data sent over that socket, the socket should be closed by the application, making the port short-lived.

One button press of a user can result in data being send over socket A, while another button press results in data being send over socket B. Often the traffic with a certain short-lived port can, therefore, be directly linked to a specific action within the application. The OS keeps track of which ports belong to which application. By analyzing network traffic and the list of which ports are assigned to which application, it is possible to get an overview of which applications are used and how they are used.

2.2. Transport Layer Security

In this research we are analyzing encrypted network traffic, specifically, we focus on network traffic encrypted using Transport Layer Security (TLS) [48], and its predecessor Secure Sockets Layer (SSL) [2]. TLS is the encryption standard for encrypting web traffic and is mainly known from the S in HTTPS, in which TLS is used to secure an HTTP connection. To get a grasp on how encrypted network traffic can be used for traffic analysis and how TLS does not prevent it, this section gives an overview of how TLS works.

TLS has the goal of providing a secure connection between two parties; providing confidentiality and integrity of the data and authentication of the parties. Authentication is achieved utilizing a Public Key Infrastructure (PKI), in which trusted third-parties attest that the communicating parties are who they say they are. There is a lot that can be said of PKI, but since this is not relevant for this thesis we will not go into it any further.

Confidentiality and integrity are achieved through the use of cryptography. Messages are encrypted using block cipher suites like Advanced Encryption Standard (AES) or stream cipher suites like ChaCha. The data integrity is checked using a keyed Message Authentication Code (MAC). During the initial handshake of the connection, information about the parameters for encryption and authentication are agreed upon.

Most TLS connections make use of a block cipher. A block cipher encrypts data blocks of a certain size. A piece of data that is to be encrypted should thus be an exact multiple of this block size. When this is not the case the data is padded with extra data to reach a total size of a multiple of the block size. For example, use AES-128, which has a block size of 16 bytes, to encrypt data of 139 bytes. The 139 bytes can be divided into 8 blocks of 16 bytes, which leaves 11 bytes. Thus to be able to encrypt all the data, 5 bytes of padding are required.

TLS is used to hide the contents of data by encrypting it. Therefore it is no longer possible to inspect the data and find sensitive information. However, TLS still uses specific sockets and each socket still has a unique port and a certain remote address that it connects to. This means that it is still possible to see how many individual sockets are being opened. With domain knowledge about specific applications and which remote servers they connect to, it is thus still possible to know which application is being used.

Some applications always sent or receive the same sized data for a specific action. Think for example about requesting a specific static web page, this is always the same size. When TLS uses a stream cipher encrypt the data the size of the encrypted data is the same as the size of the unencrypted data. It is, therefore, possible to say which page is being requested purely based on the size of the data. Block ciphers mitigate this a little bit by requiring the encrypted data to be of a size that is a multiple of the block size. However, unless the block size is very large it is fairly easy to still match the size of the encrypted data to a specific web page. The result is that TLS still leaks privacy sensitive information in some cases. The RFC of TLS acknowledges this in appendix E of RFC 8446 [48]: *TLS is susceptible to a variety of traffic analysis attacks based on observing the length and timing of encrypted packets.*

To mitigate these types of analysis TLS also gives the option to add more padding than required for a block cipher. Padding can thus be added so that the network traffic consists of packets of the same length. It is also possible to send packets that just consist of padded data, so the number of packets one action causes to be sent is changed as well. However, RFC 8446 also states that there are still other methods that can be used to figure out the size or type of unencrypted data. An attacker could for example measure the time it takes the server to process the network traffic and gain information about how much padding was added. It is not exactly clear how often TLS padding is actually enabled in practice.

2.3. VPN

A Virtual Private Network is a technique to create a private network over the public Internet. A VPN allows remote users to be part of a private network that normally requires access at a physical location. A common use is a company that uses a VPN service to allow their employees to work from home.

A VPN creates a secure tunnel from the client to the VPN. In this way the VPN acts as a private network, no one can (or should be able to) inspect or modify the connection. To connect to a VPN a client needs to connect

to a VPN server. The VPN server is an endpoint of the secure tunnel and from this server, the traffic can go to their final destination. All traffic from a client is thus routed via the VPN server. Via this VPN server, a client can access services that are only available in the Virtual Private Network or access other parts of the Internet.

The secure tunnel is established by encrypting all network traffic from either endpoint. Because all traffic is encrypted and the packets are rerouted to the VPN server there is no way of knowing what the actual destination of the network traffic is. This information is only available after the VPN server decrypts the traffic and forwards it, but now the source address is equal to the VPN server so the sender is not known anymore.

Private use of VPN's is often focused on hiding network traffic from e.g. Internet Service Providers, or malicious actors on the line. Commercial VPN providers provide servers to connect to, the public IP address of a user, therefore, becomes the IP address of the VPN server. Because many users connect to the same VPN server it becomes very hard to directly link a request made from a VPN server to a specific user.

When using a VPN there is only one connection open to the outside world, the connection to the VPN server. This means that, unlike simple TLS usage, it is no longer possible to link a single connection back to an application. Furthermore, because all traffic is sent over one connection, traffic from multiple applications are mixed, so the original size of the data from one application is no longer available even when no padding schemes are used.

2.4. Random Forests

Random Forests are a type of machine learning model that was first proposed by Tin Kam Ho [60], but in its current form credited to Breiman [9]. The model was proposed to solve the problem that complex machine learning models suffer from overfitting. Overfitting is the problem that a model is training on apparent structures in the testing data that are just noise. This means that the model will not work very well on new data as that will not contain the same noise. A Random Forest is comprised of an ensemble of Decision Trees.

2.4.1. Decision trees

To understand how Random Forests work exactly we first need to take a closer look at decision trees, the models that together form a Random Forest. Decision trees are data structures that can be used for the classification of data based on features of the data.

A decision tree consists of nodes, each node has one parent node and two or more child nodes. The input for a tree is a sample of data that should be classified. Each node poses a statement about a feature. For example: is feature X larger than 8? Based on the value feature X has for the sample in question we move to one of the children of the current node. This continues until a node is reached that has no children, this is called a leaf node. A leaf node does not pose a statement about a feature, but is an endpoint of the tree and assigns a class to the sample. An example of a decision tree to classify plants can be found in Figure 2.1. On each split of the tree, a question is posed about a feature of the plant. Based on the answer to the question we move to one of the children of the current node until we reach a leaf and have classified the plant.

A tree is created by looking at training data and figuring out which statements about the features groups features the best. For each splitting node, a feature is chosen and a statement about that feature. For complex data that is not easily clustered it can be very hard to find features that split the data cleanly and thus they can require a lot of splitting nodes. As long as the tree is large enough each sample of the training data can be described exactly by a decision tree, which will lead to overfitting. In practice, the number of nodes in a tree is often limited by setting the maximum depth of a tree; the number of levels of nodes in a tree. By tweaking this parameter overfitting can be controlled.

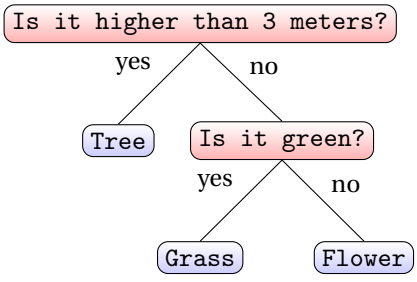


Figure 2.1: Example of a very simple decision tree for classifying a plant. The color and height of the plant are the features here. Tree, Grass, and Flower are the classes.

2.4.2. Structure

A Random Forest solves the problem of overfitting by not training one single model but instead training a lot of Decision Trees on random sub-samples of the data. In each of these trees, a random subset of features will be considered for each split. Therefore any noise will not become a prominent part of the model, as the trees will rarely make the same splits since they all consider a different part of the training data and different features

for each split. The chance that enough models will overfit so that the entire Random Forest overfits is very small. A Random Forest trains on the underlying distribution that exists in the training data and not on the exact features of the training data. A Random Forest will therefore rarely classify the training data with 100% accuracy because the Random Forest is not an exact fit of the training data. A regular Decision Tree will do this as long as the tree is large enough.

Samples that should be classified are put into each of the Decision Trees of the Random Forest. Each Decision Tree assigns a class to the sample. The Random Forest then uses a scorer to come to one classification based on the classifications of all its trees.

2.4.3. Important parameters

The training of a Random Forest can be controlled by setting several parameters. We discuss the most important parameters here. The *number of estimators* parameters sets the number of Decision Trees that are used to create the Random Forest. A higher number of Decision Trees means that there is less chance of overfitting.

The *maximum depth* sets the maximum depth each of the Decision Trees is allowed to have. A higher maximum depth means that a decision tree can use more splits to categorize the data into classes, leading to higher accuracy of the individual trees and thus the Random Forest as a whole. With a single decision tree, this setting is an important control for overfitting. In a Random Forest, this matters less as it compensates for overfitting by training on a large number of Decision Trees.

Both the *number of estimators* and the *maximum depth* have a significant influence on the time it takes to train a model because both settings control the total amount of splits that need to be found. There are various parameters to control how a splitting node should be found. These parameters can control how many features should be considered when finding the best split, how many samples are required for a split (avoids training on noisy samples), among other things.

2.4.4. Applications

Random Forests are used a wide variety of fields: biotech [18], civil engineering [44], cyber security [5, 53, 67], ecology [16], medicine [63], and many others.

Because the individual trees are trained on only a subset of the data, Random Forests can deal very well with high dimensional data (a lot of features) [6]. Each split only considers a subset of features, so the model scales very well as the amount of features rises. To get the most out of all the features in the data, the number of estimators should be increased, but this does not increase the overall complexity of the model.

Random Forests are robust to noisy data, as was already shown by Breiman [9]. The individual trees are trained on subsets of the data, so as long as the data is not too noisy the effect is mitigated as the noise will not be present in the training data for all of the trees.

Random Forests do not require a lot of pre-processing because they can deal with a lot of features and are robust to noise. This makes them very easy to use and applicable to a wide range of datasets.

3

Related Work

It has been known for more than 15 years now that privacy-enhancing technologies often fail to provide 100% privacy on the Internet. The first research about fingerprinting websites in encrypted network traffic emerged in 2003 [29]; it describes a proof of concept attack on the privacy-enhancing feature of an encrypting web proxy by analyzing the size of objects downloaded over an SSL tunnel. Research in 2006 improves on this proof of concept by doing a large evaluation of traffic analysis using statistical features on a packet level. This attack works on many encrypted tunnels other than SSL as well [7, 33]. Later research improves on the accuracies and robustness of the classification techniques by using different kinds of classifiers [28], investigates how to withstand mitigation techniques [37], and shows that highly regarded services like Tor are vulnerable to these kinds of attacks as well [45]. All of this research aims to show that there is a limitation in the privacy-preserving ability of the commonly used privacy-enhancing technologies.

Research into fingerprinting is not limited to websites only. Fingerprinting in network traffic has been successfully applied in detecting the tactics, techniques, and procedures of malicious actors [23–25, 38]. This wider usage of fingerprinting extends to mobile phone applications as well.

The mobile platform differs in many aspects from the World Wide Web and the techniques for web traffic analysis are not always transferable. Because it is relatively new, the analysis of encrypted mobile phone network traffic is not yet a mature research area. This chapter gives an overview of the state and progress in this research area and shows where there exist knowledge gaps. Most literature uses the pipeline in Figure 3.1 to go from data to a trained classifier. In this pipeline, the data is collected, processed to a usable format, and features are selected and extracted. Then the dataset is split into a training set and a testing set. The training set is used to train the classifier, and the trained classifier is used to classify the testing set. Finally, a scoring mechanism evaluates the performance of the classifier. We discuss literature on each step in this pipeline as a separate subject.

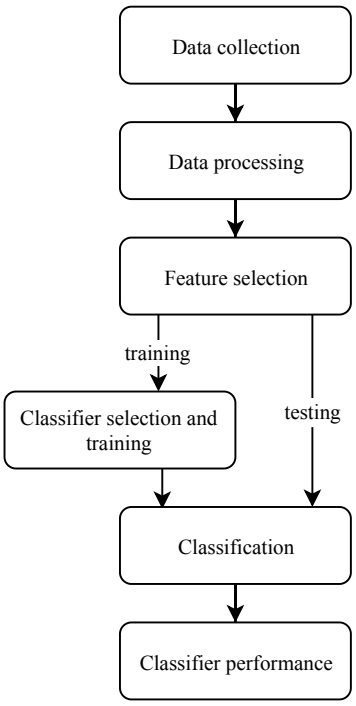


Figure 3.1: Pipeline that is used by most of the research in the area of network traffic analysis. The various steps in the pipeline are separate subjects which are discussed in this chapter.

3.1. Fingerprinting for mobile devices

As mobile phones started emerging people increasingly transferred their online activity from Personal Computers to mobile phones and with that web browsing partly switched to app usage. Therefore traffic analysis is

increasingly used to analyze the traffic from mobile devices to fill the knowledge gap on online activity [21, 66].

Early research investigates fingerprinting by analyzing unencrypted mobile phone network traffic. A problem, however, is that even in unencrypted traffic, it is not always clear from which app the traffic originates. Finding the app cannot simply be done by inspecting the HTTP traffic, because unlike the hostname for websites, the actual app name is almost never mentioned [41]. Other methods are thus required to fingerprint the apps. Dai et al. [17] built state machines of the HTTP requests generated by apps, the idea being that apps have a distinct set of HTTP requests. Miskovic et al. [41] use tokens in HTTP traffic to build a knowledge base with mappings from these tokens to apps. Tokens are generic key-value pairs in URLs and substrings of HTTP header fields. They then extract all tokens from new HTTP traffic and use their knowledge base to classify the traffic.

The wide adoption of encryption to secure network traffic on the mobile platform renders fingerprinting methods based on unencrypted traffic largely useless. A study from 2017 showed that 88% of apps in a representative dataset use HTTPS, with 50% using HTTP as well, the use of HTTPS is still increasing [47], therefore, unencrypted traffic analysis will be out of scope in this research. The same methods used for encrypted traffic analysis of websites have been applied to the mobile platform and has been a topic of research since 2013 [57].

3.2. Data collection

A major challenge for any supervised machine learning research is to obtain reliable ground truth data for training. The ground truth here means that for part of network traffic it is known from which app it originates. There are two main ways to obtain training data. One method is crowdsourcing participants and capturing data from their phones [36]. The other is to generate traffic in an automated way either through a dedicated phone or emulators. Network traffic is generated by apps when a user interacts with the phone. One way is to generate traffic is to only capture start-up traffic by launching the app [1]. It is also possible to provide random user input [58, 62], use scripts that automatically recognize and execute paths in an app [17], or write dedicated scripts that execute actions [15].

While crowdsourcing data will provide the most realistic data, it is not as scalable as automatically generating traffic. There is very little control over which apps are used. Automatically generating traffic using actual phones [15, 58, 62] can become very costly and requires complicated setups when scaling up. Using emulators [17, 41] thus provides a cheap and easy alternative to automatically generating traffic.

Labeling traffic by hand has been done for unencrypted traffic [17, 41], but these methods cannot be used reliably for encrypted traffic. Conti et al. [14], Park and Kim [46] use domain name inspection to label encrypted network traffic. However, this is an unreliable method since apps are known to share resources like software libraries and often make use of content delivery networks resulting in different apps querying the same domain names or IP addresses [1, 41, 58]. Domain name inspection can only be used for ground truth labeling if it is absolutely clear that those domain names or IP addresses are used only for one app, so this does not scale.

Taylor et al. [58, 59] solve this uncertainty by using a network logger app on smartphones that logs from which apps the network traffic originates, thus obtaining perfect ground truth data. Traffic is captured by dumping the traffic on the WiFi Access Point the smartphone is connected to. Liu et al. [36] use their own network logger [35] with similar functionality. This app also acts as a VPN provider and reroutes data to their own servers, allowing remote data collecting, which is useful for crowdsourcing. Wang et al. [62], Al-Naami et al. [1], and Dai et al. [17] run only one app at the time on their device/emulator when collecting traffic and label their traffic as the one app that was running. This might still lead to possible unwanted background traffic from the OS, so their ground truth is not perfect.

3.3. Network traffic processing

After collecting raw traffic data from mobile phones the data is processed so that it can be used as input for the classifiers. All reviewed research cleans up the TCP traffic beforehand by for example considering only TCP packets with payload and filtering out retransmitted packets. There are two competing methods of preprocessing data: burst separation [1, 57, 58, 62] and flow separation [14, 36, 58]. Burst separation is a method that aggregates packets that arrive within a predefined time of each other, this aggregation is called a burst. Bursts of network traffic are often generated by the application that is currently in use. [62] Flow separation is a method that aggregates packets on TCP sessions or which have the same source and destination IP and port number. Taylor et al. [58] first apply burst separation and then apply flow separation within the

bursts. Conti et al. [14] use a different definition of flows equivalent to a TCP session and further process these to obtain three time series of the total amount of bytes (1) incoming, 2) outgoing, and 3) combined between two peers. Al-Naami et al. [1] create time series of size and inter-arrival times of packets within uni-directional bursts, Liu et al. [36] does the same for flows.

These preprocessing methods are used both for cleaning up the classifier input as well as a means for easier labeling. There is no real evidence that one works better than the other. However, background traffic can be part of bursts. Using flow separation it is easier to filter out background traffic certain and this is also easier to classify using the logger mentioned in section 3.2.

3.4. Feature selection

Classifiers rely on features of the data to discriminate between classes and assign the correct labels. The choice for which features to use is thus an important factor in the performance of a classifier. An overview of which features were used by previous research can be found in Table 3.1. This overview shows which features can be extracted from network traffic data. The number of features used varies a lot, but some commonly used features can be identified. All research uses various time series, either Packet Size, Inter-Packet Size, or Inter-Arrival Time. These series are usually extracted for all incoming and all outgoing packets separately. To obtain more information, researchers extract various statistical features as well. There is very little motivation for why a certain feature is considered, the most common argument is that the feature is expected to have discriminatory power.

In order to select the features on which the classifiers perform best, researchers use a variety of methods. Taylor et al. [58] use the internal feature selector of a Random Forest classifier to select, from their 54 features, the features with an importance higher than 1%, a total of 40 features. They do this to avoid the curse of dimensionality. However, the RF already ignores some features to avoid the curse of dimensionality and it is very questionable whether this small reduction in the number of features would really make a difference [20, 32]. Stöber et al. [57] calculates the relative mutual information (rMI) of features, however, they do not do anything with this information and simply select all features, because according to them even features with small scores can still help the classifier. Wang et al. [62] and Al-Naami et al. [1] argue that because they use Random Forest (RF) classifiers, which do some feature selection inherently, there is no need for further feature selection, which is computationally expensive.

Research	Selection method	Features
Al-Naami et al. [1]	RF (implicit)	Bi-Burst size, time; Uni-Burst size, duration, count; Up/Dn PS; (*12*)
Conti et al. [14]	RF (implicit)	In/Out/Combined PS series; (*3*)
Coull and Dyer [15]	n/a	In/Out PS; *(2)*
Liu et al. [36]	DDFS (algorithm)	srcPort, dstPort, dstIP, protocol; 1st/2nd/3rd HPS frequency; IOPR; IOBR; PSI-x; min/max/mean/SD/SE of I/O flow IAT & PS; min/max/mean/SD of I/O IPS; min/max/mean/SD SFPS/SFIAT; I/O flow size, packet count, duration, skewness, kurtosis; (*60*)
Taylor et al. [58]	RF (explicit)	In/Out/Combined PS series: min, max, mean, MAD, SD, variance, skewness, kurtosis, percentiles (10-90), count; (*54*)
Wang et al. [62]	RF (implicit)	In/Out mean IAT & PS: all, low 20%, mid 60%, high 20%; SD IAT & PS; (*20*)

Table 3.1: Overview of the features used by different research and the selection methods used for selecting the best features. For Liu et al. only the features discarded by the best performing algorithm are listed. To save space many features are abbreviated: HPS = highest packet frequency, IAT = Inter-Arrival Time (the difference in arrival time for two subsequent packets), IOPR = incoming outgoing packet ratio, IOBR = incoming outgoing byte ratio, IPS = Inter-Packet Size (the difference in packet size for two subsequent packets), MAD = median absolute deviation, PS = Packet Size, PSI-x = distribution of packet size in 9 different ranges of Inter-Arrival Time, SFPS = "the sequence $\{mps_1, mps_2, .., mps_k\}$, where mps_i denotes the mean of packet sizes in the ith subflow", SFIAT = same as SFPS but for Inter-Arrival Time.

There is some unclarity about how well the internal feature selection algorithm of an RF works and whether feature selection beforehand is useful. It is therefore interesting to see how feature selection before input to an RF affects the performance. In trying to obtain a set of features that is robust to changes in devices or network environments Liu et al. [36] evaluate the performance of models learned on different sets of features in three different cases: 1) same mobile device set, same network, 2) different mobile device set, same network environment, 3) same mobile device set, different network environment. They compare a Basic-Feature set of 20 common features with four other feature sets from previous research and their own Hybrid-Feature set

which consists of In-flow, packet header, packet size and inter-arrival time related, and overall statistical flow features. It is shown that their Hybrid-Feature set performs best in the robustness evaluation outperforming the others by 5-25%. Using the right features thus helps with classification.

To further improve the selection of features, Liu et al. [36] developed a feature selection algorithm *DDFS* (Discrimination power and degree of Drift evaluation based Feature Extraction). The algorithm selects features based on their drift and discrimination power in data, meaning how much they change within the same class or how well classes can be distinguished from each other based on that feature. Features with outstanding discrimination power are always selected, features with a very high drift are always discarded. For all other features, a metric is used to evaluate the balance between the discrimination power and the drift is used to determine whether to use the feature or not. The features with the highest score are selected until the desired amount of total features is selected.

The performance of DDFS was compared to five other feature selection algorithms and a baseline of no feature selection algorithm. Two separate drift metrics are used with the DDFS algorithm: their own developed DDRU (Degree of Drift on Relative Uncertainty) on supervised data and HD (Hellinger Distance) [27] on unsupervised data. DDFS with HD achieves a slightly higher accuracy (2 percentage points) when validating on a previously unseen set of devices or in an unknown network environment. More than half of the algorithms improved accuracy compared to the full feature set. DDFS(HD) achieved the third-highest accuracy in case 1 (83.3%) and the highest in cases 2 (76.2%) and 3 (67.2%). Compared to using the full set, the improvement in accuracy is at most 2 percentage points for the accuracy metric using the DDFS algorithm. For many other algorithms, the full feature set as input to the RF works better. This means that the internal feature selection algorithm of the RF can be helped by selecting features during the preprocessing stage, but people should take care which algorithm to use. While the performance of DDFS is promising, it does not always perform the best. The other algorithms are not ranked in the same order for the various cases either. There is still a lot of uncertainty about the effect different data has on the performance of the algorithms and the other effects of the DDFS algorithms are not yet well studied. Therefore, and because the performance increase is not that large, we will only use the feature selection algorithm of an RF.

3.5. Classifier selection

When data is collected and features are extracted it is time to look for which classifiers work well on the data. A variety of classifiers are chosen in app fingerprinting by researchers: k-nearest neighbor (*k*-NN) [1], Support Vector Machine (SVM) [1, 58], and Random Forest (RF) [1, 14, 36, 58, 62].

None of the research uses specific selection methods similar to feature selection to select a classifier. The choice for a classifier seems to be mainly based on reported past performance in a similar dataset. Taylor et al. [58] chose SVM and RF since they are good at predicting classes when trained with features like the ones extracted from network flows. Wang et al. [62] notes that RF is a good choice for app classification because it can handle substantial noise in datasets, mitigates variance and bias somewhat, provides feature selection, provides a classification performance estimate, and is relatively fast. To select the optimal configuration of parameters of a classifier, *k*-fold cross-validation can be used on an extensive set of combinations of parameters [57, 58].

Table 3.2 gives an overview of the reported accuracy, average recall, and average precision of different classifiers used. Since in a multi-class problem there is no sense of recall and precision for a whole dataset, the recall and precision are calculated for each class individually and averaged. The Random Forest is used most often and reports the best results. Unfortunately not all research reports all the metrics even though it is important to properly evaluate the performance of the classifier. For example, the SVM used by Taylor et al. [58] reports the same accuracy and precision as the RF, but a much lower recall. The usage of the SVM would thus result in much more false negatives, which is an important factor to consider.

3.6. Classifier performance

To benchmark the performance of a classifier, researchers have come up with several validation methods. The most important distinction of validation methods is that of closed world and open-world validation, here meaning learning and validating on the same dataset for example by means of *k*-fold cross-validation and validating on previously unseen data or data generated in a different way respectively. The former type of validation gives a good idea of how well the classifier is able to use the selected features to differentiate between classes when the features don't drift too much within a class [36]. The latter gives a better overview of how well the classifier can be expected to perform in practice. It shows how well the classifier is able to classify even

Research	Classifiers	Accuracy	Macro-average Recall	Macro-average Precision	Setting
Al-Naami et al. [1]	*k*-NN, RF, SVM	n/a, n/a, 87.9	78.3, 91.2, n/a	n/a	open, open, closed
Conti et al. [14]	RF	n/a	95	95	closed
Liu et al. [36]	RF, RF, RF	83.3, 76.2, 67.2	n/a	n/a	closed, open, open
Taylor et al. [58]	RF, SVM	99.8, 99.7	82.5, 64.8	96.0, 96.1	closed
Wang et al. [62]	RF	93.96	>87.23	n/a	closed

Table 3.2: Overview of which classifiers are used in related work and the performance of those classifiers as measured by accuracy, recall, and precision. Since the validation methods are often quite different across research, the performance of classifiers should only be compared when they originate from the same research. In research where classifiers are tested on multiple datasets the average performance was taken. An open setting means that the classifier was trained and tested on completely different data, for example collected from different devices, a closed setting means that one dataset was split into testing and training data.

though the data might drift within a single class, for example, because different devices are used. As confirmed by the numbers reported in Table 3.2, this type of validation is expected to give lower accuracy, precision, and recall, which means that there are more false positives and false negatives.

The conclusions that can be drawn out of validation of the classifier depend a lot on the type of validation used. It is not possible to conclude from a closed world validation that it is doable to detect which apps people are using in a dump of network traffic, because it does not show how a classifier deals with unknown classes that might be very much like others. Another big problem is that closed world testing doesn't highlight any bias in or anything missing from the training data. For example Al-Naami et al. [1] trains a classifier on the network traffic apps generate on startup. In practice, one might expect users to have already started apps when joining a network, the fingerprint the classifier is trained on is missing, which will probably result in the classifier not detecting the app. Without validating with real-world data there is no way to tell what the real-world performance might be.

3.6.1. Effect of universe size

The performance of classifiers is impacted by the number of apps the classifier is tested on, which is equal to the number of classes in the classifiers. More classes mean a higher chance that classes are very much alike, which makes it harder for the classifiers to assign the correct class. Taylor et al. [58] tested the impact of the number of apps on the accuracy of the classifiers used, by training and testing the classifiers on sets of apps increasing with intervals of 10 from 10 to 110. They found that the accuracy decreases from 95% to 82% for a classifier that they believe would be feasible to use for classifying huge amounts of apps as the number of classes trained increases. The rate of decrease starts to lower for a higher number of classes, suggesting that eventually, the performance of the classifiers reaches a constant state. Another approach used by Taylor et al. is to train binary classifiers on each app for which are all executed in parallel, this classifier would be used for trying to fingerprint a small amount of apps. In this case, an increase in the number of apps does not affect the individual trained models. The scoring mechanism that takes into account the outcomes of all the classifiers and assigns a label is affected, but unfortunately, this was not tested.

3.6.2. Performance metrics

Because so much research uses different validation methods and datasets it is quite hard to properly compare the performance of classifiers developed. Especially since some research omits important information like recall and precision as seen in the overview of research in Table 3.2. Many papers report the accuracy of their models in the abstract and conclusion [14, 15, 57, 58], as a means to quickly provide the reader with an indication of how good their results are. However, focusing only on accuracy without context and without mention of recall, precision, and other performance indicators can be (unintentionally) misleading. Furthermore, a validation method can be framed in such a way that it seems to give better results. The following examples of research illustrate how the accuracy, recall, and precision numbers are influenced by certain design choices in either the classifier or validation methods.

Taylor et al. [58] use three different configurations for both the SVM and RF: Per Flow, Single Large, and Per App. The Per Flow classifier builds separate classifiers for each possible flow length, the Single Large classifier contains all apps as classes in one single classifier, and the Per App classifier trains a binary classifier for each app. The performance of the Per App classifier seems great since it reports an accuracy of 99.7%, however, this would be wrong since the dataset for these classifiers is very imbalanced: 99% of the trained data belongs to the negative class. Therefore, the classifier assigns the negative class to almost all flows, this

results in a very small amount of False Positive cases, but also in a rather large amount of False Negative cases, which becomes clear from the recall value, the classifier thus misses flows which do belong to the app. The results are still acceptable for closed world validation, but not as great as they might seem at first glance. Later they discuss their classification validation technique to disregard flows from unknown apps and they again reporting precision, recall, and accuracy values of over 99%. Here they only seem to consider the flows that were actually classified, completely ignoring whether the classification validator disregards flows that are not originating from an unknown app. How well this classifier is performing, thus totally depends on the requirements of the classifier. A classifier that is required to label all flows and assign *unknown* to flows it thinks belong to apps not seen before, cannot be said to perform well based on this test since flows that do belong to known apps get assigned the label *unknown* very often.

Conti et al. [14] classifies user actions in all the flows of one app. They report excellent recall and precision values (95%), which is to be expected for their setting. The amount of classes each classifier has is low (5-10 user actions per app) and a difference in network fingerprint for these often very distinct user actions is to be expected. However, this is only true under the assumption that flows are perfectly classified to a certain app. This is done by domain name inspection, but as discussed earlier this is not a reliable method. When classifying flows from a lot of different apps, some user actions might turn out to look very similar across apps. In the real world, the performance of the classifier will thus be much worse.

The Service-Feature set [22] which performed second-best (91% accuracy) in the cross-validation tests of Liu et al. [36], performs much worse in the robustness evaluation (54.6% average accuracy). This shows the importance of selecting a good feature set and highlights the huge differences that can occur in reported performance by simply using a different evaluation method.

The essence of these examples is that in order to compare the performance of a classifier to those of previous research the circumstances of the validation method must be the same. This requires a thorough analysis of the validation method since there can be a lot of variation between similar validation methods. Furthermore, the lack of reporting complete validation results including accuracy, recall, and precision hinders the comparison of research even when the same validation methods are used.

3.7. Accuracy of models in different contexts

Over time new app versions, devices, and OS versions are introduced. This has the potential to leave trained classifiers outdated as the network traffic generated by apps changes. Some research has been dedicated to investigating the accuracy of classifiers over time, also called the robustness of the classifier. As mentioned before Liu et al. [36] use feature selection as a means to combat changes in the context of mobile applications. They calculate which features can best distinguish mobile apps when classifying in data consisting of multiple mobile devices and different network environments. However, compared to using all possible features, selecting the most robust features has only a minor improvement of about 2% accuracy.

Al-Naami et al. [1] take a different approach and propose a model that can be easily updated at a later time. This is useful for when an app is updated and functionality is changed, which also changes the network fingerprint. They propose to compare the trained data with current data and analyze any difference in the data patterns, also known as concept drift. When concept drift is detected a model can be updated so that there is no more drift. Another approach is to update the models at a fixed interval, for example, every two weeks.

3.8. Countermeasures

If network traffic analysis really works to classify apps on real-world traffic, it is important to investigate countermeasures against this attack. The most intuitive solution seems to be to use a padding mechanism so all packets look the same. And indeed, this works well for classifiers that only take into account packet sizes. Coull and Dyer [15], which only use packet sizes as features, report that by adding random padding to packets, the performance of their classifier for user actions in iMessage is reduced to 0%, at the cost of overhead of more than 300%. However, often a lot more features are used, as could be seen in the overview of features used in Table 3.1. So even when removing all features related to packet size there are still quite a few features left on which classifiers can be trained.

Al-Naami et al. [1] apply traffic morphing by means of padding specifically to make the traffic generated by apps look like each other. They report a decrease of 26% accuracy for their closed world case on average, with 50% accuracy for a dataset containing 100 apps and a decrease of 33% in recall for the k-NN and 39% in recall for RF in their open-world case with about 20% in recall in the worst case (2000 apps). This is still well above random guessing, but an advantage of this approach is that the overhead can be limited when morphing to

apps with a similar traffic pattern, so it might be worth implementing for this limited privacy gain. It should however also be noted that implementation in practice will be hard as this countermeasure will require the same frequency of updates as the classifiers do.

Dyer et al. [19] investigated 10 different countermeasures against traffic analysis in encrypted website traffic. These countermeasures could also be used for encrypted network traffic from mobile phones. An overview of the effects of their countermeasures on the accuracy of classifiers based on various sets of features can be found in Table 3.3. In this table, the difference between the performance without countermeasures can be compared to the performance with countermeasures to see how effective a countermeasure is. Because the test was done for classifiers learned on different features, the table also shows which features are affected by the countermeasures. For example, it can be seen that the TIME classifier is not affected by any of the countermeasures, since they only affect the packet size. The tests are done for a universe of 128 web pages, so random guessing would give an accuracy of 0.8%. The table shows that none of the countermeasures work well enough to reduce the performance to random guessing for all classifiers.

Dyer et al. test seven padding based countermeasures and two traffic morphing countermeasures targeting the distribution of packet size and inter-arrival times. Note that this version of traffic morphing is more advanced than the one used by Al-Naami et al. [1] since it also takes time-related features into account. The results show that none of the padding based countermeasures help significantly against classifiers that take into account more than just the size of the packets. Exponential and Linear padding schemes even cause a higher accuracy, probably because these enhance small differences increasing the discriminatory power of the features. Traffic Morphing and Direct Target Sampling work much better, but the accuracy of the most robust VNG++ classifier, which combines multiple features, is still high at 80%.

Since the classifiers proved to be so robust against known countermeasures, Dyer et al. came up with their own countermeasure called Buffered Fixed-Length Obfuscator (BuFLO). The BuFLO countermeasure changes the traffic so fixed-sized packets are sent at a fixed interval for a minimum fixed amount of time. Several parameters can be set to tinker with the sizes and timings. The most robust setting (1500 byte packets, every 20 ms, for 10 seconds) tested is listed in Table 3.3. As is to be expected, the overhead on this countermeasure is significant. However, what is surprising is that the accuracy is still 4% for two of the classifiers, more than four times as much as random guessing. This means that there is still a rather large portion of websites that require a larger bandwidth per flow than is provided by these settings.

Dyer et al. conclude from these results that none of the first nine countermeasures work against fingerprinting attacks. Although they note that the circumstances favor the attacker since the training and testing data is generated in the same way and the attacker knows which classes to look for. They propose that further research into applying counter-measures in a real-world setting might prove to be more effective in mitigating the attacks.

Countermeasure	**Bandwidth overhead (%)**	**Classifier accuracy (%)**					
		LL	H	BW	TIME	VNG	VNG++
None	0	98.1	98.9	80.1	9.7	93.7	93.9
Session Random 255	7.1	40.7	13.1	54.9	9.5	87.8	91.6
Packet Random 255	7.1	80.6	40.1	77.4	9.4	91.6	93.5
Linear	3.4	96.6	89.4	79.5	9.6	93.5	94.3
Exponential	10.3	95.4	72.0	77.1	9.6	95.1	94.8
Packet Random MTU	28.8	45.8	11.2	64.6	9.5	77.8	87.6
Mice-Elephants	39.3	84.8	20.9	72.3	9.6	89.4	91.7
Pad to MTU	58.1	63.1	4.7	62.7	9.6	82.6	88.2
Traffic Morphing	49.8	31.0	6.3	43.0	9.8	81.0	86.0
Direct Target Sampling	66.5	25.1	2.7	41.2	9.7	69.4	80.2
BuFLO (best config)	418.8	4.4	0.8	n/a	n/a	n/a	4.1

Table 3.3: Overview of the effect of ten countermeasures on the accuracy classifiers based on various features as investigated by Dyer et al. [19] composed of tables from their paper. These countermeasures were applied on a dataset by Herrmann et al. [28]. Another dataset that was evaluated, which reported slightly better results for the countermeasures is omitted here. The reported results are of network traces in a universe of 128 web pages. The main features on which these naive Bayes classifiers learn are: direction and size of packets (LL), normalized direction and size of packets (H), total per-direction bandwidth (BW), total transmission time (TIME), traffic burstiness (VNG), and a combination of the last three features (VNG++). The countermeasures consist of 7 padding countermeasures and 2 that target the packet size and timing distributions. BuFLO is a countermeasure developed by Dyer et al. that tries to make all traffic uniform on size and timings.

3.9. Conclusion

The following can be concluded from the analysis of current research on the topic of fingerprinting in encrypted mobile phone traffic. It is difficult to compare the performance of classifiers but to ease the process a good description of how validation was done and complete results with explanations are required.

Most research focuses on a very small subset of apps or user actions and has well-performing classifiers. Liu et al. [36] showed that a changing context like a different mobile device or network environment can have a significant impact on the performance of classifiers. An open research question is how well this decrease in performance can be countered by creating more robust models, for example, trained on data originating from a wider range of mobile devices. Quantifying the effect of these changes in context can give insight into the feasibility of applying the classification of user actions and apps in encrypted traffic in the real world.

The results reported by research can differ quite a lot, so the performance of a classifier is dependent on various factors. Taylor et al. [58] showed that the performance is dependent on how much apps are under consideration. Various other research note that different network environments [36], different devices [36, 59], new app versions [1] or the passage of time [59] can influence classifier performance. Exactly what differences in data cause these performance changes is not yet clear. Insight into this will make it easier to explain why classifiers in certain settings do or do not work well. This insight can help with finding countermeasures against network traffic analysis.

Dyer et al. [19] showed that effective countermeasures have a very large overhead. A large overhead is problematic for adoption as bandwidth costs money. Research into effective countermeasures for the mobile platform and real-world data is lacking and might prove to have a more positive outcome than the results of Dyer et al. [19], because of the more dynamic environment.

The current best scalable ways for data collection are the use of emulators executing user actions on apps automatically. Another, less scalable way is crowdsourcing traffic data by letting users install network monitoring software on their mobile phones. Labeling a traffic dump by hand is not feasible since there is no reliable way to map flows to apps, because of the use of CDN and shared libraries. Using OS logs of which ports are opened by apps, labeling can be done in an automated and reliable way.

Open research questions are:

- How well does a classifier, that is trained on (most) of the available mobile devices and recent app versions, perform on real-world data?
- What differences in datasets influence the performance of a classifier and why?
- What are effective countermeasures against app/user action fingerprinting with an acceptable overhead?

4

Methodology

The answers to the research questions posed in chapter 1 are found through experiments with data in various settings. This chapter describes how the methods by which the results of the experiments are evaluated. The first sections describe the technical setup of the data collection and the setup of the experiments. The system can be split into two parts: data collection, and training and classification. The data collection consists of data collection through emulators and real-world data collection. The results of the experiments are evaluated using various metrics which can be calculated from the results of the classification of the data, these are discussed in section 4.6. To better understand the relation between the data and the results, the data itself is analyzed as well. The methods to compare and evaluate the data are explained in section 4.7.

4.1. Data collection

The first step in network traffic analysis is collecting network traffic. Network traffic is collected in an automated fashion using emulators or by capturing live network traffic using crowdsourcing, the latter is discussed in detail in section 4.5. After the network traffic is collected it needs to be processed so it can be used as input for the classifiers. This section explains how traffic is collected, cleaned, grouped, labeled, and how features are extracted.

4.1.1. App selection and collection

A selection of apps is required as a basis for the experiments. We randomly select 1 000 free apps from the top 10 000 apps in the Google Play Store. The top 10 000 is used, because these are the apps that can be expected to occur in the most in the wild. If network traffic analysis will not work for these apps, it is definitely not feasible to use it in the wild. The distribution of apps selected can be found in Table 4.1. By randomly selecting we attempt to get a fair representation of the apps available in the Play Store. Popular apps, which are ranked higher, are selected with a greater probability, to obtain a larger amount of data from apps that are more relevant in the real world. The apps are ranked by the number of reviews they have. This provides a better overview of popular apps than the number of installs since apps can also come pre-installed on devices, which adds to the total amount of installs. Paid apps are deemed out of scope to keep costs low. There is no evidence that using paid apps would give other results.

Rank	Amount selected
1-1 000	325
1 001-2 000	175
2 001-3 000	100
3 001-4 000	100
4 001-5 000	75
5 001-6 000	75
6 001-7 000	50
7 001-8 000	50
8 001-9 000	25
9 001-10 000	25

Table 4.1: Distribution of apps that were random selected from the Google Play Store top 10 000. The apps are ranked on the amount of reviews they have. To obtain more of the popular apps a distribution was chosen that favors higher ranked apps.

An unofficial Google Play Store API[1] is used to download the apps from the Play Store. To download apps, we must choose a phone for which the apps can be downloaded since APKs can differ per device as developers target different Android versions or require certain sensors. Furthermore, older devices might not receive the updates newer devices receive. Google Pixel is a fairly standard phone, and supports the newest versions of Android and was thus deemed a reliable choice to be able to download most of the apps available in the Google Play Store.

[1] `https://github.com/matlink/gplaycli`

Every two weeks the latest APKs of the 1 000 selected apps were downloaded, creating a version history of these APKs that can be used in the experiment. Unfortunately, about half of the APKs are only built for ARM architectures, as that is the CPU architecture all phones use. Android emulators, however, are built for x86 architectures, so about half of the downloaded APKs cannot be used. We kept all the ARM apps so they can be used in later research when we decide to collect traces on actual phones instead of emulators. As can be seen in Table 4.2, the ratio of x86 apps stays fairly consistent over time, although the ratio is often lower than the starting ratio. This means that ARM apps are updated slightly more often than x86 apps. Assuming there is no difference between x86 and ARM apps in how much a fingerprint changes for each update, this structural bias will cause our results to show slightly less performance degradation over time than when considering all apps.

Date	APKs downloaded	Build for x86	Ratio x86
Jan 21	1 000	563	0.563
Feb 4	300	146	0.487
Feb 15	199	111	0.558
Mar 4	281	150	0.534
Mar 18	257	132	0.514
Apr 1	239	133	0.556
Apr 18	292	158	0.541
Apr 29	196	95	0.485
May 21	303	165	0.545
Jun 3	219	111	0.507
Jun 17	230	127	0.552
Jul 11	325	182	0.560

Table 4.2: List of how many new versions of apps are downloaded every two weeks and how many of those apps are built for the x86 architecture and thus useful for our experiment.

4.1.2. Data collection pipeline

Training a Random Forest classifier requires a lot of data. To collect this data in a scalable way, Android emulators are used to generate network traffic from apps. The emulators are run on a machine and are controlled using a shell script. Figure 4.1 gives a conceptual overview of how the data is collected from an emulator.

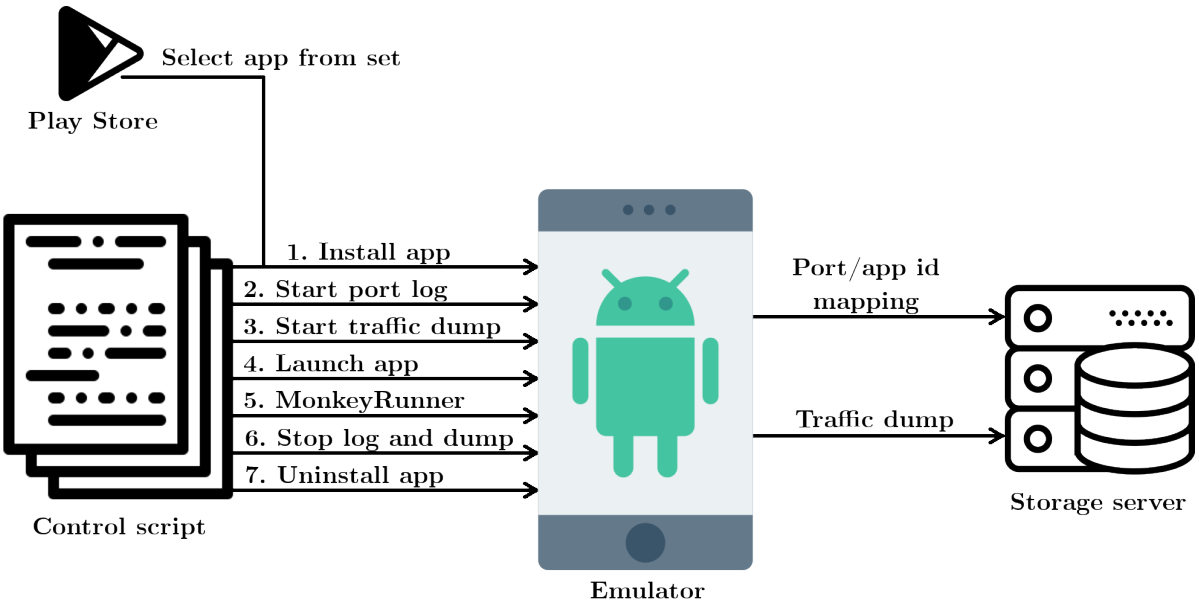


Figure 4.1: Overview of how data is collected on an Android emulator. An emulator runs on a machine can be controlled using Android Debug Bridge (ADB). Using a shell script apps are installed on the emulator, run, and uninstalled again. While the app is running, MonkeyRunner provides random user input to trigger network traffic. The network traffic and the port/app id mapping, used for labeling the data, are logged and captured and saved on the data collection server.

Several Android emulators are created and run on a machine. The apps that were previously downloaded,

as explained in subsection 4.1.1, are used for the data collection. Each app is installed on an emulator and started. To generate network traffic for an app user input is required, e.g. tapping a button to load an article or sending a message. To simulate user input *Android monkeyrunner*[2] is used. Android MonkeyRunner is a UI exercising tool that sends random user input to the emulator. The generated network traffic from starting the app and providing random user input is captured using *tcpdump* on the host machine. Afterward, the app is removed from the emulator so it will not influence the data collection for the next app. The network traffic captured by executing these steps is what we call a trace.

Not all network traffic that is collected can be attributed to the app that was in the foreground at the time of the traffic. Other stock apps or the Android operating system also generate network traffic. To label this network traffic, we log which apps use certain network ports on the emulator. Currently bound ports are listed under */proc/net/tcp*, */proc/net/tcp6*, */proc/net/udp*, and */proc/net/udp6* on linux kernels. Table 4.3 shows an example of the log that can be found in */proc/net/udp*. The log shows which local addresses currently have an open socket to which remote addresses and the UID of the user that ran the program that opened this socket. It also describes information like connection state, memory location of the socket, timeouts and more. These interfaces list which UID is the owner of this open connection. On Ubuntu each user account has a UID of 1 000 or higher, and root has UID 0. But all kinds of programs create users as well for security reasons, so for example the user *www-data* has UID 33, and user *syslog* has UID 102. Programs that use the network and are running as one of these users will thus have an entry in the interface with the UID of that user. Android works in a very similar way, a special feature, however, is that it creates a new user for each installed app. This can thus be leveraged to know exactly which open connection originates from which app and, by checking for UID 0, which traffic is background traffic from the OS. By uploading and running a shell script on the emulator that logs the changes to the interfaces, a mapping from local ports to UIDs is obtained. Subsequent lookups for the UIDs result in the corresponding app names for each connection. The log is downloaded from the emulator after the full trace has been collected and is stored for later processing. A limitation of this method is that ephemeral (short-lived) ports [56] might be missed since the script operates in a loop. We test this limitation by running the logger on an emulator, capturing its network traffic, and comparing the flows observed in the captured traffic with the flows logged by the mapper. Based on these tests we estimate that the logger misses around 12% of the connections for our setup.

…	**local_address**	**rem_address**	…	**uid**	…
…	02E8A8C0:0044	01E8A8C0:0043	…	1000	…
…	00000000:14E9	00000000:0000	…	1020	…
…	00000000:14E9	00000000:0000	…	1020	…
…	00000000:C5B6	00000000:0000	…	1020	…

Table 4.3: Example output of /proc/net/udp, showing details of current UDP connections. The log of this interface describes information about local and remote adresses, connection state, memory address of the socket, timeout and more. Most of the columns are omitted for readability. For our purposes we are only interested in the local port, remote address, and the UID.

4.1.3. Data processing

To use the data as input for the Machine Learning models, we need to process the captured network traffic. We clean the data, group packets within the data, label it and extract features. These features can be used as input for the classifiers.

Cleaning data

The data capture contains noise and data streams that we are not interested in, for example, ARP packets, HTTP data, TCP acknowledgments, or TCP retransmissions. All non-TLS data is removed from the datasets because we are analyzing encrypted flows. We choose to focus on TLS here because it is the most used protocol for encryption in the context of the mobile platform. Retransmissions are non-standard behavior and are a result of a bad link, therefore we remove those, so the classifiers can assume an error-free link and are thus less influenced by unreliable networks. All other TCP packets without payload, such as acknowledgments are also removed, as they do not contain actual data being sent and their behavior is platform-dependent, not app-dependent. Cleaning the data is done with the command line tool *tshark*[3], which outputs for each packet the source IP and port, destination IP and port, time, and length.

Many apps use libraries for placing advertisements in apps, for example, Google AdMob[4], or use trackers. These ads and trackers cause very similar network behavior across apps, so it would be better to filter out

[2] `https://developer.android.com/studio/test/monkeyrunner`
[3] `https://www.wireshark.org/docs/man-pages/tshark.html`
[4] `https://developers.google.com/admob/`

ad-related traffic. That way the classifier can make better distinctions between apps. AdBlockers usually block advertisements based on a list with known ad domain names. When an application does a DNS request for a known ad domain, the ad blocker will give a bogus response so the ad won't be loaded. We do not have that option since we cannot influence the behavior of the devices we capture data from. However, these lists can also be used to remove ads after the network traffic is captured. Using *tshark* we collect all DNS requests in the capture file and create a list of all requested hostnames. Then we match those hostnames to a list of known advertisement and tracking domains. For all hostnames, the DNS responses are checked, and thus a list of IP addresses of known advertisement and tracking domains is obtained. In case that one IP address serves both advertisements and other content, the burst with the matched IP is not removed, so no useful data is lost. Filtering out ads increases the processing time by about 50%.

Burstification

If single packets are used to train a classifier, it becomes very hard for the classifier to distinguish between classes, because packets have limited features in which they can differ from each other. So no clear fingerprint can be established for an app. Therefore a higher-order unit is required, which will highlight differences between apps. We consider a sequence of packets, so that the distribution of, for example, packet size will give a more distinct fingerprint.

A good choice for a sequence of packets is a flow, defined as all packets from and to the same IP address and port. This will result in an aggregation of packets coming from the same app, as the source-destination combination will almost always be unique to a certain app. There is a chance however that a local port is first claimed by one app and later by another and that they both connect to the same remote host. Another choice for aggregating packets is burstification. A burst is defined as all packets occurring within a certain time threshold of each other. Depending on the threshold it can be seen as the corresponding network fingerprint to user actions on the emulator. For example, loading an article will cause a lot of network activity for a short while and then nothing anymore. Alternatively streaming a video will cause a burst of network activity at regular intervals when the app is buffering. However, a burst can still contain noise from other apps.

When both methods are combined the issue of noise is solved. For each packet, we check if there is a sequence of packets from the same flow of which the last packet was received within the burst threshold and either add it to that sequence or start a new sequence. In essence, this means that we separate the traffic into flows first and then do burstification on the flows. By doing burstification on the flows, we can say with a very high degree of confidence that a sequence of packets belong to only one app since the chance that another app claims the same source port to go to the same destination is very small. This way a network fingerprint of an app is a set of packet sequences generated from functions within the app. From now on when we talk about bursts in the context of the research of this thesis we mean the bursts that are the result of burstification of the flows.

This method similar the method used in previous work by Taylor et al. [58], with the distinction that we do burstification at a flow level, while they do burstification over all the network traffic and do flow separation in the bursts. This order and their definition of a flow *"a sequence of packets (within a burst) with the same destination IP address and port number"*, both leave open the possibility that a flow contains traffic from multiple apps. This order also has the risk of losing information about the number of bursts a specific app creates. When a background app generates a lot of network traffic (e.g. by downloading), only one burst is extracted and the bursts that were caused by user actions on the foreground app are not found.

Labeling

The bursts that are extracted from the traffic originated from a single source. In combination with the port logs, it is possible to do 100% accurate labeling of the bursts. First, all the port logs are processed so that the IP addresses and ports are in decimal format. The IPv4 addresses that are mapped as IPv6 addresses in the */proc/net/tcp6* and */proc/net/udp6* interfaces turned into IPv4 address again for proper matching. For each burst, the start time and end time are matched to the timestamps of the port logs. If a match can be found the burst is labeled as originating from that app.

As previously mentioned, labeling is not possible for all bursts, because of dynamically allocated ports are sometimes missed. Another problem for labeling the bursts are mismatches between the timestamps of the bursts and the mapping. This can be partly countered by matching the timestamps with a wider margin, but it is a trade-off because then we run into issues of multiple matches. Table 4.4 gives an overview of the number of labels missing from several datasets. From these results, we can conclude that on average around 12% of bursts cannot be labeled. This means that our method of collecting labeled data is fairly efficient. Some

Dataset	Apps	Traces	Bursts	Labels missing (in %)
21 Jan	546	3 783	195 813	11.44
4 Feb	141	700	59 097	13.03
15 Feb	109	1 003	24 757	9.89
15 Feb (set 2)	109	1 050	37 102	14.75
4 Mar	143	1 430	64 069	12.76
18 Mar	122	1 219	56 548	7.27
1 Apr	125	1 244	56 551	11.15
29 Apr	93	929	32 793	16.55

Table 4.4: Overview of the fraction of bursts for which no label could be found for several datasets. Because the network capture file only contains network traffic from the emulator a perfect port logger will be able to label all bursts. These tests on many bursts show that for our setup the logger on average misses 11.8% of the opened sockets.

optimization in the port logging script could improve this some more, but without some OS-level logging, we will always miss some ephemeral ports.

4.1.4. Feature extraction

Classifiers learn how to assign a class to a burst based on features. A classifier tries to discriminate between classes based on dissimilarities in the distributions of various features. More features to discriminate on thus often improves the ability of the classifier to discriminate between classes. Looking at the data and at features used in previous research we came up with a number of features to extract. Figure 4.2 gives a complete overview of which features are extracted from different time series. We use three categories of features, informative, burst, and statistical features. Table 4.5 shows which features comprise these categories. A complete overview of all features used can be found in Appendix D. For each burst, we extract the five *informative features*. Bursts are processed in such a way that we define the source IP to always be the mobile device. This way we have the same definition for incoming and outgoing traffic for every burst. We use this to divide the burst into three time series, a complete series with all the packets, a series with only the incoming packets, and a series with only the outgoing traffic. Using these three time series instead of only one with all the packets enriches the information gained from these bursts. E.g. using YouTube to watch a video will have a lot of incoming packets while uploading photos to Instagram will have a lot of outgoing packets.

For each of the three time series, we extract three *burst features*. We further process the time series to extract three more series: a series with only packet sizes, a series with the size difference between packets (Inter Packet Size), and a series of the difference in arrival time of packets (Inter Arrival Time). The arrival time itself cannot be used as a feature as this is independent of the app, but it is used to calculate the time delta for arrival times of packets. From these three new series, we finally extract 18 *statistical features*. When we combine all the features we get a total of 176 features. Appendix D contains a complete list of all features used with their description. This can be used as a reference for when some features are mentioned later.

Informative features	Burst features	Statistical features
starting time	burst duration	minimum
source IP address	burst packet count	maximum
source port	burst total size	mean
destination IP address		variance
destination port		skewness
		kurtosis
		Standard Deviation
		Median Absolute Deviation
		Standard Error of Mean
		10^{th} percentile
		20^{th} percentile
		30^{th} percentile
		40^{th} percentile
		50^{th} percentile
		60^{th} percentile
		70^{th} percentile
		80^{th} percentile
		90^{th} percentile

Table 4.5: List of features and statistical features that are extracted for each of the bursts processed data. See Figure 4.2 for how these features are used in the different time series extracted from a burst.

The five informative features are discarded as input for the classifier, they are only collected to give more insight into the bursts for later inspection if needed. We generally do not want to use destination IP addresses because the IP address of a server can change, and therefore using them as a feature for the bursts will make the classifier outdated faster. Since an IP address will be strongly linked to an app, the classifier will rely on that feature

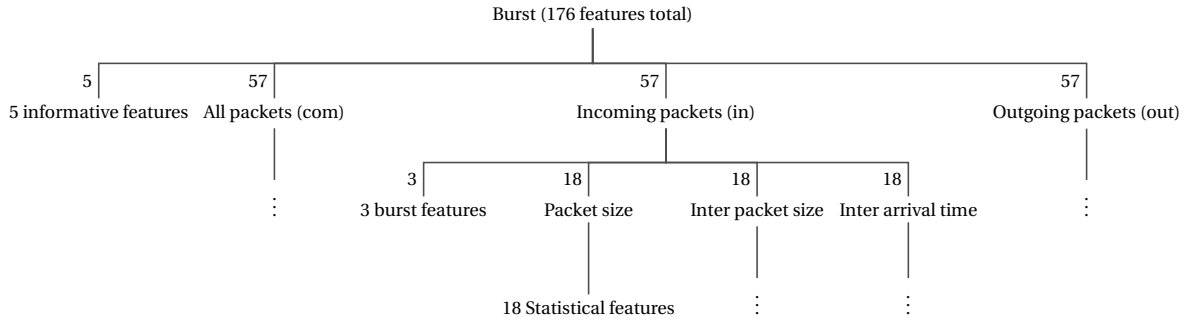


Figure 4.2: Overview of which features are extracted from each of the time series. The informative features are generally not used for training the classifier. See Table 4.5 for a description of the *informative*, *burst*, and *statistical* features. See Appendix D for a complete overview of all features used.

quite heavily, so if an IP addresses changes, a sharp drop in performance is to be expected. Furthermore not learning on IP addresses makes the classifier more robust against countermeasures like proxies. This hypothesis is further investigated in section 5.3. Source IPs should not be considered as a feature, because they are app-independent and device-dependent. Ports are often chosen randomly and therefore not considered as an important feature. The timestamp is not useful because it is independent of the app, instead, the timestamps of packets are used to create the burst duration feature.

Since Random Forests have an internal mechanism for feature ranking and selection, no custom feature selection algorithm is used. Tests with manual feature ranking and selection did not show a significant improvement in comparison with the internal feature selection methods of the Random Forest. As the feature selection of the RF already works well we refrained from implementing more elaborate feature selection algorithms like the ones proposed in previous work [36] as discussed in chapter 3.

4.1.5. Dealing with categorical data

Almost all of the features we use are numeric values. These numeric values can be used directly as input to a Random Forest classifier. They are considered continuous data, meaning that there is a sense of larger or smaller for the data. For example, when considering packet sizes it makes sense to say that small values are similar to each other and can belong to the same class. However, when we want to consider hostnames as a feature we need to deal with the fact that it is not a numeric value, but a string. It also should not be considered as continuous data, but as categorical data, a different hostname should be considered as an entirely different thing altogether. For example, it does not make sense to say that *facebook.com* is similar to *hacebook.com*; the first is owned by Facebook, the second by a Ukrainian. To be able to use categorical data as input for the classifier some pre-processing is required. There are two popular methods for dealing with categorical data that we consider: one-hot-encoding and feature hashing.

One-hot encoding in machine learning is a method where each possible value of categorical data is a single feature in a classifier. The feature takes value 0 or 1 based on whether the category is that feature or not. This is useful for when you have a predetermined amount of possible values a category takes, e.g. which protocol is used. For example, if we can choose from SSLv2, TLS1.2, and TLS1.3, three features are added to the classifier. If a burst uses TLS1.2, that feature will take value 1 and the others will take value 0. This way the Random Forest does not treat the protocol as continuous data and cannot split a tree based on an inequality relation between the protocols, which wouldn't make much sense since we cannot say that one is less or more than the other.

Feature hashing is a method that works similar to one-hot encoding. Instead of a fixed amount of possible categories, feature hashing requires a predetermined of features (or columns) for the categorical data. Each value for the category encountered is then hashed and put into one of the columns. This method has the great benefit that it is not needed to know all possible values of the category, with possible hash collisions as a downside.

There are many possible domain names that we can encounter and there is no way of knowing if we might encounter a new one when we collect new data. Even if we do know all of the possible domain names beforehand, using one-hot encoding would result in thousands of extra features, since each domain name requires an extra feature. Therefore feature hashing is a much more attractive solution. Using feature hashing we will not have to worry about determining all the possible domain names beforehand and we can keep the number of extra features manageable by finding a good balance between extra features and the chance of a

collision. We use feature hashing with 1 000 extra features, this keeps our classifiers manageable, while the chance of a collision where the other features are also very alike is very small.

4.2. Setup of machines and emulators

Most of the data was collected using the stock Google Pixel emulator of *Android SDK Tools*[5]. Other stock Android emulators were used for testing the difference in network traffic between emulators. All emulators run Android API 26 (Android 8.0 Oreo) with Play Store services. The emulators were run on two identical machines of which some details can be found in Table 4.6. Four to six emulators were run in parallel on each machine to speed up the data capture process.

OS	CPU	Number of cores	RAM
Ubuntu 16.04.6 LTS	Intel Xeon L5520 @2.27GHz	16	49.45 GB

Table 4.6: Details of the machines used to run the emulators, the two machines used were identical.

The emulators are controlled from one master script. In this script, we can define the set of apps for which we want to collect traces, the emulators to be used for collection, and the number of traces to be collected per app. From this script, we launch the emulators and launch the scripts that are going to perform the data collection, as explained in subsection 4.1.2. These collection scripts are launched using the virtual window manager *screen*. This way we can easily access the data collection scripts while they are running for inspection and debugging.

Android emulators are not very stable over time. They tend to use up more and more RAM the longer they are running and can sometimes crash. Therefore the script checks whether an emulator is still running every 15 minutes and restarts it if required. Besides emulators crashing another problem are bugs in the Android Debug Bridge (ADB). ADB starts a server that connects to the emulators and is used for sending commands to them. When this server is running for multiple hours, connections start to fail and emulators are listed as offline. The only way to bring them online again is to restart the ADB process and restart the emulator. We periodically restart ADB to limit downtime of the emulators. This causes all the running data collection scripts to fail for one trace, so therefore we only restart ADB every few hours. The scripts are fairly robust and will continue with collecting the next trace when an emulator is booted up again or ADB is restarted. Because ADB and emulators fail on an interval of the order of an hour we do not lose many traces, on average about 1%.

4.3. Training and classification pipeline

The labeled data that we have collected can be used as input to train our classifiers. This section discusses which classifier will be the most suitable for classifying the network traffic data we collect. Furthermore, it explains in short how the trained classifiers are used.

4.3.1. Selecting the best classifier

In the literature, as described in chapter 3, a few classifiers are discussed as being suitable for classifying network traffic data. These are Random Forests (RF), Support Vector Classifiers (SVC), and k-Nearest Neighbor (k-NN). In addition to these, we have also considered using other popular classifiers that were not previously reported, namely Convolutional Neural Networks (CNN) and Naive Bayes (NB) classifiers. Our goal is to select the classifier with the most potential for this type of data, others will probably reach the same conclusion and that makes this research more useful.

The most used and best-performing classifier in the literature is the RF classifier, as can be seen in Table 3.2. To confirm that this is also the case for our data we performed some preliminary tests on various classifiers to see how they perform and determine which classifier is worth looking into further. We used a dataset of 21 apps and performed 5-fold cross-validation on it using each of the classifiers. Various parameter settings were tried, the best results for each of the classifiers are displayed in Table 4.7. We are searching for a classifier that performs well for each of the displayed metrics. Like in related work, the RF comes out on top for our tests as well. The CNN has the second-best performance trailing 15 to 20 percentage points behind the RF. Other classifiers perform much worse, which is a confirmation of previous work.

[5] `https://developer.android.com/studio/command-line#tools-sdk`

An expert in neural networks might be able to get better performance for a CNN than we report since a lot of variation in which neurons to use is possible. However, finding the best performance of a CNN is considered out of scope for this research. As CNNs are not yet used by others to classify network traffic of mobile phones and it does not show great potential out of the box, the chance that it will become the standard for this type of analysis is unlikely.

We will use Random Forests in the rest of this thesis to conduct our tests. The RF is still fairly fast compared to the CNN, which gives the RF an advantage in research as it will be easier to do a lot of small experiments. It shows the best potential for classifying network traffic from mobile phones and is used most often by related work. This allows us to use the results of this thesis to explain the results of related work since we will use the same classifier.

Model	Average Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score	Execution time (ms)
CNN	0.638	0.539	0.490	0.494	1 914 920
Bernoulli NB	0.249	0.166	0.213	0.152	**316**
Gaussian NB	0.231	0.116	0.165	0.087	492
k-NN	0.608	0.448	0.449	0.442	888
RF	**0.791**	**0.726**	**0.699**	**0.698**	6 590
SVC	0.365	0.582	0.227	0.258	34 428

Table 4.7: Preliminary performance results for various classifiers on a small dataset (21 classes) using 5-fold cross validation. For each classifiers various combination of parameters were tried and the best results are displayed here. This information is used to select the classifier with the most potential. Results in bold are the best, underlined result the second best.

4.3.2. Training classifiers

Using the labeled data we can train Random Forests on the characteristics of the network traffic of apps. These classifiers are saved to disk and can be loaded later to classify other network traffic. In this research we only use the classifiers to verify whether it is possible to do this type of analysis, we do not use the classifiers in a live setting. The classifiers are fed labeled data so we can score the performance of the classifiers. Once it is verified that the classifiers can reliably predict app usage based on the network traffic, they can be fed unlabeled data to classify network traffic.

4.4. Per class classifier

The methods described until now train a single Random Forest classifier on all the data. This is perfectly feasible for a small number of features and classes. However, as the number of features or the amount of classes increase, the trees need to become much larger to distinguish between classes. Especially when using feature hashing the number of features increases significantly. For the largest dataset used for training a single classifier, this meant that upwards of 100GB of RAM was required to do classification. For real-world application either a lot of hardware or a different way of training the classifiers is required.

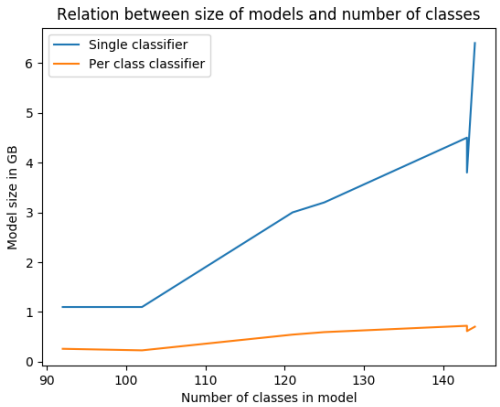


Figure 4.3: Plots showing the relation between the size of a classifier and the number of classes contained in the classifier. It can be seen that the size of a Single classifier is $\mathcal{O}(n^2)$ and the size of a Per Class classifier is $\mathcal{O}2(n)$, with n the number of classes. The largest classifier is omitted for better visualization.

One method often used is the usage of multiple binary classifiers in parallel. For each class, a binary classifier is trained on the data of that class and data from random other classes. The amount of data of other classes used can be limited to obtain a fixed size classifier for each class if desired. This results in a classifier that linearly increases in size as the number of classes increases.

The classification of new data can be done by letting all binary classifiers classify the data and using a scorer to obtain the predicted label with the highest confidence. When using binary classifiers all the trees need to

Classifier	Number of classes	Size Single classifier	Size Per class classifier	Training time Single classifier	Training time Per class classifier
21 Jan	481	58 GB	3.1 GB	1092s	13 625s
21 Jan (extra)	481	10 GB	3.5 GB	992s	12 942s
21 Jan (set 2)	478	30 GB	1.6 GB	547s	5 259s
4 Feb	144	6.4 GB	705 MB	148s	1 013s
15 Feb	102	1.1 GB	230 MB	24s	221s
4 Mar	143	4.5 GB	723 MB	111s	994s
18 Mar	121	3.0 GB	546 MB	70s	746s
1 Apr	125	3.2 GB	596 MB	82s	771s
18 Apr	143	3.7 GB	616 MB	88s	906s
29 Apr	92	1.1 GB	261 MB	26s	260s

Table 4.8: Comparison of the size on disk of a single RF classifier and the aggregated size of the per class RF classifiers. While the per class RF classifier uses much less space on disk, training takes much longer. Note that the training times are based on only one sample; the overall trend is the important takeaway from this table, not the training time for one classifier.

classify the new data, which takes longer than simply using a single classifier. On the flip side, the trees are much smaller, so the classification of a single binary classifier is also faster. Furthermore, utilizing the fact that the binary classifiers are separate objects, the classification can be done in parallel, making a classification for per class classifiers even faster than for a single classifier depending on the amount of hardware used. Previous research shows that the performance of per class classifiers is similar to or better than the performance of a single classifier [58]. This is confirmed by the experiments described in chapter 5.

In Table 4.8 an overview is given for both the single classifier and the per class classifier of the space the classifiers occupy on disk and time training them takes. It can be seen that the per class classifiers require much less disk space than the single classifiers. This is to be expected since the classifier requires exponentially more splits to properly make a distinction between classes as the number of classes increases. It results in a size of $\mathcal{O}(2n)$ for the per class classifier and $\mathcal{O}(n^2)$ for the single classifier, with n being the number of classes. This is visualized in Figure 4.3, which shows how the size of classifiers grows exponentially for a single classifier and linearly for the per class classifier. While the per class classifier uses less space on disk, the training time is much longer. However, because the per class classifier consists of many binary classifiers, the training process can easily be parallelized, so training is scalable. These results show that by using binary classifiers in parallel, the methods described in this chapter can scale to a very large amount of classes.

4.5. Real world data collection

To truly attest the feasibility of user behavior tracking through analysis of encrypted network traffic, real-world tests are required. This requires labeled real-world data, which is not trivial to come by. To obtain this data we utilized a similar technique as in Liu et al. [36], a combination of network logs and a VPN service to collect labeled data remotely.

The main problem is that we require a log of which traffic originates from an app. The research group SECUSO at Technische Universität Darmstadt developed an app that does exactly that[6]. By modifying the source code we obtained a simple app that outputs a CSV file with the mapping from ports to app ids. The most straightforward way of collecting the corresponding network traffic of users is by letting them connect to a dedicated access point on which all traffic is logged. However, this is not feasible for collecting a sufficient amount of realistic data in a short time, as this setup requires people to be near to the access point. Therefore we use a VPN service that relays all traffic from a mobile phone to a remote server where the traffic is logged. This functionality can be found in the Android Open VPN app[7]. Modifying this app allows us to embed our own Open VPN profile in the app, which has the details for setting up a VPN connection to a remote server.

Combining these two apps results in an application that users can install and run whenever they wish to so their encrypted data can be collected. Android requires a notification to be visible whenever a VPN service is running, so users can easily see when their traffic is being logged. A problem with relaying traffic to a remote server is that we add a delay to all packets. To limit this effect we make use of multiple servers, the app automatically selects the closest server to the phone, resulting in only minor added latency.

[6] https://github.com/SecUSo/privacy-friendly-netmonitor
[7] https://github.com/schwabe/ics-openvpn

4.5.1. Data processing and privacy

This thesis investigates whether there can occur a violation of privacy using traffic analysis. The data that is collected in this experiment can therefore also be sensitive. We take several measures to limit the intrusion on the privacy of participants in our experiments. Here we give an overview of which data is collected, how it is processed, and what impact this has on the privacy of participants. The collected data falls into three categories: Network traffic, App usage data, and Device information.

Network Traffic

While a user is participating in the experiment all their network traffic is tunneled through our servers using a VPN service. This means all their network traffic data is accessible on that server. Within the VPN all users get assigned their own IP from the range 10.8.0.0/24, their original IP address is thus not visible. Therefore, users cannot be identified based on their IP address, the local IP anonymizes them.

Not every app or website uses encryption for their connections. Data from these unencrypted connections is available in plain text on our servers. This data could be used to identify users and could contain sensitive information, like account details, or private messages. We, therefore, limit the amount of time this data is available on our servers to a practical minimum by removing the payload data from all captured traffic at regular intervals. Removing this data does not influence our experiment since we only require metadata. Furthermore, this has the added benefit of reducing the size of the data we store, reducing the size by at least 80%. For practical purposes it is easier to not do this live, but after the data has been captured.

The process of collecting network traffic is visualized in Figure 4.4 and can be described as follows. Using *tcpdump* all traffic on interface *tun0* is captured without any filters. *tun0* is the interface that is used by the VPN server, only the traffic of connected users is available here, so there is no noise of other traffic. This traffic data is stored in a pcap file as a cache. Every hour the current pcap file is closed and another is created to store the traffic of the coming hour.

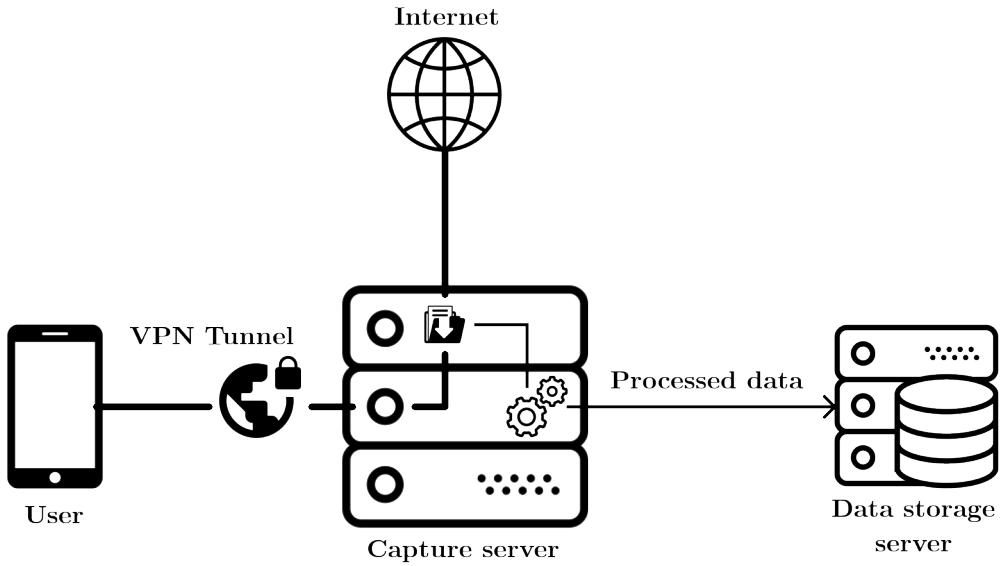


Figure 4.4: Overview of how network traffic is collected. The network traffic of a user is tunneled through the capture server and forwarded to the actual destination. On the capture server, the payload data is removed and the resulting processed data is sent to another server for storage.

The captured traffic data can then be processed to remove any payload data as visualized in Figure 4.5. When the data is captured on the Network interface it is processed. After the required information is extracted, the original captured data is deleted. First, we extract all TLS handshakes and DNS records from the captured data using the command line program *tshark* with a filter. These TLS handshakes and DNS records are required to obtain a reliable mapping from hostnames to IP addresses, which is used to remove any ambiguous data like advertisement data, which is similar across many apps. It is useful to extract the hostnames because a server with a single IP address can host multiple virtual servers. The hostnames are an extra feature to make a distinction between apps. We extract both for convenience, people could make use of a local DNS server, which wouldn't show in our logs, so TLS records could be used instead. Note that this mapping could also be

achieved using historic DNS records, so this is not an extra incursion of privacy, it is simply the easiest way to obtain this data. Next, we extract all TLS traffic from the captured data using *tshark*. This is done because once the payload data is removed from the captured traffic it is no longer possible to see which packets are TLS packets.

All required data is now extracted from the original pcap, so we can start to remove the payload data. This is done using the packet manipulation tool *scapy*, for each TCP and UDP packet, the payload data is removed, leaving only the headers. These headers are stored in a new pcap. The same process is done for the extracted TLS traffic, here the payload data is removed to save space. The non-TLS data is also stored because it might be interesting for further research.

For easy processing further in the pipeline the TLS headers, the TLS handshake messages, and DNS records are merged to a single file. The same is done for the non-TLS headers and the DNS records. To finish, five files are removed: the original pcap, TLS traffic, TLS headers, TLS handshakes, and DNS records. We are left with two pcaps: 1) all headers from the original network traffic with DNS records and 2) all TLS headers with the complete TLS handshake messages and DNS records. These pcaps are backed up to another server, and used for the experiments. The payload data never leaves the VPN server and is automatically removed within 2 hours after arriving.

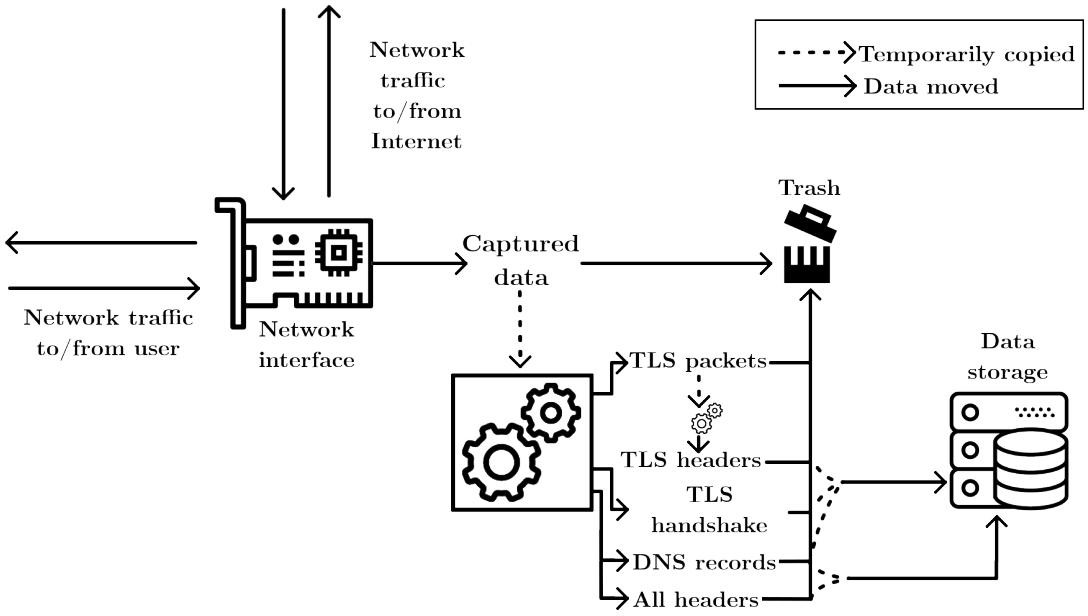


Figure 4.5: Overview of how network traffic is processed. The captured traffic is cached for a maximum of two hours before being processed. All data is deleted except for the headers, DNS records, and TLS handshake messages. Dotted lines indicate that a file is used as input for a process but is not saved.

App usage data

To obtain a ground truth of data for training and validating the classifiers, the network data needs to linked to the apps that they originate from. Therefore a log of which ports are used by apps is kept. A detailed explanation of this log can be found in subsection 4.1.2, with the main difference being that for the real-world data collection create the log in a Java environment instead of a shell. This log is uploaded to our servers at regular intervals and when a user stops participating in the experiment.

Android phones running Android 6.0+ (API 23+) go into Doze mode when the screen has been turned off for some time, the phone is not moving and the phone isn't charging. [8] The exact thresholds for when a phone goes into Doze mode can depend on the manufacturer's preferences. When a phone is in Doze mode apps cannot use the CPU or network, this means the counter till the next upload of the logs is put to sleep as well. This counter thus only progresses when the phone is active. Therefore we set the upload interval of the logs to 5 minutes, in reality, this is much less often since phones are not active at all times. Another method is to

[8] `https://developer.android.com/training/monitoring-device-state/doze-standby`

wake up the CPU at after a specific time using the *AlarmManager* [9]. However, doing this does not make much sense, since no network traffic is generated while the phone is in Doze mode anyway. Therefore logs are only uploaded for every 5 minutes the phone is active, which still ensures we lose little data.

Device information
There might be a significant difference in the network fingerprints generated by the same app on different devices or Android versions. If this is the case, it is possible to do more advanced classification by first classifying which device or Android version this traffic belongs to and then use a classifier specifically trained on data from that device or Android version. To do this, we need to be able to label the network traffic with device names or Android versions as well. Therefore we also collect some information about the device. This information comes from the Android *os.Build* package and can be accessed without requiring any extra permissions on Android. An overview of the information obtained can be found in Table 4.9.

Data	Description
baseOS	The base OS build the product is based on
developmentCodeName	Development state codename
deviceName	Name of the industrial design
manufacturer	Device manufacturer
model	Common name for the device
productName	Device codename
sdkVersion	API level of Android OS

Table 4.9: Information about the device that is collected. The information comes from the Android os.Build package[10].

4.5.2. User consent

As we are collecting and processing sensitive data we want to make sure users are well aware of the kind of data they share. It is not trivial to find out which user is behind the data we have captured. The privacy sets are quite small since we do not have many participants and do store information about the device they use. Therefore, someone with knowledge about who is participating in the experiment and which devices they use will still be able to identify people with a high degree of confidence. As we are recruiting people via the crowdsourcing platforms Mechanical Turk[11] and MicroWorkers[12], only they know who the participants are and there is an extremely small risk they will get access to the data we capture and be able to link it to their users. Still, as there is still a small risk that the data can be leaked and because we can inspect the data it is important, both from an ethical and legal perspective, to inform users about the data being processed and get their consent.

When users want to participate in the experiment and help with gathering labeled data they receive instructions, which can be found in Appendix A. Through these instructions we inform users about:

- How the app works
- Which types of data are collected
- What users should do
- Installation and usage

By informing users about the data collection and processing we allow them to make an informed decision whether they want to participate or not.

Before a user can start participating in the experiment they need to give consent to the collection and processing of their data. Furthermore, they need to acknowledge that they are 18 years or older because we do not want to collect data from a vulnerable group like minors. The consent is implemented in the form of a checkbox, see Figure B.1, which needs to be checked before the start button can be pressed. It is not possible to retract consent for already finished participation sessions since we do not know what data belongs to a user. The first time a user starts participating in the experiment they will receive another notification from the Android OS asking whether they trust our VPN provider. An example of this message can be seen in Figure B.2. The combination of these consent checks ensures that a user can be well aware of what data they are sharing.

[9] `https://developer.android.com/reference/android/app/AlarmManager`
[11] `https://mturk.com`
[12] `https://microwokers.com`

Users can see when the app is running in the app and through notifications displayed in the Status bar, the top bar on the device. Android pushes a notification by default when a VPN service is running, indicated by a little key on the right side of the Status bar. The data collection app pushes two more notifications, one with the status of the VPN connection and one indicating that the app is currently logging which ports apps use. These notifications can be seen in Figure B.3.

4.5.3. Data protection

Even though the amount of privacy-sensitive data is limited we still want to ensure the data doesn't fall into the wrong hands. As stated before, the payload data of the network traffic never leaves the server. However, there is a time window of up to two hours where it is available on the server, during which an attacker could steal this information. We use servers of the cloud provider Digital Ocean with the latest versions of Ubuntu 18 and OpenVPN and we connect to the server using SSH with key-based authentication. Therefore we have put in place all best practices for securing a server and therefore deem this threat highly unlikely. We expect only a highly motivated attacker to be able to compromise the server. If an attacker gains persistent access to the server even the shortest time window the data is stored will not be enough to prevent an attacker from stealing the data.

While the user is still connected to our VPN the app usage data and device information is uploaded through an FTP server. Therefore the log files are never sent over the Internet in clear-text. This FTP server is set up as a drop-box, it is only possible to upload files, not to view or to download files. Furthermore the upload of files is restricted to files with extensions *.csv* (for app usage data) and *.json* (for device information). Anyone could upload files, as the credentials for the FTP server are available in the code of the app. However, this can only be used for a denial-of-service attack on the server, and not for stealing any files. Every hour the device information, app usage logs, and processed pcaps are transferred to a data server for storage. This transfer of data is done over a secure connection with *rsync* using SSH with key-based authentication.

4.6. Metrics

The right metrics can give great insight into what changed between experiments. In the experiments we classify data, resulting in a list of predicted labels. We know the actual labels of the data and using this, we can create a table which shows for all labels how often it was assigned the correct label or some other label. This confusion matrix gives insight into how well the classifier performed, as it shows how often the data was assigned the correct label. However, the entire confusion matrix is not that insightful for many classes because it is very large. Therefore we use other metrics to give better insight into the results of the experiments.

Some metrics are invariant to changes in the confusion matrix and thus do not provide insight into any changes. Sokolova and Lapalme [54] analyzed which metrics are invariant for a selection of changes. From the metrics they discussed, we selected a variety of metrics such that the combination of the metrics can explain exactly what is happening in the results. The metrics used are:

- Dataset size
- Accuracy
- Micro-average Precision
- Micro-average Recall
- Micro-average F1-score
- Macro-average Precision
- Macro-average Recall
- Macro-average F1-score

The formulas for calculating the *Accuracy*, *Precision*, *Recall*, and F1-score can be found in Equation 4.1.

$$\begin{aligned} Accuracy &= \frac{TP+TN}{TP+TN+FP+FN} \\ Precision &= \frac{TP}{TP+FP} \\ Recall &= \frac{TP}{TP+FN} \\ F1-score &= \frac{2 \cdot TP}{2 \cdot TP+FP+FN} \end{aligned} \tag{4.1}$$

Dataset size is the number of samples in a dataset and gives an idea of how much influence small anomalies in the data might have influenced the results. In a smaller dataset, anomalies will have a larger effect than in large datasets.

Accuracy is the ratio of correct classifications over the entire dataset. It is calculated by taking all number of True labels and dividing it by the number of samples that was classified. Because the accuracy is calculated over the entire dataset and we are doing multiclass classification there is no sense of Positive or Negative classes, but simply True and False classifications.

For *Precision*, *Recall*, and *F1-score* we use two ways of averaging, micro-average and macro-average. Micro-averaging counts the total True/False Positives and Negatives and uses those for calculating the metrics. In the multiclass setting, this gives the same results as the Average metric, because there is no sense of Positive and Negative. So both the Precision and Recall scores calculate the ratio of correctly classified labels. F1-score is equal to $2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$, and so it is the same as the other metrics as well. We will therefore often not report the micro-average scores.

Macro-averaging averages the metric results for the individual classes. The class under consideration is the positive class, while all other classes are treated as the negative class. Because we are treating all classes with the same weight the result will not be the same as the accuracy. The macro average metrics are more suitable for measuring the performance of classes overall and give a clearer picture of the performance of smaller classes.

Precision, or Positive Predictive Value, gives the ratio of correctly classified samples out of all the samples labeled as the class under consideration. It thus indicates the chance that a predicted class actually is that class.

Recall, or True Positive Rate, gives the ratio of correctly classified samples out of all the samples that are actually the class under consideration. It thus indicates how many samples of a class are missed.

These metrics are chosen because combined they show all changes that occur to the positive and negative labels. Individually the metrics are invariant to certain changes, but for each change in the amount of True/False Positives or Negatives, there is always one metric that is not invariant to the change. When comparing the results of various experiments, we thus not only look at a single metric but look at all the metrics to be able to make clear what exactly changed between two experiments.

4.7. Comparing feature distributions

To find out how much fingerprints differ between classes and which features of the data differ more than others, we need a method to compare distributions. This gives insight into what makes a fingerprint of network traffic unique. We use the Anderson Darling (AD) test to compare the feature distributions.

The Anderson Darling test, proposed by Anderson et al. [3], is a statistical test for the hypothesis that a set of values has a specified distribution function $F(x)$. The test reports three values, a statistic, critical values, and a significance level. The significance level says at which significance level the hypothesis can be rejected. The critical values indicate which significance level should be chosen based on the statistic. The statistic is a value calculated by the AD test. The critical values are reported for 25%, 10%, 5%, 2.5%, and 1%, and their values depend on the number of samples. Figure 4.6 shows an example of how the significance level is obtained from the statistic. In this example, the test reports critical values of 0.325, 1.226, 1.961, 2.718, 3.752, and a statistic of 2.313. Therefore the significance level will be lower than 5% and higher than 2.5%, because it falls in between the critical values that correspond to those significance levels. A better approximation of the significance level is made by extrapolation, this is also used to approximate significance levels higher than 25% and lower than 1%.

This test cannot be directly used to see how similar two distributions are. When two distributions are both highly likely drawn from population F(x), we can say that they are similar. However, when two distributions are likely not drawn from the same population, we cannot say anything about how similar these two distribution are. We do not know which parts of the two distributions are not similar to the population F(x). To be able to do this we need to make use of the *k*-sample AD test.

The *k*-sample AD test was proposed by Scholz and Stephens [51]. It tests the hypothesis that *k* distributions are drawn from the same population and thus provides a means to see how similar distributions are. This test does not require the distribution function of the population to be specified. Because the test is not testing whether a distribution is drawn from a population with some static distribution function, but tests whether the *k* distributions are similar enough to be drawn from the same distribution, the significance level can be treated as a distance metric. The significance level is thus a reliable value of how much the distribution differ

from each other.

When testing whether many distributions are drawn from the same population, the AD test will report much lower significance levels on average than when testing only a few distributions. This makes sense as the chance that there are differences between many distributions is much higher than the chance that there are differences between only a few distributions. It is, however, something to take into account when we use the AD test to say something about how alike distributions are. Therefore we only use the *k*-sample AD test to compare two distributions, so that the significance level is always reported in the same context.

In this thesis the *SciPy* implementation of the *k*-sample AD test is used. This implementation sometimes has trouble extrapolating the significance levels for large statistic values. When the reported statistic is very high the significance level should be 0%, but it is often reported as a very high number, which is incorrect. In the same way, the extrapolation can cause significance levels of higher than 100% for a negative statistic. Figure 4.6 illustrates why extrapolation can cause this. We correct the extrapolation errors in all the calculations done with the AD test. When the statistic is higher than the highest critical value, which corresponds to a significance level of 1%, we check if the reported significance value is higher than 1%. If it is higher than 1%, something went wrong with extrapolating and the significance level is set to 0%. After this check, the remaining significance levels that are higher than 100% or lower than 0% are set to 100% and 0% respectively. This way all significance levels fall within the range 0% to 100% and are derived logically from the critical values.

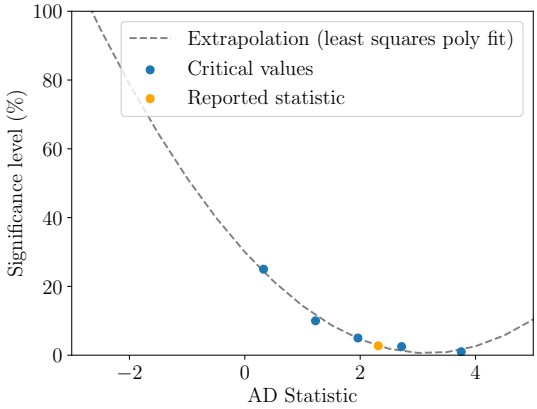


Figure 4.6: Example of how a reported statistic from the AD test is used to obtain a significance level based on the reported critical values. This figure also shows how bad extrapolation can lead to wrong significance levels. In this case, a high statistic will lead to a high significance level, instead of a significance level of (close to) 0%. The *SciPy* implementation of the AD test suffers from similar extrapolation errors.

5

Effect of new app versions on classifier performance

New app versions introduce new code and thus almost always change the network fingerprint as well. Big changes in the network fingerprint can negatively impact the performance of the classifiers, requiring more frequent model updates. This makes it harder to do accurate classification in a real-world setting, where apps are updated all the time. Furthermore, not all people update apps at the same time, so it is possible to encounter different app versions while analyzing network traffic. To see whether these different app versions are a problem for classifying mobile network traffic, this experiment tries to see how the performance of the classifier changes as it is tested on data generated by newer versions of the same apps. The experiment is performed in various settings and on various feature sets to get a better idea of what influences the performance and of the performance in various threat models. To easily build a large dataset with a consistent amount of data per app version, the data is generated using emulators (a.k.a. Android Virtual Devices). With this experiment we can find an answer to the following research question: *What is the impact of app updates on classifier performance?*

5.1. Experimental setup

A random set of 563 apps is selected from the top 10 000 of the Google Play Store. subsection 4.1.1 expands on how these apps were obtained. From 21 January to 29 April this set of apps was downloaded again roughly every two weeks, leading to a total of 8 sets of apps.

Traces were generated for each of the sets on the same Android Virtual Device (AVD). Each trace consists of 20 seconds of startup traffic and 80 seconds of traffic while using Android's Monkeyrunner to generate random user input. This way we obtain a fairly consistent amount of network traffic per trace and initiate various actions of the app. The traces were collected in the timespan of about a month. A complete overview of when the apps and traces are collected can be found in Figure 5.1. The dots show when the latest available versions of the apps were downloaded. The planes show when the traces were collected.

An RF classifier is trained on the earliest dataset of 21 January. This trained classifier is used to classify the data obtained in later sets. The performance of the classifier on each of the datasets gives an overview of the performance degradation over time. We expect the classifier to have a harder time correctly classifying the data in later datasets, as the network fingerprints start to deviate from the fingerprints the classifier was trained on.

Low-quality data can influence the performance of the classifier and thus the outcome of the experiment. The quality is assessed by performing cross-validation on the datasets. Quality here means how much anomalies the dataset contains; when a dataset contains many anomalies or outliers the cross-validation will give a worse result since the classifier would not be able to clearly distinguish classes. Anomalies are caused by errors in the data collection, e.g. due to a slow or crashed emulator, or due to rare outliers in the data. To highlight any differences in the quality of the datasets a base performance of each dataset is required. This base performance was obtained by performing 5-fold cross-validation on each of the datasets. Differences in the base performance between datasets can explain unexpected outcomes in the experiment, for example when the classifier suddenly performs much worse on a dataset.

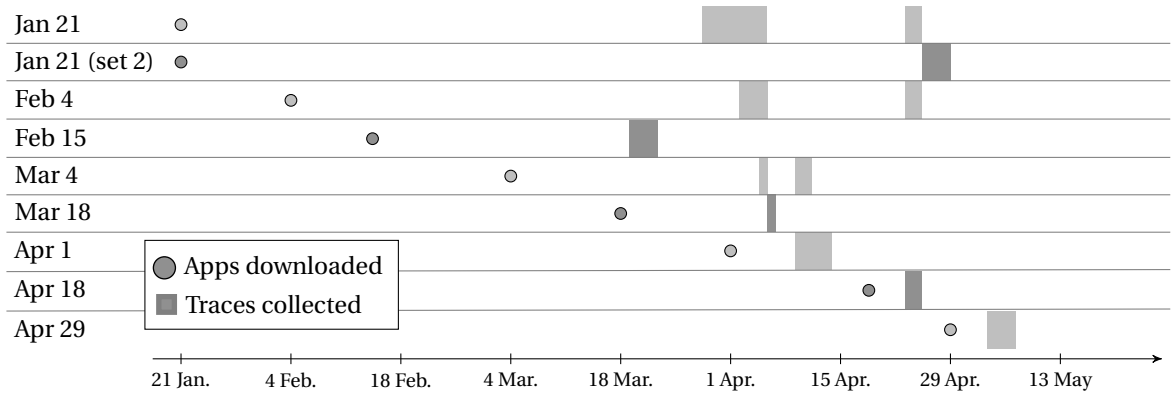


Figure 5.1: Timeline showing when the apps were downloaded for each dataset and when the traces were collected. The dots show on which day the apps were downloaded from the Play Store. The longer blocks explain during which time the traces were collected by running the apps on emulators.

5.2. Basic feature set

In this first experiment, we use a basic feature set that only contains features based on the size and timings of packets. Features like ports and IP addresses are discarded since these are known to change often and might negatively impact the performance of the classifier over time.

5.2.1. Full dataset

To simulate a setting where an attacker wants to classify all apps they encounter, we create a 'snapshot' of app versions at each date a new set of apps was downloaded. This means that each dataset contains traces from all the apps at their latest versions at that time. For example, the 4 March dataset contains traces from 135 apps with versions new on 4 March, 40 apps from 15 February and 36 from 4 February. For the remaining 251 apps that are not updated we obtain traces from the 21 January (set 2) dataset, so we do not use the training traces for those apps. The total amount of apps that are contained in the dataset is much lower than the 563 apps which have been downloaded. This is due to two reasons: 1) Some apps do not generate TLS traffic, which is the only traffic contained in the dataset. 2) Some apps only generate traffic that is associated with known ad domains, and thus the traffic is filtered from the dataset.

Dataset	Average # classes	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
21 Jan (set 1)	453	0.577	0.466	0.404	0.421
21 Jan (set 2)	450	0.595	0.500	0.417	0.435
4 Feb	452	0.612	0.502	0.427	0.445
15 Feb	447	0.629	0.544	0.468	0.486
4 Mar	446	0.637	0.542	0.473	0.489
18 Mar	447	0.663	0.546	0.476	0.493
1 Apr	447	0.655	0.531	0.466	0.480
18 Apr	445	0.639	0.534	0.459	0.475
29 Apr	446	0.639	0.530	0.449	0.469

Table 5.1: Baseline performance of the datasets. 5-fold cross validation was performed on the datasets. Each dataset contains traces of all 563 apps at the latest version at the time of the dataset. The traces do correspond to the app versions, but the dates the traces were collected vary. The amount of classes is much lower than the amount of apps, around 450, since not all apps produce (TLS) traffic and because traffic is removed by the ad filter. Since 5-fold cross validation does not always select all the classes, the average amount of classes for a fold is displayed. The accuracy metric shows the percentage of the bursts that were assigned the correct class. Precision, Recall, and F1-score show whether there are many false positives or false negatives for a class. These metrics calculated for all classes and averaged without taking into account the number of samples that belong to the class. These metrics are explained in more detail in section 4.6.

In Table 5.1 the results of cross-validation on each of the datasets can be found as reference for the quality of the datasets. This table shows the results for various metrics, so any changes in the confusion matrix can be spotted without having to inspect the matrix itself. The metrics are explained in more detail in section 4.6. For our baseline, we want to see whether datasets perform better or worse for all the metrics or just for some. When all metrics show an increase or decrease, the quality of the dataset is better or worse in general. If only

one metric changes, something else is probably wrong requiring a closer inspection of the dataset or classifier. There is a significant difference in the quality between datasets, especially the 18 March set is of higher quality when looking at all the metrics. These differences should be taken into account when evaluating the results. For our hypothesis that performance degrades over time, these quality results are not a big problem since overall the quality of later datasets is better than the quality of earlier datasets.

Training set	Validation set	Training # classes	Validation # classes	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
21 Jan	21 Jan (set 2)	453	451	0.576	0.484	0.396	0.417
21 Jan	4 Feb	453	453	0.567	0.454	0.384	0.399
21 Jan	15 Feb	453	448	0.553	0.428	0.367	0.376
21 Jan	4 Mar	453	448	0.553	0.419	0.367	0.373
21 Jan	18 Mar	453	448	0.540	0.394	0.355	0.353
21 Jan	1 Apr	453	448	0.530	0.382	0.344	0.342
21 Jan	18 Apr	453	448	0.516	0.363	0.317	0.316
21 Jan	29 Apr	453	448	0.509	0.349	0.303	0.302

Table 5.2: Results of the experiment to see the influence of newer app versions on the performance of classifiers. Full datasets were used for training and validation, meaning that all the apps at their latest version at that time were used.

The results from classifying the later datasets can be found in Table 5.2. Since we are using full datasets the number of classes in the validation (or testing) sets, are consistent and will not influence the results. When looking at the accuracy, precision, recall, and F1-score, we can see that with time the performance of the classifier goes down as well. This is exactly the behavior that is expected. As previously explained, due to changing functionality in the apps the network fingerprint of the apps changes, which makes correct classification harder. Even though the quality of the later datasets is better the decreasing performance over time can still be seen.

These results show that the changing versions are a concern for the real-world application of mobile network traffic analysis. Each month the metrics decrease 2-3 percentage points on average. To maintain an acceptable performance a model should thus be updated at least every few months, depending on the requirements of the attacker.

5.2.2. Only updated apps

The previous experiment showed the effect on the performance of a classifier averaged over a large number of apps. For specific apps, the apps that were updated, the performance degradation might be much worse. In this experiment, only the apps that were updated in the two weeks before the set of apps was downloaded are considered. The classifier is still trained on the same dataset from 21 January. This will highlight how a new app version degrades the classifier performance, without apps that haven't been updated dampening the effect.

Training set	Validation set	Training # classes	Validation # classes	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
21 Jan	21 Jan (set 2)	453	451	0.576	0.480	0.395	0.415
21 Jan	4 Feb	453	141	0.589	0.230	0.152	0.175
21 Jan	15 Feb	453	97	0.563	0.178	0.112	0.131
21 Jan	4 Mar	453	135	0.559	0.200	0.141	0.158
21 Jan	18 Mar	453	118	0.546	0.171	0.124	0.134
21 Jan	1 Apr	453	122	0.576	0.168	0.120	0.132
21 Jan	18 Apr	453	141	0.530	0.166	0.109	0.122
21 Jan	29 Apr	453	89	0.520	0.133	0.074	0.087

Table 5.3: Results of the experiment to see the influence of newer app versions on the performance of classifiers. The validation sets consist only of the apps that have been updated in the time between that set and the previous set. Notice that the Precision and Recall metrics drop significantly in performance as the app set changes. This happens because of a class imbalance, for classes with less data the classifier has a hard time assigning the correct label. When only considering the new versions the effect is more noticeable.

The results of the experiment are found in Table 5.3. The various metrics still show a drop in performance over time. Unexpectedly, however, the base performance as measured by accuracy is higher than in the experiment on the full datasets in subsection 5.2.1. We would expect the classifier to perform worse since now only app versions that were never seen before are classified. The only reason for this is that the data collected

in the 21 January (set 2) dataset is harder to classify on average than the data contained in the newer datasets. One explanation for this behavior can be that the complete set contains multiple apps that are hard to classify, which aren't present in these smaller sets. Another explanation is that the data in the 21 January (set 2) dataset was collected much later than most of the training data, as can be seen in Figure 5.1. However, this is less likely, as the effect of a different collection time is not noticeable for other datasets like the 15 February or 29 April datasets.

The Macro-average scores for this experiment are much worse than the other experiments and are also much worse compared to the accuracy. This can be explained by a class imbalance. Macro-average scores are calculated by taking the scores per class and averaging them. The poor performing classes with only a small number of bursts thus have a larger effect on the metric. Accuracy is calculated by taking all the traces and checking which are classified correctly, so it does not suffer from the class imbalance. Some apps generate very little data and others generate huge amounts. Because of this, the classifier will often misclassify the apps with a small amount of data, as there is not enough training data to build a robust fingerprint for those classes. When an app is updated and the small amount of traffic it generates is different from the training data, there is a high chance that all the samples from that app are misclassified. A misclassification of all or most of the samples results in low metric scores for that class, which is reflected in the macro average scores. When considering full datasets this effect is also present, but it is less noticeable due to the many apps that compensate the low scores of the few classes in the macro average metrics.

Training on old versions of updated apps

The previous experiment still could not highlight the influence changing app versions have as desired. The chance of misclassifications was quite high since the training set contained much more apps than the testing set. To truly see the effect of the changing app versions we should perform yet another experiment. In this experiment again all the different datasets are considered, only this time different classifiers are used for each set. For each set of updated apps, we select the traces from the same apps (only older versions) from the 21 January dataset. These traces are used to train a classifier which is then used to classify the traces of the new app versions.

Training set	Validation set	Training # classes	Validation # classes	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
21 Jan	21 Jan (set 2)	453	451	0.577	0.483	0.395	0.416
21 Jan (apps 4 Feb)	4 Feb	137	141	0.625	0.575	0.502	0.516
21 Jan (apps 15 Feb)	15 Feb	102	97	0.643	0.517	0.473	0.473
21 Jan (apps 4 Mar)	4 Mar	137	135	0.590	0.495	0.462	0.459
21 Jan (apps 18 Mar)	18 Mar	117	118	0.582	0.469	0.460	0.440
21 Jan (apps 1 Apr)	1 Apr	120	122	0.614	0.467	0.443	0.432
21 Jan (apps 18 Apr)	18 Apr	138	141	0.555	0.396	0.351	0.349
21 Jan (apps 29 Apr)	29 Apr	90	89	0.577	0.404	0.348	0.340

Table 5.4: Results of the experiment to see the influence of newer app versions on the performance of classifiers. The validation sets consist only of the apps that have been updated in the time between that set and the previous set. The training set contains all the apps that are in the validation set as their versions were on 21 January.

In Table 5.4 the results of the experiment can be found. The number of classes in the training set and the testing set is very similar. The slight difference comes from other (system) apps that are running in the background during the collection of the traces. The influence of time on the performance of the classifier does not become immediately clear from this table. Several other influences make it harder to see what we are looking for. Firstly, there is the quality of the datasets, previously showed in Table 5.1, whose influence becomes more obvious now, this explains, for example, the better performance of the 15 Feb dataset. Secondly, there is the influence of the different number of classes.

To make the trend more clear, the results are compared with the baseline from 5-fold cross-validation on the training sets, as found in Table 5.5. The results relative to the baseline are a metric that is more robust to the quality of the dataset and the number of classes in the dataset. Using the relative performance, found in Table 5.6, the the negative trend over time is visible again. The drop in performance is quite significant for the macro average metrics, suggesting that apps with a low number of training samples are much less robust to changes in versions than apps with a lot of training samples.

For the full datasets the relative performance of the results to the cross-validation on the training sets can be calculated by comparing the cross-validation results from the 21 January (set 1) to the results in Table 5.2.

Dataset	Average # classes	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
21 Jan	452	0.580	0.475	0.410	0.427
21 Jan (apps 4 Feb)	137	0.685	0.652	0.580	0.597
21 Jan (apps 15 Feb)	102	0.705	0.670	0.592	0.611
21 Jan (apps 4 Mar)	136	0.682	0.635	0.559	0.577
21 Jan (apps 18 Mar)	117	0.669	0.647	0.571	0.591
21 Jan (apps 1 Apr)	120	0.716	0.668	0.587	0.606
21 Jan (apps 18 Apr)	138	0.698	0.643	0.565	0.588
21 Jan (apps 29 Apr)	89	0.716	0.673	0.584	0.610

Table 5.5: Baseline performance of the training sets. The training sets contain the versions of the apps from each new dataset as they were on 21 January.

Training set	Validation set	Results relative to training set baseline			
		Average Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
21 Jan	21 Jan (set 2)	99.58%	101.63%	96.14%	97.35%
21 Jan (apps 4 Feb)	4 Feb	91.23%	88.18%	86.51%	86.41%
21 Jan (apps 15 Feb)	15 Feb	91.28%	77.19%	79.90%	77.40%
21 Jan (apps 4 Mar)	4 Mar	86.53%	78.04%	82.71%	79.56%
21 Jan (apps 18 Mar)	18 Mar	87.01%	72.51%	80.56%	74.48%
21 Jan (apps 1 Apr)	1 Apr	85.81%	69.94%	75.56%	71.18%
21 Jan (apps 18 Apr)	18 Apr	79.56%	61.62%	62.12%	59.33%
21 Jan (apps 29 Apr)	29 Apr	80.50%	60.04%	59.62%	55.83%

Table 5.6: Relative difference between the baseline of the training set in Table 5.5 and the results of the experiment in Table 5.4. This relative difference is more robust to the quality and the different number of classes in the training sets. It shows that the performance drops significantly as app versions become older than the training sets.

The relative results are at 88.3% for the accuracy and 71.7% for the F1-score for the 29 April dataset. This is much higher than the results we see in Table 5.6, confirming that the performance degradation is due to the changing app versions.

5.3. Extra feature set

The features of IP addresses and ports were previously discarded as they are thought to increase the performance degradation over time even more. However, for the overall performance they can be positive, perhaps so much that it outweighs the performance degradation. The destination port can be a useful feature since for almost all apps in our dataset, except four, is 443, the default HTTPS port. Table 5.7 shows the apps which don't use port 443. These abnormal destination ports are very useful to classify these specific apps since their destination ports are unique in the dataset. In larger datasets with more apps using abnormal ports, the destination port still has a strong discriminatory power for those apps.

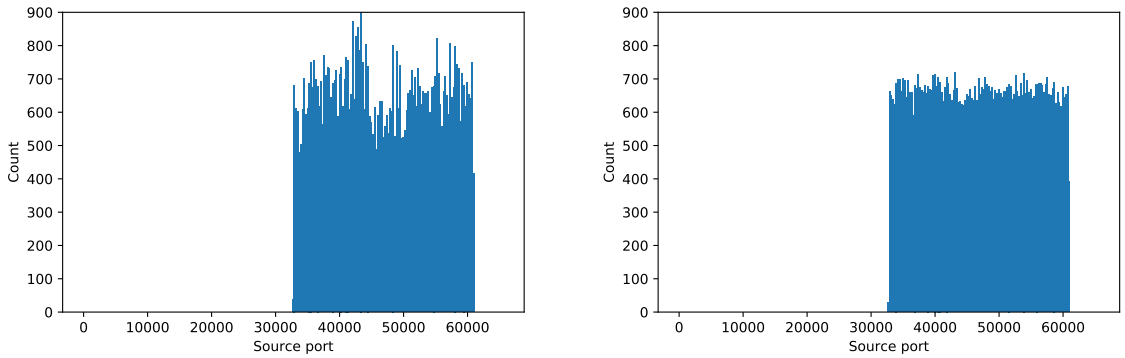
AppId	Port
com.creativemobile.DragRacing	8443
com.goodgamestudios.millennium	5222
com.metago.astro	8443
com.zeptolab.cats.google	7010, 7710

Table 5.7: List of apps which use SSL/TLS on a different port than 443. Searching for rare settings like this and adjusting the classifiers to take these into account can make some apps very easy to classify.

5.3.1. Source IP address and port

The source IP address is not used as a feature as this is entirely dependent on the user and not on the app. Source ports are usually chosen randomly, but in case (a range of) source ports are hardcoded in the app the classifier could use that as a method to classify a burst with more ease. In Figure 5.2 the distribution of the source ports in the 21 January datasets are plotted next to a uniform distribution of the same range and size. We can see that the distribution of source ports is not uniform, this would suggest that this feature can be used to discriminate between classes. As it turns out, the classifier sees this as well, because the feature importance of the source port is very high, up to 5%, while the next best feature has an importance of 1%.

However, the discriminating power of the source ports is misleading. Source ports are assigned in order from groups of ports by the Operating System. Because we have only 10-15 traces per app, each app will be



(a) 21 January dataset

(b) Uniform distribution

Figure 5.2: The distribution of source ports in the 21 January dataset are plotted as well as a uniform distribution of the same range and size. The bin size is 260. When comparing these datasets we see that the distribution of source ports is not uniform, suggesting that it is a good discriminatory feature, however this is caused by having too little training data.

associated with a small number of clusters of source ports. On average a class has 88 different source ports in the training set. With 453 classes, and about 30 000 possible ports there will be little overlap in the associated source ports between apps. Only with much more training data for each app, this bias will go away. The apps that request a specific range of ports will then emerge as having a strong correlation with the source ports. However, with the current amount of training data these patterns do not emerge. Because generating much more data is infeasible for this research, we will not use source ports as an extra feature.

5.3.2. IP addresses

The destination IP address will have a strong link with an app, but IP addresses can change over time. IP addresses change either because changing functionality does not require the apps to connect to that IP address anymore or because a server is assigned a dynamic IP address. In the first case, the IP addresses will change when the app version changes, in the second case the capture time is important. We expect the classifier to get a higher performance for the earlier datasets, where no or not many IP addresses have changed. However, we expect a much sharper drop in performance, since the RF classifier trained on the strong link between an IP address and a certain class, IP addresses will have been given a higher feature importance. When that IP address changes, the RF classifier will have a much harder time predicting the correct class. By training and testing on datasets with the destination IP address and port as extra features we test this hypothesis.

Dataset	Average # classes	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
21 Jan (set 2)	451	0.638	0.507	0.462	0.466
4 Feb	452	0.652	0.513	0.473	0.475
15 Feb	447	0.673	0.545	0.506	0.508
4 Mar	447	0.675	0.547	0.511	0.512
18 Mar	447	0.701	0.555	0.526	0.524
1 Apr	446	0.691	0.540	0.506	0.507
18 Apr	446	0.682	0.535	0.490	0.497
29 Apr	448	0.682	0.544	0.492	0.500

Table 5.8: Baseline performance of the datasets with extra features and IP addresses as continuous data. 5-fold cross validation was performed on the datasets.

Before testing the hypothesis we obtain a baseline using 5-fold cross validation, as found in Table 5.8. We see a consistent increase for all metrics of 4 percentage points on average for accuracy and recall, 3 percentage points for F1-score, and 1 percentage point for precision. Remembering the formulas from Equation 4.1, this means that by adding the extra features we mainly obtain more True Positives (TP) and (thus) less False Negatives (FN) on average. The effect is stronger in the Recall metric as its value depends on both the TP and FN. The same effect is visible in the results of the experiment with the destination IP address and port as extra features in Table 5.9. As expected, the performance of the classifier is higher, about 1 to 3 percentage

Training set	Validation set	Training # classes	Validation # classes	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
21 Jan	21 Jan (set 2)	453	451	0.606	0.478	0.427	0.432
21 Jan	4 Feb	453	453	0.601	0.454	0.415	0.415
21 Jan	15 Feb	453	448	0.588	0.430	0.397	0.393
21 Jan	4 Mar	453	448	0.586	0.421	0.396	0.389
21 Jan	18 Mar	453	448	0.571	0.397	0.382	0.369
21 Jan	1 Apr	453	448	0.560	0.386	0.370	0.357
21 Jan	18 Apr	453	448	0.549	0.362	0.345	0.330
21 Jan	29 Apr	453	448	0.540	0.346	0.329	0.314

Table 5.9: Results of the experiment to see the influence of app versions on the performance of classifiers. In this experiment we use the destination IP address and port as extra features. IP addresses are treated as continuous data, their integer values are used as feature.

points, than the performance without the extra features as previously shown in Table 5.2. However, surprisingly we do not see a sharper drop in the performance of the classifier. This is most likely due to the data being collected in a relatively short period of one month, as previously shown in Figure 5.1. Therefore, the data is less influenced by servers with dynamic IP settings. A complete overview of when data was collected for each dataset can be found in Figure 5.1. Another reason might be that the IP addresses do not change significantly more than the other features change, so the changing IP addresses have no added effect to the degradation of the performance.

IP addresses as continuous or categorical data

IP addresses can be easily converted to integers (continuous data) and used as input for the classifiers. However, this might cause the RF to treat this feature incorrectly. In the experiment above IP addresses were treated as continuous data. The problem with adding continuous data to an RF is that it implies that the feature has a sense of larger/smaller or closer/farther. This might not be intended behavior. To solve this problem we can treat IP addresses as categorical data. However, this requires some pre-processing of the feature, as explained in subsection 4.1.5, before it can be used as input for the RF classifier. The following examples explain how treating IP addresses as continuous or categorical data might impact the behavior of the classifier.

Grouping IP addresses that are close to each other might make sense. Take for example a server used by app *A* that is assigned the IP address *81.43.85.2*, but due to some changes in the configuration is assigned a new IP address from the same subnet: *81.43.85.12*. When we train our model class *A* is associated with *81.43.85.2*. Other classes are associated with different IP addresses from completely different subnets. Lets say the two closest IP addresses in the trained model are *93.12.4.2* for class *B* and *80.34.10.7* for class *C*. At a later point, we obtain new data that we want to classify, now with the new IP address *81.43.85.2*. The Random Forest will branch based on the value of the IP address. If we follow the tree Figure 5.3, we can see that for both IP addresses we end up with the correct label *A* because the values of the IP addresses are close to each other.

In the above case treating IP addresses as continuous data was positive for the quality of the model. However, there is another possible situation that highlights a case where it is negative for the model. Consider the same app *A* which connects to a server at a cloud provider with IP *81.43.85.2*. A model is trained on more data from two other apps of which one also uses the same cloud provider as app *A*. Class *B* with IP address *81.43.85.5* and class *C* with IP address *182.51.5.6*. Over time the IP address of app *A*'s server changes again to *81.43.85.12*. However, now, as we try to classify the new data, it is wrongly assigned the label *B*. In this case, we would benefit from treating the IP addresses as categorical data, since *81.43.85.12* should be considered as far from *81.43.85.5* as from *182.51.5.6* as they belong to completely different apps. Treating the IP addresses as categorical data would mean that the RF will output low confidence for any class that it assigns since all IP addresses that are in the model are considered far away. This happens because in the Random Forest there are many trees, which all have different split conditions. So in one tree we might split on whether the IP address falls into hash bucket 1 and in another tree on hash bucket 3. Overall this means that we could end up with any class depending on the hash bucket

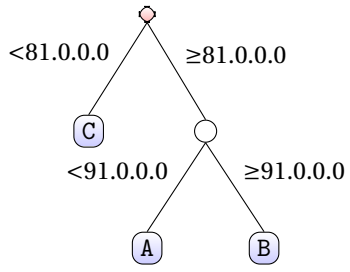


Figure 5.3: Possible decision tree for deciding a class based on using the IP address as a continuous feature. When the server used for app *A* changes its IP address to *81.43.85.2* which is in the same subnet as the previous IP address, we can see that the correct class is still assigned, because the new IP address is close to the previous one.

our new IP address is put into. We still will not get the correct class, but at least we get an indication that there is a large chance we have an incorrect label assigned unless we get a hash collision.

Training set	Validation set	Training # classes	Validation # classes	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
21 Jan	21 Jan (set 2)	453	451	0.607	0.472	0.419	0.425
21 Jan	4 Feb	453	453	0.602	0.450	0.407	0.409
21 Jan	15 Feb	453	448	0.586	0.426	0.390	0.388
21 Jan	4 Mar	453	448	0.585	0.418	0.388	0.384
21 Jan	18 Mar	453	448	0.569	0.395	0.377	0.366
21 Jan	1 Apr	453	448	0.559	0.384	0.366	0.354
21 Jan	18 Apr	453	448	0.549	0.363	0.342	0.331
21 Jan	29 Apr	453	448	0.540	0.348	0.326	0.315

Table 5.10: Results from the version experiment using extra features and IP addresses as categorical data. The model used is a single RF classifier.

Treating IP addresses as categorical data

Whether to use IP addresses as categorical data or continuous data is not immediately clear. That is why we experimented with both. We use feature hashing to treat the domain names as categorical data, this is explained in more detail in subsection 4.1.5. This results in extra 1 000 features that are set to 0 or 1 based on the domain name. The results from the experiment with IP addresses as categorical data can be found in Table 5.10. For all metrics, the difference is less than 1 percentage point on average. The difference is too small to say something conclusive about whether treating IP addresses as continuous data or categorical data is better.

Training set	Validation set	Training # classes	Validation # classes	Accuracy	Macro-avg Precision	Macro-avg Recall	Macro-avg F1-score
21 Jan	21 Jan (set 2)	453	451	0.613	0.471	0.422	0.426
21 Jan	4 Feb	453	453	0.612	0.451	0.411	0.412
21 Jan	15 Feb	453	448	0.597	0.429	0.395	0.392
21 Jan	4 Mar	453	448	0.595	0.418	0.390	0.385
21 Jan	18 Mar	453	448	0.580	0.397	0.377	0.366
21 Jan	1 Apr	453	448	0.570	0.383	0.365	0.354
21 Jan	18 Apr	453	448	0.561	0.364	0.343	0.333
21 Jan	29 Apr	453	448	0.553	0.349	0.329	0.318

Table 5.11: Results from the version experiment using extra features and IP addresses as categorical data. The model used is a per class RF classifier.

Training set	Validation set	Training # classes	Validation # classes	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
21 Jan	21 Jan (set 2)	453	451	0.633	0.497	0.449	0.454
21 Jan	4 Feb	453	453	0.630	0.473	0.435	0.436
21 Jan	15 Feb	453	448	0.622	0.454	0.421	0.418
21 Jan	4 Mar	453	448	0.614	0.439	0.412	0.407
21 Jan	18 Mar	453	448	0.601	0.418	0.401	0.390
21 Jan	1 Apr	453	448	0.588	0.405	0.389	0.376
21 Jan	18 Apr	453	448	0.581	0.385	0.368	0.355
21 Jan	29 Apr	453	448	0.575	0.372	0.354	0.341

Table 5.12: Results from the version experiment using extra features and domain names instead of IP addresses, domain names were added using feature hashing with 1 000 columns. The model used is a per class RF classifier, as 1 000 extra features requires a very large single classifier that does not fit in memory. The performance of a per class classifier is very similar to a single classifier.

5.3.3. Domain names

Server Name Indication is an extension to the TLS protocol that allows a server to manage multiple domain names on a server behind the same IP address [8]. A client indicates which domain name they try to connect to and the server serves the certificate and content of that specific domain name. One domain name might also

point to multiple IP addresses for load balancing. For example, in all datasets the *graph.facebook.com* domain name is associated with the same four IP addresses *31.13.64.16*, *31.13.92.10*, *157.240.20.15*, and *185.60.216.15*. So where for one trace an app is connecting to IP address *31.13.64.16*, a trace collected an hour later might connect to *157.240.20.15* leading to more noise in the data. Because of these reasons, domain names are expected to have a larger discriminatory power than IP addresses. Another added benefit of domain names is that, unlike IP addresses, domain names usually do not change over time. Therefore when we use domain names as a feature instead of IP addresses, we would expect less performance degradation over time compared to using IP addresses as features, simply because this is a more constant feature. But this cannot be verified because there was no significant added performance degradation when using the IP addresses.

The results of the experiment using domain names instead of IP addresses can be found in Table 5.12. We compare these results to the results in Table 5.10 of using IP addresses as categorical data. From these tables, we see that using IP addresses as categorical data yields better results, about 2 percentage points for all metrics. The reason for this performance increase is that, on average, each domain name is associated with 3.5 IP addresses and each IP address is associated with 2 domain names. So firstly, an app is more strongly bound to a specific domain name than to three different IP addresses. And secondly, the domain name is more specific than a single IP address which is used to access content behind two domain names and is associated with more classes. The performance degradation is comparable to the basic feature set and using the IP address as an extra feature. There is thus no added performance degradation because of changing domain names.

Domain names have a high discriminatory power leading to much better performance compared to using the basic feature set of Table 5.2. On average the metrics perform 2 to 6 percentage points better. The results do not show that there is an extra drop in the performance when using domain names as a feature. Over multiple months, the influence of the changing domain names could become more prominent. However, as discussed in subsection 5.2.1, it is a good idea to update the model every few months anyway. We showed that for a relatively short time span of 1 month, the effect is not noticeable.

5.4. Why does performance degrade over time?

From the experiments in this chapter, we have seen that the performance of a classifier drops when classifying data from newer app versions than the app versions of the training data. Various feature sets were used to see if a classifier could be made more robust, but all classifiers showed the same downward trend over time. In Figure 5.4 the F1-score and accuracy for all classifiers are plotted over time. Time here means the date of the app versions. No clear differences in the rate of performance degradation can be seen in these plots. It mainly confirms that using destination domain names, or IP addresses has a positive effect on classifier performance.

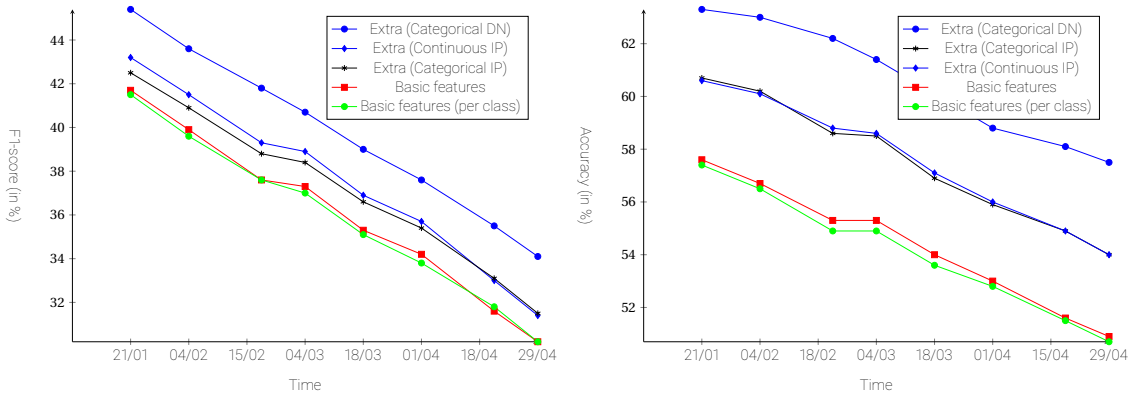


Figure 5.4: Plots of the results of the version experiment for various classifiers that are discussed in this chapter. For simplicity only the F1-score and accuracy are plotted for each classifier.

The experiments do not explain why the performance drops over time. For this, a more thorough analysis is required of the classifier and the data. Therefore in this section, we will take a closer look at which features are deemed important (or discriminatory between classes) by the classifier. In addition to this, the feature distributions are analyzed, to see how much feature distributions change over time. Knowing which features contribute the most to the performance degradation can give an insight into how the classifier can be improved by selecting other features. It also gives an insight into why classification works and which countermeasures might be the most effective.

5.4.1. Feature Importance

By analyzing the feature importances of the various classifiers that we have trained, we can see which features the classifiers considers as most discriminating between classes and contribute the most to the performance. This does not yet say something about which features change when app versions change. Table 5.13 gives an overview of the feature importances for the classifiers trained on the 21 January dataset using the basic and the extra feature set. The extra feature set contains the destination port as a feature and 1000 extra features used for the feature hashing of the destination domain name. There is a lot of overlap between the top important feature of both feature sets. The small differences can be explained by the inherent randomness of the RF classifier and the fact that the extra feature set contains a lot of extra features.

Basic set		**Extra set**	
Feature	Importance	Feature	Importance
out_iat_min	0.0109	com_iat_80_percentile	0.0089
out_size_max	0.0099	out_iat_min	0.0081
out_size_70_percentile	0.0095	out_size_total	0.0080
com_iat_70_percentile	0.0094	out_size_90_percentile	0.0078
com_iat_mean	0.0094	com_size_total	0.0077
out_size_90_percentile	0.0093	com_iat_70_percentile	0.0077
in_iat_min	0.0093	out_size_70_percentile	0.0076
out_size_60_percentile	0.0091	com_size_var	0.0076
out_size_total	0.0089	com_iat_60_percentile	0.0075
in_iat_20_percentile	0.0087	com_iat_mean	0.0073

Table 5.13: Overview of the top 10 features sorted by feature importance for the classifiers trained on the 21 January dataset using the basic and the extra feature set. Note that the extra set contains 1003 more features, because of the 1000 columns used for feature hashing the domain names. These 1000 columns have a combined importance of 0.1275.

The features of the domain name have a combined importance of 0.1275, which is very high. This is reflected in the increase in performance for the classifier that uses the extra feature set. The domain names are a good feature to discriminate between apps. Perhaps, surprisingly the destination port is given a very low importance of 0.0001, despite that we have seen that uncommon ports are strongly linked to a few apps. The reason for this is that the destination port is a discriminating feature for only 4 apps, so overall it cannot discriminate between apps all that well.

The other features come from applying statistical methods on series extracted from the bursts: 57 packet size, 54 inter-packet size, and 54 inter-arrival time features. The remaining 6 features have to do with the packet count and the duration of the bursts. The average importance (as extracted from the basic set) of the feature groups is quite similar, but the features related to count and duration have about 1/3 of the importance of the others. This tells us that the packet count and burst duration have less impact on the classifier performance. But still does not give much more insight into why performance is degrading over time, for this, we need to analyze the feature distributions.

5.4.2. Clustering classes

The classifiers can be seen as attempting to make a distinction between classes in a high-dimensional space. Depending on whether we use the basic or extra feature set this space has 171 or 173 dimensions. This high dimensional space can be reduced to a 2-dimensional space to visualize that samples of classes cluster together in this high dimensional space. A method to do this is t-distributed Stochastic Neighbor Embedding (t-SNE) [39]. t-SNE is a technique for visualizing high-dimensional data. Similar objects in high-dimensional space are modeled close to each other in low-dimensional space. Using t-SNE the structures that exist in high dimensional data can be projected onto a 2D-plane so the data can be visualized.

Using t-SNE we can visualize that classes for which the classifier has a high performance have samples that are clustered together and classes which are hard for the classifier are seemingly randomly distributed. In Figure 5.5 and Figure 5.6 t-SNE was performed on all classes with a relative high performance and low performance. t-SNE has a harder time to visualize classes with a low amount of samples, so the classes all have about the same sample size. The F1-score is used to select the best- and worst-performing classes. A complete overview of the classes, their F1-score, and sample size can be found in Table C.1.

In these two figures, we can see that the samples from the best performing apps are similar in high-dimensional space. Samples from apps are clustered together in the figures. It is easy to imagine that the Random Forest classifier can make a distinction between these classes. On the right of the figure, there is a

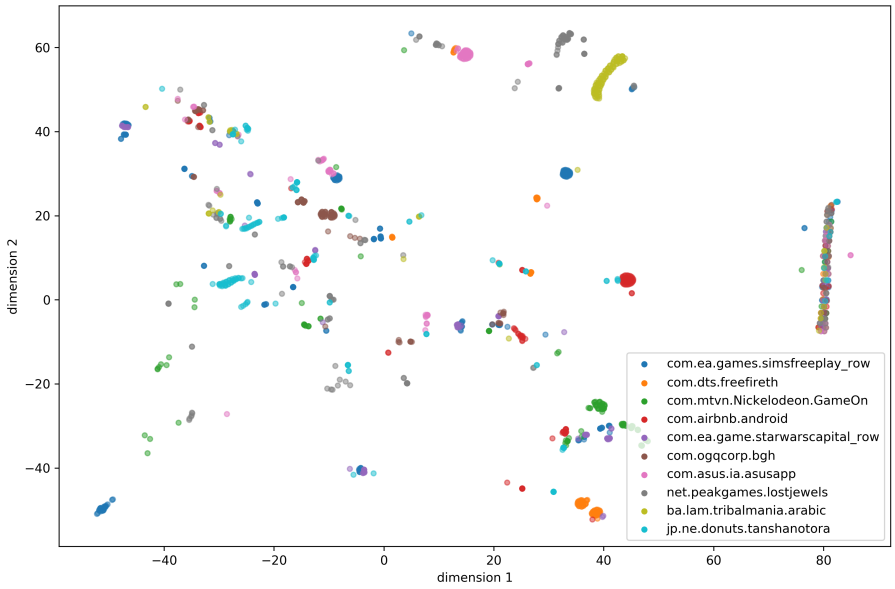


Figure 5.5: t-SNE performed on 10 well performing classes from the 21 January dataset. Compared to 10 poor performing classes in Figure 5.6 these classes can be divided into clusters easily.

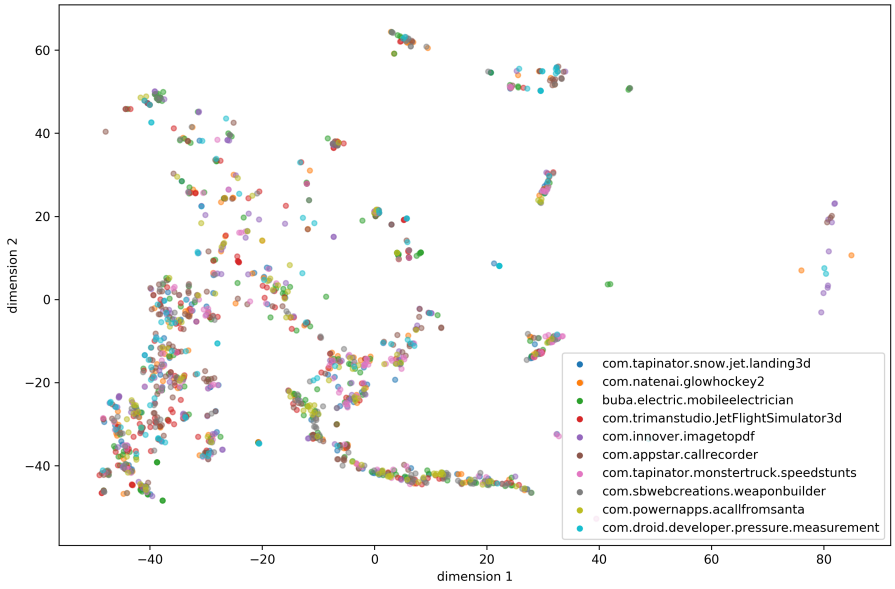


Figure 5.6: t-SNE performed on 10 poor performing classes from the 21 January dataset. The samples are not clustered well and spread out quite randomly.

cluster of samples from many different apps. Many apps have some bursts that are very much alike. A possible reason for this could be that all of these apps use some default library that generates very similar traffic every time. Another reason might be that these are bursts in which a lot of data is downloaded, so the bursts contain only packets of max size and have very short inter-arrival times. One can imagine these types of bursts occur for many apps. To find out the actual reason these samples are clustered together one would need to analyze the data deeper to find out the similarities between bursts. But whatever the reason may be, this would be hard for the Random Forest to classify. We would expect these types of clusters since the selected apps do not have a perfect score.

For the worst-performing apps, we see very few big clusters and a lot of small groups of samples or single samples. Having a lot of clusters should not be a problem and is even expected behavior, as many different bursts can be generated by an application. However, a problem that becomes apparent here is that either bursts with a specific feature distribution occur once or only a few times in our data. This means that the apps show different network behavior each time they were run during the data collection phase. A bigger problem is that we can see that all the small clusters overlap with other clusters. This means that a burst with a certain feature distribution could be multiple classes. So from this visualization, we can expect the Random Forest classifier to have a very hard time making the correct classification for a sample from one of the selected apps.

5.4.3. Selecting the best classes

From the visualizations discussed in subsection 5.4.2 we can imagine that the performance of a classifier depends a lot on the selection of apps. To verify this we took subsets of the 21 January dataset based on the F1-score of the apps from 5-fold cross-validation on the whole dataset. On these each of the subsets 5-fold cross-validation is performed; the results can be found in Table 5.14. As is expected, the top 100 performing apps in the whole dataset also have the highest score when considered as a subset. The subsequent worse-performing apps also perform worse in the expected order. An important takeaway for this is that there exist subsets of apps that perform significantly better than others. These results thus imply that it is possible to be 'lucky' with the choice in apps.

Selection	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
top 100	0.866	0.885	0.790	0.825
top 100-200	0.616	0.702	0.594	0.625
worst 100-200	0.316	0.351	0.317	0.319
worst 100	0.192	0.166	0.155	0.151

Table 5.14: 5-fold cross validation performed on subsets of apps from the 21 January dataset. The apps were first ordered on their F1-score performance when evaluating the whole dataset. Afterwards sets of best and worst performing apps were used to create separate datasets. It can be clearly seen that by choosing a specific set of apps the performance of a classifier can be boosted greatly.

Subset of 100 best apps

Classification of encrypted network traffic for all apps might not be feasible. However, there could exist a subset of apps of interest for which it is very much so feasible to do so. Consider the top 100 apps, with an F1-score of 100 to 63.3%, from the 21 January dataset. We create a dataset from those apps and add all the remaining apps as the class 'other'. Now we do a cross validation on this dataset, the results of which can be found in Table 5.15.

While the performance seems worse than when doing cross-validation only on the top 100 apps, notice that the precision is much higher in this setting. This means that when a burst is classified as one of the 100, the chance that it is correctly classified is very high, above 90%. The lower recall of around 60% for the macro setting, says that the classifier does label quite a few bursts as the negative class instead of one of the 100. However, when an attacker wants to identify which apps are used in traffic this is not a big problem since we can expect there to be other bursts that are identified. An attacker would mainly want to avoid detecting apps that are not there, and they would thus require high precision. This subset of apps might not show the same behavior when the negative class becomes much larger, consisting of much more other apps, but it is reasonable to assume that such a subset can still be found.

Subset of 100 worst apps

In the same way, as we select a subset that performs well, we can select the subset of 100 worst performing apps and show that it can be very hard to classify a stream of unknown data. The results from selecting the 100

Selection	Accuracy	Micro-average Precision	Micro-average Recall	Micro-average F1-score	Macro-average Precision	Macro-average Recall	Macro-average F1-score
100 best	0.872	0.915	0.768	0.835	0.929	0.671	0.765
100 worst	0.906	0.301	0.029	0.052	0.131	0.019	0.032

Table 5.15: The results of 5-fold cross-validation performed on the 21 January dataset with all classes other than the 100 best-performing apps labeled as the negative class. The precision, recall, and F1-score metrics are calculated on the selection of 100 best or worst performing apps only. The accuracy metric includes the majority negative class as well. This result shows that it is feasible to classify a selection of apps with a distinctive fingerprint in a dataset that contains many other apps as well.

worst performing apps, with an F1-score of 0 to 21.4%, can be found in Table 5.15 as well.

The average accuracy of 86.6% is still very high and similar to the selection of the 100 best apps. This can be explained by the classifier labeling the negative class correctly. When we look at the metrics that do not consider the negative class, a different picture emerges. There is a rather large difference between the metrics in the micro setting and the macro setting. Remember that the micro setting of the metrics averages over all the samples, and the macro setting averages over the results from each class. This difference thus implies that some large classes perform better than others, while on average the classes do not perform well at all. This difference might be the result of the slight bias that exists where classes with a low amount of samples also have a lower score. This slight bias can be seen when creating a scatter plot of the F1-scores and the number of samples in Figure 5.7 The macro average precision score at 29.1% is fairly low and too low to use in a real-world setting because there is too much uncertainty about whether data is classified correctly. The macro average recall score of only 4% tells us that for many classes no or very little bursts could be correctly classified; almost all bursts for many classes are labeled as the negative class. This means that for many apps correctly classified bursts will be rarely seen.

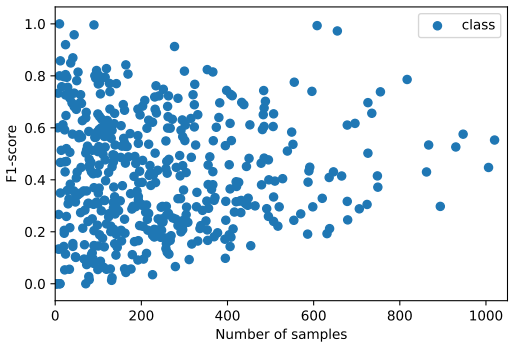


Figure 5.7: Scatterplot showing the correlation between the sample size of classes and the F1-score for the 21 January dataset. From this image, we can see that the poor performance of some classes is not necessarily caused by a low number of samples. Seven classes with more than 1050 samples and all with an F1-score higher than 0.5 are cropped out.

These experiments imply that it is possible to classify a specific set of apps in a data stream that contains network traffic from many other apps, which is considered noise. It does not prove the general applicability of this method in the real world. In a real-world setting, an attacker would want to be able to classify a specific set of apps. However, there is no way of guaranteeing that a subset of apps that works contains these specific apps.

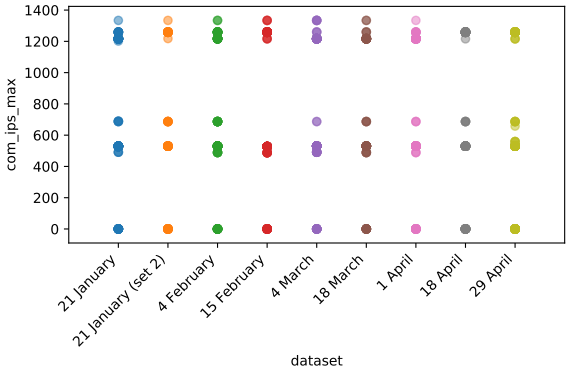
5.4.4. Some examples

We know that over time features change, however, it is useful to know which features change the most. This gives an insight into what exactly in the app updates causes the requirement for the models to update. As previously mentioned in section 4.7, the k-Sample Anderson Darling test is a method to calculate the difference in feature distributions. This tests the hypothesis that k samples are drawn from the same population. It reports a significance level which tells the degree of certainty that the hypothesis can be rejected. A lower significance level means that the hypothesis that two distributions are drawn from the same population can be rejected with a higher degree of confidence. And these values can thus be used to say something about how much distributions differ from each other.

It is not so clear from all these results why classification works. From the t-SNE plots in subsection 5.4.2 we can see that the complete feature distribution cluster on classes in high dimensional space. However, it is still not very insightful what makes the network traffic from an app unique and why it helps to extract statistical features from the obtained data. Therefore we will take a closer look at the feature distributions for a couple of

Dataset	Significance Level
21 January	-
21 January (set 2)	0.045
4 February	0.595
15 February	0.000
4 March	0.815
18 March	0.978
1 April	0.776
18 April	0.004
29 April	0.006

(a) Reported significance values



(b) Feature distributions

Figure 5.8: Comparison between datasets of the distribution of the *com_ips_max* feature for *com.airbnb.android*. The significance level is based on the two sampled Anderson Darling test between the 21 January dataset and later datasets. The differences in the distributions are subtle to the eye, but are significant according to the Anderson Darling test.

classes and try to get a deeper understanding of why all these features can produce a unique fingerprint for an app.

Take for example the app *com.airbnb.android*, this app occurs in all of the datasets. We perform the Anderson-Darling test and compare the feature distributions of the 21 January dataset with the other datasets for this app. Then we select an arbitrary feature for further analysis: the maximum inter-packet size within a bi-burst. The inter packet size is calculated as: size of the current packet – size of the previous packet.

The significance level from the Anderson Darling test says something about how similar the distribution is to the distribution from the 21 January dataset. These can be found in Figure 5.8a. The distributions of the feature can be found in Figure 5.8b. When comparing the feature distributions we can see why the distributions of some datasets are reported to be more similar to the training data than others. Take for example the 15 February set, it misses entries in the distribution around 750. The 4 March and 18 March sets appear to be very similar to the 21 January set, which is confirmed by the reported significance level.

The missing values in the 15 February set are not likely to be caused by a changing version since the datapoints come back for later sets. The missing highest values for 18 April and 29 April, could be a permanent change, which would point to a changing version being the cause. Notice that for this feature the change in distribution does not seem to be correlated with the versions all that much. It is especially strange the feature distribution for the 21 January (set 2) is that different. Perhaps the time that the traces were collected is more important here. This would make sense since Airbnb shows a selection of featured places to rent on start-up. Clicks on these places are what causes the most network traffic for this app. So when the selection of featured places changes, the network traffic fingerprint changes as well. When looking at the collection times in Figure 5.1, this does seem to be the case, although it still does not explain all the differences.

App whose performance degrades over time

To get an insight into why the performance of certain apps degrade over time we take a closer look at an app for which this is the case. 21 apps occur in all datasets, meaning they have received an update each time between the moments where new app versions were downloaded. From these apps, there are many whose performance does not degrade or whose performance even improves. We have seen from the averages over all the classes that performance does degrade as we add newer app versions, therefore we must conclude that app performance degrades more often than it improves over time. Therefore we will pick an app for which the performance clearly degrades over time and analyze that app.

For all the apps that occur in every dataset, the F1-score for those individual apps for each dataset is calculated. From these results, an app that shows a nice degradation over time

Dataset	F1-score	Samples
21 January	-	461
21 January (set 2)	0.871	309
4 February	0.809	477
15 February	0.824	299
4 March	0.781	415
18 March	0.711	536
1 April	0.750	522
18 April	0.726	688
29 April	0.683	397

Table 5.16: F1-scores for the app *com.rovio.ABstellapop* calculated for each dataset, which was classified by a classifier built on the 21 January dataset. This app was selected, because it shows a nice drop in performance over time.

was selected. This app is *com.rovio.ABstellapop* and its F1-score
over time can be found in Table 5.16. In this table, as the app is updated and thus changes more compared to the training data, the F1-score drops, this is independent of the number of samples. This app is Angry Birds POP Bubble Shooter, a popular game with more than 10 million installs. It produces quite a lot of bursts, on average about 45 per trace. This game starts downloading its data when it first starts. Therefore after each update the data that is downloaded changes and because we only capture data from freshly installed apps the fingerprint changes as well.

For this app, we perform the Anderson-Darling test comparing the feature distribution of the 21 January dataset (the training data) with the feature distributions of the later sets. From this analysis, we can find out which features do not change over time and which features change a lot over time. We would expect to see for certain features lower significance levels from the Anderson-Darling test when performing the test on later datasets, the same way as the F1-score degrades.

When we calculate the average significance level for all of the features, we see from the results in Table 5.17 that the significance level drops over time. This is exactly what we would expect; as the feature distributions start to diverge more and more from the feature distribution of the training data, the classifier has a harder time to correctly classify the bursts.

Dataset	**Average over all features**	**Average over important features**	**in_size_total**	**out_ips_mad**
21 January (set 2)	0.260	0.109	0.923	0.138
4 February	0.229	0.107	0.381	0.366
15 February	0.029	0.010	0.001	0.007
4 March	0.152	0.065	0.010	0.065
18 March	0.053	0.023	0.000	0.000
1 April	0.093	0.048	0.002	0.027
18 April	0.056	0.023	0.006	0.715
29 April	0.080	0.030	0.043	0.809

Table 5.17: Significance levels for each of the datasets for the app *com.rovio.ABstellapop*. The significance level reports the chance that the features in the 21 January dataset and the subsequent datasets are drawn from the same distribution. Important features are the features that have a higher than average importance value. Also shown: the features with the lowest and highest trend over the significance levels.

Now that we know that on average the distributions drift from the training data, it is interesting to see which features change the most. For this, we calculate the significance levels for all the features and look at the trend line over the datasets. This trend line is calculated using linear regression. For 133 out of 171 features the significance level shows a negative trend over time. The lowest slope value is -0.096 for the feature *in_size_total* and the highest slope value is 0.077 for the feature *out_ips_mad*. The significance levels for these features are also shown in Table 5.17. This shows that how a fingerprint drifts, does not become clear from a single feature, but averaging all features does show the drift of the entire fingerprint as learned by the Random Forest.

37 features have an increasing significance level as time progresses. These contain 28/54 of the Inter Arrival Time (IAT), 3/54 of the Inter Packet Size (IPS), 1/3 of the burst duration, and 5/57 size-related features. Note that this excludes almost all of the size-related features and the packet count features. This suggests that updates mainly cause changes in the number of packets and their sizes and not so much in the timings. The fact that for some IPS and IAT features the significance levels increase over time could be due to the inherent more random nature of these features. Especially timings are influenced easily by the load on the devices, both on the client-side and the server-side. This is reflected in the fact that we see features from all three types of bursts in the set of 28 IAT features. IPS is more directly related to the functionality of the app since data is usually sent in the same order. The load on the network, however, can influence the IPS as well since packets can arrive out of order.

The above analysis is over all the features, but the RF classifier does not give the same weight to all the features. So the fact that distributions change over time does not necessarily explain the drop in performance. Therefore we will take a closer look at the features which are deemed important by the RF classifier. The classifier assigns an importance value to each feature in such a way that the sum of them is 1. There are 171 features in the basic feature set, so this means that a feature with average importance will have a feature importance of $\frac{1}{171} = 0.0058$. 80 features have an importance of 0.0058 or higher.

On average the significance level from the Anderson-Darling test is lower for these important features than the average over all the features as can be seen in Table 5.17. This means that for the important features the feature distribution drifts more from the training data than for the less important features. This is not ideal since the classifier will rely heavier on the important features. However, when we calculate the slope of the regression line over these average significance levels we get a slope of -0.024 for all the features and -0.011 for the important features. So while the distribution drifts more when considering the important features, they are less sensitive to new app versions. It is possible to choose the features the classifier should use based on the drift in the data, however, it is not guaranteed that this holds for new data as well, so we risk overfitting the data if we do that.

Feature	Slope	Importance
com_iat_sem	-0.008147	0.005979
com_iat_30_percentile	-0.005605	0.006131
in_ips_50_percentile	-0.012363	0.006240
in_ips_30_percentile	-0.014031	0.007351
in_burst_duration	-0.005015	0.007705
com_iat_10_percentile	-0.001302	0.007972
in_iat_min	-0.034886	0.008563
out_ips_sem	-0.023309	0.008679
in_ips_kurtosis	-0.046167	0.009346
out_iat_var	0.000519	0.009906

Table 5.18: Random sample of 10 features from the set of features that have a higher than average feature importance. The slope of the Anderson-Darling test over all the datasets for this feature and the *com.rovio.ABstellapop* app shows how the feature distribution drifts from the training data over time. A high drift for a feature with high importance makes it difficult for the classifier to correctly classify the new data.

In Table 5.18 a random sample of 10 features from these 80 are shown, to give an idea what these look like. For almost all of these features the slope is negative. However in fact, for 61 of the 80 features with above-average importance the slope is negative, which is comparable to the overall dataset. These features do have a lower slope on average than the less important features. The average slope of is -0.0242, and the sum of all the relative slopes ($slope \cdot importance$) is -0.0240. Therefore, we can conclude that on average the features drift from the training data, and this explains the drop in performance over time for this class.

5.4.5. Generalizing this analysis

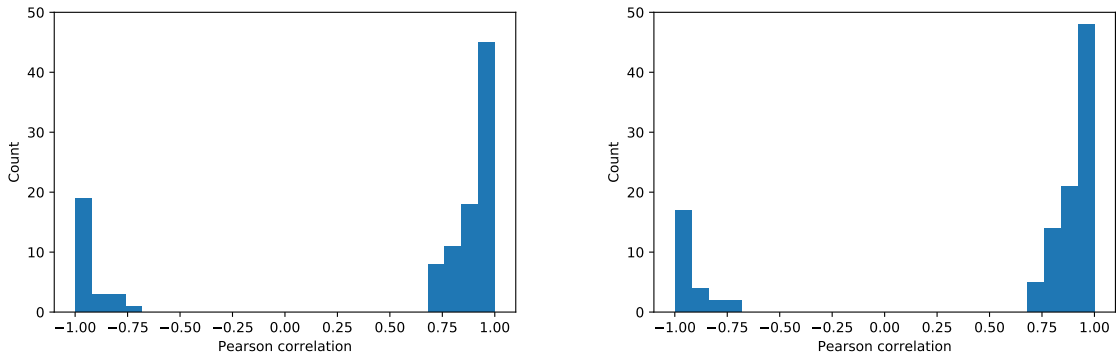
From the example of the class *com.rovio.ABstellapop* we have seen that the average slope of the significance level from the Anderson Darling test over the features is negative just like the performance for this class drops. To truly say that this drift in the feature distribution is the cause of the performance drop we will do the same analysis on all of the apps and correlate it with the slope of the trend line of the F1-scores from all apps over the datasets.

For each app, the feature distributions of the testing data is compared to the feature distributions of the training data using the Anderson Darling test. From this we calculate the average significance level for each of the testing datasets, resulting in eight average significance levels for each app. By taking the average of these significance levels we see how much the feature distributions drift on average over time. These averages can be found in Table 5.19. We clearly see that the feature distributions differ more from the training set as time progresses. These numbers are also strongly correlated with the F1-scores from the version experiment found in Table 5.2. We calculate the Pearson correlation of the F1-scores and the average significance levels of all features and over the important features. We obtain a strong correlation of 0.849 (p=0.008) and 0.889 (p=0.003) respectively. This shows that in general, the drift in feature distributions is strongly correlated to the performance of the classifier as well.

Dataset	Average over all features	Average over important features
21 January (set 2)	0.291	0.279
4 February	0.325	0.309
15 February	0.285	0.272
4 March	0.296	0.278
18 March	0.279	0.262
1 April	0.277	0.258
18 April	0.262	0.244
29 April	0.246	0.229

Table 5.19: Average significance levels for each of the datasets over all the apps. The significance level reports the chance that the features in the 21 January dataset and the subsequent datasets are drawn from the same distribution. Important features are the features that have a higher than average importance value. The reported significance levels correlate strongly with the F1-scores obtained from the version experiment on the basic features set as shown in Table 5.2.

For each app we also have the F1-score per dataset, so eight F1-scores for each app. The correlation between these F1-scores and the individual significance levels for the apps can be calculated using Pearson correlation. This says something about how much the drift in feature distributions is correlated to the F1-scores of each app. For many apps, either the F1-scores or the significance levels are the same for all the datasets. This happens either because those apps only occur in the 21 January dataset, leading to the same feature distributions, or because they are always wrongly classified, leading to an F1-score of 0. Furthermore, some apps have so little samples that the Anderson Darling test cannot be performed. For these apps the correlation cannot be calculated, this leaves us with 231 apps.



(a) All features

(b) Most important features

Figure 5.9: Histogram of the Pearson correlation between the F1-scores and average significance levels from comparing feature distributions. Figure (a) includes all the features, figure (b) only the features that are more important than average. The data includes the pearson correlation calculated for 231 apps. When feature distributions drift and the F1-scores are lower over time, we get a strong (positive) correlation. Only significant correlations (p<0.05) are displayed.

In Figure 5.9a the correlation results for the 231 apps are shown in a histogram. Only the significant correlations (p<0.05) are displayed, a total of 108 apps. From this figure, we can see that there tends to be a strong positive correlation between the F1-scores and the drift in feature distribution. This means that as F1-scores decrease, the significance levels also decrease, which in turn means that fingerprints drift more from the dataset. We also see a few apps which have a strong negative correlation, which might not be expected. This has a couple of reasons. For some apps, we see that the fingerprint becomes more like the training data over time. This can happen due to a rollback of some feature, or simply because of chance, where during collection the same actions were performed due to the random UI input. Another reason is that the fingerprint overall does not drift, but specific features do drift as the F1-score decreases. The classifier is not trained on the fingerprint as a whole, but on features with the highest discriminatory power. Therefore we can expect to see a slightly different correlation when only taking into account the features deemed important by the classifier.

First, we select all the features that have a higher feature importance than average for the trained classifier. Figure 5.9b shows the histogram of the Pearson correlations with the Anderson Darling test only performed on these features. We can see that there is a small shift in the histogram, there is a positive correlation for more apps. This is exactly as expected, only the effect is still rather small.

Like we have shown for the *com.rovio.ABstellapop* app, the drift in feature distribution directly negatively affects the performance of the classifier. For some apps a negative correlation is reported, these are deemed anomalies. Overall the drift in feature distributions is convincingly positively correlated with the performance of the classifier. The drift in the distributions of certain features, deemed important by the classifier, has a higher influence on the degradation of the performance. This is something that should be taken into account when training a model. With thorough analysis, it is possible to select a set of features which has the ideal combination of being discriminatory between classes, but whose feature distributions drift very little with app updates.

5.5. Conclusion

The performance of a classifier that is trained on a set of app versions drops when classifying data that comes from later app versions. We have shown that the accuracy drops about 7 percentage points when comparing data from app versions at T_0 to data from app versions at T+3 months. Choosing the correct features can make a big difference, by using domain names instead of IP addresses we can boost performance by 3.5%. By selecting source ports as a feature we can seriously harm performance. Selecting the correct features requires human insight into what features do and when they do or do not work in favor of the classifier performance.

By analyzing the importance the classifier gives to features and the distribution of features we give insight into the decision process of the classifier. We show that it is possible to say something about the performance an app will likely have purely based on the data, using visualizations of feature distributions or by comparing

feature distributions using the Anderson Darling test. Furthermore, we show that by selecting the correct classes the performance of the classifier can be greatly boosted. This shows that feasibility can differ greatly between adversarial models and requires a critical approach to good results on small or non-random subsets of apps.

Using specific examples we show that it is possible to analyze apps on a very low level to explain in detail what causes changes in performance over time. By generalizing the examples we show that there is a very strong link between the drift in feature distributions and the performance of the classes. These types of analyses can give a very deep understanding of why the classifier works for some types of apps and will not work for others. This can be used to make a selection of types of apps for which classification can be done robustly and gives insight into which apps will be very hard to classify.

6

Real world data

To investigate the threat of network traffic analysis on the mobile platform, we need to look into real-world feasibility. The previous chapter analyzed the feasibility of classifying network traffic generated in a controlled environment. This chapter presents the results of experiments performed to analyze the feasibility of real-world network data. This data was collected from the phones of people who volunteered or were paid to participate in the experiment. A detailed explanation of the setup of the experiment can be found in section 4.5.

The experiments focus on understanding why traffic analysis will or will not work in a real-world environment and whether the conclusions from the controlled experiments extend to the real world. We investigate the classification capabilities in three different settings. First, cross-validation on the dataset itself to investigate whether the network traffic within a class is alike. Second, training on one part of the data and classifying on a newer part of the dataset to investigate the feasibility of training a classifier in one environment and using it to classify data in that same environment. Third, training on emulator generated data and testing on the real-world data to investigate if simple generated training data can be used to train a robust classifier.

6.1. Participants

Data was collected using an online crowdsourcing platform and by finding volunteers among students at the TU Delft. The data gathered from the volunteers were mainly used to test the collection pipeline and also contains testing data from the author's phone. After this testing period, we started collecting data through crowdsourcing as well. We will only use the data gathered after the testing period because it is of higher quality. This data is mainly gathered through the crowdsourcing platform. After the evaluation of several platforms, MicroWorkers[1] was chosen as it is easily usable and quite active.

A task was posted on the crowdsourcing platform with short instructions on how to participate in the experiment. The task linked to a document with more elaborate instructions, this document can be found in Appendix A. Workers were asked to install the app, run it in the background for one hour and submit a short survey of five simple questions. Workers proved that they had done their work by submitting a screenshot of the app with their unique token visible. Workers were paid $0.70 for their efforts. This was deemed a fair price as the time workers spend on the task consists of installing the app and submitting the survey and screenshot. It was estimated that this would take workers three to five minutes, leading to an hourly wage of $8.40 to $14.00. This is above minimum wage for a vast majority of the world, and a majority of western countries as well [42]. A survey by Hara et al. [26], analyzing workers' earnings on MTurk, Amazon's crowdsourcing platform, found that the average requester pays more than $11/hour, our rewards fall into this ballpark. Rewards were the same regardless of the worker's location. We mainly target the regions of Europe, North America, and India.

SdkVersion	Common name	Occurrences
23	Android 6.0	14
24	Android 7.0	9
25	Android 7.1	7
27	Android 8.1	14
28	Android 9	38

Table 6.1: List of Android versions from the participating devices in the dataset. Submissions with a unique token and Worker id were deemed to represent a unique participating device for a total of 98. For 16 of these devices the Android version could not be determined.

[1] `https://microworkers.com`

Data was collected between 18 July and 10 September. 65 unique workers completed 74 tasks. In addition to these workers, there were also a total of 36 unclaimed submissions, with mostly very short sessions. This is probably due to workers abandoning the task, forgetting to submit it, or from people who found the app in some other way, perhaps recruited students. These combined submissions are good for more than 190 hours of data and more than 600MB of TLS data. After filtering out ads, the data contains 295 unique classes and 25 853 labeled bursts, 50 classes have 100 or more bursts. Each submission with a unique worker id and a unique token is deemed to originate from a unique device. Using this method 98 different phones generated data for this dataset, with 82 unique device models. These phones ran 5 different Android versions, the distribution of which can be found in Table 6.1. The dataset contains quite some data from web browsers; about one-third of the bursts. This is to be expected and from the results of the survey question about phone usage in Table 6.2, no bias towards web browsers or apps can be identified. Participants say they use slightly more apps than web browsers and this is reflected in the dataset. As discussed in chapter 3 web traffic is easier to classify than app traffic. However, in a real-world setting, there is no easy way of separating the web traffic from the app traffic. Therefore we keep the web browser classes in the dataset as well.

Education	
None completed	0
Elementary	1
High school	13
Associate degree	26
Bachelor degree	0
Graduate degree	13

Age	
18-24	29
25-34	22
35-44	11
45-54	3
55-64	0
65+	0

Location	
Europe	38
India	8
USA	13
Other	6

Gender	
Male	50
Female	13
Prefer not to say	2

Phone usage	
Mostly apps	8
More apps	10
Both about the same	36
More web browser	7
Mostly web browser	4

Table 6.2: Overview of the characteristics of the crowdsourced workers and answers to the question from the survey about whether workers tend to use the web browser on their phone or apps. This is relevant, because web browser traffic is mainly out of scope for this research, so we want to be sure we do not have a lot of web browser traffic.

The participants in the experiment cannot be called a representative group of society. The participants are all people who do crowdsourced work, therefore we have a bias in the dataset towards apps that enable this and other types of micropayment work. When people are running the app in the background they continue doing crowdsourced work on their phone, as can be seen in Table C.2, which shows the 20 most common apps in the dataset. There are various apps among the most common that enable micropayment work and one app with one review, which probably had to be installed as part of a crowdsourced job. However, this is not a big problem, as we mainly want to see if we can classify apps in the real world as well. We do not care that much about whether most apps are e.g. from the category Social, Education, or Communication, as we have not seen that these categories of apps can be classified better than others. There is also a bias toward men and young people, but again, for this preliminary study, we do not mind these kinds of biases. However, it is something to keep in mind before extrapolating the results to a general model. An overview of the crowdsourced workers' gender, age, education level, and location can be found in Table 6.2.

6.2. Cross validation on dataset

Data collected from the real world is expected to be more diverse than the data trained on emulators. The data generated by the emulators are based on a script that does the same thing for each trace. Even though UI fuzzing adds a random factor to the scripts, the apps are launched from the same point every time. In the real world setting, apps contain much more data and are in all kinds of different states, leading to a more diverse network fingerprint. To test if bursts are still sufficiently similar to create a proper fingerprint for each app, we do cross-validation on the whole dataset, the results of which can be found in Table 6.3.

The performance for cross-validation is very similar to the cross-validation on the emulator-generated data in chapter 5. This suggests that the feature distributions for the classes are not that much more diverse than for the emulator data. It also points to the good possibility of training a model on real-world data and being able to classify real-world data.

The results with extra features destination port and domain name are much better than the results without extra features. One reason for this is that there are a couple of unique destination ports associated with apps. More apps in the real-world dataset have this characteristic than in the closed-world dataset. An overview of all non-standard ports and the associated apps can be found in Table 6.4. Another reason is that that domain names are a good extra indicator for which app is used.

Dataset	Average # classes	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
crowdsourced basic	290	0.698	0.500	0.371	0.407
crowdsourced extra	290	0.764	0.548	0.438	0.467
crowdsourced basic without browser	282	0.690	0.502	0.393	0.421
crowdsourced extra without browser	282	0.776	0.560	0.472	0.492

Table 6.3: 5-fold cross validation on the dataset using the basic feature set and the extra feature set with destination port and domain name as extra features. Because browser data might be ambiguous, the experiment is performed including and excluding browser data, to see the effect on the performance.

We also test whether the inclusion of browser data has a big influence on the performance of the classifier. Browser data accounts for about one-third of the data and can be excluded by manually creating a list of web browser apps and filtering on those classes. As it turns out, excluding the browser data does not have a big positive influence on the performance of the classifier; about 2 percentage points. Because it is also not that easy to separate browser data from non-browser data in an unlabeled dataset, we choose to include the browser data in all our further experiments.

AppId	Destination port
com.android.chrome	53, 8888
com.samsung.vvm	993
com.android.email	993
com.samsung.android.email.provider	993
com.lilithgame.hgame.gp	1443
com.google.android.ims	5061
deezer.android.app	5222
kik.android	5223
com.sec.spp.push	5223
com.lbe.parallel.intl.arm64	5228
com.huawei.wallet	6447
com.evenwell.AprUploadService	8790
com.att.iqi	10010
org.hola	22222, 22223
com.sprint.ecid	44305

Table 6.4: List of ports other than 443 and their associated apps from the real world dataset. This shows that a unique destination port as feature is a very good way to distinguish between apps.

6.3. Training on first part, testing on second part

For real world applications it might not be necessary to aim for a very generic model. In one example attack from the introduction in chapter 1, a company wants to monitor its network for app usage and is only interested in a model that works on their network. Therefore it could work to first collect network data on their network for a certain amount of time and label this network traffic by hand. Afterward, the classifier trained on this data can be used to classify new network traffic. This approach works because we can expect there to not be many changes in the network traffic since every day the same people come to work at the company. Therefore, this method can be used as a sort of anomaly detection, where the company can monitor changes in app usage. One use of this is to check the effectiveness of policies implemented to limit the usage of social media apps during work time.

6.3.1. Splitting the data

To test whether training and testing on the same network would work, we split the collected data on a chosen date. All data before the split is used for training the classifier, all data after the split is used to test the classifier. The big problem with this approach is that we have to deal with a lot of previously unseen classes. When trying every possible split, the overlap between the training and testing data is at most 65 classes. This means that for a large part of the new data we have no training data. One solution for this is to increase the training window since more data will also mean more classes are contained in the training set. As we increase the window of the training data we start to include a larger percentage of the classes in the testing set, as can be seen in Figure 6.1 However, because our total collection time is limited, the amount of classes in the testing set also drops. For our testing purposes, we are not looking to include the highest percentage of classes but a reasonable amount of apps and a logical split in the data.

Another reason why we have little overlap is that our dataset is very diverse; the data came from a lot of different users. In e.g. a company network there will be more of the same users, so there will be more overlap between the training and testing set. Since users use the same apps during the testing phase as they used during the training phase.

We choose to split the data on 22 August, this gives us a reasonable amount of apps in both the training and testing set. It is also a logical moment where there is no data collected for a few days, as can be seen from the flat line in Figure 6.1. This splits the data in a 60/40 ratio for the number of samples. 53 classes exist in

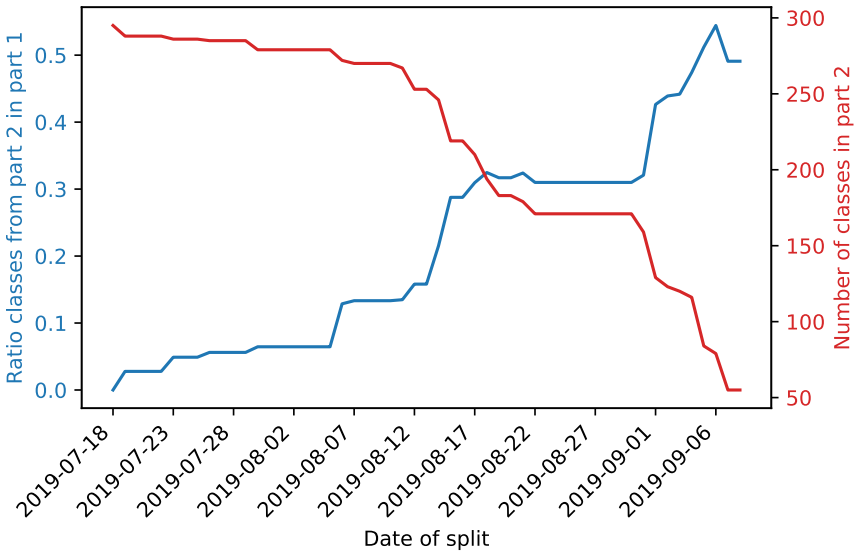


Figure 6.1: Plot showing how choosing a different split of the data affects the overlap in classes in the training and testing set and the size of the testing set.

Training set	Validation set	Training # classes	Validation # classes	Accuracy	Macro-avg Precision	Macro-avg Recall	Macro-avg F1-score
Part 1	Part 2	177	171	0.434	0.066	0.053	0.050
Part 1 extra	Part 2 extra	177	171	0.441	0.064	0.060	0.053
Part 1 (overlap)	Part 2 (overlap)	53	53	0.640	0.362	0.256	0.261
Part 1 extra (overlap)	Part 2 extra (overlap)	53	53	0.650	0.340	0.269	0.258
Part 1 + dataset E1	Part 2	215	171	0.430	0.071	0.055	0.053
Part 1 + dataset E1 extra	Part 2 extra	215	171	0.432	0.077	0.068	0.059
Part 1 + dataset E2	Part 2	215	171	0.432	0.062	0.049	0.046
Part 1 + dataset E2 extra	Part 2 extra	215	171	0.435	0.069	0.065	0.056

Table 6.5: Results from training and testing on different parts of the real world data. The data was split on 22 August, to obtain two sets with a 60/40 ratio. Training and testing was done on both the full subsets as well as only on the apps that occur in both the training and testing set. Afterwards the same tests were run, but now the training set was supplemented with data generated by emulators. First using all the apps (set E1), and later only for the apps that do not occur in the training set (set E2).

both the training set and the testing set. We ran two versions of the experiment, one with only the classes that occur in both the training set and the testing set and one with the complete training set and testing set. Both experiments are run with the basic feature set and the extra feature set. The results can be found in Table 6.5.

When only looking at the accuracy of the first experiment, the results are mediocre. We could correctly label about half of the bursts. The reason for this is that popular apps can be expected to be both in the training set as well as in the testing set, as many people use them. These apps are also apps that have more bursts, simply because they are used more. Therefore the accuracy, which is calculated over all bursts, is relatively high. However, when considering macro average scores the results are bad. When analyzing the results we find that there are no correct labels for 140 out of 171 classes, 118 of these have 0 training samples. Therefore, when taking the average F1-scores, we get a number close to 0. The 31 classes with non zero F1-scores have on average an F1-score of 0.391. All classes that occur in both datasets have on average an F1-score of 0.229. When we compare these F1-scores with the results from apps that occur often (> 100 samples) in both the training set and testing set, we see in Table 6.6 that these apps perform much better than the others. Coincidentally these are very popular (mainly social) apps. The average F1-score of these ten apps is much higher at 0.520. This shows that for a subset of apps the performance can be much better than on all apps in a network.

Adding more training data is a problem that can be solved by increasing the training window, so therefore we want to look at what happens when the training data does contain all the classes that are in the testing data. When only considering the apps that occur in both the training and testing set, the overlap, we see in the third and fourth experiment of Table 6.5 that the macro scores are boosted as well. This is mainly because we now

App id	App name	F1 score	# samples in part 1	# samples in part 2
com.google.android.gm	Gmail	0.705	274	227
com.android.chrome	Google Chrome	0.665	2 483	2 785
com.instagram.android	Instagram	0.647	1 004	386
com.facebook.katana	Facebook	0.611	957	902
org.telegram.messenger	Telegram	0.592	237	149
com.reddit.frontpage	Reddit	0.560	130	205
com.whatsapp	Whatsapp	0.553	554	113
com.android.vending	Google Play Store	0.503	391	410
com.twitter.android	Twitter	0.229	158	118
com.truecaller	Truecaller	0.139	179	129

Table 6.6: Ten of the most common apps in both the training and testing datasets. The F1-scores are the F1-scores from the results of training on the first part of the real world dataset and testing on the second part. The F1-scores of these apps are much higher than the average app in the datasets.

filter out all of the classes that were assigned a wrong class simply because they are only in the testing set. As can be seen when comparing the F1-scores of this experiment to the average F1-scores of these classes from the experiment with the full datasets. This thus shows that a better training set greatly improves performance.

6.3.2. Filtering previously unseen classes

Because we expect the testing set to contain unknown classes, it makes sense to do post-processing to account for this. The classifier gives a confidence level of how confident it is that a certain prediction is correct. We would expect previously unseen classes to have low confidence since they would not match any of the trained fingerprints. Therefore using a threshold we can label those classes as *unknown*. The ground truth for all classes that are in the testing set, but not in the training set, is labeled as *unknown*. A logical threshold would be one that is rather low, say below 0.5, so there is a small chance that we relabel correctly predicted classes.

The perfect threshold is very dependent on the dataset, therefore it does not make much sense to simply choose the best threshold for our dataset. Various threshold settings are plotted in Figure 6.2. The increase in performance is not amazingly great as the threshold increases. Note that the baseline that should be taken into account is the threshold at 0 in the figure and not the results in Table 6.5, because the relabeling essentially removed bad performing classes from the dataset, increasing the macro scores. We observe an increase in precision at a threshold value of 0.7. At this point, a significant number of already known classes that were assigned the wrong label are relabeled as unknown. So because for many classes there are less false positives, and for the *unknown* class there are more false positives, the macro average precision goes up.

Depending on which metric is preferred, various threshold settings can be considered the best. Keeping in mind that we mainly want to filter out the previously unseen classes, that are expected to perform worse, a lower threshold is better. This way we do not filter out many correctly labeled classes. The idea that previously unseen classes have low confidence is also better transferable to another dataset, while the optimal high threshold varies more per dataset. From this example, a threshold between 0.2 and 0.4 seems like a fairly safe choice. This mainly increases the accuracy of the classifier. From threshold 0.4 and up, more than half of the false positives of the *unknown* class were labeled correctly before post-processing. From this threshold relabeling is not effective anymore.

6.3.3. Supplementing training set with emulator data

There are a lot of classes missing from the first part of the dataset. With the emulator traffic collected in section 6.4, we can supplement this training set to get higher performance. This increases the overlap in classes between the training and testing set from 53 apps to 86 apps. However, as seen in Table 6.5, the performance only increases very little, the accuracy even goes down. The emulator data is quite different from the real-world data and this hurts the performance of the classifier. This dissimilarity between the training data and the testing data is also a problem in the experiment of section 6.4.

The emulator data essentially adds noise to the data, however, perhaps if we only supplement the classes that are not already in the training set we can still gain some performance. Noisy training data will work better than no training data. There are 38 classes in the emulator data that are not already in the training data. Surprisingly, from the results in Table 6.5, adding emulator traffic for only the unknown classes has barely any effect. This suggests that the emulator traffic adds so much noise that the classifier wrongly classifies previously correct traffic.

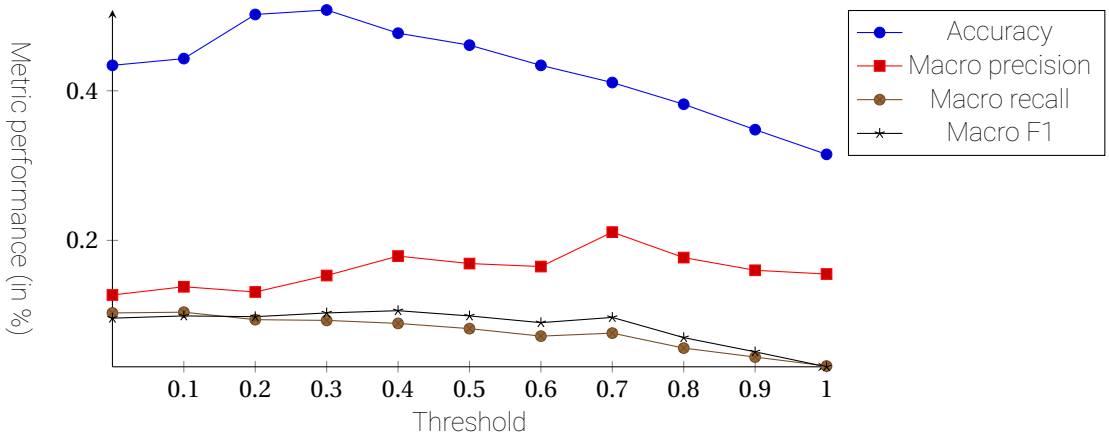


Figure 6.2: Metric scores for various threshold values on the splitted data experiment (basic feature set). When a predicted label has a confidence below the threshold, the label is changed to *'unknown'*. The macro scores at threshold 0 are higher than the reported values in Table 6.5, because all newly introduced classes in the testing set were relabeled.

6.4. Training on emulator data, testing on real-world data

There are a lot of classes with only a small amount of samples as can be seen from the histogram of the amount of samples per class in Figure 6.3. For 134 classes there are less than 10 samples. More samples help the classifier to learn the various types of bursts an app generates. So we would like more samples to be sure the model has sufficient data to learn a fingerprint. An easy way to collect a lot of samples is by using emulators to automatically collect network traffic from apps. In this section, we will investigate the effectiveness of using classifiers that are trained on emulator data to classify real-world data.

We expect a difference in the number of samples per class since not all apps require the same amount of network traffic and some apps are more popular and used more often than others. In emulator trained data, we collect a fixed number of traces, so we know that the difference in samples is simply because the app generates less network traffic. However, for real-world data, a low amount of samples can also be caused by an app not being used enough by the participants. Wherewith the emulators we can easily control the amount of time an app is running, this is not possible for our crowdsourced data.

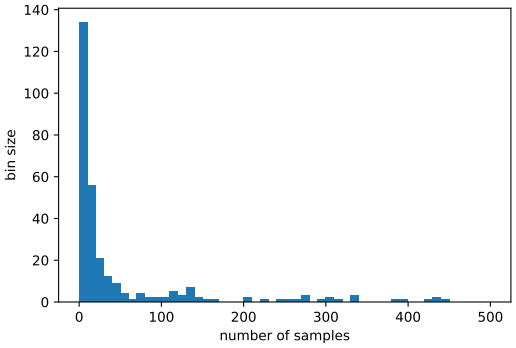


Figure 6.3: Histogram showing the distribution of samples per app in the real world dataset. Seven apps with more than 500 samples are cropped out of this figure.

Besides the low amount of samples, there is a second challenge to overcome when trying to classify real-world data. This is the problem encountered in section 6.3, where there exist classes in the testing data that have never been seen before. In a real-world setting, it would be very hard to generate training data for all possible classes that could be encountered since there exist millions of apps. An attacker would need to target a specific set of apps instead or accept that they will never be able to classify a portion of the data.

Both of these challenges can be solved by controlling the data collection better, for example by giving people specific instructions on which apps to run. However, this is a very labor-intensive task and not feasible for many attackers. If the training process could be automated it would be much easier to scale up the attack to target many apps.

The real-world data contains 309 apps, 20 of these apps also occur in the datasets we have collected using emulators, about half of these apps are installed by default. To get a larger training set the missing apps from the real world dataset were downloaded and new traces were generated on emulators. Of the 309 classes in the dataset, 189 apps could be downloaded from the Google Playstore. Some of the classes belong to system apps that cannot be downloaded or belong to apps from unofficial app stores. Of these 189 apps, 124 are built for

an x86 architecture and can be run by the emulators. For these 124 apps, 10 traces were collected each on a Google Pixel emulator running Android 8.0. It is possible to collect the traces for the ARM build apps as well by using a dedicated phone. Most of the remaining apps could be downloaded by scraping unofficial app stores as well. Therefore we will assume that it is possible to have almost all apps that can be encountered in the real world in a generated training set.

6.4.1. Experiment results

The traces collected from the 124 apps resulted in 98 classes in the training set. These same classes are also selected from the crowdsourced data, so our training and testing set contain the same classes. The experiment was performed using both the basic feature set and the extra feature set. To show what would happen in a realistic setting where our training set would not contain all the apps that are in the testing set, the same experiment was performed using the full testing set as well. The results from these experiments can be found in Table 6.8 and the baseline performance of the various datasets can be found in Table 6.7.

When classifying the complete real-world data set, the performance starts to drop a lot. This is to be expected since the classifier tries to classify everything and it does not know about one-third of the classes. Post-processing based on confidence could help the performance a little bit by labeling classes with low confidence as unknown, the same way as it did for the experiment in section 6.3. This post-processing was done for the experiment on all apps with extra features. We consider all apps that do not occur in the training dataset *unknown*. Then we relabel the predicted labels for various threshold settings, the results can be found in Figure 6.4

For this dataset, there are many more previously unseen classes compared to the splitted data experiment, so the accuracy keeps increasing as more traffic is labeled unknown. Therefore here we should mainly focus on the highest F1-score, which occurs at a threshold of 0.3, which was previously deemed a fair threshold for filtering previously unseen traffic. The macro average F1-score is increased from 0.035 to 0.049 for the threshold at 0 or 0.3 respectively. This is still a very bad score and we can conclude that using the current emulator data we cannot classify real-world data. However, more advanced techniques to automatically generate more realistic data could still be effective.

Dataset	Average # classes	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
emulator	98	0.771	0.752	0.632	0.670
emulator extra	98	0.841	0.773	0.700	0.720
real world	93	0.807	0.509	0.378	0.415
real world extra	93	0.853	0.571	0.459	0.490
real world all apps	290	0.701	0.503	0.379	0.413
real world all apps extra	288	0.762	0.534	0.416	0.450

Table 6.7: Baseline performance of the datasets, to show the quality of the sets used in the experiment where we train on emulator data and validate on real world data. 5-fold cross validation was performed on all the datasets.

Training set	Validation set	Training # classes	Validation # classes	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
emulator	real world	99	94	0.083	0.078	0.075	0.040
emulator extra	real world extra	99	94	0.102	0.128	0.127	0.081
emulator	real world all	99	295	0.033	0.010	0.023	0.006
emulator extra	real world all extra	99	295	0.040	0.014	0.039	0.011

Table 6.8: Results from training on emulator data and testing on real-world data. The testing set can contain only classes from the training set or all classes from the real-world data set. The latter simulates a more realistic setting where a classifier has to deal with previously unseen apps. The validation set can contain fewer apps because other (system) apps are also included in the training data.

6.5. Difference between training on emulator and real-world data

There are several reasons why it is hard to train a model on emulator data and use it successfully in the real world. First of all, the data we collect is always from a freshly installed app and always for the same amount of time. This means that we will miss a lot of states of the apps that occur when the app has been running for a longer time. Secondly, there are apps, especially from the Social category that require to log in, in our scripts

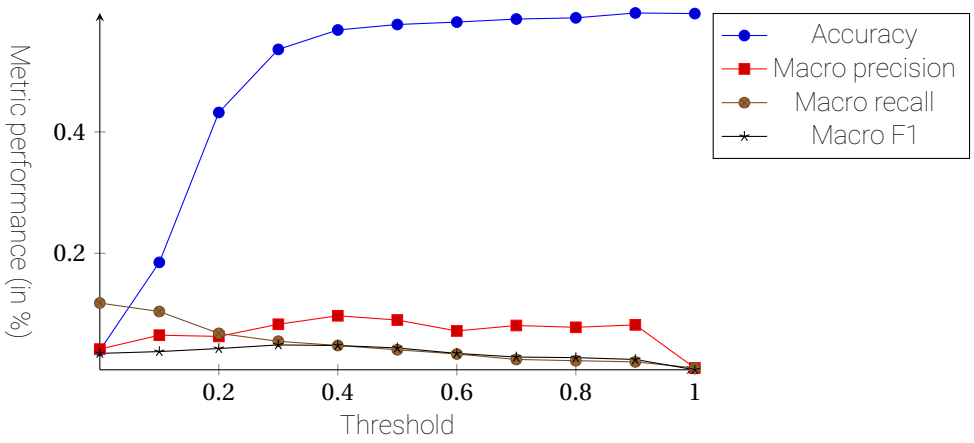


Figure 6.4: Figure showing how various thresholds for the confidence reported by the Random Forest classifier influences the performance of the classifier for the experiment where we train on emulator data and test on real-world data.

we do not take this into account. Therefore it is not possible to build reliable fingerprints for these apps. Even when creating a dummy account and writing a script that logs in, it will still not be the same as a real user. One can imagine that a dummy account will get fewer messages than someone with a lot of contacts. Simulating this kind of behavior could prove to be a large challenge. Thirdly, random user input can never truly simulate the logical steps a user takes while navigating through an app. For this to work better, it is possible to write more advanced scripts that analyze the apps and try to come up with all possible logical flows through the app.

To verify which apps this method does work and for which apps it does not, we calculate the F1-score on an app level. Because apps that do not occur in the training data have an F1-score of 0 for obvious reasons, we analyze the results from the experiment with only a selection of real-world data with extra features. We identify three categories of apps:

- Dynamic User: Apps with different content per user, e.g. social media
- Static: Apps with static content, e.g. public transport timetables
- Dynamic Time: Apps with dynamic content over time, e.g. news apps

Static content does not mean that the content stays the same, but that that the format stays the same. That is why timetables would be considered static content, even though the exact content might change over time, it still contains the same size content. For news apps, the length of articles changes and articles contain different photos and videos, this changes the network traffic significantly over time, while the content is the same for all users. Apps that require a user account often show different content based on the interest of the user, the progress of the user, or based on their social network. Therefore different network traffic is generated per user.

Based on the definitions of these categories we can hypothesize which method to collect training data is most effective for each category of apps. For apps with different content per user, real-world data collection is probably the only effective method. These are apps that serve content based on details of the user account or rely heavily on input from users, e.g. online shopping apps. Emulators do not have established user accounts, and it would require significant effort to create a diverse set of accounts for each app. With real-world data collected from enough people, this is not a problem.

For apps with static content, emulator training data will work well, since the same content is available to the emulator as it is to real users. Slight differences in data could still be caused because the random fuzzing might not always choose the most logical path through the app. This can be fixed with a more advanced simulation of user input.

For apps with dynamic content over time, emulator traffic could still work very well, because the content is static for all users at one point in time. However, for this category, the model should be updated more often, or the model should be trained on a very diverse set of data, so it learns the patterns that are invariant to the dynamic content. This category can also suffer from the same problem with logical flows through the application as the previous category.

Apps do not necessarily belong to one specific category but can have something from multiple categories. Assigning apps into specific categories can give insight into why certain training methods work better for some

apps than they work for others. This is what we are trying to prove by analyzing the results per category. We are training on emulator data and testing on real-world data, therefore we would expect mainly the Static Content category to work well.

To see we can categorize certain apps into one of these three categories we check the difference in performance when using emulator trained data or real-world training data. For this, we look for apps that occur both in the emulator data and in both the first and second parts of the real-world data from section 6.3. There are 13 apps in the intersection of these three datasets. For all of these apps, we build two models, one trained on the emulator data and one on the first part of the real-world data. We use the second part of the real-world data as a testing set. This gives insight into how a different training method can be more effective for certain apps.

In Table 6.9 the results from this experiment can be found. Even though we only have 13 apps and not all of them have a lot of samples, these apps can be still be used as examples. For some apps the F1-score for both training sets was 0, those apps are omitted from the data.

AppId	F1-score emulator	F1-score real world	# samples in emulator data	# samples in part 1	# samples in part 2
com.android.chrome	0.049	0.927	42	2 483	2 785
com.android.vending	0.434	0.779	4 107	391	410
com.ebay.mobile	0.006	0.091	577	192	8
com.google.android.gm	0.000	0.757	20	274	227
com.google.android.googlequicksearchbox	0.037	0.160	1 046	51	20
com.google.android.youtube	0.156	0.187	774	86	187
com.microsoft.office.outlook	0.000	0.842	268	14	11
in.sweatco.app	0.400	0.000	57	13	4

Table 6.9: Difference in performance as measured by F1-score for training on emulator data or the first part of the real world data and testing on the second part of the real world data. Only the 13 classes that occur in all datasets were used for training and testing. Apps that have an F1-score of 0 for both training sets are omitted.

We are left with 8 apps, which can be used for the validation of our hypothesis. *Google Chrome*, *Google Play Store*, *eBay*, *Gmail*, *Google app*, and *Outlook* perform significantly better when training on real-world data in comparison to emulator-generated data. For the mail clients *Gmail* and *Outlook* this is easily explained: our emulator does not have an e-mail account and thus cannot send and receive mail. We can safely assume that this is exactly the kind of behavior that is contained in the real-world traffic of these apps. *Google Chrome* and the *Google app* are apps that require smart user input, like typing a web address. The random fuzzing in our emulator cannot do this, as it clicks on things randomly. It can still generate some traffic from searching random strings or clicking suggested content. *Google Chrome* is also pre-installed on the emulator, so clicks on links can also generate traffic for *Google Chrome*. Therefore we do see some bursts in the emulator traffic, but not enough and not human-like enough to classify the real-world data well. These four apps are typical examples of the category *Dynamic User*.

For *eBay* we would expect the emulator trained data to still work quite well since it is possible to browser *eBay* without having a user account. However, we can see that the performance is still very poor. Perhaps the personalized content has a bigger influence than previously thought. The training data is collected from three unique users and the testing data is of 1 new unique user, so this can very well be the case. However, note that there are very few samples in the testing data, which could also influence the results. So we cannot draw this conclusion with a high degree of confidence.

The *Google Play Store* is an app that is installed by default on Android phones and it does more than just allow people to browse available apps. When collecting data on the emulators, often traffic from *com.android.vending* is generated as well. So, likely the *Google Play Store* handles other tasks used by many apps as well, but it is hard to figure out exactly what it does without analyzing decrypted network traffic. The *Google Play Store* app does not handle app updates, for this *Google Play Services* is used. This type of traffic is generated on user's phones as well, so therefore the performance is still not too bad. The emulator data does miss data that comes from browsing the Play Store, which explains the better performance for the real-world data.

in.sweatco.app is a fitness app that rewards people for the exercise they do. On the first startup, the app requires user input and SMS verification. During startup, the app also immediately starts sending and receiving network traffic. This is the traffic which is also contained in the emulator training data. This traffic is independent of the user, therefore, even though we would expect this app to fall into the category *Dynamic*

User, the start-up traffic falls into the category *Static* and we can still classify some network traffic. Note that for the basic feature set the results are the same for this app, so this is not caused by the domain name having a big influence. A likely reason why the real-world training data does not work is that there is no start-up traffic in that training set.

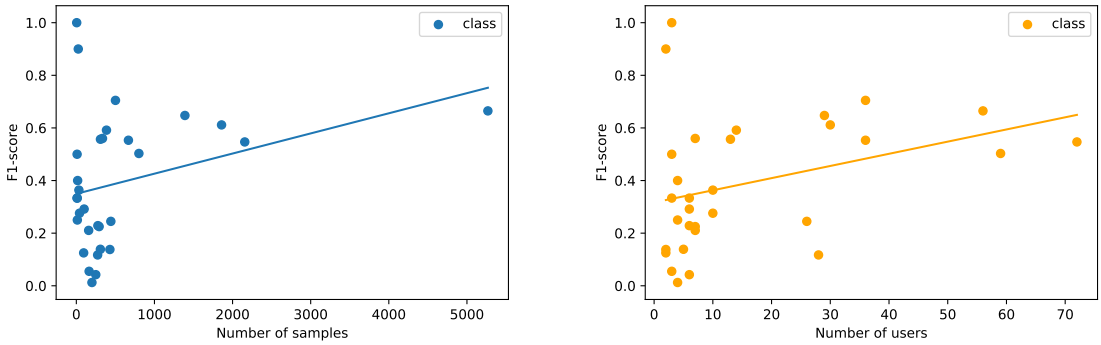
YouTube has a similar performance for both training sets. This is to be expected since streaming videos is a fairly consistent fingerprint regardless of the kind of video's someone watches. Without a user account, we are missing traffic like notifications or posting comments, but this is only a minor part of *YouTube's* traffic. This app falls into the *Static* category, because the content of the traffic does change per video, but the shape of the traffic does not change that much. The app also has some parts which fall into the *Dynamic User* category.

6.6. Influence of variety of users in training data

For apps in the category *dynamic user* we expect more differences between the feature distributions per user than for apps in the category *static* or *dynamic time*. To check whether this is true, we select a few of the most popular apps in our dataset, which have a high number of users and compare the feature distributions of the individual users. We also check the F1-scores for these apps from the splitted data experiment. That experiment trains and tests on real-world data from different users. Therefore, we expect a high F1-score for apps with little differences between the feature distributions per user and a low F1-score for apps with big differences between the feature distributions of users.

Each log with mappings between ports and App Ids has a unique token. The token is generated when our traffic log app is installed and stays the same unless the app is re-installed or all app data is cleared. Using the mappings and the tokens we can see which traffic belongs to the same unique install of our traffic log app. In theory, each unique install represents a unique user. In practice, there are a few workers who submitted a task multiple times by re-installing the app and obtaining a new token. For each submitted task we have a mapping from the token to the worker ID. We can thus filter these duplicate users by selecting unique worker IDs instead of unique tokens. For the data submitted outside of the crowdsourcing platforms, we assume that each unique token represents a unique user, since those users had no incentive to re-install the app. Our analysis thus only compares the feature distributions of unique users.

To make sure apps with a high number of users do not introduce a bias in our results, we check the correlation between the number of users and the F1-scores. The same is done for the number of samples. From the plots in Figure 6.5 we see that there is no convincing high correlation between the F1-score and either the number of samples or users. Therefore we do not expect these numbers to have a significant influence on this analysis.



(a) F1-score vs. number of samples

(b) F1-score vs. number of users

Figure 6.5: Scatter plots with best fit line showing the correlation between the F1-scores from the experiment in section 6.3 and the number of samples and users (as measured by unique tokens) per app. There are only 31 apps with an F1-score higher than 0, to prevent a bias only those are plotted.

To compare the distributions we use the k-sample Anderson Darling test. This tests the hypothesis that k distributions are drawn from the same population. The Anderson Darling tests produce a significance level. A significance value of 1 means that distributions are very likely drawn from the same population, a significance level of 0 means that they are not. For each set of k unique users we perform the AD test on all 171 features in the basic feature set. The 171 results of the AD test are averaged and this gives a measure of the likelihood

that the feature distributions of these k users are drawn from the same population. We expect that the AD test on apps that we consider *Dynamic User* produces a low significance level, while the AD test on apps that we consider *Static* produces a high significance value.

The AD test evaluates the hypothesis that distributions are drawn from the same population. When comparing a lot of distributions, the chance is higher that there are outliers in one or more of the distributions. This means that the chance is also higher that the Anderson Darling test reports a low significance level and thus rejects the hypothesis. If we perform the Anderson Darling test on all the unique users, we would thus get a biased result where the significance level would always be much lower when performing the test on more distributions. To avoid this bias we draw a random set of 5 users from the set of unique users and perform the AD test on those. The distributions for two random sets of users can differ quite a lot and therefore the significance level reported can also be vastly different. Therefore we select a random sample 1 000 times, perform the AD test and average the resulting significance levels. This gives us an expected value that the distributions for all unique users are drawn from the same population and allows us to do a fair comparison between apps.

The results of this test can be found in Table 6.10. The test was performed on the 24 apps with the most users. The table shows their number of unique users and number of samples in the training (part 1) and testing (part 2) sets, to get a feel for how much data training and testing data there is. The reported F1-score is the F1-score of the app in the splitted data experiment. The similarity between users is the average significance level from the Anderson Darling test on the feature distributions for users and says how different the feature distributions between users are. The similarity to the closest neighbor is the maximum significance level from the Anderson Darling test when comparing feature distribution of the app to all other apps and says how close the app with the most similar feature distribution is, this is used later to explain the results.

AppId	**Unique users**	**Similarity between users**	**Similarity to closest neighbor**	**# samples in part 1**	**# samples in part 2**	**F1-score**
app.uid.shared	60	0.0149	0.7653	1 135	1 021	0.547
com.android.chrome	46	0.0016	0.7351	2 483	2 785	0.665
com.android.vending	49	0.0002	0.8777	391	410	0.503
com.facebook.katana	26	0.0072	0.7307	957	902	0.611
com.facebook.orca	20	0.1504	0.6317	358	84	0.245
com.facebook.services	8	0.6022	1.0000	15	18	0.364
com.google.android.apps.docs	12	0.2491	0.7244	42	11	0.000
com.google.android.apps.maps	9	0.5001	0.7290	9	17	0.000
com.google.android.gm	32	0.0481	0.6922	274	227	0.705
com.google.android .googlequicksearchbox	14	0.5644	0.7667	51	20	0.000
com.google.android.music	6	0.7110	1.0000	4	5	0.333
com.google.android.videos	7	0.8480	1.0000	3	7	0.000
com.google.android.youtube	28	0.1168	0.5396	86	187	0.117
com.instagram.android	17	0.0072	0.7233	1 004	386	0.647
com.linkedin.android	7	0.0356	0.7787	67	91	0.211
com.netflix.mediaclient	4	0.5117	0.8696	62	38	0.292
com.reddit.frontpage	7	0.0345	0.7890	130	205	0.560
com.samsung.android.scloud	7	0.0911	0.8787	245	47	0.225
com.sec.android.app.sbrowser	6	0.0801	0.8553	224	25	0.042
com.sec.spp.push	10	0.7777	0.9860	32	9	0.276
com.snapchat.android	7	0.1902	0.6970	279	30	0.557
com.twitter.android	6	0.0000	0.8787	158	118	0.229
com.whatsapp	32	0.2626	0.7630	554	113	0.553
org.telegram.messenger	12	0.0388	0.7134	237	149	0.592

Table 6.10: Overview of comparisons between the feature distributions of the unique users for various apps as reported by the k-sample Anderson Darling test. Since the number of distributions influences the results of the AD test, a random sample of 5 users was drawn from all unique users per app. This was done a 1 000 times, to obtain an average significance value for all users. Also shown are the F1-scores from the test on the splitted data from section 6.3 (basic feature set).

There are a lot of apps in the table, so let us first take a look at the apps with the highest and lowest significance levels comparing users. These are the apps for which the feature distributions of the users are most and least alike respectively.

The five apps with the lowest significance levels with many differences in the feature distributions between users are:

- com.twitter.android - Twitter
- com.android.vending - Google Play Store
- com.android.chrome - Google Chrome
- com.instagram.android - Instagram
- com.facebook.katana - Facebook

The five apps with the highest significance levels and the most consistent feature distributions over users are:

- com.google.android.music - Google Play Music
- com.google.android.videos - Google Play Movies & TV
- com.sec.spp.push - Samsung Push Service
- com.facebook.services - some utility app used by Facebook
- com.google.android.googlequicksearchbox - Google

When we look at these apps, we can see that the apps with the lowest significant levels have something in common. They are apps that require complex user input or require an established account. For Chrome, we would expect diverse feature distributions, since users visit many different sites. Instagram, Twitter, and Facebook are typical apps for which the content differs quite a lot per user since they have different followers or friends. For the Play Store, as previously explained, it is not exactly clear what it does precisely. However, it does provide functionality used by many apps since traffic from *com.android.vending* appears when running apps on the emulators as well. So whatever it does exactly, it is influenced by the apps a user is running on their phone. This explains why the feature distributions differ per user.

The apps with a high significance level are all apps with a rather low amount of samples. Previously we said that the amount of samples is not correlated with the F1-score and thus does not matter for our experiment. However, here the low amount of samples tell us something. It tells us that these apps were not actively being used by users. Likely these are apps that were performing background tasks. For Google Music and Google Videos, we would expect a lot more traffic if they were used actively since their primary function is streaming.

Samsung Push Service, the notification service for Samsung service, is an app that does not perform actions based on user input. Occasional communication about notifications to a server would be fairly consistent across users. Therefore even the bursts for different notifications are very much alike. This becomes clear when taking a closer looking at the bursts from three users of this app in Table 6.11. These users use different phones, but the same Android version. The table shows very clear similarities across users even though the bursts are not the same between users. The second burst, with only incoming data, always occurs 1 minute after the first burst. For user C both groups of three bursts occur at the same time. Surprisingly the outgoing traffic differs in size more than the incoming traffic, while we would expect mainly the notifications (incoming) to differ in size. So perhaps the app does not only communicate notifications, but there is no way of knowing without inspecting decrypted network traffic. Either way, the point about consistent communication still stands and this example explains how a traffic fingerprint can be very consistent across users.

It is not exactly clear what *com.facebook.services* does, it is a pre-installed app on many phones and it performs some background tasks. By looking at the bursts themselves, a similar pattern as for the *Samsung Push Service*. For the *Google* app we would expect, a large difference between users, since according to the Play Store the main functionality is: "Search and browse" and "Personalised feed and notifications". However, we see that the significance level comparing the feature distributions of users is still rather high. When looking at the bursts we see bursts which consist of a lot of traffic, which would indicate search results or some other activity started by users. But we also see, for all users, bursts which consist of 1 incoming packet of 56 bytes. And when we inspect the domain names (which are not used for training and testing), we find domain names like *chromesyncpasswords-pa.googleapis.com* and *connectivitycheck.gstatic.com* which are evidence of syncing behavior. So this app generates background traffic and user-specific traffic. The background traffic is something that can easily be detected by a Random Forests classifier. There will exist a path in the tree for those specific bursts since they occur so consistently in the training data. This is data that we would consider *static*. The user-specific traffic will be much harder to detect.

User	Incoming # packets	Size of incoming data	Outgoing # packets	Size of outgoing data
A	5	2 031	4	519
A	2	111	0	0
B	5	2 036	4	524
B	2	111	0	0
C	4	2 231	4	2 718
C	1	31	0	0
C	4	2 231	3	1 017
C	4	2 231	3	886
C	4	2 231	4	2 062
C	1	31	0	0
C	1	31	0	0
C	1	31	0	0

Table 6.11: Metadata from bursts in network traffic originating from the *Samsung Push Service* app, for three different users. This shows that the bursts are very much alike for different users.

6.6.1. Correlation similar users and F1-score

We expect that the classifier performs well for apps that serve the same content for all users because their bursts will also be the same when comparing the training and testing data. Therefore a high average significance level comparing users should mean a high F1-score. In other words, we would expect a positive correlation between the significance level and the F1-score in Table 6.10. However, when we calculate the Pearson correlation between these two lists we get a negative correlation of -0.491 (p=0.017). This is surprising, but the attentive reader might have noticed before that something unexpected is happening in the table. We observe multiple examples of apps with a higher significance level for users, but a low F1-score. And also the other way around, a low significance level and a high F1-score.

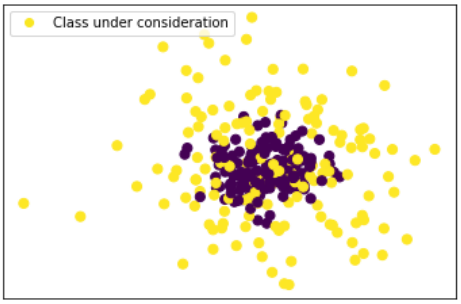
The main reason why a classifier cannot correctly classify an app is because the fingerprint of the app looks too much like another app. This can happen due to two reasons. First, the training set is not exhaustive, so the testing data is new to the classifier and it picks the closest app. Second, the app produces very similar traffic to another app and the classifier chooses the wrong app. To find out which of the two is happening and why we dive a little deeper into these two categories of apps.

Similar distribution between users, low F1-score

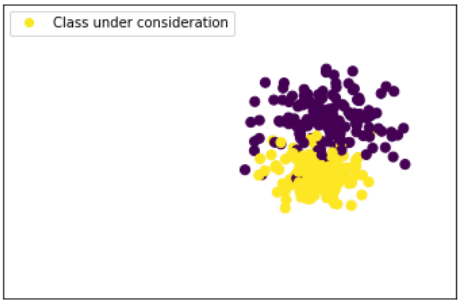
For apps with a consistent feature distribution for different users, we expect a high F1-score. However, instead, we often find a low F1-score for apps with a consistent feature distribution. The Anderson Darling test is performed on users from both the testing and training sets, so a consistent feature distribution for users also means a consistent feature distribution for the training and testing set. Therefore the low F1-score is not caused by a non-exhaustive training set or other dissimilarities between the training and testing set. The other reason is that an app produced very similar traffic to some other app. This is likely the case for this category of apps, the overall fingerprint of the app is very close to another app, which causes a high reported significance level but a low F1-score. To verify that this is indeed the case we perform the two-sample Anderson Darling test to compare the feature distributions of an app with all other apps on which the classifier was trained. Each of these AD tests gives a significance level indicating the probability that the feature distributions were drawn from the same population. A high significance level means that the feature distributions, or fingerprint, are very much alike. The highest reported significance level was added to the table and is thus an indication of how close the next neighbor is to the fingerprint of the app under consideration. A complete list of which app is the closest can be found in Table C.3.

In the table, we see that the apps with a high significance level for users (> 0.5), but a low F1-score (< 0.5) also have a high maximum significance level when comparing apps (> 0.8). There are seven apps with a high significance level for users and all of those have a low F1-score. Five of those apps also have a close neighbor, this explains why their F1-score is low; the classifier has a hard time distinguishing between this app and their close neighbor(s). The fingerprint of these apps will look like Figure 6.6b.

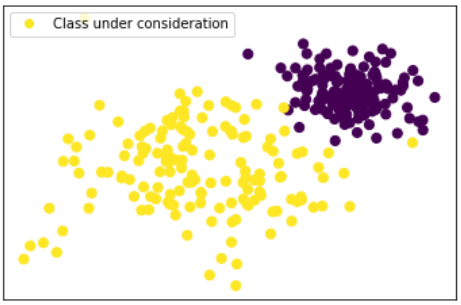
This leaves us with three apps that do not have a particularly close neighbor but still have a low F1-score. One of these is *com.google.android.googlequicksearchbox*. Previously we said that some specific bursts from the *Google* app would be easy to detect since they are so consistent across users. However, from Table 6.10 we see that the F1-score of the app is 0, so even these specific bursts are not detected. The reason for this is that there are 21 other apps in the dataset which have bursts which also consist of only one incoming packet of 56 bytes and thus have the same features. So even though there are some bursts consistent across users - which



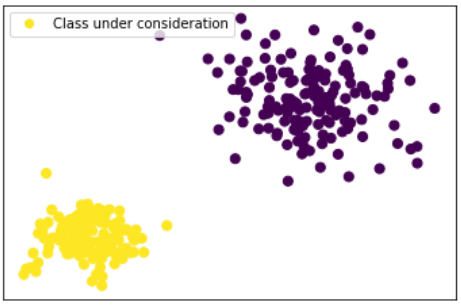
(a) Difference between users, close neighbor



(b) Little difference between users, close neighbor



(c) Difference between users, distant neighbor



(d) Little difference between users, distant neighbor

Figure 6.6: Examples of various combinations of similar/dissimilar feature distributions of users within an app and a close/distant neighbor. Overlap between two classes means that the classifier has difficulty assigning the correct label, which results in lower scores. It shows how a very spread out fingerprint can still be easily classified. While a concentrated fingerprint can be very hard to classify. These visualizations are examples and do not depict actual data.

explains the relatively high significance level - the classifier still does not classify them correctly since many apps contain these bursts. For the other bursts from this app, the differences between users are much larger, so the classifier naturally has a harder time classifying them. The same holds for *com.google.android.apps.maps*.

Different distributions between users, high F1-score

For apps with a lot of differences in the feature distributions between users, we expect a low F1-score. A lot of differences would mean a higher chance that the fingerprint of this app overlaps with another app. It also means that there is a high chance of a difference between the feature distributions of the training and testing set, which makes it harder for the classifier to assign the correct label to the testing data. However, we observe that even though the distributions are not very alike and there are a lot of different users in the data for an app, the classifier is still able to correctly classify the app. In a similar fashion as for the category of similar distributions and a low F1-score, there are two reasons for the high F1-scores here. Either, despite the differences between users, the overall fingerprint of the app is far away from the fingerprints of the other apps, or there are some common characteristics in the distributions for all the users on which the classifier trains.

There are 17 apps with a low significance level (< 0.5) for users. Of these, 10 apps have a high F1-score (> 0.5) and 9 of these have a relatively low maximum significance level for apps (< 0.8). This means that these apps have a very large (or spread out), but relatively isolated fingerprint. Which explains why the F1-score is still quite high, the classifier can distinguish well between the fingerprint of this app and others since there is little overlap. The fingerprint of these apps will look like Figure 6.6c.

The common characteristics in the distributions for all users might also be the reason why on average the fingerprint of an app is far away from other apps. Take for example *com.whatsapp*, it is known that *Whatsapp* uses some very consistent packets to, for example, signal that someone is typing [15]. We observe 37 bursts for 17 different users, which all consist of 1 incoming packet of 39 bytes and 2 outgoing packets of 3 and 35

bytes. This burst is unique to *Whatsapp* in our dataset and accounts for 5% of the total bursts of *Whatsapp*. For *org.telegram.messenger* there exist 52 bursts for 8 different users, which all have 1 incoming packet of 89 bytes and various outgoing packets. This characteristic of a burst is unique to *Telegram* in our dataset. Bursts like these, that are consistent between users and unique to an app, cause a relatively high F1-score even though overall the feature distributions between users are very much alike.

6.7. Conclusion

It is possible to collect labeled real-world data on a small scale. There is no need for labeling by hand and it is possible to find people who are willing to share their traffic metadata for a reward.

The high performance of cross-validation on real-world data shows that bursts are consistent within their class and suggests that classification is very well possible. The data is splitted into two parts and experiments are performed using this data for training and testing. The results of the experiments on splitted data show the difficulty of dealing with previously unseen apps in the real world. The experiments do show that for popular apps the classification works a lot better and that obtaining enough training data for these apps is easier. Supplementing training data with emulator data does not increase performance and mainly adds more noise to the data. Training on only emulator data and testing on real-world data gives us a very poor performance as well.

We define three categories of apps. Apps for which the network traffic is mainly *Static*, *Dynamic* over *Time* or *Dynamic* between *Users*. *Static* and *Dynamic Time* apps serve the same content to all users, but the content of *Dynamic Time* apps changes over time. *Dynamic User* apps require complex user input or an established account and serve content dependent on the user. We show that for the *Dynamic User* categories, training on emulator data does not work.

We examine the feature distributions of apps and analyze why certain apps perform better than others. We show that apps that produce very diverse data for different users can still be classified correctly. The main causes of this are that there exist common structures across users, which can be considered *Static*. Another reason is that the overall fingerprint of an app is fairly unique. We also show that apps that produce very consistent traffic can still be hard to classify when they are similar to other apps.

These results suggest that a slightly more advanced collection method on an emulator, e.g. logging in with a dummy account, can still collect valuable training data by generating the *Static* traffic. It shows that app traffic cannot be assigned an exclusive category, but that an app generates traffic from multiple categories. Furthermore, it is very important on which other apps the classifier is trained to avoid overlap in the fingerprints. These insights pose constraints on the general application of traffic analysis for app detection. However, they also explain why certain types of apps with complex functionality can, counter-intuitively, still be fairly accurately classified. And what training data is required and by which method it can be obtained. Using this information it is possible to draft threat models about which attacks are and which attacks are not possible to perform in the real world using mobile network traffic analysis.

7

Countermeasures

This and previous work [1, 14, 36, 58, 62] have shown that network traffic analysis can compromise the privacy of mobile phone users. To get a better insight into what makes the traffic analysis work and to see what can be done to counter this type of analysis we take a look at countermeasures. The main features used for traffic analysis are related to the timings and the sizes of the packets. Countermeasures should thus focus on changing those properties to confuse any trained models or to create homogeneous fingerprints for all apps, so traffic is misclassified.

7.1. Using a VPN

A Virtual Private Network (VPN) is a popular instrument nowadays to hide browsing activity. A VPN achieves this in two ways: it encrypts all traffic and it routes all traffic through a proxy server. The first ensures that no-one between you and the proxy-server can view the contents (or payload) of your traffic. This includes normally unencrypted packets like DNS traffic or packet headers, which reveal the IP addresses of the servers a user is talking to. The second mixes the traffic with traffic from other users, so it becomes very hard to identify from which user the traffic originates from.

For confusing the classifiers we are mainly interested in the first characteristic. The OpenVPN protocol breaks up packets into fragments and adds them together to create new packets of MTU (Max Transmission Unit) length. This will cause traffic to look homogeneous when many packets are sent around the same time. This, combined with the slightly changed timings of the packets due to the processing of OpenVPN and the added latency from the proxy server, can change the fingerprint of an app significantly so it cannot be recognized anymore. Since the use of VPNs is becoming more widespread this would be the easiest countermeasure against traffic analysis that a mobile phone user can implement themselves.

A VPN will work very well for the type of analysis we do, this is because a VPN hides the source/destination ports and IP addresses. Therefore when analyzing VPN traffic we cannot separate the traffic into flows. Other types of analysis, based on bursts or a packet level, however, are still possible. We are mainly interested in the effect the changing timings and packets sizes have on the performance of the classifiers and ignore the fact that in a normal environment we will not be able to do the flow separation. To still analyze this effect we amended the OpenVPN source code to log which packets are repacked into which VPN packets. Using this log we can exactly recreate the flows from the normal traffic in the VPN traffic and use this flow separated VPN traffic for the experiments.

7.1.1. Setup

To investigate the effect of a VPN we use a similar setup as previous experiments that use emulators to generate data. On a server, we collect network traces generated by running apps on emulators in the same fashion as in chapter 5. Five emulators are run in parallel to decrease the time required for collecting all the traces. The data is collected for a random selection of 50 apps from our *4 February* dataset. For each of the apps, 10 traces are collected. To get a good overview of the effects of using a VPN service, 10 different VPN servers in 10 different countries are used. An overview of the servers used can be found in Table 7.1.

Apps that were run at different times can create different network traffic since the content displayed in the app might be updated (e.g. different news articles). This might affect the network fingerprint as well. We want

to be able to detect the potentially subtle changes in classifier performance between multiple VPN servers. Therefore we should take care to limit the influence of other factors on the classifier performance, like changes in network fingerprints due to the passage of time. To do this we switch to a new VPN server after 10 traces are collected for an app, instead of first collecting all traces for one VPN server. This limits the time between the collected traces for an app significantly and ensures that there is no or minimal difference in the app behavior between traces collected for different VPN servers. The traces were collected between 19 July and 5 August 2019.

On the server where the emulators are running, we start a tunnel to NordVPN using the OpenVPN software. This is easier than running the VPN service on the emulators themselves since it is hard to control a VPN app on a windowless emulator and because emulators are unreliable and crash often. This means that the VPN encapsulation is done one step later than on an emulator, however, the effect is very similar.

Country	City	Round-trip Time (ms)
Brazil	Rio de Janeiro	253
India	Mumbai	176
Israel	Tel Aviv	100
Italy	Milan	32
Japan	Tokyo	263
Netherlands	Amsterdam	5
South Africa	Johannesburg	233
Turkey	Istanbul	70
United Kingdom	Coventry	21
United States	San Francisco	201

Table 7.1: Overview of where the VPN servers used during data collection are based. The Round-trip Time was determined by pinging the server 10 times from the TU Delft and taking the average Round-trip Time. The Round-trip time is used to verify whether an added latency has an effect on the classifier performance.

To perform the experiment we require labeled VPN data. This is not trivial to come by since our method of using port/app mappings won't work as the VPN uses entirely different ports for the traffic. To be able to label the data anyway, we require a mapping from the labeled non-VPN traffic to the VPN traffic. We achieved this by modifying the source code of the OpenVPN software. When the VPN service is running, our modified version of OpenVPN logs the unencapsulated and encapsulated packet payloads to a file. At the same time, we capture the VPN traffic leaving the server using *tcpdump* and we capture the non-VPN traffic on each of the emulators like we would in our regular setup for emulators as well. Figure 7.1 shows an overview of the setup. After a trace has been successfully collected for each emulator, we are left with the following files:

- 1 Pcap of VPN traffic (combined traffic from 5 emulators)
- 5 Pcaps of non-VPN traffic (one for each emulator)
- 5 Port logs (one for each emulator)
- 1 Mapping from unencapsulated payloads to encapsulated payloads (combined traffic from 5 emulators)

Using these files we can find out for each packet in the non-VPN pcaps which VPN packets encapsulate this payload. After we have done the regular processing of the non-VPN data and have obtained a set of labeled bursts, we form the same bursts in the VPN traffic using the mapping and can create labeled VPN bursts.

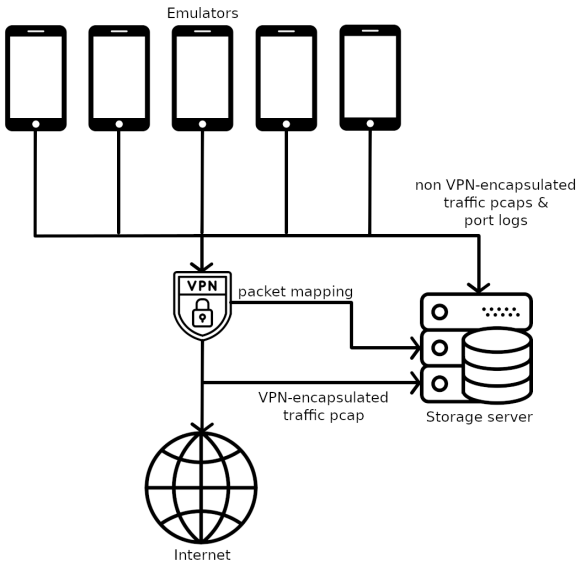


Figure 7.1: Overview of how the emulators are connected to the Internet and at which points data is captured or logged.

7.1.2. Cross validation

For each VPN setting as listed in Table 7.1 there exist two datasets: one with non VPN-encapsulated and one with VPN-encapsulated traffic. We also select the data from the 50 selected apps from the *4 February* dataset. This dataset is used to train our classifier to get an idea of whether and how much the usage of a VPN affects the performance of a classifier. We are not concerned with the actual performance of the classifier in this experiment but focus instead on the difference between the performance of the non-VPN and VPN traffic.

The US and ZA datasets were not used in this
experiment because there were a lot of issues
with mapping the traffic from regular to VPN-encapsulated traffic.

Dataset	Average # classes	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
4 February	55	0.678	0.698	0.594	0.624
VPN BR	51	0.672	0.658	0.533	0.567
regular BR	51	0.662	0.638	0.521	0.548
VPN IL	51	0.667	0.694	0.567	0.598
regular IL	51	0.654	0.707	0.558	0.598
VPN IN	50	0.695	0.702	0.548	0.597
regular IN	50	0.690	0.698	0.548	0.596
VPN IT	51	0.662	0.705	0.576	0.613
regular IT	51	0.650	0.684	0.560	0.597
VPN JP	47	0.678	0.683	0.556	0.591
regular JP	47	0.671	0.688	0.543	0.582
VPN NL	48	0.655	0.670	0.524	0.567
regular NL	49	0.636	0.689	0.521	0.571
VPN TR	47	0.401	0.432	0.340	0.351
regular TR	48	0.408	0.436	0.352	0.362
VPN UK	49	0.682	0.687	0.547	0.589
regular UK	49	0.666	0.678	0.523	0.570

Table 7.2: Baseline performance of the datasets used in the experiment to see the influence of using a VPN service on classifier performance. 5-fold cross validation was performed on the datasets.

5-Fold cross-validation was performed on each dataset, to show a base quality of the dataset. The results from these tests can be found in table Table 7.2. The baseline results show that the datasets are of similar quality, except for the TR dataset. This dataset has a lot lower baseline performance; likely something went wrong with collecting this data, for example, an unreachable VPN server. Therefore we will ignore the TR dataset in further analysis. Because the datasets are of a similar quality we will not have to take into account any bias due to the quality of the data.

7.1.3. Performance non VPN-encapsulated data

A Random Forest classifier was trained on the data of the 50 selected app from the *4 February* dataset. This classifier was used to classify both the non-VPN-encapsulated and the VPN-encapsulated traffic collected during this experiment. The results of classifying the traffic collected while a VPN service was running can be found in Table 7.3. This can be used as a baseline because it can be used to see whether performance on VPN-encapsulated traffic is poor because of the VPN-encapsulation or because the testing data is simply vastly different from the training data.

The performance on non-VPN traffic is quite poor, so there is a mismatch between the training data and the testing data. There are two main reasons for this: 1) the data was collected at a different time, so the content of apps might have changed. 2) The regular traffic is still influenced by the VPN because all traffic is still routed through the VPN. We will thus expect the timings to be slightly different than with the data collected without a VPN. This would also mean that the performance would differ per VPN server, as they have a different location and thus different timings.

This characteristic in the data can be used to test the hypothesis that it matters where the training data and testing data is collected; the influence of a network environment. By correlating the results with the Round-trip Time to the servers from the TUDelft we can see whether a difference in *capture location* between the training and testing set influences the performance. The Round-trip Time was determined by taking the average Round-trip Time (RTT) from pinging the server 10 times and can be found in Table 7.1. Correlating the RTT with the accuracy, precision, recall, and F1-score from Table 7.3 gives correlation coefficients of -0.230, -0.200, -0.842, and -0.772 with P-values of 0.514, 0.667, 0.017, and 0.042 respectively. Using a significance level of $p < 0.05$, we can say that there is a strong correlation between the RTT and recall and F1-scores.

Accuracy shows a weak correlation and Precision shows no correlation, but Recall and F1-score show a strong negative correlation. This means that as latency increases, the Recall and (therefore) the F1-score goes down, while the other metrics do not show this trend.

Training set	Validation set	Training # classes	Validation # classes	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
4 February	non-VPN BR	55	51	0.346	0.379	0.246	0.255
4 February	non-VPN IL	55	51	0.350	0.381	0.272	0.278
4 February	non-VPN IN	55	50	0.316	0.345	0.271	0.263
4 February	non-VPN IT	55	51	0.370	0.410	0.298	0.308
4 February	non-VPN JP	55	47	0.370	0.437	0.272	0.287
4 February	non-VPN NL	55	49	0.361	0.409	0.291	0.304
4 February	non-VPN UK	55	49	0.361	0.420	0.311	0.300

Table 7.3: Performance on datasets of non-VPN-encapsulated data collected while a VPN service was active. The classifier was trained on data collected on emulators without any VPN active. Both the training and testing set contain a collection of 50 apps from the 4 February dataset.

$$\begin{aligned} Accuracy &= \frac{TP+TN}{TP+TN+FP+FN} \\ Precision &= \frac{TP}{TP+FP} \\ Recall &= \frac{TP}{TP+FN} \\ F1-score &= \frac{2 \cdot TP}{2 \cdot TP+FP+FN} \end{aligned} \tag{7.1}$$

Different correlation values for different metrics might seem unexpected at first, but can be easily explained if we take a closer look at the formulas for calculating accuracy, precision, recall, and F1-score as found in Equation 7.1. When looking at these formulas we must conclude that as the RTT rises, on average, the number of True Positives decreases and the True Negatives increases. This causes a lower Recall and F1-score while having no effect on the Precision and only a small effect on the Accuracy. The following toy example in Table 7.4 illustrates this effect. In the table we see that the precision stays the same, while the accuracy only slightly decreases.

Before/after RTT increase	TP	FP	TN	FN	Accuracy	Precision	Recall	F1-score
Before	60	40	50	150	0.37	0.60	0.29	0.39
After	30	20	70	180	0.33	0.60	0.14	0.23

Table 7.4: Toy example showing the effect the change in RTT can have on the metrics. When the RTT rises, the number of True Positives decreases and the number of True Negatives increases. This explains why there is no correlation between the RTT and the accuracy or precision, since they do not change when an app is misclassified more often.

The number of True Positives decreases when the fingerprints of apps start to drift and there are more misclassifications. This drift can be confirmed by analyzing the feature distributions by performing the two sample Anderson Darling test on the training set and all the testing sets. The resulting average significance level can be correlated with the latency.

The average significance levels performing the AD test on the *4 February* set and non-VPN BR, IN, IL, IT, JP, NL, and UK datasets are 0.1373, 0.1310, 0.1332, 0.1564, 0.1438, 0.1785, and 0.1621. The Pearson correlation with the RTT gives a correlation coefficient of -0.723, with p-value 0.066. While this suggests a strong negative correlation between the RTT and the drift of the feature distributions from the training set, it is not significant at the $P<0.05$ level. When only considering the duration and Inter Arrival Time (IAT) features for the significance levels we get significance levels of 0.0931, 0.0854, 0.1056, 0.1580, 0.0964, 0.2102, and 0.1785 and a very strong and significant negative correlation of -0.877 with an p-value of 0.023. When only considering the size related features we get significant levels of 0.1594, 0.1537, 0.1471, 0.1555, 0.1674, 0.1627, and 0.1539 and a correlation coefficient of 0.364 with p-value 0.422, which means that there is no significant correlation. This shows that it is indeed mostly the time-related features that change when connecting to a different VPN server. Therefore we can conclude that the fingerprints of apps that do not make use of Content Delivery Networks - which would reduce the latency - change significantly depending on the location of the user. This is something that should be taken into account when optimizing a classifier.

7.1.4. Performance on VPN data

The results from the classification attempt of VPN-encapsulated traffic can be found in Table 7.5. The difference between these results and the results from the non-VPN-encapsulated data in subsection 7.1.3 is large. The use of a VPN service reduces the performance of the classifier significantly. It is possible that by fragmenting and joining packets the VPN service changes the fingerprint of the apps so much that they cannot be distinguished from each other anymore; the fingerprints become homogeneous. To confirm this we perform the Anderson Darling test on these classes, to see if they are drawn from the same distribution. This is done for all possible combinations of classes, 1 275 times in total. We use only the two-sample AD test since a higher sample AD test is more sensitive to subtle changes in the feature distributions and would thus always give a very low significance level. The AD test is performed on both the VPN BR dataset and the non-VPN BR dataset. A problem encountered with the AD test is that the distributions are so alike for some features that the AD test cannot be performed. The AD test requires at least 2 distinct values in the distributions under consideration. However, for example, the Inter Packet Size features are very often only 0 for a class, since all packets are of Max Transmission Unit length. Therefore the AD test reports NaN for these feature distributions, and the average significance level reported is much lower than it should be. To correct this we replace the NaN values with 1.000 if both distributions contain the same distinct value. If only one of the distributions contains one distinct value we leave the NaN result. The resulting average significance level is 0.1224 and 0.1207 for the VPN and non-VPN data respectively.

Surprisingly, there is little difference between the VPN and non-VPN data. Perhaps the distribution is still rather diverse when considering all the features, but not when considering only a certain type of features. Therefore we check the effect of only considering features related to size and counts and features only related to time in the next section.

Training set	Validation set	Training # classes	Validation # classes	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
4 February	VPN BR	55	51	0.115	0.175	0.090	0.089
4 February	VPN IL	55	51	0.140	0.212	0.102	0.103
4 February	VPN IN	55	50	0.130	0.234	0.110	0.110
4 February	VPN IT	55	51	0.127	0.248	0.104	0.108
4 February	VPN JP	55	47	0.117	0.220	0.087	0.099
4 February	VPN NL	55	49	0.136	0.237	0.095	0.105
4 February	VPN UK	55	49	0.140	0.250	0.096	0.106

Table 7.5: Performance on datasets of VPN encapsulated traffic. The VPN encapsulated traffic was mapped to form the same bursts as in the original data. The metrics show that the performance is very poor.

7.1.5. Using a different subset of features

From the way VPN works we know that packet lengths change and that the packet count per burst changes due to fragmentation and merging of packets. We train on regular data and we know that the features based on size and counts contribute to discriminate between classes. Therefore it makes sense to remove these features and not train on them since we expect that those feature distributions in the test data will be completely different anyway. The experiments from the previous sections are repeated, but now with a different subset of features. One subset without size and count related features and one subset without time-related features to confirm the hypothesis.

Removing size and count features

From all the datasets we remove the features that have to do with size, packet count, and Inter Packet Size (IPS). This leaves a set of 57 time-related features. 5-fold cross-validation is performed on all the datasets with this subset of features to see the effect of removing so many features that have discriminatory power. This acts as a baseline for the results of the classification experiment. As expected, the metrics, reported in Table C.4 are lower, about 20 percentage points on average, than the metrics are on the full feature set.

A Random Forest is trained on the selected apps from the *4 February* dataset on the subset of 57 features. This classifier is used to classify the VPN data from the various settings, results of which can be found in Table 7.6. When comparing the results to the classification on the full feature set from Table 7.5 we see that the performance decreases when only training on time-related features. This suggests that even the distributions of the time-related features are so different in the training and testing sets that the classifier cannot classify the traffic correctly.

Training set	Validation set	Training # classes	Validation # classes	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
4 February	VPN BR	55	51	0.078	0.066	0.053	0.042
4 February	VPN IL	55	51	0.081	0.056	0.045	0.040
4 February	VPN IN	55	50	0.067	0.065	0.055	0.043
4 February	VPN IT	55	51	0.102	0.105	0.068	0.063
4 February	VPN JP	55	47	0.075	0.061	0.046	0.045
4 February	VPN NL	55	49	0.120	0.098	0.068	0.069
4 February	VPN UK	55	49	0.105	0.088	0.064	0.064

Table 7.6: Performance on datasets of VPN encapsulated traffic. The VPN encapsulated traffic was mapped to form the same bursts as in the original data. For these datasets, a subset of features was used from which all features related to packet sizes and counts were excluded.

Training set	Validation set	Training # classes	Validation # classes	Avg. Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
4 February	VPN BR	55	51	0.115	0.168	0.086	0.087
4 February	VPN IL	55	51	0.133	0.164	0.093	0.094
4 February	VPN IN	55	50	0.141	0.191	0.103	0.101
4 February	VPN IT	55	51	0.115	0.198	0.092	0.095
4 February	VPN JP	55	47	0.124	0.204	0.089	0.098
4 February	VPN NL	55	49	0.111	0.174	0.078	0.087
4 February	VPN UK	55	49	0.121	0.170	0.080	0.088

Table 7.7: Performance on datasets of VPN encapsulated traffic. The VPN encapsulated traffic was mapped to form the same bursts as in the original data. For these datasets, a subset of features was used from which all features related to packet timings were excluded.

Removing time features

To verify that the time-related features change a lot due to the VPN as well, another experiment was performed where all features related duration and Inter Arrival Time (IAT) were removed from the datasets. A feature set of 114 size and count related features remains. 5-fold cross-validation was performed on all datasets with the timings removed, the results of which can be found in Table C.5. The performance is very similar to the performance on the full feature set. This suggests that the discriminatory power of the timings is not that great.

When a classifier is trained on the 50 selected apps from the *4 February* dataset this is confirmed. The results in Table 7.7 show a very similar result to the results from using the full feature set. It shows that the combined time-related features have even less discriminatory power than the combined count and size-related features. These experiments show that there is no one type of features on which the classifier would work better. And it also shows that there is no one type of features for which the fingerprints are homogeneous.

7.1.6. Training on VPN data

When looking at results in Table 7.2, there is no apparent difference between VPN and non-VPN data. This suggests that when training on VPN data, it is possible to classify VPN data with the same accuracy as regular data. This would render VPN as a countermeasure useless, as any smart attacker would simply create two models for classification: one trained on regular data and one trained on VPN data. As VPN usage is very easy to detect, the attacker can choose to apply the VPN model as soon as VPN traffic is encountered.

Analyzing the feature distributions

The fact that it is still possible to classify data when cross-validating on VPN data says that the fingerprints of the apps are not homogeneous in the VPN setting. This was previously confirmed by the results of the Anderson Darling test in subsection 7.1.4. We suggested that perhaps a certain type of feature still has discriminatory power. From subsection 7.1.5 we know that using a subset of features does not seem to help the classifier. The results suggest that the reason why the classifier produces results close to random guessing is not that it cannot distinguish between classes anymore, but because it assigns many classes the wrong label as the feature distributions drift.

We analyze the feature distributions of the classes in the *4 February* dataset and the VPN IT dataset. To check whether the fingerprints between different VPN settings stay the same, we also compare the feature distributions of VPN IT and VPN BR. If this is the case this would mean that we can train a model on VPN data to classify other VPN data. Like with previous tests, the comparison is done using the Anderson Darling test.

When comparing the feature distributions of the 50 selected apps from the *4 February* dataset and the VPN IT dataset we get an average significance level of 0.1053. The comparison of the feature distributions

between the VPN BR and the VPN IT dataset gives a much higher significance level of 0.3586. This means that the fingerprints of apps stay fairly consistent across VPN settings and that they change a lot when moving from a non-VPN to a VPN setting. Combined with the previous insight that the fingerprints are not homogeneous in the VPN setting, this means that it is possible to classify VPN data when using a classifier trained on VPN data.

In Table 7.8 we calculated the average significance level for different types of features over all classes. This shows, when comparing the fingerprints of classes of the *4 February* dataset and a VPN dataset, which features are the most consistent part of the changing fingerprint. The fact that the Inter-Packet Size (IPS) features are fairly consistent is very surprising, since we know that OpenVPN changes the size of packets by merging and fragmenting them. To further investigate this, the IPS features are split into the time series from the combined, incoming and outgoing bursts. This shows that the time series of outgoing bursts stays much more consistent than the incoming bursts. The reason for this is that the outgoing bursts consist of a much lower amount of packets than the incoming bursts. On average per class there are 4.56 packets in an outgoing burst and 15.20 in an incoming burst for the *4 February* dataset. And for the VPN IT dataset 5.56 in an outgoing burst and 56.72 in an incoming burst. A similar number is reported for the other VPN datasets. This mismatch is caused by some variation in the data collection, like an overloaded collection server.

Intuitively we would say that the resulting change in significance level is caused by OpenVPN merging packets. OpenVPN merges packets that arrive within a short time from each other. There is a lot of incoming traffic and much less outgoing traffic, therefore there is higher chance packets arrive closer together and get merged. This would mean that the IPS changes more for the incoming bursts than for the outgoing bursts since we have more packets of max length for the incoming bursts. However, when we look at the average significance levels for the non-VPN IT set in Table 7.8, we see a similar mismatch between the significance levels for incoming and outgoing bursts. So this means that it is in fact not OpenVPN that causes this mismatch, but simply the change in the collected network traffic.

Set 1	**Set 2**	**burst duration**	**packet count**	**size**	**iat**	**Inter Packet Size (ips)**			
						all	com	in	out
4 February	VPN IT	0.1651	0.1109	0.0650	0.1228	0.1252	0.0955	0.0931	0.1870
4 February	non-VPN IT	0.1642	0.1523	0.1476	0.1575	0.1641	0.1332	0.1071	0.2519
VPN BR	VPN IT	0.1510	0.4445	0.4400	0.1982	0.4399	0.4357	0.4487	0.4354

Table 7.8: Average significance level, as calculated using the two sample AD test on two datasets, for several feature types. The Inter Packet Size features are split into features from the complete burst, only the incoming packet of a burst, and only the outgoing packets of a burst.

Experiments

To test that it is indeed possible to train a classifier on VPN data and correctly classify VPN data, we use the seven VPN datasets that are of sufficient quality and split them up into training and testing sets. We start with the BR dataset as the training set and all other sets as the testing sets and move one testing set to the training sets until we only have 1 testing set and all other sets as training sets. This allows us to analyze the effect of the ratio between training and testing sets as well. Another test, starting from the UK dataset as a training set showed a slightly better (2 percentage point) performance and the same trend. Variations due to the choice of datasets are to be expected since some datasets are more like each other than others. The results for this experiment as seen in Table 7.9, confirm our hypothesis. It is very well possible to classify VPN traffic when training on VPN traffic. The results are comparable to the results for datasets with a similar number of classes in chapter 5. They also show that by using more training data the performance can be boosted significantly.

Training set	**Validation set**	**Training # classes**	**Validation # classes**	**Accuracy**	**Macro-average Precision**	**Macro-average Recall**	**Macro-average F1-score**
BR	IL to UK	51	53	0.557	0.592	0.433	0.470
BR, IL	IN to UK	52	53	0.634	0.700	0.532	0.581
BR to IN	IT to UK	52	53	0.649	0.733	0.580	0.620
BR to IT	JP to UK	52	52	0.688	0.735	0.589	0.631
BR to JP	NL, UK	52	51	0.680	0.731	0.577	0.622
BR to NL	UK	54	49	0.728	0.775	0.628	0.677

Table 7.9: Performance results of training and testing on VPN data. An increasingly larger training set and an increasingly smaller test set were used to show the effect of the ratio between testing and training sets as well. The order of the sets used is BR, IL, IN, IT, JP, NL, UK.

These results pose a question; now that the traffic shape has changed due to the VPN service, is there any change in which features are considered important? To check this we retrieve the feature importance as reported by the Random Forest classifiers trained on several datasets. Looking at Table 7.10, there is very little difference in the feature importances for the classifiers. This means that even though the fingerprint changes significantly, to the point where the *4 February* classifier cannot make any sense of the VPN BR data, the features still have a strong discriminatory power. Only the discriminatory power for the Inter Packet Size related features drops a little bit. This can be explained due to the behavior of OpenVPN that merges packets. This means there are more packets of MTU in the data, which also means that the IPS is 0 more often. This merging of packets of course also changes the packet count features, but it does not make them more ambiguous.

Training set	burst_duration	packet_count	size	ips	iat
VPN BR	0.006449	0.002814	0.006949	0.005627	0.005042
non-VPN BR	0.005252	0.002677	0.007018	0.005383	0.005287
4 February 50 apps	0.004307	0.002445	0.006943	0.005912	0.004903

Table 7.10: Feature importances of classifiers trained on various datasets. The feature importances are the average feature importances of various feature types. These types represent 3, 3, 57, 54, and 54 features respectively.

7.2. Padding related countermeasures

The easiest way to mitigate network traffic analysis is to implement countermeasures at the protocol level. This way users will not have to worry about protecting their privacy, it is instead an inherent part of the connection. TLS provides options for such countermeasures in the form of packet padding. While this option is not widely used yet, it can be implemented reasonably easy because it is already part of the standard. Packet padding changes the size-related features, which are most important for the classifier to discriminate between apps. There exist many possible padding schemes, but the TLS documentation does not provide any guidelines on which padding schemes to use. Therefore we will test a selection of padding schemes as previously also tested by Dyer et al. [19] on web traffic data.

The effect of padding schemes can be tested by changing the features of the collected traffic as if padding was applied. We extract the regular features per packet as we would normally. Only now, before separating the traffic into flows and bursts and extracting the statistical features, we add the padding length to the packet length feature. This way we obtain the data as if the padding scheme were applied on the connection. The padding schemes we test are defined as follows:

- Session random 255 (randomly choose padding length {0,8,...,248} for each TCP session)
- Packet random 255 (randomly choose padding length {0,8,...,248} for each packet)
- Linear (pad to nearest multiple of 128)
- Exponential (pad to nearest power of 2)
- Mice-elephant (pad to 128 if packet length <= 128, otherwise pad to Maximum Transmission Unit (MTU))
- Pad to MTU
- Packet random MTU padding (randomly choose padding length 0,8,...,MTU-l where l is the packet length)

To test the effectiveness of these countermeasures we apply them on the 4 February dataset. We train a classifier on the 21 January dataset and try to classify the 4 February dataset with various countermeasures applied. The feature set used includes the destination domain names and ports. We are interested in the difference between the performance of the classifier on the 4 February dataset with and without countermeasures in relation to the overhead introduced by the countermeasure. This gives us a good indication of how well a countermeasure works and if it is usable in practice.

Table 7.11 shows the results of classifying the 4 February dataset with and without various padding related countermeasures. The first thing to notice is the low macro-average scores. This is due to the fact that the training set contains many more classes than the testing set. Because bursts are classified as a class that does not occur in the testing, some classes will not have any True Positives, leading to a precision, recall, and F1-score of 0. The best padding scheme is, as might have been expected, the scheme that pads all packets to MTU; for accuracy and the F1-score, the performance is reduced by more than 98%. By using this scheme the size-related features of all classes look the same, and since our classifier relies heavily on the size-related

Countermeasure	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score	Overhead
None	0.564	0.223	0.161	0.180	0%
Session Random	0.378	0.229	0.102	0.129	5.32%
Packet Random	0.278	0.185	0.071	0.091	5.63%
Linear	0.317	0.227	0.082	0.109	3.62%
Exponential	0.323	0.207	0.087	0.111	5.62%
Mice-Elephants	0.079	0.221	0.026	0.041	21.53%
MTU	0.005	0.063	0.002	0.004	38.14%
Random MTU	0.238	0.154	0.061	0.077	18.90%

Table 7.11: The performance of a classifier trained on the 21 January dataset when classifying the 4 February dataset with and without various padding-related countermeasures using the extra feature set. The difference in performance between the dataset without countermeasures and with countermeasures shows how well a countermeasure works. The overhead for each countermeasure shows how much extra bandwidth the countermeasure requires. A large overhead makes a countermeasure less attractive for real-world application. No ads were filtered for this data since not the absolute, but the relative performance is important for this experiment. The macro-average metrics are much lower than the accuracy because bursts are classified as one of the apps that do not occur in the 4 February dataset, this means that there are many classes with only False Positives resulting in low macro precision, recall and F1 scores. This applies to the other tables in this section as well.

features, it is not able to classify any bursts correctly anymore. This good performance does come at the cost of more than 38% overhead. The Mice-Elephants scheme looks like a more attractive countermeasure since it also reduces the performance significantly at the cost of a much lower overhead of 21.5%. The Mice-Elephants scheme reduces the possible packet lengths to two options: either 128 bytes or MTU. While this leaves some information, it also reduces the performance significantly by more than 78% for the accuracy and F1-score metrics. Random MTU fails to hide the original packet length sufficiently, since large packets will often be MTU, regardless of the padding that is chosen and only reduces performance by about 58%, while still having a rather large overhead. The remaining padding schemes, while having a much smaller overhead of up to 6%, are much less effective and only reduce the performance by 30-50%.

From this experiment, we can conclude that the only padding schemes that reduce the performance of a classifier significantly come at the cost of significant overhead. This overhead is not attractive, however, if it can significantly reduce the leakage of information it might be worth implementing the scheme anyway. The padding schemes that do not introduce significant overhead are not that effective. Using these schemes as a stand-alone countermeasure is not worth it since the performance is so high that information can still be inferred. If effective countermeasures without significant overhead are required, padding schemes are not the way to go.

7.2.1. Training on countermeasure data

Using a VPN as a countermeasure, we saw that the fingerprints did not become more homogeneous and classification was still possible using a classifier trained on data with the countermeasure applied. To check the effect on the homogeneity of fingerprints of the padding related countermeasures we apply the same method. For each padding related countermeasure, a per class classifier is trained on data from the 21 January dataset with the countermeasure applied. These trained classifiers are subsequently used to classify the data from the 4 February dataset with the countermeasure applied. The effectiveness of these classifiers truly shows the effectiveness of the countermeasure, since an attacker will switch to training on data with countermeasures once the countermeasures are applied frequently.

Table 7.12 shows the results of this test. Here we again use the extra feature set. When we compare these results to the results of the classification of the data without countermeasures in Table 7.11, we see that the difference is staggering. Where previously we saw a reduction of more than 90% for the accuracy and F1-score metrics when using Mice-Elephants or MTU padding schemes, now the reduction is only 20-30%. The same difference can be observed for the other countermeasures, almost all of them are three times less effective when the classifier is trained on countermeasure data as well.

Previously we already concluded that there are no effective padding schemes without significant overhead, we now see that the padding schemes are not effective at all. This raises the question of why the difference in performance is so large and on which features the classifier bases its decisions now. Therefore, we will look into the homogeneity of the fingerprints and the feature importance assigned to the features.

Countermeasure	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score	Overhead
None	0.564	0.223	0.161	0.180	0%
Session Random	0.513	0.200	0.141	0.159	5.32%
Packet Random	0.459	0.174	0.116	0.133	5.63%
Linear	0.488	0.189	0.129	0.147	3.62%
Exponential	0.468	0.175	0.120	0.137	5.62%
Mice-Elephants	0.440	0.167	0.110	0.127	21.53%
MTU	0.431	0.163	0.105	0.122	38.14%
Random MTU	0.427	0.162	0.103	0.120	18.90%

Table 7.12: The performance of a classifier trained on the 21 January dataset when classifying the 4 February dataset with various padding-related countermeasures using the extra feature set. In this experiment the classifiers were trained on data on which the countermeasures were applied. These results show how well a countermeasure makes fingerprints homogeneous. The datasets used, are the same as in Table 7.11.

Fingerprint homogeneity

As previously noted, fingerprint homogeneity is a good measure of how well data can be classified. The goal of the countermeasures is to make the fingerprints as homogeneous as possible. To check the homogeneity of fingerprints we perform the Anderson Darling (AD) test on all possible two pairs of feature distributions. From the average of all these results, we get a sense of how similar the fingerprints of the classes in the dataset are. The padding countermeasures only change the feature distributions of the size-related features. Therefore we will only perform the AD test for those features. It is hard to put a number on the change in the fingerprint as a whole. The Random Forest classifier will simply deem features whose distributions are similar less important.

Countermeasure	Inter-Packet Size	Packet Size
None	0.032	0.019
Session Random	0.036	0.038
Packet Random	0.046	0.032
Linear	0.038	0.028
Exponential	0.040	0.027
Mice-Elephants	0.108	0.088
MTU	1.000	0.560
Random MTU	0.109	0.074

Table 7.13: Average similarity between feature distributions of the classes in the 4 February dataset for various padding schemes applied to the data. Only the similarities for Inter-Packet Size features and Packet Size features are displayed since the other features are not affected by the padding schemes. The closer a value is to 1, the higher the similarity between the feature distributions of the classes.

Table 7.13 shows how similar the fingerprint of one app is to the fingerprints of other apps. The numbers displayed are the significance levels as reported by the AD test. A significance level close to 1 means that the distributions are very similar and highly likely drawn from the same distribution. We clearly see how much the different padding schemes affect the homogeneity of the fingerprints. For the Session Random, Linear, and Exponential padding schemes the similarity between the feature distributions stays about the same. This is because these padding schemes keep a sense of small, medium, and large packets since Session Random adds static padding to the burst and the others essentially only round the packet lengths. The Packet Random scheme does add random noise to the packet length, but because it adds so little padding the original Inter-Packet Size (IPS) can still be guessed up to a size of 496 bytes. Random MTU does not suffer from this problem since it can add padding of up to MTU, therefore we see a higher similarity score for that scheme. Where Mice-Elephants significantly reduced the performance in Table 7.11, we now see that it does not introduce more homogeneity than Random MTU, because an order of small and large packets still exists in the data. Only padding to MTU makes the feature distributions of the IPS completely homogeneous since the IPS will always be 0.

A similar analysis holds for the Packet Size features. Only here we also have features related to the total size of all packets in a burst. Even with padding to MTU, this feature keeps it discriminatory power since the number of packets is not changed by the padding schemes. To create homogeneous feature distributions for these features as well, a countermeasure should be implemented that inserts a random number of packets into a burst.

Feature importance

The classifiers trained on countermeasures perform remarkably well. This means that different features are important for these classifiers since the features related to the packet size have changed significantly due to the padding. By looking at the feature importance of the classifiers we can see for which type of features a new countermeasure should be found. Since our per class classifier is a combination of many binary classifiers, we obtain the feature importance for each binary classifier and take the average of the importances.

In Table 7.14 the feature importances for various types of features can be found. We list the feature importances for each of the classifiers trained on data with a countermeasure and the classifier without countermeasure as a baseline. By looking at the importances we can see that as the discriminatory power of the size-related features (Inter-Packet Size and Packet Size) decreases, the classifier relies more heavily on time-related features and the destination name. For example, when padding to MTU, meaning that the Inter-Packet Size is always 0, the feature importance for the Inter-Packet Size is almost reduced to 0, while the importance of Inter-Arrival Time is 2.5 times higher than it is for the classifier without countermeasures. The same effect, although less drastic, can be seen for the other countermeasures.

Countermeasure	Inter-Packet Size	Inter-Arrival Time	Packet Size	Packet Count	Duration	Destination Name	Destination Port
None	0.2697	0.3178	0.2877	0.0069	0.0200	0.0963	0.0001
Session Random	0.2740	0.2918	0.2869	0.0073	0.0182	0.1204	0.0001
Packet Random	0.2604	0.2836	0.3042	0.0076	0.0174	0.1254	0.0001
Linear	0.1810	0.4987	0.1737	0.0079	0.0313	0.1058	0.0002
Exponential	0.1588	0.5428	0.1476	0.0076	0.0344	0.1075	0.0001
Mice-Elephants	0.0786	0.6614	0.0841	0.0100	0.0421	0.1226	0.0002
MTU	0.0010	0.7714	0.0208	0.0190	0.0494	0.1368	0.0002
Random MTU	0.2428	0.2764	0.3058	0.0086	0.0168	0.1483	0.0001

Table 7.14: Feature importance for the classifiers trained on the 21 January dataset with various countermeasure for the extra feature set.

7.2.2. Performance of classifiers with other countermeasures

An attacker can easily create many classifiers, all for different countermeasures, and simply take the results with the highest confidence as the true results. However, it might not even be required to train classifiers specifically for all countermeasures that are used. Since countermeasures often target the same features a classifier trained on data with one countermeasure applied will probably work quite well on data with similar countermeasures applied. Simply because the targeted feature has less of an effect on the outcome of the classifier as shown in Table 7.2.1.

To test this hypothesis we did a few tests by training a classifier on data with one classifier applied and using that classifier to classify data with another classifier applied. In Table 7.15 we show the results of these tests. We chose countermeasures that are similar to each other, for example, the *pad to MTU* scheme and the *Mice-Elephants* scheme. Both of these padding schemes often pad the packet to MTU, so the resulting feature distributions look similar. A similar case can be made for the other combinations. For the various combinations tried, we see that the performance is still very good. This makes sense since we saw in Table 7.14 that all of these classifiers rely less on the packet size, which is the feature that is changed.

Countermeasure training	Countermeasure validation	Accuracy	Macro-average Precision	Macro-average Recall	Macro-average F1-score
Session Random	Packet Random	0.440	0.169	0.110	0.125
Packet Random	Session Random	0.456	0.175	0.114	0.133
Exponential	Linear	0.400	0.178	0.098	0.115
Linear	Exponential	0.422	0.175	0.105	0.122
MTU	Mice-Elephants	0.414	0.166	0.102	0.118
MTU	Random MTU	0.418	0.165	0.103	0.119

Table 7.15: The performance of a classifier trained on the 21 January dataset with a countermeasure when classifying the 4 February dataset with a different countermeasure applied using the extra feature set. These results show that an attacker does not need to have an exhaustive set of classifiers for each countermeasure. It already helps significantly to have a classifier of a similar countermeasure.

7.2.3. Finding an effective countermeasure

The previous sections strongly show that padding is not an effective countermeasure against network traffic analysis. But since we have shown in chapter 6 that network traffic analysis is feasible in the real world it is important to find the requirements for a proper countermeasure. The goal of a countermeasure is to make sure that the classifier does not correctly classify the network traffic. This can be achieved in two ways: Firstly, by creating homogeneous fingerprints. This results in the classifier basically assigning random classes to the network traffic. Secondly, by using traffic morphing to either mimic the fingerprint of other apps or creating

truly random fingerprints. This results in the classifier assigning the wrong class to the network traffic. In this section, we discuss both types of countermeasures and give pointers to what constitutes an effective countermeasure.

Creating homogeneous fingerprints

We have shown that using a countermeasure that affects only a single (type of) feature is not effective. subsection 7.1.6 shows that using a VPN From the results in Table 7.2.1, we see that once the fingerprints based on one type of feature becomes homogeneous, the classifier locks onto different features that still have discriminatory power. In the case of using padding, it means that the classifier locks onto the destination domain name and the time-related features.

We have seen in Table 7.14 that the destination domain name is actually pretty important for the classifier, so it is interesting to see what happens if that feature becomes homogeneous as well. This can easily be achieved in practice from a user perspective by using a proxy service as a VPN. All traffic is tunneled through the same point and any attacker between the user and the proxy will only see one destination domain name: the proxy. On the infrastructure side, hiding the domain name can be achieved through the use of DNS over HTTPS (DoH) [30] and Encrypted Server Name Indication (ESNI) [49], which encrypt the DNS request and the server name in the TLS handshake respectively. Neither is widely implemented yet, but both Mozilla Firefox and Google Chrome have started implementing these techniques [4, 50, 65], so they can be expected to be widely adopted by others in the near future as well. While the IP address remains visible and can still be used as a feature, it does help in reducing the performance of the classifier as previously shown in section 5.3, this confirms the need for these techniques in battling network traffic analysis.

To simulate a countermeasure that hides the destination domain name, the experiment from subsection 7.2.1 is rerun, only now with the basic feature set; leaving out the destination domain name and port. The results can be found in Table 7.16. Again we assume a smart adversary who trains classifiers on data with the countermeasures applied. To see the effect of combining both the padding countermeasure with countermeasures that hide the domain name the results should be compared to the results of the same experiment with the extra feature set as found in Table 7.12. For all the padding schemes a significant drop in performance can be observed. Especially for the padding schemes that work better the drop is also more significant. For example pad to MTU drops 22.6% in accuracy and 9.4% in F1-score, while for Session Random the accuracy drops 10.6% and the F1-score only 2.6%. This happens simply because, for the Session Random countermeasure, the size-related features still have significant discriminatory power, so the classifier relies less on the destination name as a feature.

Countermeasure	**Accuracy**	**Macro-average Precision**	**Macro-average Recall**	**Macro-average F1-score**
Session Random	0.407	0.192	0.110	0.133
Packet Random	0.297	0.136	0.067	0.084
Linear	0.373	0.168	0.093	0.114
Exponential	0.340	0.151	0.083	0.102
Mice-Elephants	0.251	0.111	0.053	0.067
MTU	0.195	0.092	0.038	0.049
Random MTU	0.194	0.087	0.036	0.045

Table 7.16: The performance of a classifier trained on the 21 January dataset when classifying the 4 February dataset with and without various padding-related countermeasures using the basic feature set. In this experiment the classifiers were trained on data on which the countermeasures were applied. These results show what is possible when applying both padding and a countermeasure which makes the destination domain name feature useless. This table shows that even when mainly time related features are left, the performance is still well above random guessing.

Even though the performance has been reduced by combining countermeasures against the size features and the destination name feature, it is still vastly above random guessing. The classifier now mainly relies on time-related features and the order of incoming and outgoing packets. To make the time-related features homogeneous for all classes as well the packets should be sent at regular intervals. This hides the reaction time of a server or client on an incoming packet. If both sides sent packets at regular intervals the order of incoming and outgoing packets cannot be used as a discriminating feature either.

A good countermeasure thus takes care of the discriminatory power of all possible features. Any performance better than random guessing means that analysis over a long amount of time will still give an attacker enough confidence that certain apps are being used. The only effective countermeasure that renders fingerprinting impossible is to sent same-sized packets at regular intervals to a proxy server. The normal traffic can

be completely hidden in this constant stream of packets. However, this is highly impractical. To maintain the same responsiveness of applications that people are used to requires a constant stream of packets that require the full bandwidth. Otherwise, the bandwidth usage will vary based on the application that is being used, leaking information. The overhead that this requires is unacceptable in practice. The bandwidth usage could also be lowered, but this results in higher latency for applications. While for some applications like e-mail this might not be a problem, this will be unacceptable for video calls. A countermeasure that reduces classifier performance to random guessing by making fingerprints homogeneous is thus infeasible in practice. Significantly reducing performance to impractical levels is possible, but will always come at the cost of a large overhead.

Traffic morphing
Instead of trying to make all fingerprints similar it is also possible to morph the traffic so a fingerprint is changed significantly. In section 7.1 we tested traffic morphing through the use of a VPN, and we saw that the fingerprints of apps are changed completely. However, the problem there is that fingerprints are changed in a deterministic way, and thus a classifier can be trained on the changed fingerprints. This thus requires to add randomness to the changing fingerprints, so that it is no longer possible to train a classifier on it. However, it is quite hard to do this in a way that does not leak any information without ending up with a very similar system as in the previous section with significant overhead.

A better solution is to morph the traffic to mimic the fingerprint of another app. This requires a system with knowledge about the fingerprints of many other apps. The system can use specific time delays, padding, and the insertion of extra packets to morph the traffic so it can exactly recreate the fingerprint of another (random) app while hiding the original communication. In theory, this is possible, but in practice, this can be quite challenging since it requires the system to find the best fitting alternate fingerprint for the current traffic on the fly.

Most research has focused on changing the feature distribution of packet sizes to resemble feature distributions of other apps [10, 64]. However, these methods only focus on changing the *feature distribution* of packet sizes, and thus leave out features like the number of packets, total size, and timings, making the countermeasure ineffective. Chan-Tin et al. [11] cluster websites with similar fingerprints together and create a new fingerprint for each cluster, leading to minimal overhead. While this is a deterministic change of fingerprints, it makes websites in the same cluster indistinguishable from each other. A technique similar to the proposition by Chan-Tin et al. [11], where multiple sets of homogeneous fingerprints are created using traffic morphing, is very promising since it can be a good countermeasure without adding too much overhead. However, a lot of care should be taken in selecting the sets of apps since those sets can still leak some information about the app that is being used.

For these types of countermeasures to work reliably, they should become part of an Internet standard. Only when many people use the same countermeasures it becomes effective. If app developers start implementing their own traffic morphing schemes, apps could still be classified purely based on the traffic morphing scheme they use. Furthermore, they should not have to deal with securing communication anyway. The risk of mistakes leading to information leakage when app developers start implementing traffic morphing schemes themselves is high. This is exactly why standards like TLS were created in the first place. As far as we know, no propositions have yet been made for a universal standard that implements traffic morphing as a countermeasure against network traffic analysis.

7.3. Conclusion

Emulator data was collected while a VPN service was running. Both the non-VPN-encapsulated traffic and the VPN-encapsulated traffic was collected. Using a mapper the flows and bursts from the non-VPN traffic could be reconstructed in the VPN traffic, so our classification method still works. Other methods like burst separation and analysis on a packet level do not require this mapper and have been shown only have slightly worse performance than our method on normal mobile network traffic, so similar results can be expected when applying those methods to VPN data.

By training on data that was collected without active VPN we test whether a VPN is a good countermeasure against network traffic analysis. The non-VPN-encapsulated traffic is still affected by the change in timings caused by an active VPN. By leveraging this characteristic we show that there is a strong correlation between the performance of the classifier and the Round-trip Time between the locations where the training and testing data was captured. This means that classifiers trained on data from one country will not work as in another country, as the fingerprints of apps are affected by the added latency.

Using the classification of the non-VPN-encapsulated traffic as a baseline we show that the performance of a classifier on VPN-encapsulated data is very poor. However, the performance is not reduced to random guessing as might have been expected. We show that the fingerprints do not become homogeneous. In fact, the low performance is caused by a shift in the fingerprints which causes a mismatch in the fingerprints between the training data and the testing data.

We show that the fingerprints are fairly consistent across the different VPN datasets. This suggests that training and testing on VPN data yield good results. We show this is indeed the case and can classify VPN data with an accuracy of up to 73% and a macro F1-score of 68%. This is comparable to the performance of training and testing on normal data. This result shows that using a VPN as a countermeasure against network traffic analysis does not work as the VPN does not make the traffic look homogeneous.

Using the insights from the experiments with the VPN, we show that padding schemes are not effective countermeasures either. By training on data with padding schemes applied, a classifier is still able to classify network traffic with much higher accuracy than random guessing. By combining countermeasures that target a different type of features the performance of a classifier can be reduced further. However, for reducing the performance of a classifier to random guessing further steps are required. We suggest two possible types of countermeasures. One with the focus of creating homogeneous feature distributions for all classes, however, this will either require a lot of bandwidth or will result in high latency. A more promising type of countermeasure revolves around traffic morphing. By making the traffic of one app look like another random app the classifier can be tricked. Another technique is to morph the traffic of a set of apps to one new fingerprint. To make these countermeasures effective they should be applied as a universal standard. Designing such a standard requires much more research into effective countermeasures with minimal overhead.

8

Conclusion

In this thesis, we investigate the feasibility of tracking user behavior on mobile phones through the analysis of encrypted network traffic. Previous work assumes that this is feasible, purely based on small controlled experiments. We show that classifying a live network is much more complicated than assumed by previous work and that the good results in small controlled experiments are caused by favorable circumstances. By analyzing the data, we provide insight into the circumstances required for successful fingerprinting of apps in encrypted network traffic. Using this, we can form realistic conclusions about the feasibility of fingerprinting of real-world traffic and the requirements for effective countermeasures.

Using emulators, we collect automatically generated labeled network traffic from various versions of around 500 apps. This data is first used to investigate the influence of changing app versions on the fingerprints of apps. Secondly, the data is used for investigating how different setups of the controlled experiments can influence the results. Through crowd-sourcing, we collect real-world network traffic from over 65 unique people. We investigate the differences between this real-world data and the data gathered using emulators to see how real-world classification differs from classification in controlled experiments and if easy training of models using emulators is possible. Lastly, we experiment with various countermeasures, like using a VPN and adding padding to packets, to see how well they work to prevent the fingerprinting of apps in encrypted network traffic.

8.1. Requirements for comparative experiments

One goal of this research is to provide a critical review of the results of related work and show how their results could be compared. Related work often draws conclusions based on superficial results without further analysis of these results. In this section, we give more context to the results of related work that were discussed in chapter 3. The main points to take away are that:

- the setup of experiments and the selection of app sets influences results significantly;
- domain knowledge is required to make the correct choices in features;
- the methodology should be very clear to do a comparative analysis between research.

Previous research [14, 62] assumes that because classification works in a controlled environment, it will also work in a real-world setting. We show that while the classification of real-world data is possible, it is more complicated than classification on automatically generated traffic. An attack that aims to classify all network traffic in the wild is infeasible since there are simply too many apps and too many similar fingerprints, and so the attack surface is smaller than suggested in previous research.

Not all previous research is clear on how they divide their training and testing sets. E.g. Conti et al. [14] uses different users for their training and testing data, Al-Naami et al. [1] randomly selects traces for training and testing, but Wang et al. [62] and Taylor et al. [58] simply state that they split their data into two sets. We show that the method of splitting data into a training and a testing set is important information, because as seen for example in chapter 6, a random subset of the data on a burst/flow level (cross-validation) often has much better performance than using two different datasets as training and testing data, or splitting the data, which is the more realistic setting. Bursts and flows that occur close together are more alike because the environment is completely the same. So when randomly selecting bursts or flows from the data you get a very nice subset of

the possible flows and bursts, however, this is not realistic, as in the real world setting you would encounter different data from different settings all the time.

In section 5.3 we have shown that using source ports as a feature is not a good idea. The source ports should be uniformly distributed, however, this only happens when there is a lot of training data for each app. When there is not enough training data, the classifier overfits on the source port. Because this feature is noise over the entire dataset the Random Forest cannot compensate for this overfitting, resulting in wrong classifications. Liu et al. [36] use source ports as a feature in their research and do not seem to be aware of this issue. This shows that specific domain knowledge is required to apply machine learning in network traffic analysis.

As previously discussed in chapter 3 and chapter 4 previous work often does not use the same measures for their results. This, combined with the problem that other parts of the methodology either differ a lot or are unclear, as discussed above, makes it hard to compare the results from this thesis to the results in previous work.

This thesis shows that it is very well possible to give deep insight into how machine learning can be used to analyze mobile phone network traffic. We show which characteristics of an app influence the performance of a classifier. This is done not by analyzing the Random Forests, but by analyzing the feature distributions of the data. This analysis method is applicable regardless of the classifier used, and thus easily transferable to (other) black-box machine learning models. Because of this understanding of how the data works, we can predict much better what will and will not work, instead of extrapolating conclusions from experiments only.

8.2. Research questions

The results found in this thesis allow us to answer the research questions that were proposed in chapter 1. The main research question is: *Is it feasible to track user behavior on the mobile phone platform through analysis of encrypted network traffic in a real-world environment?*. To formulate an answer to this question, sub-questions were formed that can be answered with various experiments. The answers to the sub-questions give us enough information to form an answer to the main research question.

8.2.1. Sub-questions

How does the number of apps under consideration affect the classifier?

A classifier needs to distinguish between the fingerprints of apps, the more apps are under consideration the higher the chance that fingerprints overlap and thus the higher the chance of misclassifications. As we have shown in subsection 5.4.3 the performance for two same-sized sets can differ immensely. In reality, it is not the number of apps that has an impact on the performance, but the level of homogeneity of the fingerprints. However, in practice, the number of apps indirectly impacts the performance, since for more apps it becomes increasingly difficult to find a subset with unique fingerprints.

What is the impact of app updates on classifier performance?

We considered a set of more than 500 randomly selected apps and their latest versions at two-week intervals. We trained on the data from the first versions and testing on the data from the subsequent versions. There is not a single type of feature that changes when app versions change, but rather the fingerprint as a whole drift from the training data, thus harming the performance of the classifier. The performance for various metrics drops linearly for a total of 7 percentage points over three months. The impact of app updates thus proves to be significant and classifiers should be updated every few months to keep a high performance.

How well can a classifier that is trained on automatically generated data classify real-world data?

Not all that well. This can be explained by the simple user input we emulate in our classifiers. We define three categories of network traffic of apps: *dynamic over time*, *dynamic between users*, and *static*. These different categories require different training methods. More advanced methods for user input can improve the performance of emulator training data on real-world data. But, while a classifier for an app in the *static* or *dynamic over time* category can be trained on emulator data, this does not work for an app whose content depends heavily on the user.

By comparing the fingerprints of different users for the same app and the fingerprints of several apps, we show that the performance of an app is dependent both on the spread of the fingerprint and on how close other fingerprints are. Therefore, heavily depending on the other apps under consideration, *dynamic user* apps can still be classified using training data obtained from an emulator with a dummy account. However, only real people can be used to obtain exhaustive training data.

The way the targeted apps generate their content is very important for whether automatic training using emulators works. However, without a per-app analysis of the functionality, there is no way of telling which categories of network traffic an app will generate. So while automatic training can work, selecting the apps on which it will work requires manual labor.

What is the impact of different network environments on classifier performance?
We have shown that a difference in the capture location of the training and testing set influences the performance of a classifier. The added latency changes the fingerprints of apps so they are harder to recognize. This change in fingerprints will only hold for apps that do not make use of multiple servers globally. The impact on the classifier performance thus depends on which apps are considered.

How can user behavior analysis through network traffic analysis be mitigated?
We have found no effective countermeasures against network traffic analysis. While countermeasures like using a VPN or packet padding do change the fingerprints of an app, we have shown that it does not make them homogeneous. We show that by training a classifier on network traffic on which a countermeasure is applied, we can still reliably classify network traffic. A successful countermeasure should thus aim to make the fingerprints homogeneous and should attack all features. However, achieving this without significant overhead can be very difficult and is an open challenge.

8.2.2. Main research question

Is it feasible to track user behavior on the mobile phone platform through analysis of encrypted network traffic in a real-world environment?
Yes, it is feasible to track user behavior, although it requires significant effort from the attacker. We have shown that it is possible to classify real-world data on a specific subset of apps with unique fingerprints. Reliably doing this is hard and requires a lot of knowledge about the apps that are under consideration. So while a general model that can classify thousands of apps is not realistic, an attacker that is only interested in a small subset of apps can be successful.

An attacker should pay attention that they collect enough training data to build a complete model of all possible network traffic an app can generate. Collecting enough training data can be done using crowdsourcing, although this requires significant resources. The automatic generation of training data shows potential but is currently not adequate. The model should be kept up to date because as apps update, the network fingerprints change as well.

8.3. Limitations

In this section, we highlight some limitations of the research and explain how they might have influenced the results. To automatically collect our data, we use emulators with random user input to perform actions that generate network traffic. This will reach most of the parts of the apps that can be accessed by tapping buttons. However, it is not a good emulation of more complex user input, like text or a specific combination of actions that will rarely be performed by random user input. Furthermore, all traffic is collected from a fresh install of an app. This limits the diversity of the network traffic per app. As seen from the real-world data, the network traffic generated per user can differ significantly. This is the reason why our emulator generated traffic cannot be used for classification of real-world data; much of the traffic that occurs due to more complex user input is not in our data. The simple nature of the user input for our emulators does not invalidate the results, since from chapter 6 we have seen that our results transfer well to more diverse data.

As mentioned in subsection 4.1.1, about half of the apps in the Play Store are only built for the ARM architecture and cannot be run on our emulators. Automatic trace collection for these ARM-only apps requires dedicated hardware like a mobile phone or a Raspberry Pi. We have shown that from our set of apps the ARM apps are updated slightly more often than the x86 apps, making our results slightly more conservative than when considering all the apps. Because we do not know if there is any significant difference in the network fingerprints of the ARM apps compared to the x86 apps, we cannot say for sure whether our results are representative for the top 10 000 of the Play Store. However, it is unlikely that the ARM apps will show very different results since there is no evidence that there is a significant bias in the content of these apps compared to the x86 apps. The performance degradation of a classifier when encountering newer app versions is thus expected to be present for ARM apps as well.

There can be months between the app version and the collection time of a trace on that app version. All the traces were collected in a relatively short time. There is no evidence that this influenced the outcomes

of the experiments significantly. However, it is a weakness in the research since the content of some apps is updated without requiring a new version in the Play Store. Furthermore, we did not specifically verify how much the effect of different collection times is on the drift of app fingerprints.

8.4. Future work

For a thesis, there is only limited time so there are still a lot of interesting questions left in the area of classifying mobile phone network traffic.

At the start of this thesis, some preliminary tests showed that a Convolutional Neural Network (CNN) performs rather well on the data, although worse than a Random Forest for us. It would be interesting to see whether experts in deep learning can create CNNs that work better than other classifiers previously considered.

This thesis has not shown that it is possible to train a classifier on emulator generated data and classify real-world data. However, there is potential for this to work. Further efforts to enhance the automated user input for emulators are required to show whether the training of classifiers can be largely automated, which would make tracking behavior much more feasible.

Some work has been done in the area of classifying user actions within apps. It is very interesting to do similar analyses as done in this thesis on this deeper level of classification to get a thorough insight into the difficulties of classifying user actions and the feasibility of user action classification in the real world.

While this thesis does some experiments on real-world data and gives pointers to which attacks can and cannot work, experimental confirmation of the effectiveness of these attacks in the real world is still required. A more extensive analysis of the type of attackers that are capable of performing these attacks and the alternatives that exist to reach the same goals is required to develop a proper threat profile.

Lastly, more research into countermeasures against network traffic analysis is required. This is hard as countermeasures often also introduce a significant overhead, which makes them unattractive to implement. Traffic morphing is an attractive countermeasure since it can confuse classifiers without introducing much overhead and is robust against training on countermeasure data. More research into applying this technique effectively on a large scale is required. Once proper countermeasures are found, they should be incorporated into the standards of the Internet Engineering Task Force (IETF) so people can use the Internet without having to worry about their protecting their privacy.

A

Collection app instructions

Investigating tracking of android users using encrypted network traffic analysis

Master thesis project
Wilko Meijer

Research description
Nowadays almost all data that is sent over the Internet is encrypted. This encryption protects the content from being inspected by third parties. However, even though the content is not visible, there is still metadata available to third parties that might provide insights into the behavior of users. In this research we investigate whether it is possible to identify which apps people are using only by looking at the encrypted network traffic their mobile phones produce. When proven feasible we will look into countermeasures against this type of analysis.

To train the models used for analysing the network traffic we need to obtain a ground truth of data. Therefore we want to obtain network traffic from real users for which we already know from which mobile app it originates. To assist in the experiment we've created an app that allows for easy participation in this experiment.

How does the app work?
The experiment app tunnels traffic through a VPN to a TUDelft server. A log of the metadata of the tunnelled traffic is kept on this server.

At the same time the experiment app logs which apps are used on the device. When you stop the experiment, this log is uploaded to our servers.

Once the log is uploaded, we can combine it with the logged metadata to get a dataset of metadata labelled with the apps it belongs to.

Which data is used by us?
We only use metadata of the encrypted network traffic. We cannot see the contents of your messages. The destination, size, duration and timestamps of network traffic are used. So we are able to see which websites or services are connected to and which apps you use, but not which pages you visit or what you do on the apps. We are not able to link the network traffic to you.

Unencrypted traffic is also routed through our servers, but not stored or looked at.

What should you do?
Mainly you should use apps as you do normally. To help us collect more useful data you could try to use a wide variety of apps and limit the use of your browser. You can participate as short or as long as you like, from minutes to hours and split up over as many sessions as you like. However, to prevent any unforeseen errors and loss of data it is best to not leave the service on for longer than a day and to split it up into sessions. Lastly we ask you to please not do anything illegal; in general, but especially while participating in the experiment.

Installation and usage instructions
To install the app, become a tester and download it from the Google Playstore (https://play.google.com/apps/testing/nl.tudelft.ewi.androidapptracking).
The app is available for Android devices from Android 4.0.

Link to Playstore

After you have installed the app you will be asked to consent to the collection and processing of your network traffic and the data about your app usage. Now you are able to press the play button in the bottom right corner to start participating in the experiment. These two steps are highlighted in figure 1. The first time you do this you will get a request to confirm that you agree that your traffic is routed through our servers, as can be seen in figure 2. After you accept this a connection will be made to our servers and all traffic will be captured as previously explained. At the same time the app usage will be logged. While the experiment is running two notifications will be visible, as can be seen in figure 3, indicating that your traffic is tunnelled through the VPN and that the app usage is logged.

When you want to stop participating in the experiment press the stop button. The app usage logs will be uploaded and the phone's connection will work as before.

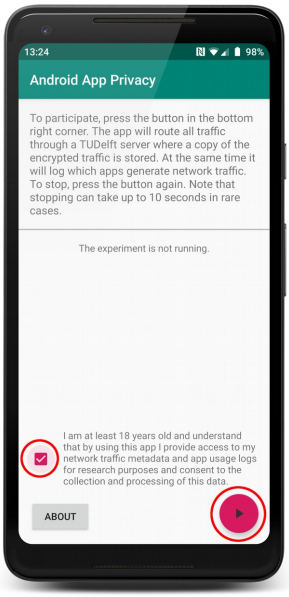


Figure 1: Circled: the steps to take to start participating in the experiment.

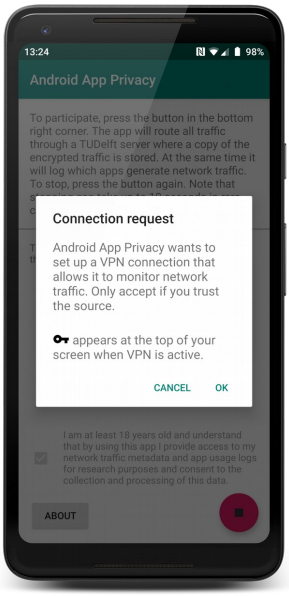


Figure 2: Permission request for VPN tunnel, only visible when first starting the app

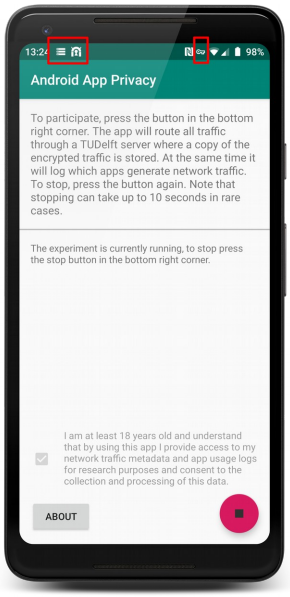


Figure 3: Within the red box the notifications can be seen that are visible when the experiment is running

B

Collection app screenshots

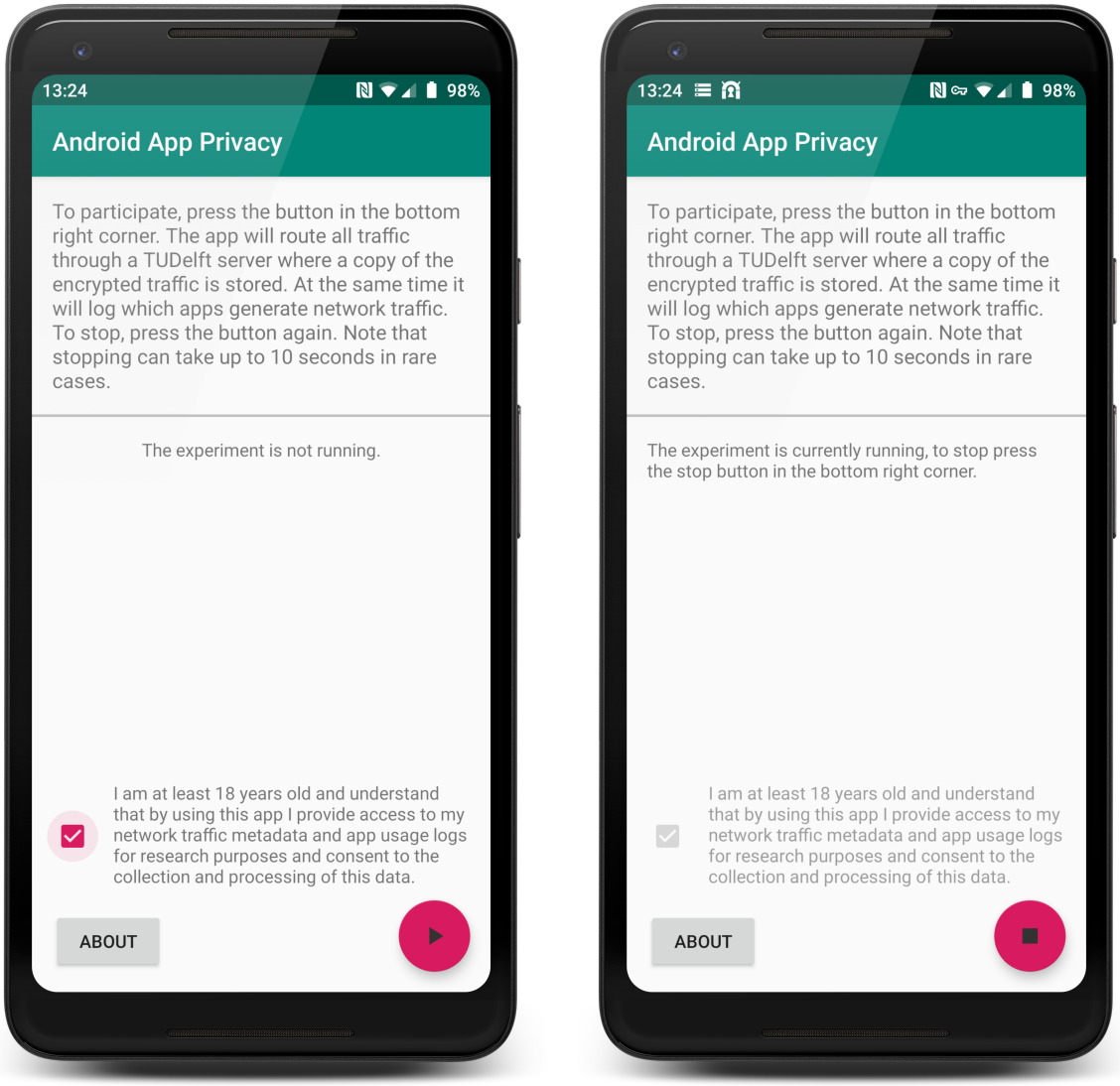


Figure B.1: Overview of the data collection app, the consent checkbox needs to be checked before a user can start participating in the experiment. Left, the view when the experiment is not running; right, the view when the experiment is running.

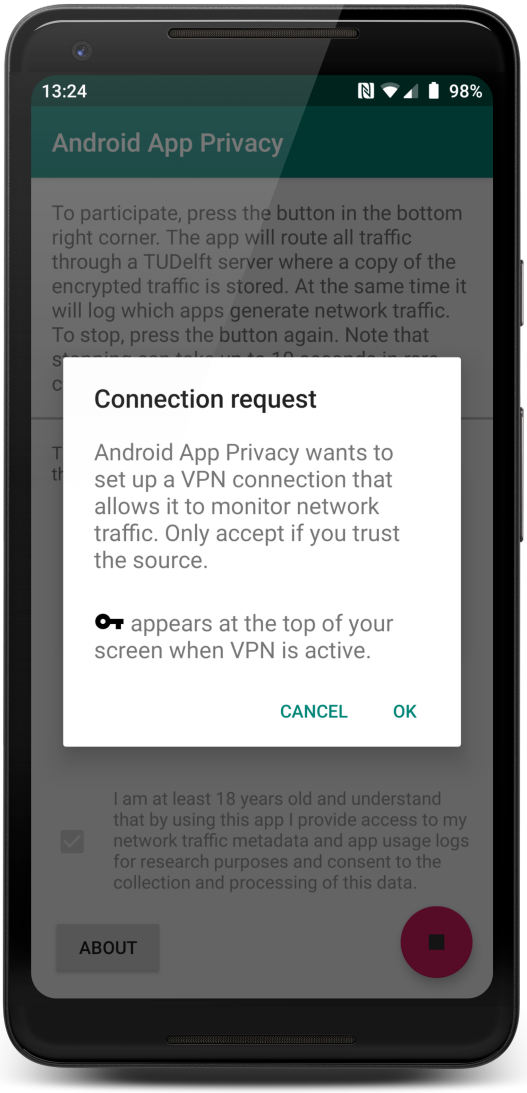


Figure B.2: Android OS notification informing users about the risks of using a VPN. This notification is only shown the first time a user connects to our VPN.

Figure B.3: Notifications displayed in the Status bar when the app is running.

C

Tables

Class	F1-score	# samples
com.tapinator.snow.jet.landing3d	0.013	131
com.natenai.glowhockey2	0.023	132
buba.electric.mobileelectrician	0.034	226
com.trimanstudio.JetFlightSimulator3d	0.043	164
com.innover.imagetopdf	0.055	119
com.appstar.callrecorder	0.056	176
com.tapinator.monstertruck.speedstunts	0.057	128
com.sbwebcreations.weaponbuilder	0.057	161
com.powernapps.acallfromsanta	0.063	112
com.droid.developer.pressure.measurement	0.086	105
com.ea.games.simsfreeplay_row	0.761	234
com.dts.freefireth	0.768	100
com.mtvn.Nickelodeon.GameOn	0.770	142
com.airbnb.android	0.770	126
com.ea.game.starwarscapital_row	0.772	103
com.ogqcorp.bgh	0.794	116
com.asus.ia.asusapp	0.801	101
net.peakgames.lostjewels	0.806	169
ba.lam.tribalmania.arabic	0.827	119
jp.ne.donuts.tanshanotora	0.842	164

Table C.1: Overview of the classes used in the t-SNE visualizations found in Figure 5.5 and Figure 5.6. This is a selection of the better and poorer performing classes when performing 5-fold cross validation on the 21 January dataset.

AppId	# bursts	Category	# reviews
com.android.chrome	5268	Web browser	16.3M
app.uid.shared	2156	Android utility	n/a
com.facebook.katana	1859	Social	90.3M
com.instagram.android	1390	Social	87.3M
com.android.vending	801	Android utility	n/a
com.whatsapp	667	Communication	95.5M
com.google.android.gm	501	Communication	6.1M
com.facebook.orca	442	Communication	68.3M
com.UCMobile.intl	435	Web browser	20.2M
com.sotg.surveys	430	Micropayments	69.0K
com.android.browser	425	Web browser	n/a
org.mozilla.firefox	390	Web browser	3.3M
org.telegram.messenger	386	Communication	4.0M
com.ibotta.android	336	Micropayments	417.7K
com.reddit.frontpage	335	News & Magazines	1.2M
com.cubegridltd.instagramdownloader	334	Tools	1
com.duolingo	315	Education	8.0M
com.snapchat.android	309	Social	20.1M
com.truecaller	308	Communication	10.9M
com.sec.android.app.sbrowser	292	Web browser	1.1M

Table C.2: List of the 20 most common apps in the collected real-world data and their corresponding category and popularity as measured by the number of reviews in the Play store. A relatively low amount of reviews for apps indicates a bias in the dataset.

AppId	Significance level	Closest neighbor
app.uid.shared	0.7653	com.puzzleenglish.main
com.android.chrome	0.7351	com.puzzleenglish.main
com.android.vending	0.8777	com.puzzleenglish.main
com.facebook.katana	0.7307	org.lineageos.updater
com.facebook.orca	0.6317	com.samsung.android.dialer
com.facebook.services	1.0000	org.lineageos.updater
com.google.android.apps.docs	0.7244	com.google.android.syncadapters.calendar
com.google.android.apps.maps	0.7290	com.google.android.apps.paidtasks
com.google.android.gm	0.6922	com.puzzleenglish.main
com.google.android.googlequicksearchbox	0.7667	net.androgames.level
com.google.android.music	1.0000	com.google.android.apps.tasks
com.google.android.videos	1.0000	com.prodege.answer
	1.0000	com.akbank.android.apps.akbank_direkt
	1.0000	com.mylol.spotafriend
	1.0000	com.sprint.ecid
	1.0000	com.puzzleenglish.main
	1.0000	com.samsung.android.app.reminder
com.google.android.youtube	0.5396	com.mylol.spotafriend
com.instagram.android	0.7233	com.sprint.ecid
com.linkedin.android	0.7787	com.podbean.app.podcast
com.netflix.mediaclient	0.8696	com.podbean.app.podcast
com.reddit.frontpage	0.7890	com.whatsapp.w4b
com.samsung.android.scloud	0.8787	com.podbean.app.podcast
com.sec.android.app.sbrowser	0.8553	com.samsung.android.oneconnect
com.sec.spp.push	0.9860	com.sec.android.app.samsungapps
com.snapchat.android	0.6970	org.lineageos.updater
com.twitter.android	0.8787	com.puzzleenglish.main
com.whatsapp	0.7630	com.sprint.ecid
org.telegram.messenger	0.7134	org.lineageos.updater

Table C.3: List of the closest neighbor for all apps under consideration in section 6.6. The closest neighbor is obtained by performing the 2-sample Anderson Darling test on the feature distributions of the app and all apps in the training set of the classifier. The significance level says how likely the two distributions are drawn from the same population. A high significance level indicates a high likely hood that they are drawn from the same population. When the fingerprints of two apps are close neighbors they are easily confused for each other, which harms the performance of the classifier. In case the significance level is the same for multiple apps, all those apps are displayed.

Dataset	Avg. # classes	Avg. Accuracy	Macro-avg Precision	Macro-avg Recall	Macro-avg F1-score
4 February	55	0.382	0.324	0.261	0.276
VPN BR	51	0.523	0.491	0.370	0.398
non-VPN BR	51	0.498	0.448	0.357	0.376
VPN IL	51	0.453	0.465	0.338	0.368
non-VPN IL	51	0.442	0.451	0.332	0.359
VPN IN	50	0.516	0.523	0.366	0.410
non-VPN IN	50	0.507	0.506	0.355	0.394
VPN IT	51	0.465	0.434	0.341	0.365
non-VPN IT	50	0.442	0.420	0.317	0.341
VPN JP	47	0.540	0.553	0.423	0.455
non-VPN JP	47	0.510	0.529	0.411	0.437
VPN NL	49	0.464	0.486	0.346	0.382
non-VPN NL	49	0.440	0.448	0.317	0.350
VPN UK	49	0.476	0.472	0.334	0.370
non-VPN UK	49	0.461	0.454	0.316	0.352

Table C.4: The baseline performance of the datasets used in the experiment to see the influence of using a VPN service on classifier performance. 5-fold cross-validation was performed on the datasets. For these datasets, a subset of features was used from which all features related to packet sizes and counts were excluded.

Dataset	Avg. # classes	Avg. Accuracy	Macro-avg Precision	Macro-avg Recall	Macro-avg F1-score
4 February	55	0.680	0.705	0.596	0.624
VPN BR	51	0.672	0.647	0.549	0.569
non-VPN BR	50	0.658	0.643	0.521	0.552
VPN IL	51	0.669	0.696	0.545	0.587
non-VPN IL	51	0.680	0.680	0.556	0.592
VPN IN	50	0.690	0.700	0.549	0.595
non-VPN IN	50	0.678	0.661	0.516	0.559
VPN IT	51	0.674	0.702	0.573	0.609
non-VPN IT	50	0.654	0.676	0.548	0.582
VPN JP	47	0.661	0.694	0.556	0.594
non-VPN JP	47	0.656	0.676	0.547	0.580
VPN NL	49	0.663	0.712	0.566	0.607
non-VPN NL	48	0.651	0.679	0.542	0.582
VPN TR	48	0.400	0.445	0.345	0.352
non-VPN TR	48	0.405	0.434	0.336	0.350
VPN UK	49	0.678	0.664	0.520	0.560
non-VPN UK	49	0.672	0.653	0.506	0.549

Table C.5: The baseline performance of the datasets used in the experiment to see the influence of using a VPN service on classifier performance. 5-fold cross-validation was performed on the datasets. For these datasets, a subset of features was used from which all features related to packet timings were excluded.

D

List of features

These features are extracted for each burst. When we are talking about all, outgoing, or incoming packets, we mean the packets that are contained in one burst.

Feature	Description
time	Timestamp of the first packet
src_ip	Source IP address
src_port	Source TCP port
dst_ip	Destination IP address
dst_name	Destination hostname
dst_port	Destination TCP port
com_burst_duration	Duration of the burst (all packets)
com_packet_count	Number of all packets
com_size_total	Total size of all packets
com_size_min	Minimum size of all packets
com_size_max	Maximum size of all packets
com_size_mean	Mean size of all packets
com_size_var	Variance in size of all packets
com_size_skewness	Skewness in distribution of sizes of all packets
com_size_kurtosis	Kurtosis in distribution of sizes of all packets
com_size_std	Standard deviation in size of all packets
com_size_mad	Median absolute deviation in size of all packets
com_size_sem	Standard error of the mean in size of all packets
com_size_10_percentile	10th percentile size of all packets
com_size_20_percentile	20th percentile size of all packets
com_size_30_percentile	30th percentile size of all packets
com_size_40_percentile	40th percentile size of all packets
com_size_50_percentile	50th percentile size of all packets
com_size_60_percentile	60th percentile size of all packets
com_size_70_percentile	70th percentile size of all packets
com_size_80_percentile	80th percentile size of all packets
com_size_90_percentile	90th percentile size of all packets
out_burst_duration	Duration of the burst (outgoing packets)
out_packet_count	Number of outgoing packets
out_size_total	Total size of outgoing packets
out_size_min	Minimum size of outgoing packets
out_size_max	Maximum size of outgoing packets
out_size_mean	Mean size of outgoing packets
out_size_var	Variance in size of outgoing packets
out_size_skewness	Skewness in distribution of sizes of outgoing packets
out_size_kurtosis	Kurtosis in distribution of sizes of outgoing packets
out_size_std	Standard deviation in size of outgoing packets
out_size_mad	Median absolute deviation in size of outgoing packets
out_size_sem	Standard error of the mean in size of outgoing packets
out_size_10_percentile	10th percentile size of outgoing packets
out_size_20_percentile	20th percentile size of outgoing packets
out_size_30_percentile	30th percentile size of outgoing packets
out_size_40_percentile	40th percentile size of outgoing packets
out_size_50_percentile	50th percentile size of outgoing packets
out_size_60_percentile	60th percentile size of outgoing packets

Feature	Description
out_size_70_percentile	70th percentile size of outgoing packets
out_size_80_percentile	80th percentile size of outgoing packets
out_size_90_percentile	90th percentile size of outgoing packets
in_burst_duration	Duration of the burst (incoming packets)
in_packet_count	Number of incoming packets
in_size_total	Total size of incoming packets
in_size_min	Minimum size of incoming packets
in_size_max	Maximum size of incoming packets
in_size_mean	Mean size of incoming packets
in_size_var	Variance in size of incoming packets
in_size_skewness	Skewness in distribution of sizes of incoming packets
in_size_kurtosis	Kurtosis in distribution of sizes of incoming packets
in_size_std	Standard deviation in size of incoming packets
in_size_mad	Median absolute deviation in size of incoming packets
in_size_sem	Standard error of the mean in size of incoming packets
in_size_10_percentile	10th percentile size of incoming packets
in_size_20_percentile	20th percentile size of incoming packets
in_size_30_percentile	30th percentile size of incoming packets
in_size_40_percentile	40th percentile size of incoming packets
in_size_50_percentile	50th percentile size of incoming packets
in_size_60_percentile	60th percentile size of incoming packets
in_size_70_percentile	70th percentile size of incoming packets
in_size_80_percentile	80th percentile size of incoming packets
in_size_90_percentile	90th percentile size of incoming packets
com_iat_min	Minimum Inter Arrival Time between all packets
com_iat_max	Maximum Inter Arrival Time between all packets
com_iat_mean	Mean Inter Arrival Time between all packets
com_iat_var	Variance in Inter Arrival Time between all packets
com_iat_skewness	Skewness in distribution of Inter Arrival Time between all packets
com_iat_kurtosis	Kurtosis in distribution of Inter Arrival Time between all packets
com_iat_std	Standard deviation in Inter Arrival Time between all packets
com_iat_mad	Median absolute deviation in Inter Arrival Time between all packets
com_iat_sem	Standard error of the mean in Inter Arrival Time between all packets
com_iat_10_percentile	10th percentile Inter Arrival Time between all packets
com_iat_20_percentile	20th percentile Inter Arrival Time between all packets
com_iat_30_percentile	30th percentile Inter Arrival Time between all packets
com_iat_40_percentile	40th percentile Inter Arrival Time between all packets
com_iat_50_percentile	50th percentile Inter Arrival Time between all packets
com_iat_60_percentile	60th percentile Inter Arrival Time between all packets
com_iat_70_percentile	70th percentile Inter Arrival Time between all packets
com_iat_80_percentile	80th percentile Inter Arrival Time between all packets
com_iat_90_percentile	90th percentile Inter Arrival Time between all packets
out_iat_min	Minimum Inter Arrival Time between outgoing packets
out_iat_max	Maximum Inter Arrival Time between outgoing packets
out_iat_mean	Mean Inter Arrival Time between outgoing packets
out_iat_var	Variance in Inter Arrival Time between outgoing packets
out_iat_skewness	Skewness in distribution of Inter Arrival Time between outgoing packets
out_iat_kurtosis	Kurtosis in distribution of Inter Arrival Time between outgoing packets
out_iat_std	Standard deviation in Inter Arrival Time between outgoing packets
out_iat_mad	Median absolute deviation in Inter Arrival Time between outgoing packets
out_iat_sem	Standard error of the mean in Inter Arrival Time between outgoing packets
out_iat_10_percentile	10th percentile Inter Arrival Time between outgoing packets
out_iat_20_percentile	20th percentile Inter Arrival Time between outgoing packets
out_iat_30_percentile	30th percentile Inter Arrival Time between outgoing packets
out_iat_40_percentile	40th percentile Inter Arrival Time between outgoing packets
out_iat_50_percentile	50th percentile Inter Arrival Time between outgoing packets
out_iat_60_percentile	60th percentile Inter Arrival Time between outgoing packets
out_iat_70_percentile	70th percentile Inter Arrival Time between outgoing packets
out_iat_80_percentile	80th percentile Inter Arrival Time between outgoing packets
out_iat_90_percentile	90th percentile Inter Arrival Time between outgoing packets
in_iat_min	Minimum Inter Arrival Time between incoming packets
in_iat_max	Maximum Inter Arrival Time between incoming packets
in_iat_mean	Mean Inter Arrival Time between incoming packets
in_iat_var	Variance in Inter Arrival Time between incoming packets
in_iat_skewness	Skewness in distribution of Inter Arrival Time between incoming packets
in_iat_kurtosis	Kurtosis in distribution of Inter Arrival Time between incoming packets
in_iat_std	Standard deviation in Inter Arrival Time between incoming packets
in_iat_mad	Median absolute deviation in Inter Arrival Time between incoming packets
in_iat_sem	Standard error of the mean in Inter Arrival Time between incoming packets

Feature	Description
in_iat_10_percentile	10th percentile Inter Arrival Time between incoming packets
in_iat_20_percentile	20th percentile Inter Arrival Time between incoming packets
in_iat_30_percentile	30th percentile Inter Arrival Time between incoming packets
in_iat_40_percentile	40th percentile Inter Arrival Time between incoming packets
in_iat_50_percentile	50th percentile Inter Arrival Time between incoming packets
in_iat_60_percentile	60th percentile Inter Arrival Time between incoming packets
in_iat_70_percentile	70th percentile Inter Arrival Time between incoming packets
in_iat_80_percentile	80th percentile Inter Arrival Time between incoming packets
in_iat_90_percentile	90th percentile Inter Arrival Time between incoming packets
com_ips_min	Minimum Inter Packet Size between all packets
com_ips_max	Maximum Inter Packet Size between all packets
com_ips_mean	Mean Inter Packet Size between all packets
com_ips_var	Variance in Inter Packet Size between all packets
com_ips_skewness	Skewness in distribution of Inter Packet Size between all packets
com_ips_kurtosis	Kurtosis in distribution of Inter Packet Size between all packets
com_ips_std	Standard deviation in Inter Packet Size between all packets
com_ips_mad	Median absolute deviation in Inter Packet Size between all packets
com_ips_sem	Standard error of the mean in Inter Packet Size between all packets
com_ips_10_percentile	10th percentile Inter Packet Size between all packets
com_ips_20_percentile	20th percentile Inter Packet Size between all packets
com_ips_30_percentile	30th percentile Inter Packet Size between all packets
com_ips_40_percentile	40th percentile Inter Packet Size between all packets
com_ips_50_percentile	50th percentile Inter Packet Size between all packets
com_ips_60_percentile	60th percentile Inter Packet Size between all packets
com_ips_70_percentile	70th percentile Inter Packet Size between all packets
com_ips_80_percentile	80th percentile Inter Packet Size between all packets
com_ips_90_percentile	90th percentile Inter Packet Size between all packets
out_ips_min	Minimum Inter Packet Size between outgoing packets
out_ips_max	Maximum Inter Packet Size between outgoing packets
out_ips_mean	Mean Inter Packet Size between outgoing packets
out_ips_var	Variance in Inter Packet Size between outgoing packets
out_ips_skewness	Skewness in distribution of Inter Packet Size between outgoing packets
out_ips_kurtosis	Kurtosis in distribution of Inter Packet Size between outgoing packets
out_ips_std	Standard deviation in Inter Packet Size between outgoing packets
out_ips_mad	Median absolute deviation in Inter Packet Size between outgoing packets
out_ips_sem	Standard error of the mean in Inter Packet Size between outgoing packets
out_ips_10_percentile	10th percentile Inter Packet Size between outgoing packets
out_ips_20_percentile	20th percentile Inter Packet Size between outgoing packets
out_ips_30_percentile	30th percentile Inter Packet Size between outgoing packets
out_ips_40_percentile	40th percentile Inter Packet Size between outgoing packets
out_ips_50_percentile	50th percentile Inter Packet Size between outgoing packets
out_ips_60_percentile	60th percentile Inter Packet Size between outgoing packets
out_ips_70_percentile	70th percentile Inter Packet Size between outgoing packets
out_ips_80_percentile	80th percentile Inter Packet Size between outgoing packets
out_ips_90_percentile	90th percentile Inter Packet Size between outgoing packets
in_ips_min	Minimum Inter Packet Size between incoming packets
in_ips_max	Maximum Inter Packet Size between incoming packets
in_ips_mean	Mean Inter Packet Size between incoming packets
in_ips_var	Variance in Inter Packet Size between incoming packets
in_ips_skewness	Skewness in distribution of Inter Packet Size between incoming packets
in_ips_kurtosis	Kurtosis in distribution of Inter Packet Size between incoming packets
in_ips_std	Standard deviation in Inter Packet Size between incoming packets
in_ips_mad	Median absolute deviation in Inter Packet Size between incoming packets
in_ips_sem	Standard error of the mean in Inter Packet Size between incoming packets
in_ips_10_percentile	10th percentile Inter Packet Size between incoming packets
in_ips_20_percentile	20th percentile Inter Packet Size between incoming packets
in_ips_30_percentile	30th percentile Inter Packet Size between incoming packets
in_ips_40_percentile	40th percentile Inter Packet Size between incoming packets
in_ips_50_percentile	50th percentile Inter Packet Size between incoming packets
in_ips_60_percentile	60th percentile Inter Packet Size between incoming packets
in_ips_70_percentile	70th percentile Inter Packet Size between incoming packets
in_ips_80_percentile	80th percentile Inter Packet Size between incoming packets
in_ips_90_percentile	90th percentile Inter Packet Size between incoming packets

E

List of apps

This appendix contains a complete list of all apps used in this research and in which datasets they are included. The displayed app names are the app names as are available from the Google Play Store. If an app is not available in the Google Play Store, no name is included in the table.

The datasets are defined as follows:

A. 21 January

B. 4 February

C. 15 February

D. 4 March

E. 18 March

F. 1 April

G. 18 April

H. 29 April

I. Real world

J. VPN (50 apps randomly selected from the 4 February dataset)

App Id	**App name**	**Dataset**									
		A	B	C	D	E	F	G	H	I	J
air.bg.lan.Monopoli	Rento - Dice Board Game Online	✓									
air.com.boostr.Air	Urban Rivals	✓		✓			✓		✓		
air.com.peoresnada.casimillonario	Almost Millionaire	✓	✓		✓			✓			
air.com.tensquaregames.letsfish	Let's Fish: Sport Fishing Games. Fishing Simulator	✓		✓		✓	✓				
app.buzz.share	Helo - Best Interest-based Community App									✓	
app.uid.shared										✓	
ar.com.develup.pasapalabra	Alphabet Game	✓							✓		
au.gov.bom.metview	BOM Weather									✓	
ba.lam.tribalmania.arabic	Tribal Mania	✓									
bbc.iplayer.android	BBC iPlayer	✓				✓	✓	✓			
block.app.wars	Block City Wars: Pixel Shooter with Battle Royale	✓									
br.com.band.guiatv	BAND	✓									
br.com.escolhatecnologia.vozdonarrador	Narrator's Voice	✓			✓	✓	✓				
br.com.globosat.android.sportvplay	SporTV Play	✓									

App Id	App name	Dataset									
		A	B	C	D	E	F	G	H	I	J
br.com.tapps.deardiary	Dear Diary - Teen Interactive Story Game	✓									
br.com.tapps.dinoquest	Dino Quest - Dinosaur Discovery and Dig Game	✓	✓								✓
br.com.tapps.fluffyshuffle	Fluffy Shuffle - Cute Match-3 Puzzle Adventure	✓									
br.com.tapps.foxevolution	Fox Evolution - The Secret of The Mutant Foxes	✓									
br.com.tapps.leagueofgamers	League of Gamers - Be an E-Sports Legend!	✓			✓		✓				
br.com.tapps.myboo	My Boo - Your Virtual Pet Game	✓		✓							
br.com.tapps.myicecreamtruck	My Ice Cream Truck - Make Sweet Frozen Desserts	✓									
br.com.tapps.mypizzashop	My Pizza Shop - Italian Pizzeria Management Game	✓									
buba.electric.mobileelectrician	Mobile electrician	✓	✓				✓				✓
cc.pacer.androidapp	Pedometer - Step Counter, Weight & Calorie Tracker									✓	
chat.rocket.android	Rocket.Chat									✓	
chessfriends.online.chess	Chess Online	✓		✓		✓					
club.fromfactory	Club Factory - Online Shopping App		✓		✓		✓	✓	✓	✓	✓
cn.wps.moffice_eng	WPS Office - Word, Docs, PDF, Note, Slide & Sheet									✓	
cn.xender	Xender - Share Music&Video,Share Photo,Share File									✓	
coconut.island.nk	Naughty Kitties - Cats Battle	✓									
com.accuweather.android	AccuWeather: Live local weather forecast & alerts	✓	✓		✓	✓		✓	✓	✓	✓
com.aceviral.angrygrantoss	Angry Gran Toss	✓									
com.acidcousins.chilly	Chilly Snow	✓									
com.acidcousins.fdunk	Flappy Dunk	✓			✓						
com.adda247.app	IBPS PO, IBPS Clerk, SSC CGL, Railway RRB & SBI PO									✓	
com.adobe.reader	Adobe Acrobat Reader: PDF Viewer, Editor & Creator									✓	
com.Ahmad.Kzb		✓									
com.airbnb.android	Airbnb	✓	✓	✓	✓	✓	✓	✓	✓		✓
com.akbank.android.apps .akbank_direkt	Akbank									✓	
com.alibaba.aliexpresshd	AliExpress - Smarter Shopping, Better Living									✓	
com.alibaba.intl.android.apps .poseidon	Alibaba.com - Leading online B2B Trade Marketplace									✓	
com.allocine.androidapp	AlloCine	✓	✓	✓	✓		✓	✓			
com.almondstudio.animalquiz	Scratch and guess the animal	✓			✓				✓		
com.alohamobile.browser	Aloha Browser - private fast browser with free VPN									✓	
com.alper.okey101.hd	101 Okey HD İnternetsiz - Yüzbir Okey HD	✓	✓		✓						
com.amazon.mp3	Amazon Music	✓	✓			✓			✓	✓	✓
com.amazon.mShop.android										✓	
com.amazon.mShop.android .shopping	Amazon Shopping - Search Fast, Browse Deals Easy									✓	
com.amphibius.prison_break_free	Prison Break: Lockdown (Free)	✓									
com.amvg.hemlak	Hürriyet Emlak									✓	
com.andigames.dirtontires		✓									
com.android.browser										✓	
com.android.chrome	Google Chrome: Fast & Secure	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
com.android.email										✓	
com.android.gallery3d										✓	
com.android.thememanager										✓	
com.android.updater										✓	
com.android.vending		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
com.ankama.dofustouch	DOFUS Touch	✓	✓								
com.ankama.tactilwar	Tactile Wars	✓									

App Id	App name	Dataset									
		A	B	C	D	E	F	G	H	I	J
com.ansangha.drparking4	Dr. Parking 4	✓					✓	✓			
com.antivirus	AVG AntiVirus Free & Mobile Security, Photo Vault	✓	✓		✓			✓	✓		
com.antutu.ABenchMark	AnTuTu Benchmark	✓			✓		✓		✓		
com.app.downloadmanager	Downloader & Private Browser - Kode Browser	✓		✓			✓	✓			
com.applepie4.mylittlepet.en	Hellopet - Cute cats, dogs and other unique pets	✓		✓							
com.applicaster.caracoltv	Caracol Televisión	✓									
com.appstar.callrecorder	Automatic Call Recorder	✓				✓		✓	✓		
com.apusapps.tools.unreadtips	APUS Message Center—Intelligent management	✓				✓					
com.artificialsolutions.teneo.va.prod	Lyra Virtual Assistant	✓									
com.aspiro.tidal	TIDAL Music - Hifi Songs, Playlists, & Videos									✓	
com.asus.filemanager	File Manager	✓									
com.asus.ia.asusapp	MyASUS - Service Center	✓						✓			
com.att.iqi										✓	
com.audible.application	Audiobooks from Audible	✓	✓		✓			✓			✓
com.aura.oobe.samsung										✓	
com.autoscout24	AutoScout24 - used car finder									✓	
com.avito.android		✓	✓	✓	✓	✓	✓	✓	✓		✓
com.azumio.android.lifecoin	LifeCoin - Rewards for Walking & Step Counting									✓	
com.bauermedia.tvmovie	TV Movie - TV Programm	✓					✓				
com.baviux.voicechanger	Voice changer with effects	✓			✓	✓		✓			
com.bazon.stickmanbowmasters	Stickman Bow Masters: Archers Bloody Arena	✓									
com.beatgridmedia.panelsync .mediarewards	Media Rewards									✓	
com.bendingspoons.live.quiz	Live Quiz - Win Real Prizes									✓	
com.berobo.android.scanner	Police Scanner	✓									
com.bestcoolfungames.antsmasher	Ant Smasher	✓		✓							
com.bhimaapps .mobilenumbertraker	Mobile Number Call Tracker	✓				✓					
com.binnazabla.kahvefali	Binnaz - Fortune Teller	✓	✓	✓	✓		✓		✓		
com.binteraktive.kniffel.live	Dice Clubs	✓	✓	✓		✓			✓		
com.blayzegames.iosfps	Bullet Force	✓	✓		✓	✓		✓			✓
com.blizzard.bma	Blizzard Authenticator	✓									
com.booking	Booking.com: Hotels, Apartments & Accommodation	✓	✓		✓	✓	✓	✓			
com.boukhatem.app.android .soundeffects	Sound Effects	✓									
com.Bow3.TheClaw	Clawbert						✓	✓			
com.brave.browser	Brave Privacy Browser: Fast, safe, private browser									✓	
com.budgestudios .HelloKittyNailSalon	Hello Kitty Nail Salon							✓			
com.buildingcraftinggames .block.girls.craft.sims.minecraft .fashion.dream.house.design.build .exploration.crafting.and.building	Dream House Craft: Design & Block Building Games	✓									
com.buildingcraftinggames.warcraft .loot.mine.craft.magic.hero.fantasy .hobbit.princess.quest.knight.castle .dragon.sword.dungeons	Fantasy Craft: Kingdom Builder	✓									
com.bumble.app	Bumble — Date. Meet Friends. Network.									✓	
com.bunstudio .superadventurefteddy	Super Adventures of Teddy	✓									
com.bushiroad.en.bangdreamgbp	BanG Dream! Girls Band Party!						✓		✓		
com.busuu.android.enc	Busuu: Learn Languages - Spanish, English & More									✓	
com.byeline.hackex	Hack Ex - Simulator	✓									
com.byril.seabattle	Sea Battle	✓									
com.byril.seabattle2	Sea Battle 2	✓						✓			
com.callpod.android_apps.keeper	Password Manager - Keeper	✓	✓		✓						

App Id	App name	Dataset									
		A	B	C	D	E	F	G	H	I	J
com.campmobile.snow	SNOW	✓	✓		✓	✓	✓	✓			✓
com.cashitapp.app.jokesphone	JokesPhone - Joke Calls	✓					✓				
com.cgv.android.movieapp	CGV	✓			✓				✓		
com.chase.sig.android	Chase Mobile	✓			✓			✓			
com.chess	Chess · Play & Learn	✓	✓	✓			✓				
com.chillingo.warfriends.android.gplay	WarFriends: PvP Shooter Game	✓	✓				✓				
com.chimbori.hermitcrab	Hermit • Lite Apps Browser	✓						✓			
com.cleanmaster.mguard	Clean Master - Antivirus, Applock & Cleaner									✓	
com.clickworker.clickworkerapp	clickworker									✓	
com.clue.android	Period Tracker Clue - Ovulation and Cycle Calendar									✓	
com.cmcm.live	LiveMe - Video chat, new friends, and make money									✓	
com.cmtelecom.texter										✓	
com.cocoplay.highschoolcrush	High School Crush - First Love	✓									
com.cocoplay.shoppinggirl	Shopping Mall Girl - Dress Up & Style Game	✓									
com.coloros.pictorial										✓	
com.coloros.weather.service										✓	
com.com2us.acefishing.normal.freefull.google.global.android.common	Ace Fishing: Wild Catch	✓	✓					✓			
com.com2us.enjoyyut.kakao.freefull.google.global.android.common		✓				✓					
com.com2us.smon.normal.freefull.google.kr.android.common	Summoners War	✓	✓	✓	✓	✓	✓	✓	✓		
com.contextlogic.wish	Wish - Shopping Made Fun	✓	✓	✓	✓	✓	✓	✓	✓		✓
com.crayzoo.zueiras	Zuapp	✓	✓								
com.creativemobile.DragRacing	Drag Racing	✓				✓					
com.criticalforceentertainment.criticalops	Critical Ops: Multiplayer FPS	✓	✓		✓	✓	✓	✓			
com.crowdstar.covetHome	Design Home	✓	✓		✓	✓		✓	✓		
com.cryptotab.android	CryptoTab Browser									✓	
com.csam.icici.bank.imobile	iMobile by ICICI Bank	✓			✓	✓		✓	✓		
com.cubegridltd.instagramdownloader	HD Downloader And Repost App for Instagram									✓	
com.cubegridltd.whatsappstatussaver	Status Saver for WhatsApp									✓	
com.cyberpony.stickman.warriors.epic		✓									
com.cygames.Shadowverse	Shadowverse CCG	✓	✓		✓						✓
com.cyou.privacysecurity	LOCX Applock Lock Apps & Photo	✓									
com.d2l.brightspace.student.android	Brightspace Pulse									✓	
com.dankolab.hotelsilence	Hotel Silence	✓									
com.daon.daonlab	DaonCollect									✓	
com.dencreak.dlcalculator	ClevCalc - Calculator	✓	✓	✓	✓		✓	✓	✓		
com.developandroid.android.girls	Princess memory game for kids	✓									
com.dianxinos.optimizer.duplay		✓	✓					✓			
com.digibites.calendar	DigiCal Calendar Agenda	✓									
com.dino.simulator.free	Dinosaur Simulator	✓									
com.discord	Discord - Chat for Gamers	✓	✓		✓	✓	✓		✓		
com.discovery.realscaryspiders		✓									
com.dollargeneral.android	Dollar General - Digital Coupons, Ads And More									✓	
com.dowino.ABlindLegend	A Blind Legend	✓									
com.dragonplay.dragonplaypoker	Dragonplay™ Poker Texas Holdem	✓							✓		
com.droid.developer.pressure.measurement	Fingerprint Lock Screen Prank	✓									
com.droidhen.car3d	Car Conductor: Traffic Control	✓									
com.droidhen.irunner	iRunner	✓									
com.dropbox.android	Dropbox	✓	✓		✓	✓	✓		✓	✓	
com.dts.freefireth	Garena Free Fire: Spooky Night		✓				✓				
com.duapps.recorder		✓	✓		✓	✓	✓	✓		✓	✓

App Id	App name	Dataset									
		A	B	C	D	E	F	G	H	I	J
com.duolingo	Duolingo: Learn Languages Free	✓	✓	✓	✓	✓	✓	✓	✓	✓	
com.dynamicgames .worldtruckdrivingsimulator	World Truck Driving Simulator	✓		✓	✓	✓	✓		✓		
com.ea.game.easportsufc_row	EA SPORTS UFC®	✓		✓			✓				
com.ea.game.nfs14_row	Need for Speed™ No Limits		✓		✓	✓		✓	✓		
com.ea.game.starwarscapital_row	Star Wars™: Galaxy of Heroes	✓			✓	✓					
com.ea.games.simsfreeplay_row	The Sims FreePlay	✓	✓			✓	✓				
com.easygame.marblelegend	Marble Legend	✓									
com.easytech.gogh	Glory of Generals HD	✓									
com.ebay.mobile	eBay: Online Shopping Deals - Buy, Sell, and Save									✓	
com.ehawk.antivirus.applock.wifi	Virus Cleaner - Antivirus, Cleaner & Booster	✓			✓		✓	✓			
com.elbylabs .brickbreakerrestructured	Brick Breaker Arcade	✓									
com.enflick.android.TextNow	TextNow: Free Texting & Calling App									✓	
com.enflick.android.tn2ndLine	2ndLine - Second Phone Number									✓	
com.estrongs.android.pop		✓	✓		✓	✓	✓	✓			
com.etermax.pictionary		✓									
com.etermax.preguntados.lite	Trivia Crack	✓	✓	✓	✓	✓	✓	✓	✓		✓
com.etsy.android.soe	Sell on Etsy									✓	
com.evenwell.AprUploadService										✓	
com.evil.cogs.game	Evil Cogs	✓									
com.extreme.motorbike.jump	Extreme Motorbike Jump 3D	✓									
com.eyewind.colorfit.mandala	Mandala Coloring Book	✓									
com.eytg.ssjj	Shooter Of War-FPS	✓									
com.f84games.survivalrun	Survival Run with Bear Grylls	✓									
com.facebook.appmanager										✓	
com.facebook.katana	Facebook									✓	
com.facebook.lite	Facebook Lite									✓	
com.facebook.mlite	Messenger Lite: Free Calls & Messages									✓	
com.facebook.orca	Messenger – Text and Video Chat for Free									✓	
com.facebook.services										✓	
com.faldiyari.apps.android	Fal Diyarı Kahve Falı	✓			✓		✓				
com.fatsecret.android	Calorie Counter by FatSecret									✓	
com.fdgentertainment.bananakong	Banana Kong	✓									
com.feelingtouch.strikeforce	Gun & Blood	✓									
com.fetchrewards.fetchrewards.hop	Fetch Rewards: Scan Receipts, Earn Gift Cards									✓	
com.flaregames.nskchuck	Nonstop Chuck Norris - RPG Offline Dungeon Crawler	✓						✓			
com.fluik.Explosions	Grumpy Bears	✓									
com.foap.android	Foap - sell your photos									✓	
com.fortune.astroguru	Astro Guru: Horoscope, Palmistry & Tarot Reading	✓		✓	✓	✓	✓	✓	✓		
com.freecharge.android	Freecharge - Recharges & Bills, UPI, Mutual Funds									✓	
com.freegamesstudio .thesurvivalhuntergames2	The Survival Hunter Games 2	✓			✓						
com.frograms.watcha	Watcha - Movies, TV Series Recommendation App	✓	✓	✓	✓	✓		✓			
com.frojo.bear	Happy Bear - Virtual Pet Game	✓									
com.frojo.chicken	My Chicken - Virtual Pet Game	✓									
com.frojo.cooking.android	Moy Restaurant Chef	✓									
com.frojo.moy3.android	Moy 3 Virtual Pet Game	✓									
com.ftw_and_co.happn	happn – Local dating app	✓	✓	✓		✓	✓	✓	✓		
com.fugugo.clashofclans .strategybases	Strategy Coc Base Layout	✓									
com.fullfat.android.agentdash	Agent Dash - Run Fast, Dodge Quick!	✓									
com.fungames.flightpilot	Flight Pilot Simulator 3D Free	✓	✓								✓
com.fungames.landsimww		✓									
com.funplus.kingofavalon	King of Avalon: Dragon War | Multiplayer Strategy									✓	

App Id	App name	Dataset									
		A	B	C	D	E	F	G	H	I	J
com.futurextending.taximetro	TAXIMETRO	✓									
com.g5e.paranormalsocietypg .android	The Paranormal Society: Hidden Object Adventure					✓		✓			
com.g5e.secretsociety	The Secret Society - Hidden Objects Mystery	✓	✓		✓	✓		✓			
com.gaana	Gaana Music- Hindi English Telugu MP3 Songs Online	✓				✓		✓			
com.gamebasics.osm	Online Soccer Manager (OSM) - 2019/2020	✓	✓	✓							
com.gamedevltd.modernstrike	Modern Strike Online: PRO FPS	✓	✓				✓		✓		
com.gameeapp.android.app	GAMEE - Play games with your friends							✓			
com.gamefirst.survivalsavage		✓									
com.gameinsight.tribez	The Tribez: Build a Village	✓	✓	✓	✓	✓	✓	✓	✓		
com.gameloft.android.ANMP .GloftDYHM	Disney Magic Kingdoms: Build Your Own Magical Park	✓	✓	✓			✓	✓			✓
com.gameloft.android.ANMP .GloftGHHM	March of Empires: War of Lords	✓		✓			✓	✓			
com.gameloft.android.ANMP .GloftGOHM	Gods of Rome	✓									
com.gameloft.android.ANMP .GloftSIHM		✓									
com.gameloft.android.ANMP .GloftSXHM	Six-Guns: Gang Showdown								✓		
com.gameloft.android.ANMP .GloftW2HM	War Planet Online: Global Conquest	✓			✓			✓			
com.gameone.RTOCard		✓			✓		✓				
com.gamesofa.android .gunsrushmobile		✓									
com.geargames.aow	Art of War 3: PvP RTS modern warfare strategy game	✓	✓		✓		✓		✓		
com.geargames.pfp		✓									
com.geaxgame.pokerkingpro	Texas Holdem Poker Pro	✓									
com.generagames.gladiatorheroes	Gladiator Heroes Clash: Fighting and Strategy Game	✓	✓	✓		✓					
com.generagames.resistance	Cover Fire: Offline Shooting Games	✓			✓		✓		✓		
com.gentoozero.memecreator	Meme Creator	✓									
com.gingersoftware.android .keyboard	Ginger Keyboard - Emoji, GIFs, Themes & Games	✓	✓	✓	✓		✓				✓
com.giu.diceme	Dice Me Online FREE	✓									
com.gmstdiostrtrck .truckdriving3droadspeddd	Truck Speed Driving 3D	✓									
com.gobrainfitness.durak	Durak	✓				✓					
com.gojek.app	Gojek - Ojek Taxi Booking, Delivery and Payment									✓	
com.gojek.life	GoLife | One App for Every Service Need									✓	
com.goodgamestudios.millennium	Empire: Millennium Wars	✓									
com.google.android.apps.adm	Google Find My Device	✓									
com.google.android.apps.ads .publisher	Google AdSense	✓									
com.google.android.apps.books	Google Play Books - Ebooks, Audiobooks, and Comics	✓						✓	✓		
com.google.android.apps.docs	Google Drive	✓								✓	
com.google.android.apps.docs .editors.docs	Google Docs									✓	
com.google.android.apps.docs .editors.sheets	Google Sheets									✓	
com.google.android.apps.docs .editors.slides	Google Slides									✓	
com.google.android.apps.maps	Maps - Navigate & Explore	✓	✓	✓				✓		✓	✓
com.google.android.apps.messaging	Messages	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
com.google.android.apps.nbu.files	Files by Google: Clean up space on your phone	✓	✓		✓		✓				
com.google.android.apps.paidtasks	Google Opinion Rewards									✓	
com.google.android.apps.photos	Google Photos									✓	

App Id	App name	Dataset									
		A	B	C	D	E	F	G	H	I	J
com.google.android.apps.searchlite	Google Go: A lighter, faster way to search									✓	
com.google.android.apps.tachyon	Google Duo - High Quality Video Calls		✓			✓		✓	✓		
com.google.android.apps.tasks	Google Tasks: Any Task, Any Goal. Get Things Done									✓	
com.google.android.apps.translate	Google Translate									✓	
com.google.android.apps.wallpaper	Wallpapers				✓		✓				✓
com.google.android.calendar	Google Calendar									✓	
com.google.android.contacts	Contacts									✓	
com.google.android.dialer	Phone	✓	✓	✓	✓	✓	✓	✓	✓		✓
com.google.android.gm	Gmail									✓	
com.google.android.gm.lite	Gmail Go									✓	
com.google.android .googlequicksearchbox	Google	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
com.google.android.ims	Carrier Services									✓	
com.google.android.inputmethod .latin	Gboard - the Google Keyboard	✓	✓	✓	✓	✓	✓	✓	✓		✓
com.google.android.instantapps .supervisor										✓	
com.google.android.music	Google Play Music	✓								✓	
com.google.android.syncadapters .calendar										✓	
com.google.android.videos	Google Play Movies & TV	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
com.google.android.youtube	YouTube	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
com.gotv.nflgamecenter.us.lite	NFL									✓	
com.grillgames.guitarrockhero2	Rock Hero 2	✓									
com.gta.real.gangster.crime	Real Gangster Crime	✓	✓	✓	✓	✓	✓				✓
com.halfbrick.fruitninjafree	Fruit Ninja®	✓	✓		✓	✓	✓				✓
com.halig.seka	Seka	✓		✓	✓	✓	✓	✓			
com.hamropatro	Hamro Patro : The Best Nepali Patro	✓			✓	✓	✓				
com.happylabs.hotelstory	Hotel Story: Resort Simulation	✓						✓	✓		
com.hasbro.riskbigscreen	RISK: Global Domination	✓	✓	✓		✓	✓	✓			
com.he2apps.kelimelik	Kelimelik									✓	
com.herogame.gplay.hopelessland	Hopeless Land: Fight for Survival	✓									
com.hitcents.stickmanepicfree	Draw a Stickman: EPIC Free	✓									
com.huawei.android.hwouc	System Update									✓	
com.huawei.android.totemweather										✓	
com.huawei.appmarket										✓	
com.huawei.hwid	Huawei Mobile Services									✓	
com.huawei.wallet										✓	
com.hulu.plus	Hulu: Stream TV, Movies & more									✓	
com.ibotta.android	Ibotta: Cash Back Savings, Rewards & Coupons App									✓	
com.ifreenow.app										✓	
com.igg.castleclash	Castle Clash: Brave Squads									✓	
com.igoldtech.streetchaser	Street Chaser	✓							✓		
com.illusiveray.zcs	Zombie Combat Simulator	✓	✓			✓		✓			
com.imangi.templerun	Temple Run	✓				✓					
com.imdb.mobile	IMDb Movies & TV Shows: Trailers, Reviews, Tickets	✓	✓			✓	✓	✓	✓		
com.imo.android.imoim	imo free video calls and chat									✓	
com.inapp.incolor	InColor - Coloring Book for Adults	✓			✓			✓			
com.indeed.android.jobsearch	Indeed Job Search									✓	
com.ing.mobile	ING Bankieren									✓	
com.innover.imagetopdf	Image to PDF Converter	✓									
com.instagram.android	Instagram									✓	
com.intellectualflame.ledflashlight .washer		✓	✓					✓			
com.Ironnos .PowerRangersDashSaban	Power Rangers Dash	✓									
com.ironsrc.aura.tmo										✓	
com.ismaker.android.simsimi	SimSimi	✓	✓		✓		✓				
com.itchmedia.ta3	Baseball Boy!	✓									
com.jaekkl.forest2		✓									
com.jagoapp.jagobd	Jagobd - Bangla TV(Official)	✓									

App Id	App name	Dataset									
		A	B	C	D	E	F	G	H	I	J
com.jaumo	JAUMO Dating – Singles & Flirting	✓	✓	✓	✓	✓	✓	✓			✓
com.jb.emoji.gokeyboard	GO Keyboard - Cute Emojis, Themes and GIFs	✓	✓								
com.jb.gokeyboard	GO Keyboard Lite - Emoji keyboard, Free Theme, GIF	✓				✓					
com.jdpapps.brisca	Briscola Online HD - La Brisca	✓	✓	✓							
com.jio.join	JioCall									✓	
com.jio.myjio	MyJio: For Everything Jio									✓	
com.jjplaycardgames.big2	Big 2	✓		✓	✓	✓					
com.jotterpad.x	JotterPad - Writer, Screenplay, Novel	✓									
com.joxdev.orbia	Orbia: Tap and Relax	✓			✓		✓	✓			
com.joycity.god	Game of Dice	✓		✓					✓		
com.just4funmobile.catwalksinphone	Cat Walks in Phone Cute Joke	✓		✓				✓			
com.kabam.marvelbattle	MARVEL Contest of Champions	✓		✓		✓		✓	✓		
com.kb.mobfree	Backgammon Masters Free	✓	✓		✓						✓
com.ketchapp.bottleflip		✓									
com.ketchapp.jellyjump	Jelly Jump	✓									
com.ketchapp.twist	Twist	✓									
com.killi_app	Killi									✓	
com.kms.free	Kaspersky Mobile Antivirus: AppLock & Web Security									✓	
com.knipsoft.klaverjassen.lite	Klaverjas HD Free	✓									
com.kongregate.mobile.epicskater.google	Epic Skater	✓									
com.kristanix.android.mahjongsolitairetitan	Mahjong Titan	✓	✓				✓				
com.kubi.kucoin	KuCoin									✓	
com.lafourchette.lafourchette	TheFork - Restaurants booking and special offers									✓	
com.lastpass.lpandroid	LastPass Password Manager	✓								✓	
com.lazygeniouz.saveit	Status Saver									✓	
com.lbe.parallel.intl	Parallel Space - Multiple accounts & Two face									✓	
com.lbe.parallel.intl.arm64	Parallel Space - 64Bit Support									✓	
com.lbe.security.miui										✓	
com.lemongame.klondike.solitaire	Solitaire	✓						✓			
com.lenovo.anyshare.gps	SHAREit - Transfer & Share									✓	
com.level4studios.tarot	Your Free Tarot	✓						✓			
com.life360.android.safetymapd	Life360 - Family Locator, GPS Tracker									✓	
com.lilithgame.hgame.gp	AFK Arena									✓	
com.linecorp.linelite	LINE Lite: Free Calls & Messages									✓	
com.linkdokter.halodoc.android	Halodoc - Doctors, Medicine & Appointments									✓	
com.linkedin.android	LinkedIn: Jobs, Business News & Social Networking									✓	
com.livescore	LiveScore: Live Sport Updates									✓	
com.lonedwarfgames.tanks.android	Tank Recon 3D (Lite)	✓									
com.lrhsoft.shiftercalendar	Work Shift Calendar	✓									
com.macropinch.pearl	Battery	✓									
com.macropinch.swan	Weather	✓		✓							
com.madfingergames.legends	SHADOWGUN LEGENDS - FPS PvP and Coop Shooting Game	✓		✓	✓			✓			
com.marothiatechs.gulel	Shoot Girl's Fruits : Gulel	✓									
com.maysalward.Dominoes	Dominoes Pro		✓	✓							✓
com.megafox.geometry.defense	Geometry Defense 2	✓									
com.melodis.midomiMusicIdentifier.freemium	SoundHound - Music Discovery & Lyrics		✓				✓		✓		
com.memrise.android.memrisecompanion	Learn Languages with Memrise - Spanish, French...	✓	✓	✓	✓			✓			
com.mentormate.android.inboxdollars	InboxDollars									✓	
com.metago.astro	File Manager by Astro (File Browser)	✓	✓	✓			✓				✓
com.metropcs.metrozone	MetroZone									✓	

App Id	App name	Dataset									
		A	B	C	D	E	F	G	H	I	J
com.mgc.miami.crime.simulator.two	Vegas Crime Simulator	✓	✓	✓	✓	✓		✓			
com.mi.android.globalFileexplorer	File Manager : free and easily									✓	
com.microclip.secret.agent.usarmy.mission	Secret Agent US Army Mission	✓		✓			✓				
com.microsoft.appmanager	Your Phone Companion - Link to Windows									✓	
com.microsoft.launcher	Microsoft Launcher									✓	
com.microsoft.office.outlook	Microsoft Outlook	✓	✓	✓	✓	✓	✓	✓		✓	✓
com.microsoft.skydrive	Microsoft OneDrive									✓	
com.microsoft.todos	Microsoft To Do: List, Task & Reminder	✓	✓	✓	✓	✓	✓	✓	✓		
com.milos.design										✓	
com.mindy.grap1	Stickman Hook			✓	✓						
com.ministrycentered.PlanningCenter	Planning Center Services	✓	✓	✓	✓	✓					
com.miui.analytics										✓	
com.miui.android.fashiongallery	Mi Wallpaper Carousel									✓	
com.miui.global.packageinstaller										✓	
com.miui.home										✓	
com.miui.hybrid										✓	
com.miui.msa.global										✓	
com.miui.notes										✓	
com.miui.videoplayer	Mi Video									✓	
com.miui.weather2										✓	
com.mobeyeapp	Mobeye - Earn money									✓	
com.mobgames3dgp.minibussim2017	Minibus Simulator 2017	✓									
com.mobilemotion.dubsmash	Dubsmash - Create & Watch Videos	✓	✓		✓	✓	✓		✓		✓
com.mochibits.hexario.google	Hexar.io - io games	✓	✓					✓			
com.moregames.makemoney	PlaySpot - Make Money Playing Games									✓	
com.motionvolt.flipdiving	Flip Diving	✓									
com.MoustacheBanana.TacticalWarSimulator	Tactical Battle Simulator	✓									
com.movile.playkids	PlayKids - Educational cartoons and games for kids				✓	✓	✓	✓	✓		
com.mtvn.Nickelodeon.GameOn	Sky Whale	✓						✓			
com.mxtech.videoplayer.ad	MX Player									✓	
com.mxtech.videoplayer.pro	MX Player Pro									✓	
com.myairtelapp	Airtel Thanks - Recharge, Bill Pay, Bank, Live TV	✓				✓	✓	✓		✓	
com.myfitnesspal.android	Calorie Counter - MyFitnessPal	✓		✓	✓			✓			
com.mylol.spotafriend	Spotafriend - Meet Teens App									✓	
com.myyearbook.m	MeetMe: Chat & Meet New People									✓	
com.nanovationlabs.slipawayandroid	Slip Away	✓									
com.narvii.amino.master	Amino: Communities and Chats	✓	✓	✓	✓	✓	✓	✓	✓		
com.natenai.glowhockey2	Glow Hockey 2	✓									
com.naturalmotion.customstreetracer2	CSR Racing 2 - #1 in Racing Games	✓		✓			✓	✓			
com.nc.robot.car	Robot Car	✓									
com.ne.hdv	HD Video Downloader	✓				✓					
com.nedl.app	nēdl									✓	
com.nemo.vidmate										✓	
com.netflix.mediaclient	Netflix									✓	
com.netmarble.mherosgb	MARVEL Future Fight	✓	✓		✓		✓		✓		
com.netqin.ps	Vault - Hide Pics & Videos, App Lock, Free Backup	✓				✓					
com.newstargames.newstarsoccer	New Star Soccer	✓	✓	✓							✓
com.nexonm.oz	Oz: Broken Kingdom™	✓		✓			✓				
com.nhl.gc1112.free	NHL									✓	
com.nianticlabs.hpwu.prod	Harry Potter: Wizards Unite									✓	
com.nianticlabs.pokemongo	Pokémon GO									✓	
com.niceapp.instasave	Photo & Video Downloader for Instagram									✓	

App Id	App name	Dataset									
		A	B	C	D	E	F	G	H	I	J
com.NikSanTech.FireDots3D	Fire Balls 3D	✓	✓		✓		✓				
com.nimblebit.pockettrains	Pocket Trains	✓									
com.ninechat.android.chat	9CHAT: International Forum, Groups & Communities									✓	
com.ninetyninegames.dhoomthree	Dhoom:3 The Game	✓									
com.nirogames.cuanto.sabes .primaria	Cuanto Sabes de Primaria	✓									
com.nj.callbreak.android	Call Break	✓									
com.NNGames.starchef_android	Star Chef™ : Cooking & Restaurant Game	✓	✓		✓						
com.noodlecake.altosadventure	Alto's Adventure	✓									
com.noodlecake.happyjump	Happy Jump	✓									
com.nordcurrent.canteenhd	Cooking Fever	✓	✓			✓					
com.nordcurrent.murderinalps	Murder in the Alps	✓									
com.nordvpn.android	NordVPN: Best VPN Fast, Secure & Unlimited									✓	
com.notabasement.mangarock .android.lotus										✓	
com.nstudio.weatherhere	NOAA Weather Unofficial (Pro)									✓	
com.nuance.swype.emui										✓	
com.nullapp.piano	Piano	✓									
com.oddrok.powerhover	Power Hover	✓						✓			
com.ogqcorp.bgh	Backgrounds HD (Wallpapers)	✓	✓	✓		✓		✓			✓
com.olacabs.customer	Ola. Get rides on-demand									✓	
com.onegogo.explorer	APUS File Manager (Explorer)	✓	✓		✓						✓
com.onehundredpics .onehundredpicsquiz	100 PICS Quiz - Guess Trivia, Logo & Picture Games	✓				✓	✓	✓			
com.oneq.askvert	1Q									✓	
com.onexsoftech .fingerbodytemperature-prank	Body Temperature Info	✓									
com.opera.browser	Opera browser with free VPN									✓	
com.oppo.market										✓	
com.orangefish.app.pokemoniv	IV Go (get IV for Pokemon)	✓				✓		✓			
com.ovilex.drivingschool2016	Driving School 2016	✓									
com.OwlGamesStudio.QuizRoyale	Quiz Royale Online	✓									
com.pandora.android	Pandora - Streaming Music, Radio & Podcasts									✓	
com.paxful.wallet	Paxful Bitcoin Wallet									✓	
com.perblue.disneyheroes	Disney Heroes: Battle Mode	✓	✓	✓		✓		✓			
com.pescapps.kidspaint	Paint Easy	✓									
com.picsart.studio	PicsArt Photo Editor: Pic, Video & Collage Maker									✓	
com.pinterest	Pinterest									✓	
com.piriform.ccleaner	CCleaner: Memory Cleaner, Phone Booster, Optimizer	✓	✓								
com.piviandco.uglybooth	UglyBooth	✓									
com.pixeltoys.freeblade	Warhammer 40,000: Freeblade	✓									
com.pixonic.wwr	War Robots. 6v6 Tactical Multiplayer Battles	✓	✓			✓			✓		
com.plarium.throne	Throne: Kingdom at War	✓									
com.plarium.vikings	Vikings: War of Clans	✓	✓			✓	✓	✓	✓		
com.playgendary.partymasters	Partymasters - Fun Idle Game	✓									
com.playway.cms2017	Car Mechanic Simulator 18	✓			✓						
com.podbean.app.podcast	Podcast App & Podcast Player - Podbean									✓	
com.pof.android	Plenty of Fish Free Dating App	✓	✓		✓	✓		✓	✓		
com.polyplayinc.combatassault	Combat Assault: SHOOTER	✓				✓	✓				
com.popularapp.periodcalendar	Period Tracker - Period Calendar Ovulation Tracker	✓			✓				✓		
com.powerd.cleaner	Fast Cleaner - Speed Booster & Cleaner	✓	✓		✓	✓	✓	✓			✓
com.powernapps.acallfromsanta	A Call From Santa Claus! + Chat (Simulation)	✓									
com.premise.android.prod	Premise - Take Photos Earn Money									✓	
com.prezi.android	Prezi Viewer	✓									
com.prntscr.app	Lightshot (screenshot tool)									✓	

App Id	App name	Dataset									
		A	B	C	D	E	F	G	H	I	J
com.prodege.answer	SB Answer - Surveys that Pay									✓	
com.prodege.swagbucksmobile	Swagbucks - Best App that Pays									✓	
com.Project100Pi.themusicplayer	Pi Music Player - MP3 Player, YouTube Music Videos									✓	
com.protostar.sling	Sling Kong	✓	✓		✓	✓		✓			✓
com.psafe.msuite	dfndr security: antivirus, anti-hacking & cleaner	✓	✓	✓	✓	✓	✓	✓			✓
com.pubcolor.paint	Paint a picture	✓									
com.pubgmafia.gtacrime.stickman.crimegangster	Grand StickMan Vegas Mafia Crime Fight To Survive	✓			✓						
com.punjabkesari.yum	Yum Recipes									✓	
com.pureprofile.android.pureprofile	Pureprofile									✓	
com.puzzleenglish.main	Puzzle English									✓	
com.qualcomm.qti.qms.service.connectionsecurity										✓	
com.RaiserGames.ZooEvolution		✓									
com.realmofempires.roedroid	Realm of Empires	✓									
com.reddit.frontpage	Reddit									✓	
com.REDSUNworks.RunFastRun	Run Fast Run!	✓									
com.reliancegames.urf	Ultimate Robot Fighting	✓		✓							
com.revolut.revolut	Revolut - A Radically Better Account									✓	
com.rkgames.smashywanted	Smashy Road: Wanted	✓									
com.rms.gamesforkids.coloring.princess	Kids coloring book: Princess	✓									
com.RockyHong.BeatStomperAndroid	Beat Stomper	✓									
com.roposo.android	Roposo - Video Status, Earn Money, Friends Chat									✓	
com.rottzgames.logic	Einstein's Riddle Logic Puzzles	✓									
com.rovio.ABstellapop	Angry Birds POP Bubble Shooter	✓	✓	✓	✓	✓	✓	✓	✓		✓
com.rovio.angrybirdsstarwars.ads.iap	Angry Birds Star Wars	✓									
com.rovio.baba	Angry Birds 2	✓	✓	✓	✓		✓		✓		
com.rtve.clan	Clan RTVE	✓									
com.ruanshaomin.game	Unblock Car	✓									
com.rubycell.pianisthd	Piano +	✓									
com.sablo.plowfarming.harvester	Forage Plow Farming Harvester	✓									
com.sahibinden	sahibinden.com: Emlak,Araba,Alışveriş ve Diğerleri									✓	
com.samsung.android.app.notes	Samsung Notes									✓	
com.samsung.android.app.reminder										✓	
com.samsung.android.dialer										✓	
com.samsung.android.email.provider	Samsung Email									✓	
com.samsung.android.game.gos										✓	
com.samsung.android.mobileservice	Samsung Experience Service									✓	
com.samsung.android.oneconnect	SmartThings									✓	
com.samsung.android.provider.stickerprovider										✓	
com.samsung.android.rubin.app										✓	
com.samsung.android.scloud										✓	
com.samsung.android.sm.devicesecurity										✓	
com.samsung.oh	Samsung Members									✓	
com.samsung.vvm										✓	
com.sbs.pixelcombat	Pixel Fury: Multiplayer in 3D	✓			✓		✓	✓	✓		
com.sbwebcreations.weaponbuilder	Weapon Builder	✓									
com.scee.psxandroid	PlayStation App									✓	
com.science.yarnapp	Yarn - Chat Fiction	✓	✓	✓	✓		✓	✓			✓
com.scimob.ninetyfour.percent	94% - Quiz, Trivia & Logic	✓	✓	✓	✓	✓		✓	✓		
com.sec.android.app.camera										✓	
com.sec.android.app.samsungapps										✓	
com.sec.android.app.sbrowser	Samsung Internet Browser									✓	
com.sec.android.app.shealth	Samsung Health									✓	
com.sec.android.daemonapp										✓	

App Id	App name	Dataset									
		A	B	C	D	E	F	G	H	I	J
com.sec.android.inputmethod	Samsung Keyboard									✓	
com.sec.spp.push	Samsung Push Service									✓	
com.sega.sonicdash	Sonic Dash	✓			✓		✓				
com.setsnail.daddylonglegs	Daddy Long Legs	✓						✓			
com.SevenBulls .CounterAttackShooter	Counter Attack - Multiplayer FPS	✓		✓		✓			✓		
com.shazam.android	Shazam							✓	✓	✓	
com.shootbubble.bubbledexlue	Shoot Bubble Deluxe	✓									
com.shopback.app	ShopBack - The Smarter Way | Shopping & Cashback									✓	
com.shopkick.app	Cashback Deals & Rewards on Shopping with Shopkick									✓	
com.shutterstock.contributor	Shutterstock Contributor									✓	
com.simpler.backup	Easy Backup - Contacts Transfer and Restore	✓	✓				✓				
com.simplesoft.launchpad	DJ Electro Mix Pad	✓									
com.skype.raider	Skype - free IM & video calls									✓	
com.smartme.analytics	Smartme App									✓	
com.sn.generaladaptiveapps	sn	✓									
com.snapchat.android	Snapchat									✓	
com.snapdeal.main	Snapdeal Online Shopping App - Shop Online India									✓	
com.sngict.rummy	Rummy - Offline	✓	✓					✓	✓		✓
com.socialping.lifequotes	The Best Life Quotes	✓	✓			✓					✓
com.softwego.liedetector	Lie Detector Simulator Fun	✓									
com.sololearn	SoloLearn: Learn to Code for Free									✓	
com.sony.nfx.app.sfrc	News Suite by Sony	✓	✓		✓		✓	✓	✓		✓
com.sonymobile.sketch		✓	✓	✓		✓			✓		
com.sopt.mafia42.client	Mafia42	✓	✓	✓	✓		✓	✓	✓		
com.sotg.surveys	Surveys On The Go®									✓	
com.soundcloud.android	SoundCloud - Music & Audio									✓	
com.sp.protector.free	AppLock - Fingerprint									✓	
com.spaceapegames.fastlane	Fastlane: Road to Revenge	✓			✓						
com.spigo.yatzy	Yatzy	✓									
com.splendapps.splendo	To Do List	✓				✓	✓				
com.spotify.music	Spotify: Listen to new music, podcasts, and songs									✓	
com.spring.bird.game2048plus	2048 Plus	✓									
com.sprint.ecid										✓	
com.sprint.ms.smf.services	Carrier Hub									✓	
com.starmakerinteractive.starmaker	StarMaker: Sing with 50M+ Music Lovers									✓	
com.stickman.dismount.heroes	Stickman Destruction 3 Heroes	✓									
com.studio8apps.instasizenocrop	Square InPic - Photo Editor & Collage Maker	✓	✓								
com.suntv.sunnxt	Sun NXT	✓	✓						✓		✓
com.supercell.clashofclans	Clash of Clans	✓	✓		✓			✓	✓		
com.supercell.clashroyale	Clash Royale									✓	
com.superluckycasino.viponly.slots .vegas.android.free	Slots: VIP Deluxe Slot Machines Free - Vegas Slots	✓									
com.supersolid.honestfood	Food Street - Restaurant Management & Food Game	✓	✓	✓	✓			✓			
com.supertapx.drawin	Draw In	✓			✓		✓				
com.surpax.ledflashlight.panel		✓						✓			
com.surveysampling.mobile .quickthoughts	QuickThoughts: Take Surveys Earn Gift Card Rewards									✓	
com.suzy.droid	Crowdtap: Opinions & Rewards									✓	
com.sword.game.bubble	Bubble Break	✓									
com.tangiappsit.shiva .cyclingadventure	Shiva Cycling Adventure	✓			✓						
com.tap4fun.reignofwar	Invasion: Modern Empire	✓	✓		✓	✓	✓	✓	✓		
com.tapastic	Tapas – Comics, Novels, and Stories	✓	✓	✓		✓	✓	✓			
com.tapblaze.hairsalonmakeover	Hair Salon Makeover	✓									
com.tapinator.bikerracing.stunt	City Bike Race Stunts 3D	✓									
com.tapinator.monstertruck .speedstunts	Monster Truck Speed Stunts 3D	✓									

App Id	App name	Dataset									
		A	B	C	D	E	F	G	H	I	J
com.tapinator.snow.jet.landing3d	Snow Cargo Jet Landing 3D	✓									
com.tapinator.taxidriver3d	Modern Taxi Driving 3D	✓									
com.teamstartup.farm	My Happy Farm Daily	✓									
com.tencent.ig	PUBG MOBILE									✓	
com.tencent.mm	WeChat									✓	
com.ternopil.fingerpaintfree	Paint Free	✓									
com.teslacoilsw.launcher	Nova Launcher							✓			
com.them.ok	The Onion Knights	✓									
com.TheRefik.SkyLand	SkyLand	✓									
com.thetisgames.fsxtremefree	FlyWings Flight Simulator 2013 Rio de Janeiro	✓									
com.thirdrock.fivemiles	5miles: Buy and Sell Used Stuff Locally									✓	
com.tictactoe.MobileAppsPro	Tic Tac Toe	✓									
com.tinder	Tinder									✓	
com.tinyco.potter	Harry Potter: Hogwarts Mystery									✓	
com.tinydragonadventuregames .pony.crafting.unicorn.game.girls .color.quest.pet.building.minecraft .cube.pixel.rainbow.secret.free	Pony Girls Craft: Exploration	✓									
com.tivola.doghotel	Dog Hotel – Play with dogs and manage the kennels	✓	✓								
com.tmobile.services.nameid	T-Mobile NAME ID									✓	
com.toast.comico.th	comico	✓					✓	✓	✓		
com.tobiapps.android_100fl	100 Floors - Can you escape?	✓									
com.todoist	Todoist: To-Do List, Tasks & Reminders	✓	✓		✓	✓	✓				
com.toluna.webservice	Toluna									✓	
com.topfreegames .bikeracefreeworld	Bike Race Free - Top Motorcycle Racing Games	✓		✓				✓			
com.topfunfreegames.adultcoloring	Adult Coloring Book Premium	✓			✓			✓			
com.touchtype.swiftkey	SwiftKey Keyboard									✓	
com.traveloka.android	Traveloka: Book Hotel, Flight Ticket & Activities									✓	
com.trendsource.thesource	The Source Mobile									✓	
com.trimanstudio .JetFlightSimulator3d	F18 3D Fighter Jet Simulator	✓									
com.tripadvisor.tripadvisor	TripAdvisor Hotels Flights Restaurants Attractions	✓	✓		✓	✓	✓	✓			
com.trivago	trivago: Compare hotel prices	✓		✓	✓	✓		✓	✓		
com.tru	YuppTV - LiveTV Movies Shows	✓					✓		✓		
com.truecaller	Truecaller: Caller ID, block robocalls & spam SMS									✓	
com.ttxapps.drivesync	Autosync for Google Drive	✓	✓		✓	✓	✓		✓		
com.tumblr	Tumblr									✓	
com.turbochilli.gks	Good Knight Story	✓									
com.turbochilli.hoppyfrog2	Hoppy Frog 2 - City Escape	✓									
com.turborilla.bmx2	Mad Skills BMX 2	✓			✓						
com.twitter.android	Twitter						✓			✓	
com.twitter.android.lite	Twitter Lite									✓	
com.ubisoft.colparis	City of Love: Paris	✓									
com.uc.browser.en	UC Mini- Download Video Status, Movie, Funny Video									✓	
com.UCMobile.intl	UC Browser- Free & Fast Video Downloader, News App									✓	
com.us.tubers.life.tycoon	Tubers Life Tycoon	✓									
com.usps.myusps_mobile_app _android	Informed Delivery®									✓	
com.utternik.utternik	Utternik: Opinion Rewards - #1 Cash Earning App									✓	
com.varomoney.varo	Varo: Save money with a no fee mobile bank account									✓	
com.vector3d.uatc	Unmatched Air Traffic Control	✓									
com.verizon.mips.services										✓	
com.vg.AirplaneFlightSimulator	3D Airplane Flight Simulator	✓									
com.vg.cityroads.buildersim		✓									
com.vg.FiretruckParking3D		✓									

App Id	App name	Dataset									
		A	B	C	D	E	F	G	H	I	J
com.vg.prisonbus		✓									
com.vg.shipsofbattleageofpirates	Ships of Battle - Age of Pirates - Warship Battle	✓	✓	✓	✓		✓				
com.viber.voip	Viber Messenger - Messages, Group Chats & Calls									✓	
com.vpgame.eric	VPGAME-E-sports live stream									✓	
com.WatchMoviesOnline	watch movies online free	✓									
com.weather.Weather	Weather maps & forecast, with The Weather Channel	✓	✓	✓	✓		✓	✓			✓
com.whatsapp	WhatsApp Messenger									✓	
com.whatsapp.w4b	WhatsApp Business									✓	
com.wikia.singlewikia.pokemon	FANDOM for: Pokemon	✓									
com.wire	Wire • Secure Messenger									✓	
com.WNYume.GodOfStickMan3	God of Stickman 3	✓									
com.Woof.Game		✓									
com.works.timeglass.logoquiz	Picture Quiz: Logos	✓	✓	✓	✓						
com.xaltimate.gemscoccalc	Gems Calc for Clash of Clans	✓									
com.xiaomi.account										✓	
com.xiaomi.discover										✓	
com.xiaomi.mipicks										✓	
com.xiaomi.xmsf										✓	
com.xnxx.app										✓	
com.xvideos.app										✓	
com.xvideostudio.videoeditor	VideoShow Video Editor, Video Maker, Photo Editor									✓	
com.yahoo.mobile.client.android.mail	Yahoo Mail – Organized Email									✓	
com.yahoo.mobile.client.android.weather	Yahoo Weather	✓	✓								
com.yandex.toloka.androidapp	Yandex.Toloka									✓	
com.yodo1.crossyroad	Crossy Road	✓		✓	✓			✓	✓		
com.youdagames.pokerworld	Poker World - Offline Texas Holdem	✓				✓			✓		
com.yy.hiyo	HAGO - Play With New Friends									✓	
com.zenstudios.ZenPinball	Zen Pinball	✓									
com.zentertain.flashlight3	Best Flashlight	✓									
com.zeptolab.cats.google	CATS: Crash Arena Turbo Stars	✓	✓	✓	✓	✓		✓			
com.zeptolab.timetravel.free.google	Cut the Rope: Time Travel	✓									
com.zhiliaoapp.musically	TikTok - Make Your Day									✓	
com.zhiliaoapp.musically.go	TikTok Lite									✓	
com.zingmagic.bridgevfree	Bridge V+, 2019 Edition	✓		✓							
com.zombodroid.MemeGeneratorClassic	Meme Generator (old design)	✓	✓	✓	✓	✓	✓	✓	✓		✓
com.zplay.willhero	Will Hero	✓	✓					✓			
com.zynga.empires2	Empires and Allies	✓		✓	✓	✓		✓			
de.axelspringer.yana.zeropage										✓	
de.lotum.whatsinthefoto.es	4 Fotos 1 Palabra	✓			✓		✓	✓			
de.motain.iliga	Onefootball - Soccer Scores	✓	✓		✓	✓	✓	✓			✓
deezer.android.app	Deezer Music Player: Songs, Playlists & Podcasts									✓	
dvortsov.alexey.princess	Running Princess	✓							✓		
es.mamba.liedetector		✓									
es.mrcl.app.juasapp	Juasapp - Joke Calls	✓					✓				
eu.bandainamcoent.galagawars	Galaga Wars	✓									
eu.livesport.FlashScore_com										✓	
fb.video.downloader	Video Downloader for Facebook	✓			✓	✓					
fi.LinnamaEntertainment.hydraulicpress	Hydraulic Press Pocket	✓									
fi.twomenandadog.zombiecatchers	Zombie Catchers	✓			✓	✓		✓	✓		
flashlight.led.clock	Flashlight	✓									
fr.two4tea.fightlist	Fight List - Categories Game	✓								✓	
goldenshorestechnologies.brightestflashlight.free	Brightest Flashlight Free ®	✓									
hr.podlanica	Music Volume EQ - Equalizer & Booster	✓	✓			✓	✓		✓		✓
im.ecloud.ecalendar		✓									
imagepro.hdwallpapers	10,000+ Wallpapers HD	✓		✓			✓				

App Id	App name	Dataset									
		A	B	C	D	E	F	G	H	I	J
imoblife.toolbox.full	All-In-One Toolbox: Cleaner, More Storage & Speed									✓	
in.amazon.mShop.android.shopping	Amazon India Online Shopping and Payments									✓	
in.sweatco.app	Sweatcoin Pays You To Get Fit									✓	
in.swiggy.android	Swiggy Food Order & Delivery									✓	
info.simpleapi.simpleapi										✓	
io.voodoo.dune	Dune!							✓			
io.voodoo.firedup	Fire Up!	✓			✓						
io.voodoo.paper2	Paper.io 2	✓				✓		✓	✓		
io.voodoo.stackjump	Stack Jump	✓									
iSurvey.Android	Merchandiser by Survey.com									✓	
it.rortos.airnavyfighterslite	Air Navy Fighters Lite	✓									
it.telecomitalia.centodiciannove	MyTIM									✓	
it.uniud.hcilab.prepareforimpact	Prepare for Impact	✓									
je.fit	JEFIT Workout Tracker, Weight Lifting, Gym Log App									✓	
jp.co.hit_point.tabikaeru		✓				✓					
jp.co.ponos.battlecatsen	The Battle Cats	✓		✓	✓	✓		✓			
jp.ne.donuts.tanshanotora		✓		✓	✓	✓	✓	✓			
jurassic.survival.craft.z	Jurassic Survival	✓									
kik.android	Kik									✓	
kr.co.mokey.mokeywallpaper_v3	HD Free Wallpaper(Backgrounds)	✓									
lemmingsatwork.quiz	Quiz: Logo game	✓									
manastone.game.wcc.Google	World Chess Championship	✓									
marble.egyptian.quest	Marble Shoot – Egyptian – Marble shooting	✓				✓					
mk.g6.crackyourscreen		✓									
mobi.byss.gun3	Major GUN : War on Terror	✓									
modpixelmon.modpecraft	Mod Pixelmon MCPE	✓									
name.markus.droesser.tapeatalk	Tape-a-Talk Voice Recorder	✓									
net.androgames.level	Bubble level									✓	
net.darksky.darksky	Dark Sky - Hyperlocal Weather									✓	
net.defensezone.first	Defense Zone - Original	✓									
net.fieldagent	Field Agent									✓	
net.ib.mn	Kpop Idol CHOEAEDOL	✓	✓	✓	✓		✓	✓	✓		
net.IntouchApp	InTouch Contacts: CallerID, Transfer, Backup, Sync	✓	✓		✓			✓	✓		✓
net.kernys.aooni	Zombie High School	✓	✓		✓		✓	✓	✓		
net.lionbird.google.dragonEvolutionWorld	Dragon Evolution World	✓									
net.mbc.gobo		✓	✓								
net.peakgames.lostjewels	Lost Jewels - Match 3 Puzzle	✓	✓	✓			✓	✓			
net.smsprofit.app										✓	
net.zedge.android	ZEDGE™ Wallpapers & Ringtones	✓	✓		✓	✓	✓	✓			✓
netgenius.bizcal	Business Calendar	✓	✓								
news.buzzbreak.android	BuzzBreak - Read News, Win Lucky Money!									✓	
nl.uitzendinggemist	NPO Start	✓	✓	✓							
nl.wiebetaaltwat.webapp	WieBetaaltWat									✓	
np.com.nepalipatro	Nepali Patro	✓		✓				✓			
org.chromium.webview_shell											✓
org.hola	Hola Free VPN Proxy Unblocker									✓	
org.imperiaonline.android.v6	Imperia Online - Medieval empire war strategy MMO	✓				✓	✓	✓			
org.lineageos.updater										✓	
org.mozilla.firefox	Firefox Browser fast & private									✓	
org.mozilla.focus	Firefox Focus: The privacy browser									✓	
org.softmotion.fpack.lite	Family's Game Travel Pack Lite	✓									
org.telegram.messenger	Telegram									✓	
org.thoughtcrime.securesms	Signal Private Messenger									✓	
org.toshi	Coinbase Wallet — Crypto Wallet & DApp Browser									✓	
org.zwanoo.android.speedtest	Speedtest by Ookla	✓		✓	✓			✓	✓		
pl.evelanblog.prawdaczyfalsz	Prawda czy Fałsz AKTUALIZACJA	✓									
pl.lukok.draughts	Checkers					✓		✓			

App Id	App name	Dataset									
		A	B	C	D	E	F	G	H	I	J
pl.mb.calendar	Calendar	✓									
pl.pawelbialecki.jedilightsaber	Force Saber of Light	✓									
pt.sibs.android.mbway	MB WAY									✓	
ro.mercador	OLX.ro - Anunturi gratuite									✓	
ru.appforge.domino	Domino	✓									
ru.prodigydev.minemaze3d	Mine Maze 3D	✓									
ru.waxtah.jadv	Jack Adventures	✓				✓					
scare.your.friends.prank.maze .halloween		✓									
se.ace.whatif	What if..	✓									
se.hellothere.kungfurygame		✓									
sg.gumi.bravefrontier	Brave Frontier									✓	
sgs.urb.pixel.survival.online	URB: Last Pixels Battle Royale	✓	✓								
smart.calculator.gallerylock	Photo,Video Locker-Calculator									✓	
software.simplicial.nebulous	Nebulous.io	✓	✓	✓	✓	✓		✓			
software.simplicial.orborous	Orborous	✓									
tv.peel.mobile.app										✓	
tv.periscope.android	Periscope - Live Video						✓	✓			
tv.sliver.android	SLIVER.tv - Watch Streams & Earn TFuel									✓	
tv.twitch.android.app	Twitch: Livestream Multiplayer Games & Esports	✓	✓	✓	✓	✓	✓	✓	✓		
tw.mobileapp.qrcode.banner	QR code reader / QR Code Scanner	✓	✓		✓		✓	✓			
uk.co.aifactory.spadesfree	Spades Free	✓									
video.like	Likee - Formerly LIKE Video - celebrity look alike									✓	
videos.share.rozdhan	Roz Dhan: Earn Money, Read News, and Play Games									✓	
vsin.t16_funny_photo	Photo Lab Picture Editor + Halloween makeup art	✓	✓	✓	✓	✓	✓	✓	✓		
ws.xsoh.taqwemee	Hijri Calendar - Taqwemee	✓				✓					

Bibliography

[1] Khaled Al-Naami, Swarup Chandra, Ahmad Mustafa, Latifur Khan, Zhiqiang Lin, Kevin Hamlen, and Bhavani Thuraisingham. Adaptive encrypted traffic fingerprinting with bi-directional dependence. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 177–188, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4771-6. doi: 10.1145/2991079.2991123. URL http://doi.acm.org/10.1145/2991079.2991123.

[2] Paul C. Kocher Alan O. Freier, Philip Karlton. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101, Internet Engineering Task Force (IETF), August 2011. URL https://tools.ietf.org/html/rfc6101.

[3] Theodore W Anderson, Donald A Darling, et al. Asymptotic theory of certain" goodness of fit" criteria based on stochastic processes. *The annals of mathematical statistics*, 23(2):193–212, 1952.

[4] Kenji Baheux. Experimenting with same-provider dns-over-https upgrade, September 2019. URL https://blog.chromium.org/2019/09/experimenting-with-same-provider-dns.html.

[5] Siddhartha Bhattacharyya, Sanjeev Jha, Kurian Tharakunnel, and J. Christopher Westland. Data mining for credit card fraud: A comparative study. *Decision Support Systems*, 50(3):602 – 613, 2011. ISSN 0167-9236. doi: https://doi.org/10.1016/j.dss.2010.08.008. URL http://www.sciencedirect.com/science/article/pii/S0167923610001326. On quantitative methods for detection of financial fraud.

[6] Gérard Biau. Analysis of a random forests model. *Journal of Machine Learning Research*, 13(Apr):1063–1095, 2012.

[7] George Dean Bissias, Marc Liberatore, David Jensen, and Brian Neil Levine. Privacy vulnerabilities in encrypted http streams. In George Danezis and David Martin, editors, *Privacy Enhancing Technologies*, pages 1–11, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-34746-0.

[8] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport layer security (tls) extensions. RFC 3546, Internet Engineering Task Force (IETF), June 2003. URL https://tools.ietf.org/html/rfc3546.

[9] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001. ISSN 1573-0565. doi: 10.1023/A:1010933404324. URL https://doi.org/10.1023/A:1010933404324.

[10] L. Chaddad, A. Chehab, I. H. Elhajj, and A. Kayssi. Mobile traffic anonymization through probabilistic distribution. In *2019 22nd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pages 242–248, Feb 2019. doi: 10.1109/ICIN.2019.8685871.

[11] Eric Chan-Tin, Taejoon Kim, and Jinoh Kim. Website fingerprinting attack mitigation using traffic morphing. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1575–1578. IEEE, 2018.

[12] Josh Constine. Facebook pays teens to install vpn that spies on them. *Techcrunch*, Jan 2019. URL https://techcrunch.com/2019/01/29/facebook-project-atlas/.

[13] Josh Constine. Google will stop peddling a data collector through apple's back door. *Techcrunch*, Jan 2019. URL https://techcrunch.com/2019/01/30/googles-also-peddling-a-data-collector-through-apples-back-door/.

[14] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde. Analyzing android encrypted network traffic to identify user actions. *IEEE Transactions on Information Forensics and Security*, 11(1):114–125, Jan 2016. ISSN 1556-6013. doi: 10.1109/TIFS.2015.2478741.

[15] Scott E. Coull and Kevin P. Dyer. Traffic analysis of encrypted messaging services: Apple imessage and beyond. *SIGCOMM Comput. Commun. Rev.*, 44(5):5–11, October 2014. ISSN 0146-4833. doi: 10.1145/2677046.2677048. URL http://doi.acm.org/10.1145/2677046.2677048.

[16] D Richard Cutler, Thomas C Edwards Jr, Karen H Beard, Adele Cutler, Kyle T Hess, Jacob Gibson, and Joshua J Lawler. Random forests for classification in ecology. *Ecology*, 88(11):2783–2792, 2007.

[17] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song. Networkprofiler: Towards automatic fingerprinting of android apps. In *2013 Proceedings IEEE INFOCOM*, pages 809–817, April 2013. doi: 10.1109/INFCOM.2013.6566868.

[18] Ramón Díaz-Uriarte and Sara Alvarez De Andres. Gene selection and classification of microarray data using random forest. *BMC bioinformatics*, 7(1):3, 2006.

[19] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *2012 IEEE Symposium on Security and Privacy*, pages 332–346, May 2012. doi: 10.1109/SP.2012.28.

[20] Chesner Désir, Simon Bernard, Caroline Petitjean, and Laurent Heutte. One class random forests. *Pattern Recognition*, 46(12):3490 – 3506, 2013. ISSN 0031-3203. doi: https://doi.org/10.1016/j.patcog.2013.05.022. URL http://www.sciencedirect.com/science/article/pii/S003132031300246X.

[21] Hossein Falaki, Dimitrios Lymberopoulos, Ratul Mahajan, Srikanth Kandula, and Deborah Estrin. A first look at traffic on smartphones. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 281–287, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0483-2. doi: 10.1145/1879141.1879176. URL http://doi.acm.org/10.1145/1879141.1879176.

[22] Y. Fu, H. Xiong, X. Lu, J. Yang, and C. Chen. Service usage classification with encrypted internet traffic in mobile messaging apps. *IEEE Transactions on Mobile Computing*, 15(11):2851–2864, Nov 2016. ISSN 1536-1233. doi: 10.1109/TMC.2016.2516020.

[23] Vincent Ghiette, Norbert Blenn, and Christian Doerr. Remote identification of port scan toolchains. In *IFIP International Conference on New Technologies, Mobility and Security*, 2016.

[24] Vincent Ghiette, Harm Griffioen, and Christian Doerr. Fingerprinting tooling used for ssh compromisation attempts. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019.

[25] Harm Griffioen and Christian Doerr. Discovering collaboration: Unveiling slow, distributed scanners based on common header field patterns. In *IEEE/IFIP Network Operations and Management Symposium*, 2020.

[26] Kotaro Hara, Abigail Adams, Kristy Milland, Saiph Savage, Chris Callison-Burch, and Jeffrey P. Bigham. A data-driven analysis of workers' earnings on amazon mechanical turk. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 449:1–449:14, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5620-6. doi: 10.1145/3173574.3174023. URL http://doi.acm.org/10.1145/3173574.3174023.

[27] M. Hazewinkel. *H*, pages 344–516. Springer Netherlands, Dordrecht, 1989. ISBN 978-94-009-5997-2. doi: 10.1007/978-94-009-5997-2_3. URL https://doi.org/10.1007/978-94-009-5997-2_3.

[28] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. Website fingerprinting: attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 31–42. ACM, 2009.

[29] Andrew Hintz. Fingerprinting websites using traffic analysis. In Roger Dingledine and Paul Syverson, editors, *Privacy Enhancing Technologies*, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-36467-2.

[30] P. Hoffman and P. McManus. Dns queries over https (doh). RFC 8484, Internet Engineering Task Force (IETF), October 2018. URL https://tools.ietf.org/html/rfc8484.

[31] S. Landau. Making sense from snowden: What's significant in the nsa surveillance revelations. *IEEE Security Privacy*, 11(4):54–63, July 2013. doi: 10.1109/MSP.2013.90.

[32] Liran Lerman, Romain Poussier, Gianluca Bontempi, Olivier Markowitch, and François-Xavier Standaert. Template attacks vs. machine learning revisited (and the curse of dimensionality in side-channel analysis). In Stefan Mangard and Axel Y. Poschmann, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 20–33, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21476-4.

[33] Marc Liberatore and Brian Neil Levine. Inferring the source of encrypted http connections. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 255–263, New York, NY, USA, 2006. ACM. ISBN 1-59593-518-5. doi: 10.1145/1180405.1180437. URL http://doi.acm.org/10.1145/1180405.1180437.

[34] Timothy Libert. Exposing the hidden web: An analysis of third-party HTTP requests on 1 million websites. *CoRR*, abs/1511.00619, 2015. URL http://arxiv.org/abs/1511.00619.

[35] Z. Liu and R. Wang. Mobilegt: A system to collect mobile traffic trace and build the ground truth. In *2016 26th International Telecommunication Networks and Applications Conference (ITNAC)*, pages 142–144, Dec 2016. doi: 10.1109/ATNAC.2016.7878798.

[36] Zhen Liu, Ruoyu Wang, Nathalie Japkowicz, Yongming Cai, Deyu Tang, and Xianfa Cai. Mobile app traffic flow feature extraction and selection for improving classification robustness. *Journal of Network and Computer Applications*, 125:190 – 208, 2019. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2018.10.018. URL http://www.sciencedirect.com/science/article/pii/S1084804518303400.

[37] Liming Lu, Ee-Chien Chang, and Mun Choon Chan. Website fingerprinting and identification using ordered feature sequences. In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *Computer Security – ESORICS 2010*, pages 199–214, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-15497-3.

[38] Mark Luchs and Christian Doerr. The curious case of port 0. In *IFIP Networking*, 2019.

[39] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[40] Ewen MacAskill, Julian Borger, Nick Hopkins, Nick Davies, and James Ball. Gchq taps fibre-optic cables for secret access to world's communications. *The Guardian*, June 2013. URL https://www.theguardian.com/uk/2013/jun/21/gchq-cables-secret-world-communications-nsa.

[41] Stanislav Miskovic, Gene Moo Lee, Yong Liao, and Mario Baldi. Appprint: Automatic fingerprinting of mobile applications in network traffic. In Jelena Mirkovic and Yong Liu, editors, *Passive and Active Measurement*, pages 57–69, Cham, 2015. Springer International Publishing. ISBN 978-3-319-15509-8.

[42] OECD. Nominal minimum wages. 2018. doi: https://doi.org/10.1787/data-00314-en. URL https://www.oecd-ilibrary.org/content/data/data-00314-en. Accessed: 10 Sep 2019.

[43] Paul Ohm. The rise and fall of invasive isp surveillance. *University of Illinois Law Review*, page 1417, 2009. URL https://ssrn.com/abstract=1261344.

[44] Mahesh Pal. Random forest classifier for remote sensing classification. *International Journal of Remote Sensing*, 26(1):217–222, 2005.

[45] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website fingerprinting in onion routing based anonymization networks. In *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society*, WPES '11, pages 103–114, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1002-4. doi: 10.1145/2046556.2046570. URL http://doi.acm.org/10.1145/2046556.2046570.

[46] Kyungwon Park and Hyoungshick Kim. Encryption is not enough: Inferring user activities on kakaotalk with traffic analysis. In Ho-won Kim and Dooho Choi, editors, *Information Security Applications*, pages 254–265, Cham, 2016. Springer International Publishing. ISBN 978-3-319-31875-2.

[47] Abbas Razaghpanah, Arian Akhavan Niaki, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Johanna Amann, and Phillipa Gill. Studying tls usage in android apps. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, pages 350–362, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5422-6. doi: 10.1145/3143361.3143400. URL http://doi.acm.org/10.1145/3143361.3143400.

[48] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Internet Engineering Task Force (IETF), August 2018. URL https://tools.ietf.org/html/rfc8446.

[49] Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher Wood. Encrypted server name indication for tls 1.3. Internet-Draft draft-ietf-tls-esni-05, IETF Secretariat, November 2019. URL https://tools.ietf.org/html/draft-ietf-tls-esni-05.

[50] Eric Rscorla. Encrypted sni comes to firefox nightly, October 2018. URL https://blog.mozilla.org/security/2018/10/18/encrypted-sni-comes-to-firefox-nightly/.

[51] F. W. Scholz and M. A. Stephens. K-sample anderson–darling tests. *Journal of the American Statistical Association*, 82(399):918–924, 1987. doi: 10.1080/01621459.1987.10478517. URL https://doi.org/10.1080/01621459.1987.10478517.

[52] Suranga Seneviratne, Aruna Seneviratne, Prasant Mohapatra, and Anirban Mahanti. Predicting user traits from a snapshot of apps installed on a smartphone. *SIGMOBILE Mob. Comput. Commun. Rev.*, 18(2):1–8, June 2014. ISSN 1559-1662. doi: 10.1145/2636242.2636244. URL http://doi.acm.org/10.1145/2636242.2636244.

[53] Kamaldeep Singh, Sharath Chandra Guntuku, Abhishek Thakur, and Chittaranjan Hota. Big data analytics framework for peer-to-peer botnet detection using random forests. *Information Sciences*, 278:488 – 497, 2014. ISSN 0020-0255. doi: https://doi.org/10.1016/j.ins.2014.03.066. URL http://www.sciencedirect.com/science/article/pii/S0020025514003570.

[54] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4):427 – 437, 2009. ISSN 0306-4573. doi: https://doi.org/10.1016/j.ipm.2009.03.002. URL http://www.sciencedirect.com/science/article/pii/S0306457309000259.

[55] StatCounter Global Stats. Mobile operating system market share worldwide, 2019. URL https://gs.statcounter.com/os-market-share/mobile/worldwide.

[56] W Richard Stevens, Bill Fenner, and Andrew M Rudoff. *UNIX network programming*, volume 1. Addison-Wesley Professional, 2004.

[57] Tim Stöber, Mario Frank, Jens Schmitt, and Ivan Martinovic. Who do you sync you are?: Smartphone fingerprinting via application behaviour. In *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '13, pages 7–12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1998-0. doi: 10.1145/2462096.2462099. URL http://doi.acm.org/10.1145/2462096.2462099.

[58] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic. Appscanner: Automatic fingerprinting of smartphone apps from encrypted network traffic. In *2016 IEEE European Symposium on Security and Privacy (Euro S&P)*, pages 439–454, March 2016. doi: 10.1109/EuroSP.2016.40.

[59] Vincent F Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. Robust smartphone app identification via encrypted network traffic analysis. *IEEE Transactions on Information Forensics and Security*, 13 (1):63–78, 2018.

[60] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282 vol.1, Aug 1995. doi: 10.1109/ICDAR.1995.598994.

[61] Diana Walsh, James M. Parisi, and Katia Passerini. Privacy as a right or as a commodity in the online world: the limits of regulatory reform and self-regulation. *Electronic Commerce Research*, 17(2):185–203, Jun 2017. ISSN 1572-9362. doi: 10.1007/s10660-015-9187-2. URL https://doi.org/10.1007/s10660-015-9187-2.

[62] Q. Wang, A. Yahyavi, B. Kemme, and W. He. I know what you did on your smartphone: Inferring app usage over encrypted data traffic. In *2015 IEEE Conference on Communications and Network Security (CNS)*, pages 433–441, Sept 2015. doi: 10.1109/CNS.2015.7346855.

[63] Frederick Wolfe, Daniel J Clauw, Mary-Ann Fitzcharles, Don L Goldenberg, Robert S Katz, Philip Mease, Anthony S Russell, I Jon Russell, John B Winfield, and Muhammad B Yunus. The american college of rheumatology preliminary diagnostic criteria for fibromyalgia and measurement of symptom severity. *Arthritis care & research*, 62(5):600–610, 2010.

[64] Charles V Wright, Scott E Coull, and Fabian Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *NDSS*, volume 9. Citeseer, 2009.

[65] Alice Wyman, Michele Rodaro, Joni Savage, and Lamont Gardenhire. Firefox dns-over-https, 2019. URL `https://support.mozilla.org/en-US/kb/firefox-dns-over-https`.

[66] Qiang Xu, Jeffrey Erman, Alexandre Gerber, Zhuoqing Mao, Jeffrey Pang, and Shobha Venkataraman. Identifying diverse usage behaviors of smartphone apps. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 329–344, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1013-0. doi: 10.1145/2068816.2068847. URL `http://doi.acm.org/10.1145/2068816.2068847`.

[67] Jiong Zhang, Mohammad Zulkernine, and Anwar Haque. Random-forests-based network intrusion detection systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(5):649–659, 2008.