AppScanner: Automatic Fingerprinting of Smartphone Apps From Encrypted Network Traffic

Vincent F. Taylor*, Riccardo Spolaor†, Mauro Conti† and Ivan Martinovic*

*Department of Computer Science
University of Oxford, Oxford, United Kingdom
{vincent.taylor, ivan.martinovic}@cs.ox.ac.uk

†Department of Mathematics
University of Padua, Padua, Italy
{riccardo.spolaor, conti}@math.unipd.it

***Abstract*—Automatic fingerprinting and identification of smartphone apps is becoming a very attractive data gathering technique for adversaries, network administrators, investigators and marketing agencies. In fact, the list of apps installed on a device can be used to identify vulnerable apps for an attacker to exploit, uncover a victim's use of sensitive apps, assist network planning, and aid marketing. However, app fingerprinting is complicated by the vast number of apps available for download, the wide range of devices they may be installed on, and the use of payload encryption protocols such as HTTPS/TLS. In this paper, we present a novel methodology and a framework implementing it, called AppScanner, for the automatic fingerprinting and real-time identification of Android apps from their encrypted network traffic. To build app fingerprints, we run apps automatically on a physical device to collect their network traces. We apply various processing strategies to these network traces before extracting the features that are used to train our supervised learning algorithms. Our fingerprint generation methodology is highly scalable and does not rely on inspecting packet payloads; thus our framework works even when HTTPS/TLS is employed. We built and deployed this lightweight framework and ran a thorough set of experiments to assess its performance. We automatically profiled 110 of the most popular apps in the Google Play Store and were later able to re-identify them with more than 99% accuracy.**

1. Introduction

Smartphone and mobile device usage continues to grow at a remarkable pace as devices become more powerful, feature-rich and more affordable. Gartner reports that sales of smartphones to consumers exceeded one billion units in 2014 alone, up 28.4% over 2013 [1]. Additionally, they report that two-thirds of mobile handsets sold in the world were smartphones. Flurry, a mobile analytics company, reports that overall app usage grew by 76% in 2014 [2]. Smartphones are well-equipped out of the box, but users regularly download and install add-on applications, called apps, to introduce additional features and functionality. The intense demand for smartphones, and rapid increase in app usage, makes the mobile platform a prime target for any individual or organisation looking to identify the presence of specific apps on users' smartphones, whether for benevolent or malevolent reasons.

On personal computers, many techniques have been used to identify types of network traffic, as well as the applications that generated this traffic. Nguyen and Armitage [3] survey machine learning techniques for Internet traffic classification. Traditionally, TCP/IP traffic may be identified by port number, since it is common for applications to use "well-known" destination port numbers that are reserved for each type of service. In the case of identifying multiple sources of traffic from services that use the same port number (for example web browsing), it can sometimes suffice to rely on the HTTP host header or destination IP address to identify the recipient of the communication. However, in the mobile landscape, traffic fingerprinting is complicated by the fact that many apps communicate exclusively with their servers by sending and receiving data using HTTP/HTTPS. In the case where developers opt to use HTTPS, the payload is encrypted and thus cannot be inspected to help identify the app that the traffic originated from. Additionally, for scalability reasons, many developers and ad networks use content distribution networks (CDNs) to deliver content and provide APIs to their apps. The use of CDNs and APIs means that more than one app may send (and receive) data to (and from) the same IP address or IP address range, thus frustrating app identification attempts that rely on IP addresses.

Users typically install apps in line with their interests. Thus, merely knowing what apps a user has installed on their device can provide valuable profiling information about the user [4]. This profiling information is valuable to advertisers, governments, or rogue individuals intent on invading that

978-1-5090-1752-2/16 $31.00 © 2016 IEEE
DOI 10.1109/EuroSP.2016.40

IEEE computer society

individual's privacy. On the other hand, the list of apps users have installed on their devices may be very useful to network administrators concerned with network planning, security, or traffic engineering. We consider an actor capable of passively monitoring network traffic or otherwise being able to obtain network traces. We motivate our work by outlining four concrete scenarios where app fingerprinting and identification may be useful to such an actor.

Attackers targeting specific apps. An adversary in possession of exploits (perhaps zero-day exploits) for particular apps may use app fingerprinting to identify vulnerable apps on a network. The adversary can build a fingerprint of a vulnerable app (or vulnerable version of an app) "offline" and then later use it to identify these apps in the wild. Once vulnerable apps have been identified, the adversary may then exploit these vulnerabilities for their own benefit. It is particularly worrying to consider an adversary fingerprinting and scanning for vulnerable mobile banking apps on users' devices. By performing app fingerprinting, the adversary increases their accuracy in targeting victims, and becomes more discreet when attacking by not needing to "broadcast" their attack to users who are not vulnerable.

Attackers targeting specific users. App fingerprinting may also be used in situations where there are specific targets. By joining a victim's network (or merely staying within wireless range without associating with the network), an adversary could surreptitiously monitor and fingerprint the victim's traffic to identify what apps the victim was using or had installed on his/her device. For high-profile clients this may be highly undesirable since merely knowing what apps the victim uses on their smartphone may be quite significant. For example, a competitor may think it would be interesting to the general public to know that a married politician was using a dating/flirting app on his/her device. The gravity of this problem is highlighted when one considers the Advanced Persistent Threat (APT) context where high-profile persons are targeted. Once a list of apps have been identified, the adversary may then go on to obtaining the relevant exploits to attempt to take control of the victim's device or data. In this scenario, app fingerprinting is used to reduce the potential cost (in terms of both time and money) for exploiting a victim by quickly and easily enumerating the services that the victim uses. Presumably, the adversary will then use the most cost-effective avenue to attack the victim.

Network management. App fingerprinting provides valuable data about the types of apps and usage patterns of these apps within an organisation. In the current era of bring-your-own-device (BYOD), this information would be invaluable to network administrators wanting to optimize their networks. For example, knowing the most popular apps and their throughput and latency requirements for good user experience, administrators could then configure their network so that particular apps performed more efficiently. Additionally, app fingerprinting may be used to determine whether disallowed apps were being used on an enterprise network. The administrator could then take appropriate action against the offender.

Advertising and market research. App fingerprinting can be a valuable aid to market research. Suppose an analytics firm wants to know the popularity of apps in a particular location or during a particular event (e.g. during a football match). This firm could potentially fingerprint apps and then go into their location of interest to identify app usage from within a crowd of users. By fingerprinting app usage within a target population, advertisers may be better able to build profiles of their target market, and consequently target advertisements to users more efficiently.

1.1. Contributions

In this paper we introduce AppScanner, a framework implementing a robust and extensible methodology for the automatic fingerprinting and real-time identification of Android apps from their network traffic, whether this traffic is encrypted or unencrypted. We have built and tested AppScanner with Android devices and apps. However, due to its modular design, AppScanner can be easily ported to fingerprint and identify apps on other platforms such as iOS/Windows/Blackberry. Our main contributions are the following:

- Enumerating strategies for network traffic preprocessing that enable accurate extraction of features that can be reliably used to re-identify an app.
- Outlining a method of obtaining perfect ground truth of what app is responsible for each network transmission using a novel demultiplexing strategy.
- Providing a highly-scalable supervised learning framework that can be used to accurately model and later identify traffic flows from apps.
- Outlining a method for real-time classification of intercepted Wi-Fi traffic leveraging live packet capture.
- Comparing the performance of various classification strategies for identifying smartphone apps from encrypted network traffic.

The rest of the paper is organised as follows: Section 2 discusses work related to traffic analysis and fingerprinting; Section 3 outlines the design of AppScanner and how the different components work together to fingerprint an app; Section 4 discusses the classification strategies that were tested; Section 5 evaluates the performance of AppScanner; Section 6 discusses the limitations of AppScanner and highlights areas of future work; and finally Section 7 concludes the paper.

2. Related Work

Traffic analysis and fingerprinting is by no means a new area of research, and indeed much work has been done on analysing traffic from workstations and web browsers [5]. At first glance, it may seem that traffic analysis and fingerprinting of smartphone apps is a simple translation of existing work. While there are some similarities, such as end-to-end communication using IP addresses/ports, there are nuances in the type of traffic sent by smartphones and the way in which it is sent that makes traffic analysis in the realm of smartphones distinct from traffic analysis on traditional workstations [6][7][8]. In the remainder of this section, we consider traditional traffic analysis approaches on workstations (Section 2.1), and then we look at traffic analysis on smartphones (Section 2.2).

2.1. Traditional Traffic Analysis on Workstations

Traditional analysis approaches have relied on artefacts of the HTTP protocol to make fingerprinting easier. For example, when requesting a web page, a browser will usually fetch the HTML document and all corresponding resources identified by the HTML code such as images, JavaScript and style-sheets. This simplifies the task of fingerprinting a web page since the attacker has a corpus of information (IP addresses, sizes of files, number of files) about the various resources attached to an individual document.

Many app developers, for scalability, build their app APIs on top of content delivery networks (CDNs) such as Akamai or Amazon AWS [9]. This reduces (on average) the space of endpoints that apps communicate with. In the past, it may have been useful to look at the destination IP address of some traffic and infer the app that was sending the traffic. Presently, requests to *graph.facebook.com*, for example, may possibly be from the Facebook app but they may also be from a wide range of apps that query the Facebook Graph API. With the advent of CDNs and standard web service APIs, more and more apps are sending their traffic to similar endpoints and this frustrates attempts to fingerprint app traffic based on destination IP address only.

In the literature, several works considered strong adversaries (e.g., governments) that may leverage traffic analysis. Those adversaries are able to capture the network traffic flowing through communication links [10]. Liberatore et al. [11] showed the effectiveness of proposals aiming to identify web-pages via encrypted HTTP traffic analysis. Subsequently, Herman et al. [12] outperformed Liberatore et al. by presenting a method that relies on common text mining techniques to the normalized frequency distribution of observable IP packet sizes. This method correctly classified some 97% of HTTP requests. Similar work was proposed by Panchenko et al. [13]. Their proposal correctly identified web pages despite the use of onion routing techniques such as Tor. More recently, Cai et al. [14] presented a web page fingerprinting attack and showed its effectiveness despite traffic analysis countermeasures (e.g., HTTPOS). Unfortunately, these pieces of work were not designed for smartphone traffic analysis. Indeed, the authors focus on identifying web pages on a traditional PC and leverage the fact that the HTTP traffic can be very unique depending on how the web page is designed. On smartphones, although apps communicate using HTTP, they do so usually through text-based APIs, removing rich traffic features present in typical HTTP web page traffic.

2.2. Traffic Analysis on Smartphones

A number of authors have proposed different schemes for identifying smartphone apps and smartphones themselves from smartphone traffic. These schemes have relied on inspecting IP addresses and packet payloads among other things. The methodology and framework we propose in this paper uses IP addresses only for flow separation (i.e., not for feature generation, as explained in Section 3) and does not leverage any information contained in packet payloads.

Dai et al. [15] propose NetworkProfiler, an automated approach to profiling and identifying Android apps using dynamic methods. They use a user-interface (UI) fuzzing technique to automatically try different execution paths in an app, while the network traces are being monitored. They inspect HTTP payloads and thus this technique suffers from the fact that it only works on unencrypted traffic. Dai et al. did not have the full ground truth of the traffic traces they were analysing, so it is difficult to systematically quantify how accurate NetworkProfiler was in terms of precision, recall, and overall accuracy.

In what is probably the most directly related work, Wang et al. [16] propose a system for identifying smartphone apps from encrypted 802.11 frames. They collect data frames from target apps by running them dynamically and training classifiers with features from this data. This work shows promise but suffers from the fact that the authors only test 13 arbitrarily chosen apps from eight distinct app store categories and collect network traces for only five minutes. Indeed, the authors discover that longer training times have an adverse effect on accuracy when classifying some apps with their system. Moreover, the authors use an insufficient sample size (i.e., only 13 apps) to validate their results. By taking into account a large set of apps, in Section 5 (specifically Fig. 5), we show how increasing the number of apps negatively influences classifier accuracy. Additionally, it is not known whether Wang et al. chose that specific set of apps because it offered good classification performance or whether a statistically suitable set size will yield similar good performance. The authors also do not provide precision/recall measurements so it is difficult to judge their system performance. Finally, it is problematic to quantify their results, in general, since the authors have no way to collect accurate ground truth, i.e., a labelled dataset that is free of noise from other apps. Indeed, our methodology calls for running a single app at a time to reduce noise, and we still had to filter out 13% of our raw dataset because it was noise. AppScanner solves the aforementioned problems by going several steps further to systematically investigate this important topic. We use 110 randomly chosen apps (from the most popular apps in the Google Play Store) from 26 different categories and collect network traces for 75 minutes each. We pre-process these network traces using a novel demultiplexing technique to obtain perfect ground truth. We examine two classification algorithms, two feature generation approaches, and three overall classification strategies. Finally, we identify and validate reasons for traffic misclassification and propose mitigation strategies.

Conti et al. [17] identify specific actions that users are performing within their smartphone apps. They achieve this through flow classification and supervised machine learning. Like AppScanner, their system also works in the presence of encrypted connections since they only leverage coarse flow information such as packet direction and size. The authors achieved more than 95% accuracy for most of the considered actions. This work suffers from its specificity in identifying discrete actions. By choosing specific actions within a limited group of apps, Conti et al. may benefit from the more distinctive flows that are generated. Their system does not scale well since a manual approach was taken when choosing and fingerprinting actions. Indeed, the authors had to choose a subset of apps and a subset of actions within those apps to train their classifiers on. AppScanner is different in that it has a less specific classification aim (identifying entire apps) and it is highly scalable since fingerprints can be built for any app automatically.

Stöber et al. [18] propose a scheme for fingerprinting entire devices by identifying device-specific traffic patterns. They contend that 70% of smartphone traffic belongs to background activities happening on the device and that this can be leveraged to create a fingerprint. The authors posit that 3G transmissions can be realistically intercepted and demodulated to obtain side channel information from a transmission such as the amount of transmitted data and the timing. They leverage 'bursts' of data to generate features since they cannot analyse the TCP payload directly. Using supervised learning algorithms, the authors build a model of the traffic they want to fingerprint. This model is then capable of identifying similar bursts of data at a later time. The authors conclude that using approximately 15 minutes of captured traffic can result in a classification accuracy of over 90%. This work is similar to AppScanner in that they both leverage bursts of traffic to generate fingerprints. However, AppScanner is different because we leverage bursts to identify a single specific app at a time, and are not able to take advantage of the rich information that is present when leveraging multiple interleaved traffic bursts to gain a more unique fingerprint. Additionally, Stöber et al. [18] need 6 hours of training and 15 minutes of monitoring to achieve reliable fingerprint matching, while AppScanner uses 75 minutes of captured traffic per app for training (which can be done on the attacker's own device) and can then classify unknown traffic in real-time.

3. System Design

The main idea underpinning AppScanner is the focus on traffic flows from an app that can be used to identify that app. Traffic flows from apps may be interactive or non-interactive; that is, they may be generated with or without user interaction. A newsreader app may generate non-interactive traffic flows if it polls a server in the background for the latest news. Interactive traffic flows are generated by user action such as launching an app or navigating the app's user interface. For our fingerprinting and identification methodology, we focused primarily on interactive app traffic. Our main design goals were:

- To develop a highly-scalable framework that could be used to fingerprint and identify smartphone apps.
- To ensure that models for new or updated apps could be built in an automated way and added to the system.
- To ensure that the models were portable, i.e., they could be built and reused in a new deployment without suffering a penalty for retraining.
- To deliver a system that could perform real-time (or near real-time) classification of flows as they were observed on a network.

3.1. Definitions

Before going any further, we define some terms used later in the paper and explain other key concepts central to the AppScanner framework.

Burst - A burst is the group of all network packets (irrespective or source or destination address) occurring together that satisfies the condition that the most recent packet occurs within a threshold of time, the *burst threshold*, of the previous packet. That is, packets are grouped temporally and a new group is created only when no new packets have arrived within the amount of time set as the burst threshold. This is visually depicted in the *Traffic Burstification* section of Fig. 1, where we can see Burst A and Burst B separated by the *burst threshold*. We use the concept of a burst to logically divide the network traffic into discrete, manageable portions, which can then be further processed. The concept of a burst was previously used by Stöber et al. [18] and is used similarly here.

Flow - A flow is a sequence of packets (within a burst) with the same destination IP address and port number. That is, within a flow, all packets will either be going to (or coming from) the same destination IP address/port. Flows are not to be confused with TCP sessions. A flow ends at the end of a *burst*, while a TCP session can span multiple bursts. Thus, flows typically last for a few seconds, while TCP sessions can continue indefinitely. AppScanner leverages flows instead of TCP sessions to achieve real-time/nearer-to-real-time classification. From the *Flow Separation* section of Fig. 1, we can see that a burst may contain one or more flows. Flows may overlap in a burst if a single app, *App X*, initiates TCP sessions in quick succession or if another app, *App Y*, happens to initiate a TCP session at the same time as *App X*. We explain how we accurately attribute flows to their originating app in Section 3.3. The notion of flows has been used previously by Conti et al. [17] and are used similarly here.

We use supervised machine learning for pattern recognition on flows. In AppScanner, the supervised learning algorithms are provided with labelled examples of flows (or statistical features extracted from these flows) from each app which are then used to build models. These models can then be used to classify unlabelled flows. The models need to be lightweight since we need AppScanner to be deployable even in environments with limited processing/memory resources and still perform app classifications in real-time or near real-time.

3.2. Equipment Setup

The setup used to collect network traces from apps is shown in the *Equipment Setup* section of Fig. 1. The workstation was configured to forward traffic between the Wi-Fi access point (AP) and the Internet. To generate traffic from which to capture our training data, we used scripts that communicated with the target smartphone via USB using the Android Debug Bridge (ADB). These scripts were used to simulate user actions on the test device and thus elicit network flows from the apps. Traffic flowing through the workstation was captured and exported to a comma-separated value (CSV) file with each row containing the details of a captured packet. We collected packet details such as time, source address, destination address, ports, packet size, protocol and TCP/IP flags. The payload for each packet was also collected but was not used to provide features since it may or may not be encrypted. Our aim is for AppScanner to be able to identify apps whether their traffic is encrypted or unencrypted. Although physical hardware was used for network traffic generation and capturing, this process can be massively automated and parallelized by running apps within Android emulators on virtual machines.

3.3. Fingerprint Making

The fingerprint making process consisted of a number of steps that we outline in Fig. 1, and describe below.

Network Trace Capture: The Network Trace Capturing process entailed running user simulation scripts on the hardware setup. These scripts generated app launches, touches and button presses to elicit interactive traffic from apps which was then collected using a packet sniffer. We ran only one app at a time to minimise 'noise' in the network traces. We observed that all the apps in our test set generated network flows when they were launched. User simulation can be done in various ways, but we leveraged the standard Android SDK UI exerciser tools, *monkey* and *monkeyrunner*. AppScanner does not leverage patterns of flows or other such artefacts that may be emphasized in human-generated app traffic. For this reason, we can automate the training process (for high-scalability) using UI exercising tools. The use of UI automation may cause us to not obtain all of an app's unique flows, but the flows that are obtained would still be real-world traffic flows that would have come from the app if a human user were using it. For example, an app would still check-in with its server, send standard API queries, load online resources, and other similar tasks. In general, the more diverse the connections an app makes, the more distinguishing the traffic will be when used for feature generation and the subsequent training of classifiers.

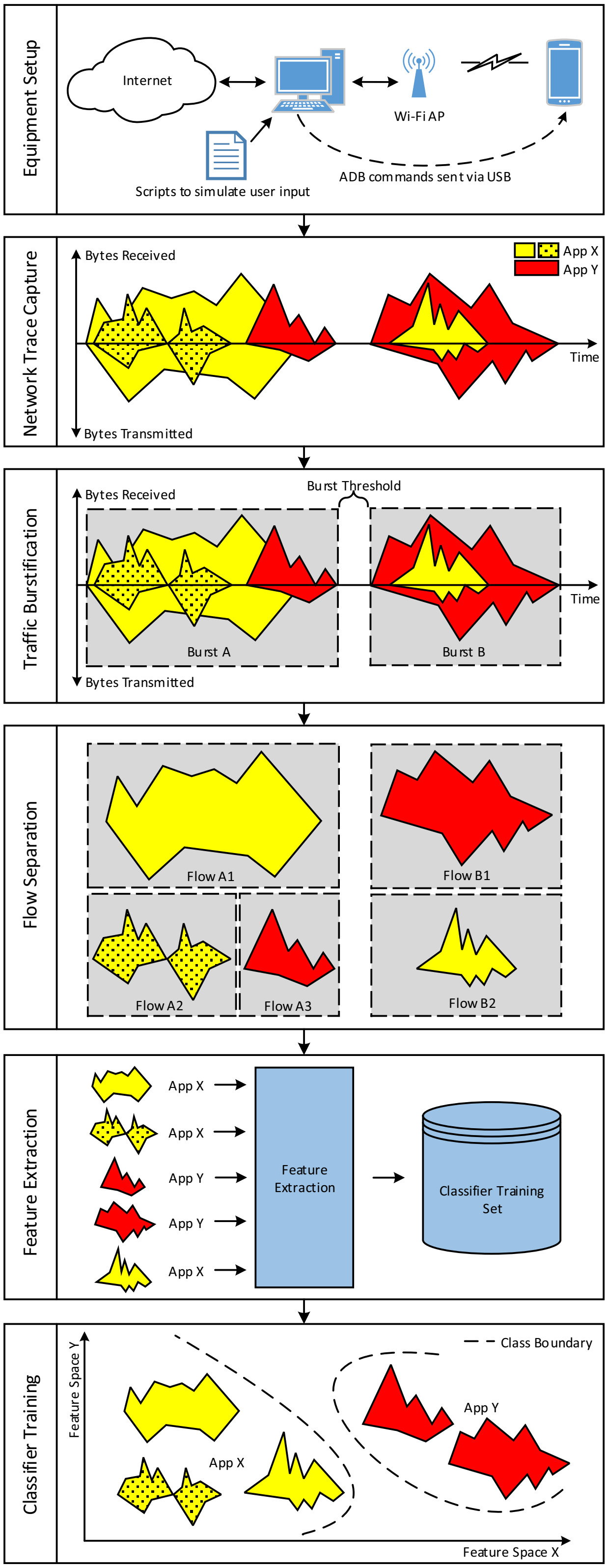


Figure 1: Visualisation of bursts and flows within TCP/IP traffic, and a high-level representation of the classifier training steps performed by AppScanner.

Greater coverage of all the network flows in an app may theoretically be obtained by using advanced UI fuzzing techniques provided by frameworks such as Dynodroid [19], or by recruiting human participants. However, we consider these approaches to be out of scope for this paper.

After data collection, the network traffic dumps were filtered to include only TCP traffic that was error free. For example, we filtered to remove packet retransmissions that were as a result of network errors. However, these dumps potentially contained traffic from other Android apps running (in the background) on the smartphone that could interfere with and taint the fingerprint making process. In addition to the target apps, another open-source app, Network Log [20], was installed and started on the target device. Network Log was used to identify the app responsible for each network flow coming from the test device. In this way, we obtained perfect ground truth of what flows came from what app. Using logged data from Network Log combined with a 'demultiplexing' script, all traffic that did not originate from the target app was removed from the traffic dump for that app. At this point, each network dump only contained error-free TCP traffic from the target app.

Traffic Burstification and Flow Separation: The next step was to parse the network dumps to obtain network traffic bursts. Traffic was first discretized into bursts to obtain ephemeral chunks of network traffic that could be sent immediately to the next stage of AppScanner for processing. This allows us to meet the design objective of real-time or near real-time classification of network traffic. Falaki et al. [21] observed that 95% of packets on smartphones "are received or transmitted within 4.5 seconds of the previous packet". During our tests, we observed that setting the burst threshold to one second instead of 4.5 seconds only slightly increased the number of bursts seen in the network traces. This suggested to us that network performance (in terms of bandwidth and latency) has improved since the original study. For this reason, we opted to use a burst threshold of one second to favour more overall bursts and nearer-to-real-time performance. These bursts were separated into individual flows (as defined in Section 3.1 and depicted in Fig. 1) using destination IP address/port information. We enforced a minimum flow length and maximum flow length that would be considered by AppScanner. This is simply to ensure that AppScanner safely ignores abnormal traffic when deployed in the real-world.

It is important to note that while destination IP addresses were used for flow separation, they were not leveraged to assist with app identification. We also opted to not use information gleaned from DNS queries or flows with unencrypted payloads. These were deliberate design decisions taken to understand how AppScanner would perform in the worst case, as well as avoid the reliance on domain-specific knowledge that frequently changes. Concretely, these additional sources of data may be considered unsuitable for the following reasons:

- **IP addresses** - Destination IP addresses contacted by an app can change if DNS-based load-balancing/high-availability is used. Additionally, many apps contact similar IP addresses because they utilise the same CDN or belong to the same developer.
- **DNS queries** - DNS queries are not always sent/observed due to the use of client-side DNS caching. Also, multiple apps may send the same DNS queries, for example, to resolve advertisement server domain names.
- **Packet payloads** - Many app developers are becoming more privacy-aware and are opting to use HTTPS/TLS to encrypt packet payloads. Thus features extracted from TCP payloads will become less useful over time.

While the aforementioned data sources may be carefully used to assist with app identification, we consider their use to be out of scope for this paper and leave such analysis to future work.

Feature Extraction and Classifier Training: Once we obtained individual flows falling within the prescribed flow length thresholds, features were generated from them and used to train classifiers. Raw packet lengths from flows were used as features, as well as the statistical properties of these flows. We elaborate on feature generation and classification approaches/strategies in Section 4.

3.4. Fingerprint Matching

The fingerprint matching phase follows steps similar to those of the fingerprint making phase, up to the end of the feature extraction step. At this point, the features are instead passed to the pre-built models to be classified, followed by what we call the 'classification validation' phase at the end. During fingerprint matching, the network traffic capturing phase is also somewhat different, since we may perform fingerprint matching during a live network capture or on a saved network trace.

Network Traffic Capturing: AppScanner can work in both online and offline mode for capturing and processing network traffic.

- *Online Mode* - Network traffic from the target smartphone is sniffed directly from the air on a live network using *tshark* (the terminal version of Wireshark) or a similar tool and passed on to the traffic capturing module of AppScanner by means of a *tshark* wrapper library. The *traffic burstification* buffer collects the incoming network packets and passes them on to the *flow separation* module as a burst whenever the burst threshold amount of time elapses with no new packets being seen. Thus, AppScanner performs app identification in real-time or near real-time.
- *Offline Mode* - A pre-collected network trace can be fed into AppScanner for 'batch processing'. The network trace is parsed into bursts and passed on to the *flow separation* module just as in online mode.

TCP Pre-processing and Flow Classification: The *flow separation* module, upon receiving a burst of network traffic, uses source and destination IP addresses to separate it into flows. Before flows are passed on to the next stage in AppScanner, they are discarded if they contain any TCP retransmissions or other errors or if they fall outside of the flow length thresholds. Flows containing TCP retransmissions or other errors are discarded since they would introduce noise into the flow that should not be there. As mentioned before, flow length thresholds are set to ensure that very lengthy (and most likely anomalous) flows do not enter the system. We discuss the actual flow length thresholds used in Section 5. Feature generation (see Fig. 2 for an outline of how this is done) is performed on these error-free, validated flows and the features of each flow are passed on to the classifiers for identification. The result of this classification is then passed to the final phase, called classification validation.

Classification Validation: The classification validation stage is crucial for one primary reason. Machine learning algorithms will always attempt to place an unlabelled example into the class it most closely resembles, even if the match is not very good. Given that our classifiers will never be trained with the universe of flows from apps, it follows that there will be some flows presented to AppScanner which are simply unknown or never-before-seen. If left unchecked, this can cause an undesirable increase in the false positive (FP) rate. Additionally, as we discuss in Section 5, some traffic flows from different apps are very similar to each other, and this will also cause an undesirable increase in the FP rate of AppScanner if left unchecked.

To counteract these problems, we leverage the prediction probability feature available in the classifiers to understand how certain the classifier is about each of its classifications. The prediction probability is a measure reported by a classifier that gives an indication of how confident the classifier is about its assignment of a particular label to an unknown sample. For example, if the classifier labelled an unknown sample as *com.facebook.katana*, we would check its prediction probability value for that classification to determine the classifier's confidence. If this value is below the classification validation threshold, AppScanner will not make a pronouncement. However, if this value exceeds the threshold, AppScanner would report it as a match for that particular app. In Section 5, we discuss how varying this threshold impacts the precision, recall, and overall accuracy of AppScanner, as well as how this affects the percentage of total flows that the classifiers are confident enough to classify.

4. Classifier Design

Since AppScanner is modular, it is possible to use different machine learning algorithms with minimal effort required if modifications are made. We designed and thoroughly tested six classification approaches as shown in Table 1. Each approach used either a Support Vector Classifier (SVC) or a Random Forest Classifier. These two classifiers were chosen because they are particularly suited for predicting classes (in our case, apps) when trained with the features that we extracted from network flows.

A Support Vector Classifier models training examples as points in space, and then divides the space using hyperplanes to give the best separation among the classes. In the case of non-linearly separable problems, the Support Vector Classifier can rely on kernel functions to project the data into a high-dimensional feature space to make it linearly separable. A Random Forest Classifier is an ensemble method that uses multiple weaker learners to build a stronger learner. This classifier constructs multiple decision trees during training and then chooses the mode of the classes output by the individual trees. It is also able to rank the importance of the features that it has selected for use (as shown in Table 3).

In Table 2, we outline additional characteristics of each of the six classification approaches that show why one would favour a particular approach over another. The classifiers are compared in terms of their speed of training, size of classifier, average confidence per classifications, whether they can measure true negatives, and whether they are robust against out-of-order packets. In general, the *Per Flow Length Classifiers* are smaller and faster to train since they have smaller training sets (only flows of a certain length). We include average speeds and sizes for the approaches when trained using our dataset. Only the binary classifiers are able to understand true negatives. Only the classifiers using statistical features are robust against out-of-order packets because the other classifiers would incorrectly assign features when presented with swapped packets.

The features used to train the classifiers were either the actual flow vectors of raw packet lengths or statistical features derived from these flow vectors. Fig. 2 shows broadly the two approaches of using flow vectors or statistical features. From the figure, the flow pre-processor simply changes the sign (i.e., makes negative) the length of incoming packets. The output of the flow pre-processor is then passed as a (variable length) flow vector to the classifiers that use packet lengths as features, or to the Statistical Feature Extraction function for the other classification strategies.

Statistical Feature Extraction involves deriving 54 statistical features from each flow (regardless of flow length). For each flow, three packet series are considered: incoming packets only, outgoing packets only, and bi-directional traffic (i.e. both incoming and outgoing packets). For each series (3 in total), the following values were computed: minimum, maximum, mean, median absolute deviation, standard deviation, variance, skew, kurtosis, percentiles (from 10% to 90%) and the number of elements in the series (18 in total). These statistical features were computed using the Python pandas libraries [22].

The features were then passed through the Feature Scaler function, which is a min-max scaler (i.e., the minimum and the maximum value for a specific feature in the training set corresponds to 0 and 1 respectively). In order to avoid the curse of dimensionality, a Feature Selection function was used to choose the best features. The

TABLE 1: The six different classification approaches that were tested in AppScanner.

Approach #	Type of Classifier	Algorithm	Features Used	Number of Features	Details
1	Multi-class	SVC	Flow vectors	7-260	Single classifier per flow length
2	Multi-class	Random Forest	Flow vectors	7-260	Single classifier per flow length
3	Multi-class	SVC	Statistical features	40 (narrowed from 54)	Single large classifier with all apps
4	Multi-class	Random Forest	Statistical features	40 (narrowed from 54)	Single large classifier with all apps
5	Binary	SVC	Statistical features	40 (narrowed from 54)	Single classifier per app
6	Binary	Random Forest	Statistical features	40 (narrowed from 54)	Single classifier per app

TABLE 2: Additional characteristics of the six classification approaches that would help one to determine what approach is more suitable for their particular deployment. RF means Random Forest Classifier. Large SVC and Large RF refer to Approaches 3 and 4 each having a Single Large Classifier. Avg. confidence per classification was determined based on the results of our extensive tests.

Approach	Per Flow SVC	Per Flow RF	Large SVC	Large RF	Per App SVC	Per App RF
#	1	2	3	4	5	6
Speed of training	Fast (2secs.)	Fast (2secs.)	Slow (2hrs.)	Medium (1hr.)	Medium (1hr.)	Medium (1hr.)
Size of classifier	Small (1MB)	Small (1MB)	Large (350MB)	Large (180MB)	Medium (35MB)	Small (4MB)
Avg. confidence per classification	Medium	High	Low	High	Very high	Very high
Provides true negatives	No	No	No	No	Yes	Yes
Robust against out-of-order packets	No	No	Yes	Yes	Yes	Yes

Feature Selection function leverages the Gini Importance metric used by a Random Forest classifier that was run on the training set [23]. This metric relies on the Gini impurity index which is computed during estimator building. At the end of training, the classifier gave a score to each feature according to its significance. At this point, we selected only those features with a score higher than 1%, for a total of 40 features of the original 54. In Table 3, we report the score for each of the Top 40 features.

Approach 1-2 - Multi-Class Classification using a Classifier Per Flow Length: These approaches involve training a multi-class Support Vector Classifier and a Random Forest Classifier with the features being a vector of packet sizes from each flow. For the Support Vector Classifier, we used an *rbf* kernel with parameters *gamma=0.0001, C=10000*. For the Random Forest Classifier, we used parameters *criterion=gini, max_features=None, n_estimators=150*. An exhaustive search on a wide set of hyperparameters (with 5-fold cross-validation) was used to optimize these parameters. The length of the feature array from a flow is equal to the amount of packets in the flow and thus the classifier for flow length *n* will be trained with *n* features per training example. Only one classifier per flow length is possible (since each training example in a classifier needs to have the same amount of features) and thus we have up to the *maximum flow length* amount of classifiers.

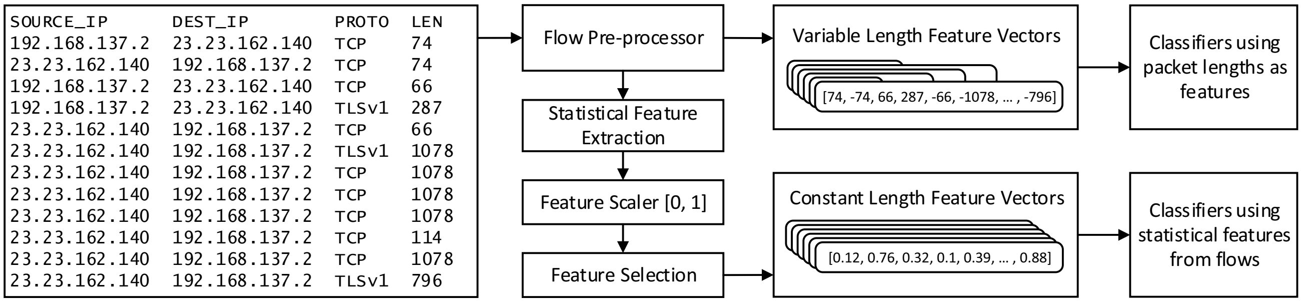
Approach 3-4 - Multi-Class Classification using a Single Large Classifier: These approaches involve training a multi-class Support Vector Classifier and a Random Forest Classifier with the features being statistical features derived from the vector of packet sizes from each flow. In these approaches, each classifier is very large and contains all the apps in the test set of apps. The parameters for the Support Vector Classifier were *kernel=linear, C=100*. For the Random Forest Classifier, we used parameters *criterion=gini, max_features=sqrt, n_estimators=150*. An exhaustive search on a wide set of hyperparameters (with 5-fold cross-validation) was used to optimize these parameters.

Approach 5-6 - Binary Classification using a Single Classifier Per App: These approaches involve training a binary Support Vector Classifier and a binary Random Forest Classifier with the features being statistical features derived from the vector of packet sizes from each flow. For the Support Vector Classifier, we used an *rbf* kernel with *gamma=0.001, C=100*. For the Random Forest Classifier, we used parameters *n_estimators=10*. In these approaches, each classifier was a binary classifier and was trained to identify only one app. Since the classifiers were of a binary nature, unlabelled flows were passed to each of the 110 classifiers in parallel when they were to be classified.

After training the classifiers, the models were saved to a persistent state using serialization. By serializing the trained classifiers, they could be loaded almost instantly the next time they were used without suffering a penalty for retraining.

5. System Evaluation

In this section, we present the experiment settings and the results of the tests that we performed on AppScanner. To build and test our framework, we used a Motorola XT1039 (Moto G) smartphone running Android 4.4.4 (KitKat). The smartphone was connected to the internet via a Linksys E1700 Wi-Fi Router/AP that had its internet connection routed through a Dell Optiplex 9020 workstation with two network interface cards. Each app was exercised automatically (using the procedure outlined in Section 3.2) in 150 rounds for a period of 75 minutes and the resulting network traffic was collected using Wireshark. We built the classifiers in Python using the scikit-learn machine learning



```
SOURCE_IP       DEST_IP         PROTO  LEN
192.168.137.2   23.23.162.140   TCP    74
23.23.162.140   192.168.137.2   TCP    74
192.168.137.2   23.23.162.140   TCP    66
192.168.137.2   23.23.162.140   TLSv1  287
23.23.162.140   192.168.137.2   TCP    66
23.23.162.140   192.168.137.2   TLSv1  1078
23.23.162.140   192.168.137.2   TCP    1078
23.23.162.140   192.168.137.2   TCP    1078
23.23.162.140   192.168.137.2   TCP    1078
23.23.162.140   192.168.137.2   TCP    114
23.23.162.140   192.168.137.2   TCP    1078
23.23.162.140   192.168.137.2   TLSv1  796
```

Flow Pre-processor
Statistical Feature Extraction
Feature Scaler [0, 1]
Feature Selection
Variable Length Feature Vectors
[74, -74, 66, 287, -66, -1078, ... , -796]
Constant Length Feature Vectors
[0.12, 0.76, 0.32, 0.1, 0.39, ... , 0.88]
Classifiers using packet lengths as features
Classifiers using statistical features from flows

Figure 2: Feature Extraction from flows - AppScanner's two main approaches for generating features from flows for classifier training.

TABLE 3: Table showing the percentage scores given to the 40 statistical features which exceeded the threshold of 1%.

Rank	Feature	Score
1st	Complete Maximum	4.32%
2nd	Outgoing Maximum	4.15%
3rd	Complete Skew	3.43%
4th	Outgoing Variance	3.35%
5th	Outgoing Standard deviation	3.35%
6th	Outgoing Kurtosis	3.01%
7th	Outgoing Skew	3.01%
8th	Outgoing Median Absolute Deviation	2.99%
9th	Outgoing 90th percentile	2.96%
10th	Complete Mean	2.86%
11th	Complete Kurtosis	2.70%
12th	Outgoing Mean	2.68%
13th	Complete Variance	2.62%
14th	Complete Standard Deviation	2.56%
15th	Complete 90th percentile	2.48%
16th	Outgoing 80th percentile	2.43%
17th	Complete Median Absolute Deviation	2.33%
18th	Incoming Variance	2.32%
19th	Incoming Skew	2.30%
20th	Incoming Standard Deviation	2.24%
21st	Incoming Kurtosis	2.22%
22nd	Incoming Median Absolute Deviation	2.19%
23rd	Complete Number of packets	2.19%
24th	Outgoing 70th percentile	2.11%
25th	Outgoing Number of packets	2.06%
26th	Incoming Number of packets	2.01%
27th	Incoming Mean	1.79%
28th	Incoming 30th percentile	1.79%
29th	Incoming 40th percentile	1.74%
30th	Incoming 60th percentile	1.65%
31st	Complete 10th percentile	1.62%
32nd	Complete 20th percentile	1.55%
33rd	Incoming 50th percentile	1.53%
34th	Incoming 20th percentile	1.49%
35th	Complete 80th percentile	1.48%
36th	Complete 30th percentile	1.45%
37th	Incoming 30th percentile	1.38%
38th	Incoming 10th percentile	1.28%
39th	Outgoing 60th percentile	1.16%
40th	Incoming 80th percentile	1.09%

libraries [24]. At the end of the training process, the classifiers were serialized in a process called pickling. Pickling is a feature provided by Python that allows the translation of the classifier data structures and object state into files.

The aim of the experiment was to find out how accurately we could fingerprint and re-identify apps from their interactive traffic as captured from the network. For these tests, AppScanner was trained with interactive traffic from 110 apps. These apps were chosen at random from the 150 Top Free Apps as listed in the Google Play Store in July 2015. (Please see Table 9 in the Appendix for the list of apps that were used to test AppScanner.)

We chose the most popular apps because we believe that these apps represent a very large cross-section of the total install base of apps across the world. If AppScanner performs well on these apps, it points to the usefulness of AppScanner as a framework for identifying apps on a global scale. Furthermore, we used free apps because free apps tend to contain more advertisements than paid apps and thus would generate more advertisement traffic. Since advertisement traffic supplied by a particular ad network would tend to be similar, AppScanner would have a more difficult task in classifying advertisement flows as belonging to one app from the group of apps that use the same ad network. Indeed, our results confirm this. For this reason, we believe the results we obtain from AppScanner being tested on free apps is the worst case performance figure.

5.1. Measuring AppScanner's Performance

Before training the classifiers, we needed to choose a suitable value for the minimum flow length that would be considered. For this test, we chose our *Per Flow Random Forest Classifier* (Approach 2) and varied the minimum flow length threshold while keeping the maximum flow length threshold constant at infinity. Fig. 3 shows the effect that changing minimum flow length had on classification accuracy. Classifier accuracy increased sharply from a flow length of one packet to a flow length of seven packets and remained constant (more or less) afterwards. This is understandable since shorter flows carry less information, and as a result, we expect the classifiers to make more errors when classifying shorter flows. A flow length of seven is a good choice of minimum flow length because it is the length of the shortest "complete" flow; i.e., a flow containing a TCP handshake (three packets) followed by an HTTP request,

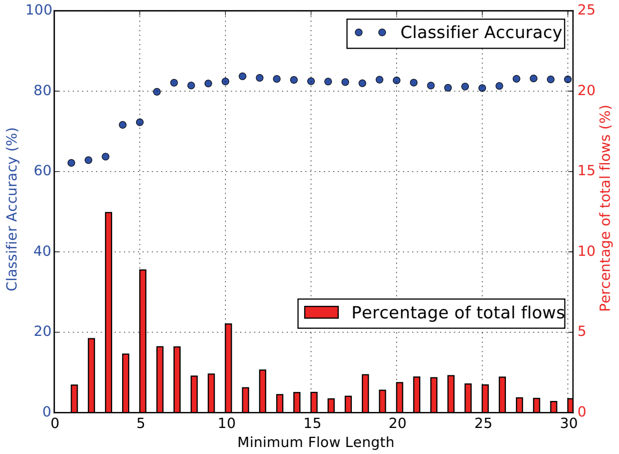


Figure 3: Impact of minimum flow length on classifier accuracy for the Per Flow Random Forest Classifier.

response, and acknowledgements (four packets). Note that we do not consider TCP session termination packets in the length of a shortest complete flow. This is because the *burst threshold* will usually occur before the TCP session termination packets, and as such they would never be a part of a flow. For these reasons, a minimum flow length of seven was used for the remainder of the tests. Of course, this can be easily adjusted based on any other specific needs. The other classification approaches yielded plots with a similar behaviour for classifier accuracy vs. minimum flow length and are omitted for brevity.

Using a minimum flow length of seven and an arbitrary maximum flow length of 260, our interactive traffic contained 131,736 flows which was split 75%/25% for the training/testing sets respectively. We used a maximum flow length of 260 since this was the length of the longest flow observed in our training data. This value can be easily adjusted depending on the maximum flow length expected in a typical usage scenario. We trained the classifiers with features from the training set and their accuracy was measured by comparing their predictions to the ground truth from the testing set. For this first round of tests, no classification validation was used. This was to aid our understanding of how the classifiers would perform without any additional post-processing. Fig. 4 shows the resulting confusion matrix for our *Per Flow Random Forest Classifier* (Approach 2). For brevity, we show only one confusion matrix since the other classification approaches yielded similar plots. Furthermore, in the confusion matrix itself, instead of showing app names, each of the 110 apps are assigned a unique number (0-109) on the axes. The y-axis shows the true apps responsible for the flows, while the x-axis shows the predicted apps that were output from the classifiers. The cells in the confusion matrix show how well each flow was classified, with a darker colour depicting more accurate classification.

Next we calculated precision, recall, and accuracy for our six classification approaches. Where *TP* refers to the number of true positives, *FP* refers to the number of false positives, *FN* refers to the number of false negatives, and *TN* refers to the number of true negatives: precision was calculated using the formula $TP/(TP + FP)$, and recall was calculated using the formula $TP/(TP + FN)$. For Approaches 1-4 (the multi-class classifiers), accuracy was calculated as the total number of correct classifications divided by the total number of classifications. Approaches 5-6 involved binary classifiers so accuracy was calculated as $(TP+TN)/(TP+FP+TN+FN)$. The results are reported in Table 4. Without using classification validation, AppScanner had best overall performance with *Per App Random Forest Classifiers* trained on statistical features from network flows (Approach 6). These classifiers had an overall precision of 96.0%, recall of 82.5%, and accuracy of 99.8% for our test set of 110 apps. Our *Per App Support Vector Classifiers* (Approach 5) had comparable precision and accuracy, but a lower recall of 64.8%. Given that no classification validation had been used with the results presented in Table 4, these are the worst case performance figures that can be expected from AppScanner using these classification approaches.

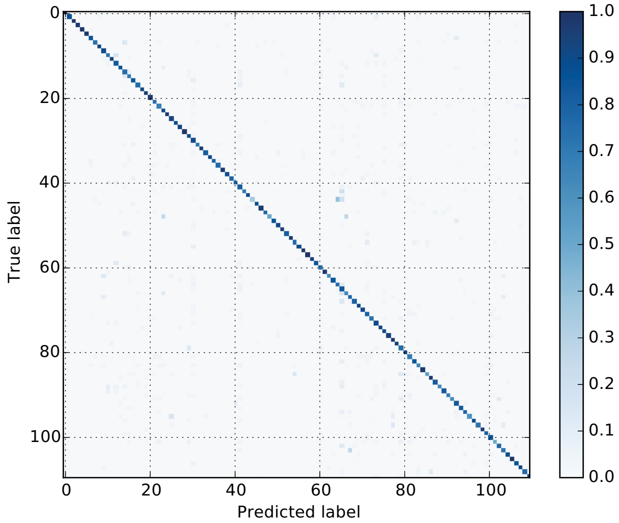


Figure 4: Normalized confusion matrix showing actual classes vs. predicted classes for the Per Flow Random Forest Classifier.

For the next round of tests, we wanted to measure the impact that increasing the number of classes had on classification accuracy for our multi-class classifiers (Approach 1-4). We started with a set size of 10 apps which were chosen randomly from our test set of 110 apps. The classification performance was measured. This test was repeated 50 times (with random sets of the same set size) and the results averaged. This entire process was repeated, each time increasing the app set size by 10, until we had the maximum set size of 110. The result of these tests are shown in Fig. 5. From the figure we can see that increasing the number of apps in the classifiers causes precision, recall, and overall accuracy of classification to decrease. This is not

TABLE 4: Table showing classifier performance for the six classification approaches: Per Flow SVC, Per Flow Random Forest Classifier, Single Large SVC, Single Large Random Forest Classifier, Per App SVC, and Per App Random Forest Classifier.

#	Approach	Precision	Recall	Accuracy
1	Per Flow SVC	77.1%	71.9%	71.5%
2	Per Flow Random Forest	84.4%	83.1%	82.1%
3	Single Large SVC	51.3%	60.2%	42.4%
4	Single Large Random Forest	89.5%	85.9%	86.9%
5	Per App SVC	96.1%	64.8%	99.7%
6	Per App Random Forest	96.0%	82.5%	99.8%

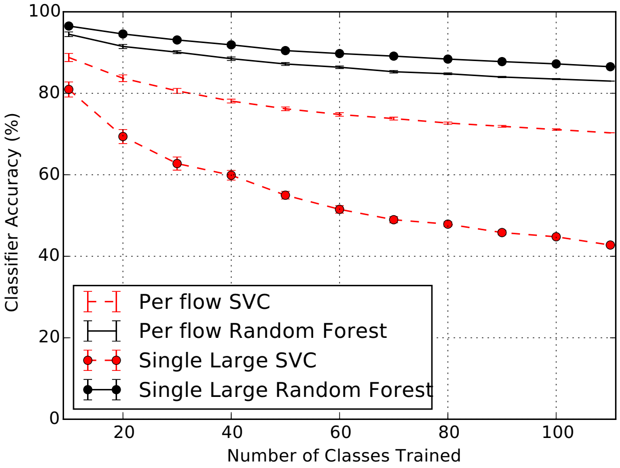


Figure 5: Impact of the number of apps trained in the classifiers on classifier performance for the four multi-class classifiers. Error bars show 95% CI for the mean.

unexpected, since the accuracy of a multi-class classifier is a function of the number of classes that an unknown input can be matched to. What is important to note, however, is that as the number of classes is increased, the rate of decrease in classifier performance decreases. Thus we expect classifier performance to eventually level off and remain constant when the number of classes is significantly increased.

If one wanted to fingerprint the universe of apps, they would use the *Classifier Per Flow Length* strategy (Approach 1-2 in Section 4). This would ensure that no single classifier would contain a very large number of apps, since not all apps generate flows for each flow length. For this reason, we believe that scaling up AppScanner to identify the universe of apps is feasible. Such large-scale 'appscanning' would not be common, though, since we believe the typical application scenario would be to use AppScanner to target a certain subset of apps. For example, by fingerprinting the Top 10,000 apps, one would have a realistic coverage of all the apps that would be seen (with a non-trivial likelihood) on a given network. In other cases, we expect AppScanner to be deployed to only identify very specific apps (such as apps allowed/disallowed by company policy), in which case the *Single Classifier Per App* strategy (Approach 5-6 in Section 4) would be used.

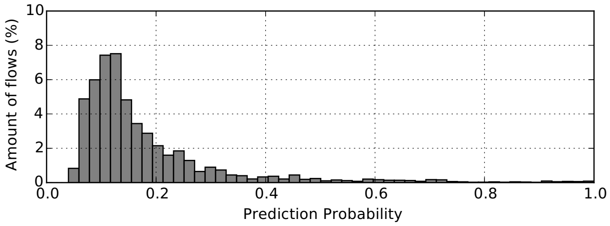


Figure 6: Histogram of prediction probabilities for each classification as outputted by the Single Large SVC.

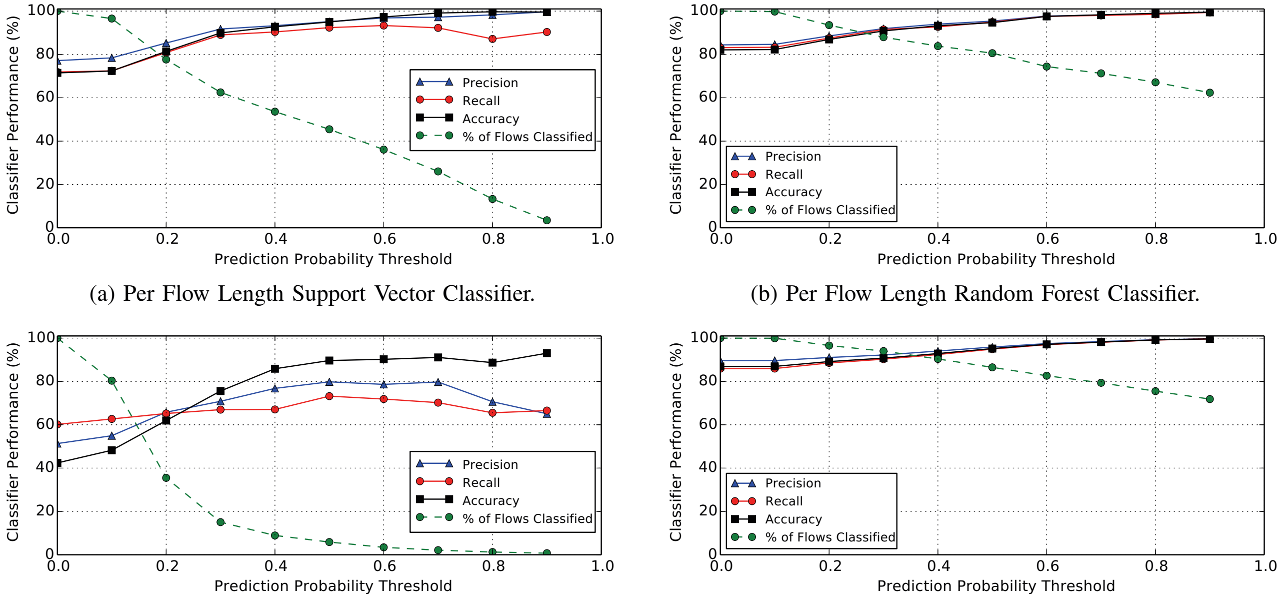
5.2. Using Classification Validation to Improve Performance

To understand the utility of the classification validation post-processing function (as detailed in Section 3.4), we looked at the confidence that our multi-class classifiers reported with each of their classifications. Fig. 6 shows a histogram of the prediction probabilities reported by our worst performing classifier (*Single Large SVC*) for the $\approx 33,500$ flows that were in the testing set. The prediction probability had a mean $\mu = 0.18$ with standard deviation $\sigma = 0.14$. For the vast majority of classifications, the classifier was less than 20% certain about its decision. Indeed, we can see from the figure that the classifier was only around 10-12% confident for a large number of choices.

In the case where more than one apps had similar flows, such as ad/analytics traffic or querying similar APIs, it is understandable that the classifiers would not be very confident in their classifications. This is so, because the class boundaries would not be as distinct as in the case where all apps had perfectly unique traffic. This suggests that classification validation can be a useful strategy for improving classification performance since we can set 'minimum standards' for what we will accept from the classifier as a confident classification. By using classification validation, we can free AppScanner from the task of making a decision on flows that are genuinely very ambiguous to the classifier.

Table 5 summarises the (sometimes) dramatic improvement in classification performance that we obtained by using classification validation. In general, the Random Forest Classifiers outperformed the Support Vector Classifiers for our dataset, whether a *Classifier Per Flow Length* or a *Single Large Classifier* was used. The Random Forest Classifiers use aggregated decision trees which, in turn, reduce bias. Also, they are better able to handle noise since they are an ensemble learning method. The Support Vector Classifiers are not very confident about their predictions and indeed it can be seen that the percentage of flows they were confident enough to classify falls off sharply as the prediction probability threshold is increased. The overall winner is the *Single Large Random Forest Classifier* (Approach 4) that used statistical features derived from flows. We now detail the performance of these classification approaches.

Fig. 7a shows classifier performance for our *Per Flow Length Support Vector Classifiers* (Approach 1). With no classification validation in use, precision, recall, and accu-



(a) Per Flow Length Support Vector Classifier.

(b) Per Flow Length Random Forest Classifier.

(c) Single Large Support Vector Classifier.

(d) Single Large Random Forest Classifier.

Figure 7: Impact of prediction probability threshold on classifier performance.

racy was 77.1%, 71.9%, and 71.5%, respectively, with the classifiers making a judgement on all the unlabelled flows. By setting the prediction probability threshold to a modest 0.5, precision, recall, and accuracy increased to 95.1%, 92.4%, and 95.0% respectively with the classifiers making judgements on just under a half (45.5%) of the unlabelled flows. From the figure it can be seen that accuracy in excess of 99% (accuracy of 99.1%, precision of 97.2%, recall of 92.3%) can be achieved by setting the prediction probability threshold to 0.7. At a threshold of 0.7, however, AppScanner will only be confident enough to make a judgement on roughly a quarter (26.0%) of flows. At higher thresholds, the number of flows classified falls off sharply with negligible improvement in performance.

Fig. 7b shows classifier performance for our *Per Flow Length Random Forest Classifier* (Approach 2). With no classification validation in use, precision, recall, and accuracy was 84.4%, 83.1%, and 82.1% respectively. At a threshold of 0.5, accuracy jumps to 94.7% and we can exceed 98% accuracy at a threshold of 0.7, while still classifying over 71% of flows. Although the *Per Flow Length Support Vector Classifiers* have a higher peak accuracy, the percentage of flows they can classify at higher thresholds makes them useful in only very specific circumstances.

Fig. 7c shows classifier performance for our *Single Large Support Vector Classifier* (Approach 3). This is our worst performing classifier. With no classification validation in use, precision, recall, and accuracy was 51.3%, 60.2%, and 42.4% respectively. By tuning the prediction probability threshold, additional performance can be squeezed from this classifier but it comes at the detriment of percentage of flows classified. The amount of flows classified falls off even more sharply than the same type of classifier used in a *Per Flow Length Classifier* approach (Approach 1). At a threshold of 0.5, accuracy was less than 90% and to achieve this, the classifier could only classify 5.9% of flows.

Fig. 7d shows classifier performance for our *Single Large Random Forest Classifier* (Approach 4). This is our best performing classifier. With no classification validation in use, precision, recall, and accuracy was 89.5%, 85.9%, and 86.9% respectively. At a threshold of 0.7, all three of precision, recall, and accuracy exceeded 98%, and at a threshold of 0.9, precision, recall, and accuracy all exceeded 99.5%. This near perfect accuracy is achieved while still being able to classify roughly three-quarters of all flows that were seen.

5.3. Understanding Classification Errors

Some classification approaches performed better than others both in terms of precision/recall/accuracy as well as percentage of flows classified when using classification validation. Some of the apps themselves also performed better than others when being classified by AppScanner. We expect that the better performing apps are those that have traffic flows that are very distinct from the flows of other apps. To test this hypothesis, we analysed our *Per Flow Random Forest Classifier* (Approach 2), the classifier from the group that had overall performance somewhere in the middle (not the best and not the worst). Table 6 shows the apps that were most accurately classified by this classifier (without using classification validation). We removed classification validation for this step to get a fuller idea of the types of flows that were being classified incorrectly. The package *air.uk.co.bbc.android.mediaplayer* was perfectly classified

TABLE 5: Table summarising multi-class classifier performance when classification validation is used.

Classification Approach	Prediction	Probability	Threshold
Per flow SVC	**0.5**	**0.7**	**0.9**
Precision	95.1%	97.2%	99.7%
Recall	92.4%	92.3%	90.4%
Accuracy	95.0%	99.1%	99.6%
% Flows classified	45.5%	26.0%	3.4%
Per flow Random Forest	**0.5**	**0.7**	**0.9**
Precision	95.4%	98.1%	99.5%
Recall	95.0%	98.0%	99.4%
Accuracy	94.7%	98.3%	99.5%
% Flows classified	80.6%	71.3%	62.3%
Single Large SVC	**0.5**	**0.7**	**0.9**
Precision	79.8%	79.7%	65.0%
Recall	73.2%	70.2%	66.5%
Accuracy	89.6%	91.1%	93.0%
% Flows classified	5.9%	2.1%	0.7%
Single Large Random Forest	**0.5**	**0.7**	**0.9**
Precision	95.9%	98.4%	99.7%
Recall	94.9%	98.0%	99.6%
Accuracy	95.2%	98.2%	99.6%
% Flows classified	86.5%	79.4%	72.0%

TABLE 6: Best 10 Apps for classification by AppScanner.

Rank	Package Name	Accuracy
1^{st}	*air.uk.co.bbc.android.mediaplayer*	100.0%
2^{nd}	*com.fivestargames.slots*	99.1%
3^{rd}	*bbc.mobile.weather*	98.7%
4^{th}	*me.pou.app*	97.3%
5^{th}	*com.machinezone.gow*	97.2%
6^{th}	*com.prettysimple.criminalcaseandroid*	97.0%
7^{th}	*com.justeat.app.uk*	95.8%
8^{th}	*com.king.farmheroessaga*	95.0%
9^{th}	*com.playfirst.cookingdashx*	94.6%
10^{th}	*com.whatsapp*	92.1%

in all cases with another 15 apps exceeding a classification accuracy of 90%. Other apps performed much worse as shown in Table 7. None of the flows from the package *com.google.android.apps.plus* were classified correctly by AppScanner when not using classification validation. Another 13 apps from our test set of 110 apps performed below the 50% mark with this setting. These apps seem to generate flows are harder to classify and thus produce more false positive and false negative results. To understand if this was the case, we did an in-depth analysis of the incorrectly classified flows to gain additional insight. With classification validation still removed, we performed another set of tests where AppScanner would make its best guess at what app a flow belonged to. We did this analysis using classification Approach 1; another approach where performance was in the middle (not the best and not the worst). We collected the $\approx 10,000$ flows (of $\approx 33,500$) that were classified incorrectly (using this classification approach) and performed manual/semi-automated analysis on them by destination IP address.

The $\approx 10,000$ incorrectly classified flows were going to some 1,467 unique destination IP addresses. It is interesting to note that the Top 25 of these IP addresses accounted for more than 30% of the incorrectly classified flows. For brevity, we report on the Top 10 destinations (for incorrectly classified flows) in Table 8. The table shows the domain names, the amount of incorrectly labelled flows going to each domain, the number of apps sending flows to each domain, and the type of service running at that domain. Note that the type of service running at a domain was inferred by manually doing research on the domain in the form of WHOIS queries, visiting the domain, analysing the subdomain etc. and thus may not be perfectly accurate in all cases.

TABLE 7: Worst 10 Apps for classification by AppScanner.

Rank	Package Name	Accuracy
101^{st}	*com.imo.android.imoim*	47.4%
102^{nd}	*com.snapchat.android*	41.8%
103^{rd}	*com.twitter.android*	41.0%
104^{th}	*com.zentertain.photoeditor*	40.8%
105^{th}	*com.mixradio.droid*	40.3%
106^{th}	*com.imangi.templerun2*	32.9%
107^{th}	*com.tayu.tau.pedometer*	28.6%
108^{th}	*com.meetup*	28.0%
109^{th}	*com.google.android.apps.inbox*	24.3%
110^{th}	*com.google.android.apps.plus*	0.0%

TABLE 8: Top 10 destinations for incorrectly labelled flows, number of flows going to these destinations, number of different apps sending these flows, and the type of service at each destination.

Domain Name	Flows	Apps	Type
graph.facebook.com	994	44	Standard API
googleads.g.doubleclick.net	426	16	Ads/Analytics
data.flurry.com	142	26	Ads/Analytics
googleads.g.doubleclick.net	127	5	Ads/Analytics
data.flurry.com	121	26	Ads/Analytics
android.clients.google.com	98	14	Standard API
spotlight-endpoint.appspot.com	96	1	Standard API
api.meetup.com	70	1	App-specific API
d.appsdt.com	70	9	Ads/Analytics
d.appsdt.com	67	8	Ads/Analytics

According to Table 8, with the exception of two domains, all domains in our Top 10 ‘worst classification domains’ list received traffic from more than one app. The remainder of these ‘multi-app’ domains served either standard developer APIs or advertisement/analytics resources. This supports our hypothesis that similar flows were indeed being sent by more than one apps as a result of them contacting the same standard web services. In this case, there is not much that can be done to assist AppScanner to differentiate the exact source of these network flows without leveraging additional features. However, classification accuracy could be improved if AppScanner were allowed to be more general and label a flow as being advertisement/analytics/standard-API traffic instead of naming a specific app.

6. Discussion and Future Work

AppScanner is the implementation of a novel methodology that leverages machine learning and traffic analysis to automatically fingerprint and identify smartphone apps. The smartphone landscape offers unique challenges to traffic analysis, such as less available features and the need for automation and high-scalability. Our classification framework based on flows offers novel insights. We explore three general classification strategies (i.e. classifier per flow length, single classifier with all apps, and single classifier per app) and explore and enumerate the trade-offs of each strategy in terms of time taken to train classifier, size of resulting classifier, and classification performance.

Our framework is able to very accurately identify apps from their network traffic but it also has some limitations. We discuss these limitations in Section 6.1, we compare AppScanner to the state of the art in Section 6.2, and talk about future work in Section 6.3.

6.1. Limitations

Table 7 shows that AppScanner was much worse at identifying some apps such as Temple Run, Pedometer, MeetUp, Inbox by Gmail, and Google+. This, we think, is as a result of these apps having very generic flows of traffic. This hypothesis is supported by Table 8, where we see that the most incorrectly labelled flows were from multiple apps going to similar destinations. The simple fact is that ambiguous flows are harder to classify and AppScanner (or any other system) would not be able to reliably differentiate between these flows without leveraging additional features.

AppScanner was built with a single device that generated the training and testing flows. It is possible that apps may behave differently on different devices or different versions of Android. It is also possible that different flavours of TCP on different devices may cause our classifiers to misclassify if they were trained using network traces from a different device. We plan to test this by generating flows from apps using various Android emulators running on virtual machines.

6.2. Comparison with website fingerprinting methods

Since the domain of this paper is smartphone app fingerprinting, and the closest related work we identified in Section 2 focuses mainly on website fingerprinting, a direct comparison between AppScanner and the related work cannot be made. However, we validate the necessity and utility of AppScanner by showing how the existing website traffic analysis techniques in the literature (for which there exists ground truth) perform below par when classifying the smartphone app traffic from our dataset. The results of our comparison are shown in Fig. 8.

The first group of approaches considered for the comparison are the ones proposed by Liberatore et al. in [11]: a classifier leveraging the Jaccard similarity metric (i.e., *Liberatore Jaccard*) and another leveraging a Naive Bayes classifier (i.e., *Liberatore NB*). Among these two classifiers, *Liberatore NB* achieves the best accuracy with 50.8%. The second group of approaches were presented by Herrmann et al. in [12]. The difference with the proposals in this group is that transformations are applied to the dataset: no transformation (i.e., *Herrmann Pure*), Term Frequency transformation (i.e., *Herrmann TF*), and Cosine Normalization applied after a TF transformation (i.e., *Herrmann Cos*). The best performance is 50.2% accuracy, achieved by the TF transformation without the Cosine Normalization, i.e., *Herrmann TF*. Finally, the method proposed by Panchenko et al. in [13] performed best with an accuracy of 64.5%. As we can see from Fig. 8, this is the approach with performance closest to ours. However, five out of six of our classifiers outperform it, with our two best approaches outperforming it by some 35% accuracy. Our worst four approaches, when using classification validation (with a modest prediction probability threshold of 0.5) outperform Panchenko et al. by 25%-30% accuracy.

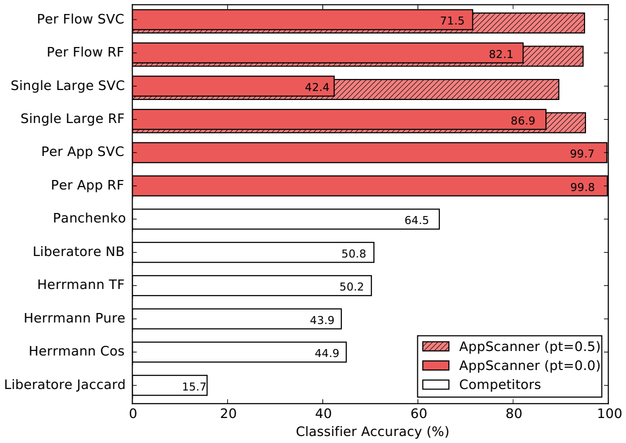


Figure 8: AppScanner's accuracy compared to existing approaches from the literature. We use *pt* to denote the prediction probability threshold used for classification validation. RF means Random Forest Classifier.

6.3. Future Work

For future work, we will examine ways of grouping flows to more reliably determine the originating app. For example, three flows may be ambiguous when analysed separately, but when assessed as a group, they may match a particular app that always sends three of these flows together. We will also look at other approaches for identifying apps, such as active network probing, which can be used to elicit further identifying network traffic from apps. We intend to use different modelling tools, such as Hidden Markov Models and finite state machines, for app classification. We also plan to improve classification accuracy by identifying and using other features from flows, such as packet inter-arrival time. Other readily available information such as whether a flow always occurs within a burst with multiple flows or whether it contains HTTPS/TLS packets can also be leveraged to improve accuracy. Finally, we plan to examine the extent to which app fingerprinting can be done at the MAC layer in the presence of MAC layer encryption.

7. Conclusion

In this paper, we presented AppScanner, a framework implementing a novel methodology for the automatic fingerprinting and real-time identification of smartphone apps from their encrypted network traffic. Our evaluation shows that apps can indeed be identified with over 99% accuracy even in the presence of encrypted traffic streams such as HTTPS/TLS. We validated that multi-class classifiers can be used to fingerprint and identify a wide variety of apps in a single classifier. We also showed that binary classifiers can also be used to obtain very high precision and overall accuracy in the case where only certain apps are of interest. Undoubtedly, smartphone usage will continue to increase as app developers continue to provide new apps to consumers to satisfy their insatiable appetites. As a result, more and more actors will become interested in fingerprinting and identifying these apps for both benevolent and malevolent reasons. By continuing research in this area we hope to gain a better understanding of the privacy and security risks that end users currently face. In this way, we can continue on the path of helping to preserve privacy and security now and into the future.

Acknowledgement

Vincent F. Taylor is supported by a Rhodes Scholarship and the UK EPSRC. Mauro Conti is supported by a Marie Curie Fellowship (PCIG11-GA-2012-321980). This work is also supported by the projects EU TagItSmart! (H2020-ICT30-2015-688061), EU-India REACH (ICI+/2014/342-896), Italian PRIN TENACE (20103P34XC), and University of Padua PRAT-2013 on Malware Detection.

References

[1] Gartner. (2015, March) Gartner says smartphone sales surpassed one billion units in 2014. [Online]. Available: http://www.gartner.com/newsroom/id/2996817

[2] Flurry. (2015, January) Shopping, productivity and messaging give mobile another stunning growth year. [Online]. Available: http://www.flurry.com/blog/flurry-insights/shopping-productivity-and-messaging-give-mobile-another-stunning-growth-year

[3] T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 4, pp. 56–76, October 2008.

[4] S. Seneviratne, A. Seneviratne, P. Mohapatra, and A. Mahanti, "Predicting User Traits from a Snapshot of Apps Installed on a Smartphone," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 18, no. 2, pp. 1–8, Jun. 2014.

[5] A. Hintz, "Fingerprinting Websites Using Traffic Analysis," in *Privacy Enhancing Technologies*. Springer Berlin Heidelberg, 2003, vol. 2482, pp. 171–178.

[6] H.-S. Ham and M.-J. Choi, "Application-level traffic analysis of smartphone users using embedded agents," in *14th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, September 2012, pp. 1–4.

[7] J. Yang, S. Zhang, X. Zhang, J. Liu, and G. Cheng, "Analysis of smartphone traffic with mapreduce," in *22nd Wireless and Optical Communication Conference (WOCC)*, May 2013, pp. 394–398.

[8] S.-W. Lee, J.-S. Park, H.-S. Lee, and M.-S. Kim, "A study on smartphone traffic analysis," in *13th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, Sept 2011, pp. 1–7.

[9] H. Bacic. (2015, March) Are You Using A Content Delivery Network For Your Website Yet? You Should Be. [Online]. Available: http://www.forbes.com/sites/allbusiness/2015/03/16/are-you-using-a-content-delivery-network-for-your-website-yet-you-should-be/

[10] J.-F. Raymond, "Traffic analysis: Protocols, attacks, design issues, and open problems," in *Designing Privacy Enhancing Technologies*. Springer, 2001.

[11] M. Liberatore and B. N. Levine, "Inferring the Source of Encrypted HTTP Connections," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS '06. ACM, 2006, pp. 255–263.

[12] D. Herrmann, R. Wendolsky, and H. 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*, ser. CCSW '09. ACM, 2009, pp. 31–42.

[13] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel, "Website Fingerprinting in Onion Routing Based Anonymization Networks," in *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society*, ser. WPES '11. ACM, 2011, pp. 103–114.

[14] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson, "Touching from a Distance: Website Fingerprinting Attacks and Defenses," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. ACM, 2012, pp. 605–616.

[15] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song, "NetworkProfiler: Towards automatic fingerprinting of Android apps," *Proceedings of the 32nd IEEE INFOCOM*, pp. 809–817, Apr. 2013.

[16] Q. Wang, A. Yahyavi, M. Kemme, and W. He, "I Know What You Did On Your Smartphone: Inferring App Usage Over Encrypted Data Traffic," in *3rd IEEE Conference on Communications and Network Security (CNS)*, 2015.

[17] M. Conti, L. Mancini, R. Spolaor, and N. Verde, "Analyzing Android Encrypted Network Traffic to Identify User Actions," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 1, pp. 114–125, Jan 2016.

[18] T. Stöber, M. Frank, J. Schmitt, and I. 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*, ser. WiSec '13. ACM, 2013, pp. 7–12.

[19] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An Input Generation System for Android Apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 224–234.

[20] Pragmatic Software, "Network Log," April 2014. [Online]. Available: https://play.google.com/store/apps/details?id=com.googlecode.networklog

[21] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin, "A First Look at Traffic on Smartphones," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '10. New York, NY, USA: ACM, 2010, pp. 281–287.

[22] W. McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Conference on Python in Science*, S. van der Walt and J. Millman, Eds., 2010, pp. 51 – 56.

[23] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[25] Google Inc. Top charts - Android Apps on Google Play. [Online]. Available: https://play.google.com/store/apps/top

Appendix

TABLE 9: List of apps in the AppScanner testing set. App details were obtained from the Google Play Store [25].

#	Package Name	Number of Installs
1	com.amazon.kindle	100-500 million
2	com.dictionary	10-50 million
3	com.iconology.comics	1-5 million
4	com.google.android.gm	1-5 billion
5	com.imo.android.imoim	50-100 million
6	kik.android	50-100 million
7	com.facebook.orca	500 million-1 billion
8	com.skype.raider	500 million-1 billion
9	com.viber.voip	100-500 million
10	com.whatsapp	1-5 billion
11	com.yahoo.mobile.client.android.mail	100-500 million
12	com.digidust.elokence.akinator.freemium	10-50 million
13	bbc.iplayer.android	10-50 million
14	air.uk.co.bbc.android.mediaplayer	10-50 million
15	com.imdb.mobile	50-100 million
16	air.ITVMobilePlayer	5-10 million
17	com.itv.loveislandapp	50,000-100,000
18	com.netflix.mediaclient	100-500 million
19	hu.tonuzaba.android	10-50 million
20	com.bskyb.skygo	1-5 million
21	tv.twitch.android.app	1-5 million
22	com.miniclip.agar.io	1-5 million
23	com.yodo1.crossyroad	10-50 million
24	com.imangi.templerun2	100-500 million
25	com.joycity.warshipbattle	1-5 million
26	com.prettysimple.criminalcaseandroid	10-50 million
27	com.rovio.angrybirds	100-500 million
28	com.nordcurrent.canteenhd	5-10 million
29	com.umonistudio.tile	50-100 million
30	com.halfbrick.fruitninjafree	100-500 million
31	com.robtopx.geometryjumplite	50-100 million
32	com.boombit.RunningCircles	1-5 million
33	com.boombit.Spider	5-10 million
34	com.kiloo.subwaysurf	100-500 million
35	com.mobilityware.solitaire	50-100 million
36	com.leftover.CoinDozer	50-100 million
37	com.fivestargames.slots	5-10 million
38	com.outplayentertainment.bubbleblaze	5-10 million
39	com.king.candycrushsaga	100-500 million
40	com.king.candycrushsodasaga	100-500 million
41	com.playfirst.cookingdashx	1-5 million
42	com.gameloft.android.ANMP.GloftDMHM	100-500 million
43	air.com.puffballsunited.escapingtheprison	5-10 million
44	com.king.farmheroessaga	100-500 million
45	com.outfit7.mytalkingtomfree	100-500 million
46	com.jellybtn.cashkingmobile	10-50 million
47	me.pou.app	100-500 million
48	com.king.alphabettysaga	10-50 million
49	com.ciegames.RacingRivals	10-50 million
50	com.igg.android.finalfable	1-5 million
51	com.fungames.flightpilot	1-5 million
52	com.miniclip.eightballpool	50-100 million
53	com.BitofGame.MiniGolfRetro	1-5 million
54	com.supercell.boombeach	10-50 million
55	com.supercell.clashofclans	100-500 million
56	com.hcg.cok.gp	10-50 million
57	com.bigkraken.thelastwar	1-5 million
58	com.machinezone.gow	10-50 million
59	com.myfitnesspal.android	10-50 million
60	com.tayu.tau.pedometer	1-5 million
61	com.runtastic.android	10-50 million
62	com.northpark.drinkwater	5-10 million
63	info.androidz.horoscope	10-50 million
64	uk.co.dominos.android	1-5 million
65	com.gumtree.android	1-5 million
66	com.justeat.app.uk	1-5 million
67	com.tinder	10-50 million
68	com.mobilemotion.dubsmash	50-100 million
69	com.google.android.youtube	1-5 billion
70	com.mixradio.droid	1-5 million
71	com.shazam.android	100-500 million
72	com.soundcloud.android	50-100 million
73	com.spotify.music	50-100 million
74	com.cnn.mobile.android.phone	10-50 million
75	net.zedge.android	100-500 million
76	com.instagram.layout	10-50 million
77	com.zentertain.photoeditor	50-100 million
78	com.google.android.apps.photos	1-5 million
79	com.dropbox.android	100-500 million
80	com.google.android.apps.inbox	5-10 million
81	com.microsoft.office.outlook	10-50 million
82	com.surpax.ledflashlight.panel	100-500 million
83	com.amazon.mShop.android.shopping	10-50 million
84	com.ebay.mobile	100-500 million
85	com.groupon	10-50 million
86	com.shpock.android	5-10 million
87	com.contextlogic.wish	10-50 million
88	com.badoo.mobile	50-100 million
89	com.facebook.katana	1-5 billion
90	com.google.android.apps.plus	1-5 billion
91	com.instagram.android	500 million-1 billion
92	com.chatous.pointblank	5-10 million
93	com.meetup	1-5 million
94	com.pinterest	50-100 million
95	com.snapchat.android	100-500 million
96	com.twitter.android	100-500 million
97	com.qihoo.security	100-500 million
98	com.adobe.air	100-500 million
99	com.cleanmaster.mguard	100-500 million
100	com.lazyswipe	50-100 million
101	com.avast.android.mobilesecurity	100-500 million
102	uk.co.nationalrail.google	1-5 million
103	com.thetrainline	1-5 million
104	com.airbnb.android	5-10 million
105	com.booking	10-50 million
106	com.joelapenna.foursquared	10-50 million
107	com.google.earth	100-500 million
108	com.google.android.apps.maps	1-5 billion
109	com.tripadvisor.tripadvisor	100-500 million
110	bbc.mobile.weather	5-10 million