Gerald Palfinger, BSc.

# SCAnDroid: Automatic Side-Channel Detection Framework for the Android API

**MASTER'S THESIS**

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Software Engineering and Management

submitted to

**Graz University of Technology**

Supervisor

Raphael Spreitzer

Institute of Applied Information Processing and Communications

Assessor

Prof. Stefan Mangard

Graz, May 2018

## AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

# Abstract

In the last few years, numerous side channels have been identified in the Android API and reported to Google. Most of the side channels have been detected by manually inspecting a small subset of the Android API. However, by manually choosing methods to be investigated, information leaks may be overlooked. Therefore, we propose a more systematic approach by introducing the automatic side-channel detection framework SCAnDroid. The framework triggers events of interest while invoking methods of the Android API and recording the return values. These recordings are then analysed for correlations with the help of dynamic time warping.

We demonstrate the applicability of the SCAnDroid framework by exposing side channels in several parts of the Android API. We show that these side channels can be exploited to infer various events, such as website invocations, application launches, and Google Maps search queries. In particular, the global network statistics released via the `android.net.TrafficStats` API allow us to deduce the executed events with high accuracy. Furthermore, storage statistics released via the Java `java.io.File` API, Android's `android.os.storage.StorageManager`, and `android.app.usage.StorageStatsManager` APIs also leak information about the executed event. Some of these methods have only recently been added in Android 8, the latest stable release. Compared to a manual investigation, SCAnDroid can be automatically applied to recently added methods in new versions of the Android API. Therefore, SCAnDroid represents a valuable framework for hardening the Android API against side-channel attacks.

# Kurzfassung

In den letzten Jahren wurden unzählige Seitenkanäle in der Android API identifiziert und an Google gemeldet. Die meisten der Seitenkanäle wurden durch die manuelle Inspektion eines kleinen Teiles der Android API erkannt. Durch die manuelle Auswahl der zu untersuchenden Methoden können Seitenkanäle jedoch übersehen werden. Deswegen stellen wir mit dem automatischen Seitenkanalerkennungsframework SCAnDroid eine systematischere Herangehensweise vor. SCAnDroid startet Ereignisse, während es gleichzeitig Methoden der Android API aufruft und die Rückgabewerte dieser Methoden speichert. Diese Aufnahmen werden dann mit Hilfe von Dynamic-Time-Warping auf Korrelationen untersucht.

Wir demonstrieren die Anwendbarkeit von SCAnDroid durch das Aufdecken von Seitenkanälen in verschiedenen Teilen der Android API. Wir zeigen, dass die gefundenen Seitenkanäle ausgenutzt werden können um auf verschiedene Ereignisse zurückzuschicken, wie zum Beispiel das Öffnen von Webseiten, das Starten von Applikationen, oder Sucheingaben in Google Maps. Vor allem die globalen Netzwerkstatistiken, welche über die `android.net.TrafficStats` API abgerufen werden können, erlauben es auf die ausgeführten Ereignisse mit hoher Genauigkeit aus den veröffentlichten Daten zu schließen. Zusätzlich ermöglichen es uns auch Speicherstatistiken, wie sie über die Java `java.io.File` API, `android.os.storage.StorageManager` API, und die `android.app.usage.StorageStatsManager` API veröffentlicht werden, auf die ausgeführten Ereignisse zurückzuschließen. Einige der genannten Methoden wurden erst kürzlich, mit Version 8 des Android Betriebssystems, eingeführt. Im Vergleich zu einer manuellen Untersuchung kann SCAnDroid automatisch auf neu hinzugefügte API Methoden angewandt werden. Dadurch ist SCAnDroid ein nützliches Framework, um die Android API gegen Seitenkanalattacken abzusichern.

# Acknowledgements

I would like to express my gratitude to my thesis supervisor Raphael Spreitzer for his helpful comments, support, and assistance during the implementation and writing of this thesis. I am also thankful for his review of the draft of this thesis. I would also like to thank my parents, Helga and Kurt Palfinger, for their endless support throughout my studies and during the realisation of my master thesis. I am also grateful to Professor Stefan Mangard who introduced me to the topic, supervised, and assessed this work.

# Contents

# List of Figures

# List of Tables

# 1. Introduction

Smartphones have become omnipresent in our daily lives. We use them to store our most personal photos, exchange secret messages with friends and relatives, and may even use our personal device at work due to the trend of bring your own device (BYOD). Therefore, a strict separation between the different applications running on the device is needed. On Android, this separation is achieved by sandboxing each application with the help of the underlying Linux kernel. Thus, an application is prevented from directly accessing data stored by another application. Furthermore, access to resources which are considered sensitive is managed by Android's permission system. However, shared resources which are deemed as insensitive and, therefore, available without a permission, may still leak our private data to untrusted applications. Several attacks have been conducted in the literature by exploiting these seemingly benign resources. In such side-channel attacks, the resources are used to deduce, for example, sensitive events and, therefore, circumvent the previously mentioned security measures. The identified side channels include usage statistics which can be utilised to infer, e.g., opened websites [49, 77, 78] or running applications [11, 36, 78, 82], and sensor readings which may leak, for example, entered unlock patterns and passwords [4, 5, 7, 8, 62, 76, 81], or even the user's daily commuting routes [45, 58, 59].

Until recently, most of the published side-channel analyses on smartphones have been identified manually, by choosing to profile specific resources. However, manually finding and analysing possible side-channel candidates is an elaborate and time-consuming undertaking. To automate this task, Spreitzer et al. [78] have introduced a more systematic approach in a tool called ProcHarvester. The framework is designed to automatically find and analyse side channels in the procfs [74] of Android, which contains various process-specific statistics. However, to our knowledge, no such tool exists

for the Android API. Therefore, we introduce SCAnDroid, a framework for the automatic detection and analysis of side channels in the Android API. In the first step, SCAnDroid searches for possible side-channel candidates in the Android API. In the second step, it invokes these methods while launching events of interest and, in the third step, it analyses the gathered data for correlations. Using this approach, we make the process of finding and evaluating side channels in the Android API more systematic and, in turn, improve the security of the stored and processed private information on Android smartphones by reporting our findings to Google and by open-sourcing SCAnDroid.

## 1.1. Contributions

- **Implementation:** We introduce the SCAnDroid framework for the automatic detection and analysis of side channels in the Android API. We extend the ProcHarvester framework [78] to automate the profiling of methods in the Android API. SCAnDroid utilises the analysis framework based on dynamic time warping from the ProcHarvester implementation to detect side channels. Furthermore, it builds on the communication framework used in ProcHarvester to launch events and communicate between the smartphone and the PC.
- **Utilisation:** We apply SCAnDroid to explore the Android API for possible side channels systematically. We assess the information leakage by recording the return values of API methods while triggering different events of interest, namely website invocations, application starts, Google Maps searches, and keyboard gestures.
- **Evaluation:** By running SCAnDroid on a device with the latest Android release, version 8.1, we uncover several side channels in different parts of the Android API, allowing us to infer the launched events of interest.

The results of this work have been published at ACM WiSec 2018.

## 1.2. Outline

In Chapter 2, we discuss the background and related work. In Chapter 3, we give an overview of the SCAnDroid framework and the steps taken to profile the Android API for information leaks automatically. In Chapter 4, we demonstrate the capabilities of SCAnDroid by investigating four different attack scenarios and present the results. In Chapter 5, we discuss possible countermeasures, how SCAnDroid can help to harden new APIs, limitations of the framework, and possible future work. Finally, Chapter 6 concludes this work.

# 2. Preliminaries

In this chapter, we give an overview of the related work and present background information.

## 2.1. Related Work

In this section, we shortly introduce general side-channel attacks, explore related work in the field of software-based side-channel attacks with a focus on smartphones, and tools to automate the software-based side-channel analysis process.

### 2.1.1. Side-Channel Attacks

The discovery of side-channel-based information leakage dates back to at least World War II [39]. Back then, Bell Labs detected that (part of) the plaintext their encryption device was processing could be inferred from, e.g., electromagnetic radiation and acoustic emanations. Conductors like signal or power lines could amplify the signal and made it possible to infer secrets over considerable distances. Since then, side channels have been found in many devices and software products handling sensitive information. In general, side channels are not considered as an exploit of specific vulnerabilities in, e.g., software or hardware. Side channels represent leaks in information published either unintentionally, as in the example above, or intentionally by publishing seemingly harmless information [79]. Side-channel attacks include attacks on cryptographic implementations of mathematically secure primitives (see [80]). However, side channels have also been found in numerous other applications, such as web applications (see

[12]), operating systems (see [46]), or mobile devices [79]. The attack vectors include, for example, electromagnetic emanations [2], timing behaviour [40], and power consumption [50] of a device. Attackers can also play an active role and may, for example, try to cause faults [6].

### 2.1.2. Software-Based Side-Channel Attacks

In this section, we will concentrate on software-based side-channel attacks. Specifically, we focus on software-based side channels which have been demonstrated on Android smartphones. All of the attacks mentioned can be carried out remotely via the installation of a seemingly benign application. The attacks are also passive, *i.e.*, the attacker does not try to influence the device actively and only observes specific properties of the smartphone and its operating system. The information used in these attacks is usually gathered via the Android API or the procfs/sysfs.[1] All of the side channels are exploitable without requiring permissions that have to be granted by the user. However, some of the exploited information channels have been restricted in newer versions of Android.

#### Memory Usage Statistics

Jana and Shmatikov [49] were able to deduce a user's browsing behaviour using the RAM usage of the web browser, such as Chrome and Firefox. The attack utilised the process-specific memory usage of the browser which was published via the procfs. They exploited the observation that, depending on the opened website, the memory footprint of the browser differs. The authors used the Jaccard index for classification. To further improve the accuracy of their work, they also used CPU scheduling statistics to distinguish the entered URLs by their length.

[1] The procfs and sysfs are pseudo file systems inherited from the Linux kernel, which provide various statistics and data about the system, such as process statistics or CPU details [74, 75].

### Interrupt Timings

Simon et al. [73] proposed an attack to infer entered text on keyboards using gestures. In their work, they used the interrupt counter of the smartphone display and the global context-switch counter. These system-wide statistics are provided via the procfs. Whenever the user touches the screen, the interrupt counter of the display gets increased. Furthermore, the keyboard application switches between kernel and user space while processing the events, increasing the context-switch counter. Even though other applications running in the background may also increase the context-switch counter, they were able to deduce the entered text from a list of sentences of interest. For the inference, a recurrent neural network was used in the paper. The side channels also allowed them to identify users in message boards based on inferring the messages they submitted.

Diao et al. [36] exploited the interrupt counters of the display sub-system to infer the unlock pattern and running applications. This information is again released via the procfs. In their paper, a hidden Markov model was used to deduce possible unlock patterns which significantly reduced the search space compared to random guessing. For the application inference, dynamic time warping was used.

### Shared Memory Statistics

Chen et al. [11] utilised process-specific shared memory counters released via the procfs to infer currently running applications and their state. Android, like many other operating systems, uses shared memory to communicate between the current application and the window compositor. The window compositor is responsible for drawing the application window on the screen. As different activities use a different number of user interface elements, the shared memory usage differs. The authors used a hidden Markov model to infer transitions between activities and the activity itself. In their experiments, they deduced the running activity in seven different applications, such as WebMD, Gmail, or the banking application Chase. To improve the inference accuracy, they also considered the process-specific CPU-utilisation time and process-specific network usage statistics, such

as the size of sent packets and destination IP addresses. They showcased the power of these side channels by launching a fraudulent login activity directly after an application started its login activity. Therefore, this attack could be used to steal user data. The authors presented such an attack against two of the profiled applications. Furthermore, they used the side channel to infer when the camera was opened and later released by the Chase application to capture images of cheques while the user was still pointing at them.

### Data Usage Statistics

Zhou et al. [87] used process-specific data usage statistics as provided by the Android API and via the procfs to infer sensitive information about the user. They were able to collect the user identities by observing the data statistics of the Twitter application. Furthermore, the data usage of the WebMD application allowed them to infer medical conditions. The data usage of the Yahoo! Finance application enabled them to infer stocks a user is observing. The authors built a signature database by executing the events while running a network sniffing application on their device. The signatures accounted for differences in the size of the sent and received network packets. For example, the size of a sent Twitter message may have varied by 140 bytes, which was the maximum message size of a Twitter message back then. For the WebMD application, they built a finite state machine comprising of such signatures. In the attack phase, they compared the recorded data usage with the pre-built signatures. Process-specific data usage statistics have since been restricted.

Spreitzer et al. [77] showed that it was possible to use the application-specific data usage statistics to infer the web browsing behaviour of the user. The experiments were conducted with different browser applications, including Chrome and Firefox. Even when surfing via the anonymity network Tor using the Tor Browser for Android (Orfox), these statistics allowed to infer website launches with high accuracy. They used a classifier based on the Jaccard index. The authors stressed that, in combination with sensor-based side-channel attacks [56, 62, 64], this attack could be used to infer the visited websites based on the data usage and to steal the login credentials with the

help of the sensor data. Therefore, combinations of the presented attacks allow even more devastating attack scenarios which do not only pose a threat to the privacy but also the security of the user.

### Speaker Status

In addition to inferring personal information from applications via the data usage statistics as outlined in the previous section, Zhou et al. [87] were also able to deduce the user's location via the speaker status published through the Android API. Even though the speaker status only represents the binary information on or off, the researchers were able to infer driving routes of the user based on the length of the navigation guidance from the navigation application. The length was determined by polling the speaker status with high frequency. The authors simulated different driving routes to collect the length of the navigation guidance on these routes and utilised a variant of the Jaccard index for classification. They used Google Maps as navigation application as this application also announces the road name in addition to the driving direction. However, any navigation application announcing road names should be susceptible to this attack.

### Power Consumption Statistics

Michalevsky et al. [55] leveraged the power consumption of the smartphone to deduce the user's location. This attack is possible because the device consumes more power if it is further away from the connected cell tower or if obstacles are between the phone and the tower. Therefore, the signal strength can be approximated by the overall power consumption of the device. The authors used subsequence dynamic time warping and optimal subsequence bijection for the classification of the gathered data. These algorithms allowed them to match parts of the recorded routes to parts of the known routes. The power consumption has been obtained from the battery statistics in the sysfs.

Yan et al. [82] exploited the global power consumption statistics to infer running applications or to identify the UI state of the currently running

application. Similar to the previously mentioned attacks, this information could again be used by a malicious entity to conduct a phishing attack by starting a fake login screen when the user is supposed to enter username and password. Furthermore, the authors used the power consumption to infer the length of entered passwords. While they were not able to guess the actual password, the number of characters can still help to reduce the search space in brute-force attacks.

### Sensor Readings

Compared to traditional desktop computers, virtually all modern mobile devices, such as smartphones and tablets, contain a plethora of sensors. While these sensors enable a multitude of new features and use cases, such as automatic screen rotation, motion-steered games, and virtual reality applications, these sensors also pose a privacy and security threat. The sensors are available to developers via the Android API. Most of them do not require dedicated permissions. To use a sensor, an application usually requests access to the sensor by registering a callback interface. This interface gets called whenever the sensor reading changes. Different sensors have been exploited to infer, for example, user input or location.

Cai et al. [9] and Raij et al. [70] laid the groundwork for sensor-based attacks on smartphones by analysing the threat landscape and user awareness. Later on, Cai and Chen [8] utilised the accelerometer to derive touch input on a number-only keyboard. The probability density function of the Gaussian normal distribution was used to infer the keystrokes.

Owusu et al. [62] used the accelerometer to infer keypresses on a QWERTY keyboard. They used the random forest classification method. Furthermore, the researchers used their findings to reduce the search space for guessing 6-character passwords significantly.

In further publications, sensors were used to deduce the PIN or pattern which is entered to unlock the phone. Xu et al. [81] used the accelerometer and the orientation sensor to infer PIN input. The researchers constructed different features from the obtained data and used k-means clustering for

classification. Furthermore, Aviv [4] and Aviv et al. [5] used the accelerometer to deduce both PINs and patterns. Logistic regression and the hidden Markov model were used for the classification of the extracted features. The logistic regression classifier was used to deduce the entered PIN out of a set of 50 candidates, while the hidden Markov model was used to infer random PINs and patterns from a set of smaller training sequences. Furthermore, Cai and Chen [7] utilised the accelerometer and the gyroscope to infer 4-digit PINs. They used dynamic time warping and support vector machines (SVM) as classifiers. They evaluated the attack among different users, devices, and keyboard layouts and found that the sensors leaked information about the entered PIN code in all cases. Furthermore, they noted that the gyroscope readings collected in the study lead to a higher inference accuracy than the accelerometer traces.

Miluzzo et al. [56] inferred touch locations and entered English text by combining the readings of the gyroscope and the accelerometer. They used an ensemble approach for the inference, *i.e.*, they connected the output of multiple classifiers, namely k-nearest neighbour (kNN), SVM, random forests, multinomial logistic regression, and bagged decision trees. Again, the authors concluded that the gyroscope contributed the most to the inference accuracy. Ping et al. [64] also combined the gyroscope and the accelerometer to infer more extended bodies of text, such as typed Twitter messages and emails. The entered Twitter messages allowed the researchers to deduce the identity of the user by searching for the inferred text on Twitter. They used the shared memory side channel from [11] to detect the presence of the keyboard on the screen and the timing of user input. Furthermore, they applied language models to correct the inferred keypresses, and thereby, increased the accuracy on a letter and word level. For the classification of the typed letters, the authors used an ensemble approach combining linear logistic regression, random forests, SVM, and kNN classifiers.

Michalevsky et al. [54] exploited readings from the gyroscope to recover speech patterns of nearby persons. Due to the limited sampling rate of the gyroscope, not every word could be reconstructed from the gyroscope readings. Therefore, the authors limited the dictionary of recognisable words to single digits. Still, such a side channel could be used to, for example, deduce spoken credit card numbers. Combining gyroscope readings of the same statement from multiple smartphones allowed them to increase the

inference accuracy further. Before applying the speech recognition, they first identified the gender of the speaker, and subsequently also the speaker. For the gender identification, they used a binary SVM classifier, and for the speaker identification, they utilised a multi-class SVM and a Gaussian mixture model. For the actual speech recognition, dynamic time warping was used, as the same word could differ in its length when spoken multiple times or by different persons.

Spreitzer [76] proposed an attack using the ambient-light sensor to infer PIN input from a set of 50 candidates. The author used multiclass logistic regression, discriminant analysis, and kNN as classifiers. The attack was conducted successfully under different lighting conditions. The only premise for this attack to work is that the readings of the ambient-light sensor have to change during PIN input, which may not be the case in very bright or dark environments.

In addition to text input, the accelerometer and the gyroscope can also be exploited to infer driving routes. Nawaz and Mascolo [59] used the accelerometer and the gyroscope readings to detect repeated driving routes. They applied a dynamic time warping approach on the angular velocities to compensate for different journey times caused by, for example, traffic congestion and driving styles. Narain et al. [58] combined the accelerometer, gyroscope, and magnetometer to deduce driving routes in a city. They used a maximum likelihood estimator to calculate a list of the most likely courses based on the recorded trajectory. Possible routes were based on OpenStreetMap data of a specific city. The authors argued that the gyroscope readings provided the most useful data as the readings allowed to infer turn angles and road curvature best.

Hemminki et al. [43] utilised the accelerometer to detect the user's current transportation mode. They used a hidden Markov model and AdaBoost for classification. They were able to differentiate between resting, walking, transportation by bus, and railway transportation, *i.e.*, tram, train, and metro. Similarly, Sankaran et al. [72] used the barometer to deduce the transportation mode. They distinguished more coarsely between resting, walking, and general transportation by vehicle, which includes both motorised and unmotorised vehicles. Compared to accelerometer-based transportation type inference, the barometer also classified the resting state correctly if the user

was creating movements by operating the device in her hands. They classified the barometer readings through different observations, for example, the height change in a given time frame or by calculating the standard deviation of the height in a predefined period. The classification occurred in real time on the phone. Furthermore, Ho et al. [45] used the barometer to estimate the elevation and subsequently infer driving routes. They used dynamic time warping to deduce the most likely driving route from a fixed set of paths.

### 2.1.3. Automated Side-Channel Analysis

Most of the attacks in the literature rely on the manual selection and exploitation of possible side channels. Unlike testing frameworks, which do exist for most programming languages, frameworks for the automatic detection of side channels are much less ubiquitous. We are aware of one framework made specially for Android and two tools for the discovery of side channels in individual web applications. The framework for Android is called ProcHarvester [78]. It automates the detection of side channels in the procfs. For the analysis of web applications, we identified Sidebuster [83] for GWT (Google Web Toolkit) applications and a more generic tool by Chapman et al. [10], which, unlike Sidebuster, is not constrained to a single web application framework. As side channels in web applications are a threat to smartphone users as well, we will take a closer look at these two tools before we move on to ProcHarvester.

Compared to side channels specific to smartphones, the two web application frameworks consider a different adversary model. As described in Section 2.2.1, with smartphone-specific side channels we (usually) consider a seemingly benign application or game which the user installs. In the case of web applications, on the other hand, no installation of an application is required. Instead, the user only opens the web application in a browser. In this adversary model, the web application, furthermore, uses encryption to protect information transferred between the server and the client. Therefore, the sensitive information is not directly available to anyone observing the communication. The attacker resides on the wire and, hence, cannot directly execute code on the client. Such an attacker could, for example, be the Internet service provider (ISP) or other users on the local network. The

attacker has access to the encrypted web traffic and any metadata which is sent unencrypted, such as packet lengths and sequences. The attacker uses this information to infer the encrypted data. Furthermore, the tool by Chapman et al. [10] also considers an adversary who has only access to encrypted WLAN traffic, *i.e.*, no direct access to individual packet metadata. Thus, the adversary in this second threat model only sees the size of the transferred data and whether the traffic is outgoing or incoming.

### Automated Side-Channel Analysis in Web Applications

Sidebuster [83] is a tool to help detect side channels in web applications. It has been designed for applications based on Google Web Toolkit. The tool first uses a white-box approach to find interactions with the investigated website which are possibly leaking sensitive information. For this step, the developer of the web application has to tag variables containing sensitive information. Sidebuster then checks whether or not the sensitive information is in fact sent over the network by examining the data flow and control flow of the application. In a second step, the tool assesses how much sensitive information is leaking by running the possibly leaking interaction in a simulated browser based on HtmlUnit [3]. The result of this step should help the developer to evaluate whether or not an information leak is worth fixing. In their paper, the researchers used the tool to assess six different web applications. Three of them were replicas of websites with known side-channel vulnerabilities. Sidebuster was able to detect the known issues and also found side channels in the other three examined applications. However, the researchers note that most of the web applications built on GWT are closed source and can, therefore, not be analysed using Sidebuster unless the source code is made available. Furthermore, Sidebuster does not work with the various other web frameworks used in practice.

Chapman et al. [10] proposed another tool to detect side channels in web applications. Unlike Sidebuster, their framework is based on a black-box approach. Therefore, it does not need access to the source code and is applicable to any web application. The implementation is based on Crawljax [53], which is in turn based on Selenium [69], a browser automation application primarily used for testing. To detect possible side channels, the tool requires

a so-called crawl specification for each website. This specification contains, for example, the elements to click, input fields and corresponding values to fill, or login data for individual sites. The tool was tested on parts of six different real-world web applications, consisting of four search engines and two health-related websites. Five of them had known side-channel leaks which could be verified with the tool. In the sixth application, a symptom checker, they were able to infer the entered information with high accuracy.

#### Automated Side-Channel Analysis on Smartphones

In the mobile space, only recently Spreitzer et al. [78] introduced a tool called ProcHarvester. The tool automatically searches for possible side channels in the procfs of Android [74]. In a second step, it records traces of these side-channel candidates while automatically triggering events of interest. The recorded time series are later analysed for correlations. In their work, application starts, website launches, and keyboard gestures were profiled. While launching these events, ProcHarvester was able to identify a known side channel, but also found several new side channels in Android 7 and Android 8. The recognised side channels allowed the researchers to infer the profiled events with high accuracy. However, no such tool exists for the Android API. Therefore, SCAnDroid closes this gap by profiling the Android API for side channels. It builds on the concept of ProcHarvester. However, unlike ProcHarvester, which uses the File API to read from the procfs, SCAnDroid has to create complex objects for the invocation of methods and constructors.

## 2.2. Background

In this section, we present our adversary model, discuss how Android implements the isolation between different applications via the concept of sandboxing and permissions, show the machine learning technique used in this paper, and give an introduction to Java Reflection.

### 2.2.1. Adversary Model

In this work, we consider so-called template attacks, which are a form of profiling attacks. The attacks consist of two different steps, the *training* and the *attack* step. In the training step, the attacker creates a template of the different events of interest. The attacker does this step on a device of her choice. To obtain data for the attack phase, the attacker then releases an application to an application store like the Google Play or the Amazon App Store. The application may fulfil some useful purpose or is an addictive game to provide an incentive for the user to install the application. However, besides this legitimate purpose, the application also runs in the background recording data while the user may perform events of interest. To make it look even more benign, the application does not require any permissions which would need to be granted by the user to obtain the data. The gathered data is then analysed with the help of the template built in the previous step. To keep the battery consumption reasonable, this step may be done on a more capable remote machine. As every application on Android is allowed to access the Internet without user consent, this should not raise any further suspicion by the user. The application only gathers data and does not try to modify or influence the operation of the device. Hence, in our adversary model, the attacker is a passive attacker.

### 2.2.2. Android Permission Types

In recent years, smartphones have emerged as the computing device of choice for many people. Smartphones have become much more capable, and we carry them with us almost anywhere. Therefore, these devices accumulate a lot of sensitive information about the user. Smartphones allow the user to, for example, shoot and store personal photographs, create notes, hold conversations, and manage contacts. Furthermore, with the trend of BYOD, personal devices are increasingly used in the work environment to process sensitive corporate information. On the other hand, application markets like Google Play or the Amazon App Store allow users to install potentially untrusted applications and games.

To keep personal and corporate information safe from untrusted applications, Android uses the concept of sandboxing to separate applications from each other. Android uses security features from the Linux kernel to enforce the sandboxing concept. To achieve this, the operating system assigns a unique user ID (UID) to each application. Furthermore, since version 4.3, Android also uses security-enhanced Linux (SELinux) to refine the sandbox further [68]. SELinux is a Linux kernel module which provides support for mandatory access control. It allows specifying policies for different applications. Violations of these policies are enforced by the kernel to prevent malicious activity.

The security techniques described in the last paragraph allow Android to separate applications from each other. However, some applications need access to information and resources outside the sandbox, like access to the internal storage, contacts, or camera. Applications can ask for access to such resources by requesting the appropriate permissions. Before Android 6.0, these permissions were granted at installation time, *i.e.*, the user only had the choice of granting all requested permissions or not installing applications requiring too many permissions. However, with the introduction of runtime permissions in Android 6.0, the user can now deny or revoke individual permissions from applications [17].

The permissions are grouped into different protection levels. In general, we distinguish three different types of applications corresponding to the type of permissions they request. These are as follows:

- **Zero-Permission Applications:** Zero-permission applications do not request any permission from the user of the device. On Android, these applications can request so-called *normal permissions* [29]. These normal permission allow an application to access certain resources outside its sandbox which are considered as having little risk to the user's privacy. They are granted to the application at install time without requiring user interaction and cannot be revoked by the user. Examples in this category include the `SET_ALARM` or `INTERNET` permission.
- **Dangerous-Permission Applications:** Dangerous-permission applications are allowed to access certain private information from the device. Since Android 6.0 [17], these permissions need to be granted by the user either via a pop-up dialogue or from the settings application.

Examples of dangerous permissions are the `READ_EXTERNAL_STORAGE` permission to read the persistent storage of the device or the `READ_CONTACTS` permission to access the contacts of the user.

- **System-Level/Signature-Permission Applications:** In general, system-level and signature permissions are only available to system applications or applications signed with a special key respectively. However, some of these permissions can also be requested by third-party applications. These permissions are listed in [41], annotated with `android:protectionLevel="appop"`. Of special interest here is the `PACKAGE_USAGE_STATS` permission [26], which allows to query various usage statistics. Compared to dangerous permissions, system-level and signature permissions, if available to third-party applications, can only be granted from the settings application.

For this work, we primarily search for side channels in the Android API which can be exploited by zero-permission applications, as these applications should not be able to access or infer any personal information.

### 2.2.3. Java Reflection

To systematically evaluate the Android API, we use the Java Reflection API [61]. In general, reflection is the capability of a program to analyse and change itself at runtime [52]. Reflection allows us to programmatically examine the Android API by retrieving all API classes and invoking the methods of these classes. Thus, the methods do not need to be called manually by invoking each individual method in the source code of SCAnDroid. With thousands of methods in the Android API, a manual inspection would be nearly impossible. With Reflection, on the other hand, it is possible to dynamically retrieve a class based on a string containing the package and class name. The Reflection API then allows us to query all constructors and to create an object of the class. In case a constructor takes any parameters, the type of the parameters can be queried, and objects can be instantiated before invoking the constructor. Furthermore, the declared methods of the class can be retrieved and invoked via Reflection. Similar to constructors, the parameters of a specific method can also be queried and instantiated before the method is called.

An example of retrieving the class `android.net.TrafficStats` and invoking each of the methods inside this class can be found in Listing 2.1. As non-static methods need an object for invocation, we first create objects by calling all available constructors. In case a constructor or method requires parameter objects, these are also created accordingly. Moreover, Reflection allows calling methods and retrieving fields which are marked as private or protected and, thus, inaccessible from regular Java code outside the respective class or package. For more information on Java Reflection, we refer to the official Java API documentation [61] and specialist publications on the topic such as [38].

### 2.2.4. Dynamic Time Warping

In this work, we record return values of methods over time, while triggering events of interest on the smartphone. These recorded values form so-called traces or time series. Let us assume we recorded the data usage of a browser while opening the same website twice. When disregarding disk cache usage and any dynamic content that may change between site visits, the browser's data usage should be the same between the two recordings. However, due to network congestion, routing differences, or other influencing factors, the data usage recorded at a specific point in time may not be exactly the same. To account for these temporal variations, we use dynamic time warping (DTW) [57]. DTW warps the two time series in a non-linear fashion to find the warping path with minimal distance. Compared to a linear alignment, such as produced by applying the Euclidean or Manhattan distance, a non-linear alignment such as produced by dynamic time warping leads to much better accuracies for time series. Dynamic time warping has been used in automatic speech recognition [71] but has since been applied in other fields, such as bioinformatics [44] and gesture recognition [51], as well. In essence, DTW calculates the distance between each point of the two time series and then backtracks to find the ideal warping path. However, in practice, several constraints are applied to reduce the computational overhead and to avoid some problems. According to [57], these are as follows:

- **Boundary Condition:** The warping path starts at the beginning of the time series and stops at the end, *i.e.*, the algorithm considers both time

```
// Retrieve class and get declared constructors
Class clazz = Class.forName("android.net.TrafficStats");
Constructor[] cstors = clazz.getDeclaredConstructors();
// Create array for saving the constructed class objects
ArrayList<Object[]> classObjects = ...;
for(Constructor constructor : cstors) {
  // Get type of constructor parameters and create objects
  Class <?>[] paramTypes = constructor.getParameterTypes();
  ArrayList<Object[]> parameterObjects =
     getParamObjs(paramTypes);
  // Create class objects by invoking each constructor
  for(Object[] paramObject : parameterObjects) {
    classObjets.add(constructor.invoke(paramObject));
  }
}
// Get all declared methods
Method[] methods = clazz.getDeclaredMethods();
for(Method method : methods) {
  for(Object classObj : classObjects) {
    // Get parameter types and create objects
    ArrayList<Object[]> parameterObjects =
       getParamObjs(method.getParameterTypes());
    // Invoke each method to retrieve the return value
    for(Object[] paramObj : parameterObjects) {
      Object retValue = method.invoke(classObj, paramObj);
    }
  }
}
ArrayList<Object[]> getParamObjs(Class <?>[] paramTypes) {
  ArrayList<Object[]> parameterObjects = ...;
  for(Class parameterType : paramTypes) {
    // Parameter objects may need to be created recursively
    parameterObjects.add(...);
  }
  return parameterObjects;
}
```

Listing 2.1: Retrieving the return value of all methods of a specific class.

series as a whole and not just subsequences.
- **Monotonicity Condition:** The warping path does not go back in time, *i.e.*, features are not repeated.
- **Step Size Condition:** A basic step size condition is to move the warping path one point forward in each step, either in one of the two time series or in both. Therefore, the warping path does not jump in time, and no features are skipped. However, using this basic step size condition does not prevent that short parts of one time series are matched to very long parts of the other time series. This problem can be circumvented by constraining the slope of the warping path. These so-called slope constraints ensure that the warping path is neither too steep nor too flat.
- **Warping Window:** Considering the three constraints above, it is still possible that one point of a time series gets matched to many points in the other time series. To avoid this, a warping window is introduced. The warping window represents a global constraint which reduces the number of admissible warping paths.

In our experiments, we use a k-nearest neighbour classifier on the distance matrix computed with dynamic time warping. In particular, we utilise the python library cdtw [42] in combination with the machine learning framework scikit-learn [63]. As shown in [37], the combination of DTW and a kNN classifier produces a high accuracy. Figure 2.1 depicts the alignment produced by dynamic time warping when applying the algorithm to two of the time series SCAnDroid has gathered during our experiments.

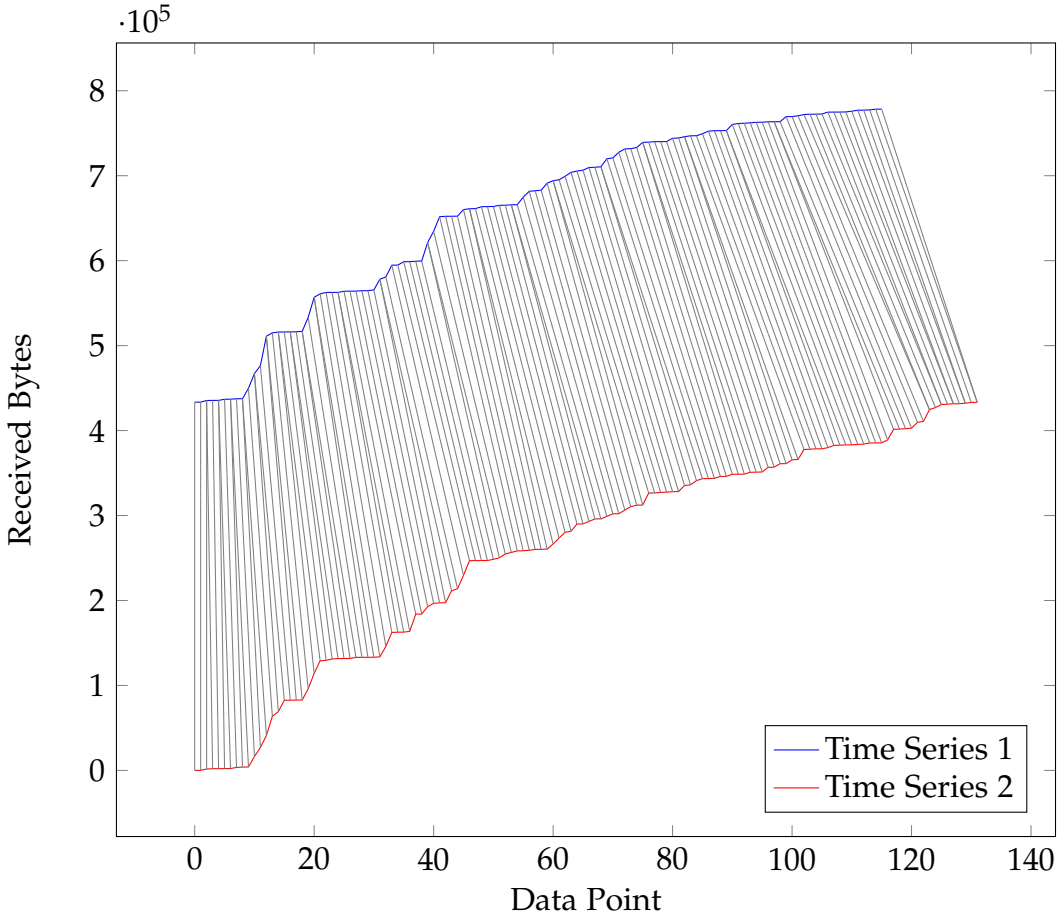


Figure 2.1.: Alignment of two time series produced by applying dynamic time warping. The time series show the received bytes while launching the application Spiegel Online.

# 3. SCAnDroid

SCAnDroid is a generic framework for side-channel detection and analysis of the Android API. In general, SCAnDroid automatically triggers events of interest (e.g., opening applications or websites) on a smartphone (or emulator) running Android. Concurrently, SCAnDroid invokes a specified set of methods in the Android API and monitors the return values for changes. In a second step, the gathered data will be analysed for correlations with the help of machine learning techniques.

A graphical overview of the structure of the SCAnDroid framework can be found in Figure 3.1. The framework consists of three main components. These components are called the Parser, the Backend, and the Service. The Parser and the Backend run on the PC, while the Service is executed on the smartphone. Before the profiling, the Parser fetches a list of methods, constructors, and the corresponding package structure from the Android Developers [20] website. During the actual profiling, this information is then used by the Service component to invoke methods of interest by means of Java Reflection. The profiling phase is controlled by the Controller component of the Backend. In this profiling phase, the Service first detects potentially leaking methods by monitoring the return values for changes, while the Backend continuously triggers events of interest. After side-channel candidates have been found, the Backend starts the actual profiling. It invokes predefined events of interest while the Service records the return values of the candidate methods. When the profiling is done, the Backend pulls the recorded files from the Android device for analysis by the Analysis component. The Backend and parts of the Service component are based on the ProcHarvester [78] implementation. The following sections will provide more details about the features and configuration options of the components.

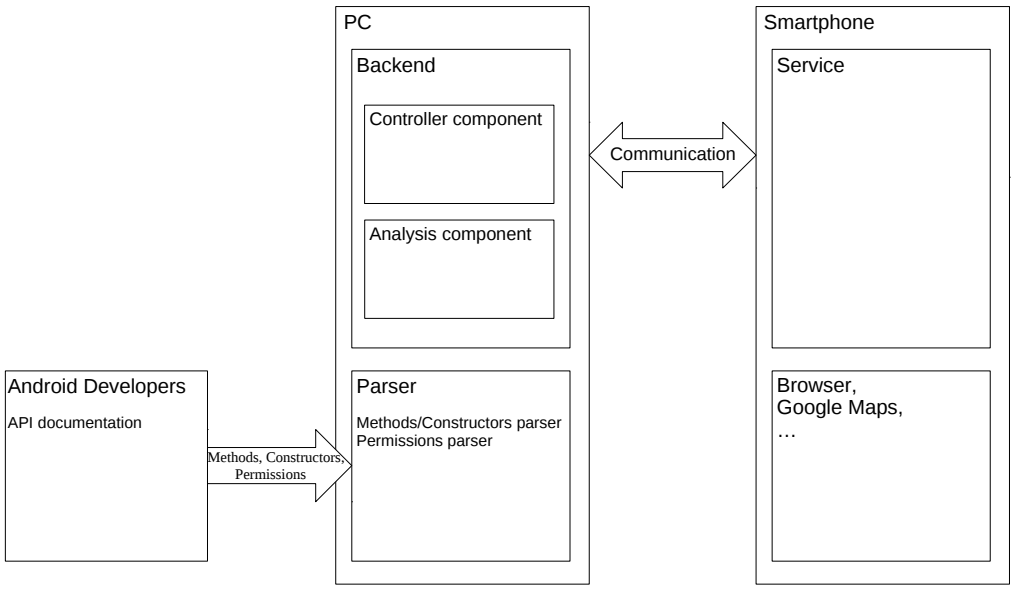


Figure 3.1.: Overview of the structure of SCAnDroid.

## 3.1. Parser

The Parser collects data from the Android Developers documentation. Some of the data collected by the Parser cannot be gathered during runtime of the Service using Java Reflection. In particular, these are the parameter names of methods and constructors. While parameter names are not strictly needed to call methods, the parameter names often convey the meaning of different parameters. Therefore, they allow us to define parameters of our choice with predefined values instead of randomly guessing them. Thus, the parameter names are needed by the Service to associate the predefined parameter values and statements with the parameters of methods and constructors. With Android 8 now supporting version 8 of the Java API [18], it is theoretically possible to query parameter names at runtime. The Java 8 standard specifies a method called `getName()` [28] in the `java.lang.reflect.Parameter` class. However, this method only provides the real parameter names as specified in the source code if the class of the method or constructor is compiled with

debug information. As the Android API on a regular phone is not compiled with debug information, the method `getName()` only provides generic parameter names in the form of `arg0`, `arg1`, `...` , `arg`$N$. These generic names make it harder to correctly associate the predefined parameters with the parameters of methods and constructors. Therefore, we parse the parameter names from the Android Developers documentation.

The Parser creates two files. One file contains information about methods and the other one about constructors. We call these the methods file and the constructors file respectively. An example of the methods file can be found in Figure 3.2. It consists of the method names and the name of their parameters grouped by package and class. The constructors file contains all parameter names of the constructors grouped by package and class. The constructors file has a similar structure as the methods file. The grouping by package and class is needed because the same method name can exist in different classes and packages. The files, however, do not contain the types of the parameter. These can be queried via the Reflection API. Thus, the Service checks if a predefined parameter type fits before calling a method or creating a class object. The parsed files can be edited to remove specific methods or to probe only a particular subset of the Android API. The Parser can also be configured to only parse methods and constructors of a specified API level or up to a specified API level. This option can be helpful to investigate possible side channels introduced in a specific version of Android.

The Parser also collects all Android permissions provided by the operating system itself. These permissions are directly written into valid Android App Manifest XML files [22]. Three different files are created. The first one contains only normal permissions which do not require any user interaction. The second one also encompasses dangerous permissions which need to be granted by the user via a popup. The third one contains all permissions, including system-level and signature permissions which have to be granted via the settings application. Each Manifest file represents a different build variant in Android Studio. These variants are installed as separate applications on the phone. Each application acts like either a zero-permission application, a dangerous-permission application, or a system-level permission application respectively. They are distinguished by the Backend via their package name. When starting the recording process in the Controller

**Structure**

```
<package.class>
 Methods: <number of methods in class>
  <method_name> | <param1>, <param2>,…
```

**Example**

```
android.content.Context
 Methods: 117
  bindService | service, conn, flags
  …
```

Figure 3.2.: Structure and example of the methods file.

component of the Backend, the variant is chosen by a parameter. While normal permissions are the most interesting as they can be accessed without user consent, SCAnDroid is also able to explore methods which require permissions. This feature can be useful for debugging or comparison purposes (e.g., between similar leaking methods where one method requires a permission and the other one does not).

## 3.2. Backend

The process of running SCAnDroid can be split up into two different phases — the profiling phase and the analysis phase. In the profiling phase, a smartphone (or emulator) running some version of Android and a PC are needed to execute actions on the phone. The communication between the Controller component of the Backend on the PC and the Service on the smartphone is done via the Android Debug Bridge (ADB) [19] by means of Intents. The analysis phase, on the other hand, is executed entirely on a PC. The Controller and Analysis components are both written in Python.

**Controller component:** The Controller component controls the Service on the phone during profiling. It starts the Service on the phone and waits until the Service has loaded completely. Afterwards, the Controller component randomly triggers a set of predefined events of interest while the Service records the return values of the potentially leaking methods. The Controller also sends the corresponding label of the event to the Service. After triggering an event, the Controller component waits for a specified amount of time until the next event is triggered. Currently, the Controller component can trigger website invocations, application launches, Google Maps search queries, and keyboard gestures. Furthermore, the framework can be easily extended with new events by adding them to the Controller component. Changes to the Service are usually not needed as the concept of recording return values stays the same. To automatically profile the events, it should be possible to trigger the events via ADB or some other automatic execution mechanism. The framework also supports triggering events by a human manually. This feature is useful to rule out interference caused by the automatic triggering of the events.

**Analysis component:** In the analysis phase, the data recorded in the profiling phase is checked for correlations utilising machine learning. The framework uses the cdtw library [42] which implements dynamic time warping, and the machine learning framework scikit-learn [63] for analysis. In our experiments, we record changing return values together with the respective time stamp and label them with the current event. These form the so-called traces or time series. The time series is first normalised by subtracting the mean of the recorded values from the time series. Afterwards, the time series gets interpolated. The interpolation ensures that the values are distributed evenly over the time series while assuring that each value occurs at least once in the series.

Let us denote the combination of all the preprocessed time series as $TS$. Let us further assume that we recorded 8 launches of 20 different events. Therefore, we have 160 time series in $TS$. We use k-fold cross-validation to split up the recorded time series $TS$ into training and test data. In our example, we use 8 folds to split up the data into 140 training traces and 20 test traces. The folds are split up in a stratified fashion, *i.e.*, the percentage of each event is preserved in the folds. For each test trace $t_{test}$, the distances between $t_{test}$ and each of the 140 training traces $t_{train}$ are

calculated. The Analysis component then takes the nearest neighbour, *i.e.*, the training trace $t_{train}$ with the lowest distance to $t_{test}$, as the inferred event. If the inferred event corresponds to the label of the test trace $t_{test}$, the event has been inferred correctly. The proportion of the number of correctly inferred events and the total number of events in this fold depicts the accuracy of the inference. This accuracy is averaged over all folds. In case the accuracy is higher than random guessing, we consider the method as leaking. The results of our experiments can be found in Section 4.2. We also tried to extract features with the help of tsfresh [13], used a convolutional neural network (CNN), and long short-term memory (LSTM) network based on Tensorflow [1]. However, in our experiments, the results were not as promising as the DTW approach.

## 3.3. Background Service

The Background Service running on the smartphone is implemented as an `IntentService`. A background service on Android does not need a user interface and is thus invisible to the user. In general, the Service is responsible for recording traces while the Backend triggers events of interest on the device. In our experiments, we consider methods with a special prefix as relevant. The prefixes have been chosen because they are usually associated with retrieving information. These prefixes are:

- get
- has
- is
- query

We apply several preprocessing steps as detailed in the following sections. Even though we restrict the list of relevant methods as discussed above, these preprocessing steps still require a lot of RAM. Therefore, we split up the list of methods and constructors into four parts for our experiments with a Nexus 5X having 2 GB of RAM. For each of these parts, we apply the following preprocessing procedure consisting of three phases.

### Loading Phase

Before searching for side-channel candidates, we invoke each method of interest once and remove methods which cannot be called, e.g., due to invalid parameters. For methods which can be invoked successfully, we save the created parameters, and in case of non-static methods, we also save the class objects. These are needed for invocation of the methods in the following phases.

**Class Object Creation:** To create objects for the invocation of non-static methods, the framework queries all available constructors of a class and invokes them. To invoke some constructors, we first need to create parameter objects. There are four cases to consider:

1. **No Parameters:** The constructor does not take any parameters. It can be invoked directly without creating any parameter objects.
2. **Primitive Parameters:** The constructor takes one or more primitive parameters. These can be either predefined or will be randomly guessed by SCAnDroid.
3. **Non-Primitive Parameters:** The constructor takes an object of a non-primitive class type. In this case, the framework will recursively create the needed objects until it reaches a constructor of type 1 or 2. In case there are multiple constructors for the non-primitive parameter objects, the framework will call all constructors and combine them in all possible combinations, *i.e.*, it will create all permutations of the created parameter objects.
4. **Primitive and Non-Primitive Parameters:** In case the constructor takes both primitive and non-primitive parameters, primitive parameters are handled as described in case 2 and non-primitive parameters are handled as in case 3.

**Method Invocation:** At the beginning of the method invocation phase, the framework queries all available methods. When at least one object of the class was created (or if the class has static methods), the framework creates parameter objects for each method that is also defined in the methods file to invoke the methods. The framework distinguishes the same four cases as when invoking constructors. For each of the created parameter objects and each of the previously created class objects, the framework invokes

the method. If the method can be successfully invoked, we distinguish two different cases based on the type of the return value:

1. **Primitive Return Value:** In case the method returns a primitive return value, e.g., a return value of type `int` or `boolean`, or if it returns an array or collection of type `java.util.Collection` containing primitive types, the method, parameter objects, and the corresponding class object will be saved. These will be used in the following phases to invoke the method again.
2. **Non-Primitive Return Value:** If the method returns an object of non-primitive type, the framework explores the object recursively, *i.e.*, the framework explores all methods of the returned class objects until it reaches a predefined recursion depth. For our experiments, we have set the recursion depth to 2.

By default, the framework only saves the objects on which the method has been invoked and does not recreate this object or its parent objects, *i.e.*, it flattens the recursion to reduce the processing time. However, in some cases, it may be required to recreate this object for each invocation. For example, consider a returned object containing the two fields `int x` and `int y` and the corresponding getter methods `getX()` and `getY()`. The fields are set at creation time via the constructor and, thus, invoking any of the two getter methods will not change the values of the fields. Therefore, `getX()` and `getY()` will always return the same value when invoked on the same object. However, recreating the object itself may lead to different values of `x` and `y`. Therefore, it is possible to configure classes for which the object should be recreated.

#### Exploration Phase

After the Service has loaded all invokable methods and the corresponding objects into memory, the Backend starts to trigger events of interest continuously. Meanwhile, the Service calls each of the saved methods and records the return value. Each method will be invoked a specified amount of times. In these subsequent invocations, the Service will only record the return value if it has changed compared to the previously saved value. At the end of the exploration phase, the Service discards all methods where

the return value has not changed at least once. This phase significantly reduces the number of methods to be profiled in the next phase, because the return value does not change for most methods. Thus, the potentially leaking methods can be invoked at a higher rate in the recording phase.

### Recording Phase

In the recording phase, the Backend triggers a predefined set of events a predefined number of times in randomised order. In the meantime, the Service keeps running in the background, invoking the potentially leaking methods and recording the return values. The return value of each method is written to a separate file for later evaluation. For each triggered event, the Backend sends a label to the Service which appends it to each trace. At the end of the recording phase, the Backend pulls the recorded files for evaluation by the Analysis component.

### Predefining Parameter Values

As randomly guessing parameter values does not always lead to successful invocations of methods, parameter values for constructors and methods can be preconfigured in the parameters file. The structure of this file and an example can be found in Figure 3.3. There are two types of values which can be set. The first type are constant values of primitive types like `boolean`, `int`, or `float`. Specifying character strings of type `java.lang.String` is also supported. The second type, on the other hand, are more elaborate statements. These statements are interpreted by the Service before calling constructors and methods in the loading phase. The Service uses BeanShell [60] as the interpreter for the statements. These can range from simple additions to sophisticated invocations of the Java and Android API. As BeanShell does not know about Android's concept of Application Contexts [23], the context of the Service is supplied to BeanShell as a predefined variable. The parameter values and statements need to be associated with methods or constructors. Three different options are available to do this. The most general option is to specify only a parameter name. In this case, the value or statement is used whenever a parameter with the chosen name

Structure

```
primitiveParameterValues:
    <param_name>: <value>
    <package>:
        <method_name or wildcard>:
            <param_name>: <value>
            <param_name>: {value: [<value1>, <value2>, …],
                           parameter_is_array: true|false}
parameterStatements:
    <param_name>: <statement>
    <package>:
        <method_name or wildcard>:
            <param_name>: <statement>
```

Example

```
primitiveParameterValues:
    pid: 1234
    android.net.TrafficStats:
        "*":
            uid: 10083
    android.content.Context:
        getSystemService:
            name: {value: ["storagestats", "netstats", "storage"],
                   parameter_is_array: false}
    …
parameterStatements:
    context: "context"
    android.app.usage.NetworkStatsManager:
        "*":
            startTime: "System.currentTimeMillis()"
    …
```

Figure 3.3.: Structure of the parameter file and an example configuration of different parameter values.

is encountered. A more detailed option is to specify a value or statement for all methods and constructors in a specific class. The most granular option is to constrain the value or statement to methods with a particular name in a specific class. Regardless of choice, the parameter value or the evaluation of the statement is checked for type compatibility before calling a method or a

constructor. It is also possible to set multiple different values or statements for a single parameter. The Service will then call the method multiple times with each of the predefined values. As multiple values are specified with array syntax, a flag has to be set to distinguish them from regular arrays.

# 4. Evaluation

In the following section, we evaluate the coverage of SCAnDroid. The subsequent section details the side channels we have identified by using SCAnDroid to profile the Android API in four different scenarios automatically.

## 4.1. Coverage Analysis

To get a perspective of how many methods were analysed in this work, we performed a coverage analysis of the Android API. The coverage analysis has been conducted on the stock Android 8.1 system image on the Nexus 5X with security patches from February 2018. The metrics obtained by the Parser from the Android Developers documentation are based on API level 27 (Android 8.1). An overview of the coverage analysis is depicted in Table 4.1. According to the Reflection API, 53 913 methods are available in the packages and classes parsed from the Android Developers documentation. However, in the documentation of the classes only 36 339 methods are listed. We attribute this discrepancy to the fact that the Reflection API also finds methods which are, for example, private or protected. These methods are normally only available in the corresponding class or package but not directly to developers.

From our evaluation of the `android.net.TrafficStats` class, we found that the undocumented methods in this class are mostly internal helpers for the documented methods. For example, the publicly accessible method `getMobileTxBytes()` uses the private method `getMobileIfaces()` to retrieve all mobile interfaces. Some other public methods may also be hidden from the documentation deliberately, like for instance the method

`getTxBytes(String iface)`. We have also tried to evaluate the private and protected methods in our study. However, a lot of these methods crash in the native code, due to missing sanity checks already performed in the public methods. As these crashes generally cannot be caught inside Java, they make an automatic assessment of the private and protected methods by SCAnDroid challenging. It is also harder to choose meaningful predefined values for the parameters of these methods as the name of the parameter is unknown (see Section 3.3). Thus, our focus in this work is on the 12 012 documented methods which have the prefix `get`, `has`, `is`, or `query`.

| | |
|---|---|
| Declared methods according to Java Reflection | 53 913 |
| Methods in the Android Developers Documentation | 36 339 |
| **Methods documented with relevant prefix (get, has, is, query)** | 12 012 |
| − Methods in abstract classes or interfaces | 2 860 |
| − Methods where class could not be found (e.g., `android.test`) | 208 |
| − Methods removed (e.g., due to segmentation faults in native code) | 511 |
| − Methods where invocation throws an error (e.g., due to wrong parameters) | 692 |
| − Non-static methods where object creation fails | 3 664 |
| = Methods to be profiled | 4 077 |
| **Methods actually profiled** | 5 056 |
| Methods which do not change during website inference | 5 020 |
| Methods which change during website inference | 36 |
| Methods which do not change during application inference | 5 019 |
| Methods which change during application inference | 37 |
| Methods which do not change during Google Maps search query inference | 5 020 |
| Methods which change during Google Maps search query inference | 36 |
| Methods which do not change during keyboard gesture inference | 5 034 |
| Methods which change during keyboard gesture inference | 22 |

Table 4.1.: Coverage of SCAnDroid running on a Nexus 5X stock image based on Android 8.1 (API level 27).

Out of the 12 012 methods of interest, 2 860 possibly relevant methods are in abstract classes or interfaces. Thus, they cannot be called directly. Another 208 methods cannot be invoked because the class cannot be found via Reflection. This set contains the deprecated `android.test` package and the `junit.runner` package. We have also removed a small set of classes due to problems encountered in the pre-profiling. These classes include 511 possibly relevant methods. The biggest part of them are located in the `android.graphics` package. Several methods and constructors inside this package caused segmentation faults in the underlying native code when invoked with incorrect parameters, which, like in the case of private and protected methods, cannot be caught inside Java code. Another 692 methods cannot be invoked due to, e.g., wrong parameters or missing permissions. For 3 664 non-static methods, we cannot directly create an object. Reasons for this include wrong parameters for the constructor and missing publicly accessible constructors in the class itself or in the parameters of the constructor. Factoring in all of the aforementioned limitations, we can theoretically invoke 4 077 methods. However, some of the missing objects can later be retrieved by recursively exploring non-primitive return values. Thus, we improve our coverage by around 1 000 methods to 5 056 invoked methods. Examples include the method `getSystemService(String)` [24] of the class `android.content.Context`, which returns various system classes like the `android.app.usage.StorageStatsManager`.

In our experiments, we profiled four different events of interest. During the website inference and Google Maps search query inference, the return values of 36 methods changed at least once, while 5 020 did not change. For the application inference, the return value of one additional method changed. We started the application via Android's UI/Application Exerciser Monkey [35]. Therefore, the method `android.app.ActivityManager.isUserAMonkey()`, which reports whether or not an input has been triggered via the UI/Application Exerciser Monkey, returned true while the application was launched. In total, 37 methods had changing return values during the application inference, while the return values of 5 019 methods did not change. In our last experiment, the keyboard gesture inference, the return values of 22 methods changed, while 5 034 did not change. In this case, fewer methods reacted because the return values of methods providing network statistics did not change during the execution of keyboard gestures.

## 4.2. Findings

In our evaluation, we used a Google Nexus 5X equipped with 2 GB of RAM. The device is running Android 8.1 (either the stock image or LineageOS). The following sections detail our results across the four chosen attack scenarios: website inference, application inference, Google Maps search query inference, and keyboard gesture inference. These attack scenarios are by no means exhaustive; however, they show the applicability of SCAnDroid by exposing information leaks in different parts of the Android API.

The experiments detailed in the following sections have been realised using the mobile network. For the sake of completeness, we have also executed the experiments with a WLAN connection. The results were similar and have, therefore, been omitted. Naturally, API methods reporting mobile data usage did not leak when using the WLAN connection.

### 4.2.1. Website Inference

In this attack scenario, an attacker wants to infer visited websites. The ability for any application to detect visited websites poses a privacy threat. Knowing visited websites could allow an application to, e.g., learn about preferences of the user like sexual orientation, political views, or medical conditions a user suffers. It could also allow an attacker to deduce when the user is visiting a shopping or banking website to launch a targeted phishing attack. Therefore, starting with Android 6.0, the ability for third-party applications to query the browsing history and bookmarks has been removed entirely [16]. In previous versions of Android, it was possible to retrieve this information with the `READ_HISTORY_BOOKMARKS` permission. However, side channels in the procfs, as uncovered by Spreitzer et al. [78] with the ProcHarvester tool, still make it possible to infer visited websites.

We show that inferring visited websites is also possible with information gathered via the Android API from methods identified by SCAnDroid. We have conducted the experiment on Android 8.1 running Chrome 64. The attack, however, is not limited to Android 8.1 or Chrome, respectively. We have also verified the applicability of the attack on Android 7.1 and other

browsers like Mozilla Firefox or Firefox Klar. The results for Android 7.1 are available in Appendix A. In our experiments, we have profiled 20 different websites. Each website has been opened eight times with a profiling time of ten seconds. The profiling time has been chosen so that each website has enough time to load completely under normal circumstances. The websites have been chosen according to the Alexa Top 500 ranking [48]. We have removed duplicates like `google.de` and `google.co.uk`. The complete list of the 20 profiled websites is depicted in Table 4.2.

| | |
|---|---|
| http://www.360.cn | http://www.netflix.com |
| http://www.amazon.com | http://www.qq.com |
| http://www.baidu.com | http://www.reddit.com |
| http://www.facebook.com | http://www.sina.com.cn |
| http://www.google.com | http://www.sohu.com |
| http://www.imgur.com | http://www.tmall.com |
| http://www.instagram.com | http://www.vk.com |
| http://www.jd.com | http://www.wikipedia.org |
| http://www.linkedin.com | http://www.yahoo.com |
| http://www.live.com | http://www.yandex.ru |

Table 4.2.: The URLs of the 20 profiled websites.

### Normal Permissions

In this section, we present the results of inferring website visits by considering API methods which can be accessed without a dangerous or system-level permission. An overview of the results is depicted in Table 4.3. Several of the methods reside in the class `android.net.TrafficStats`. This class provides network traffic statistics. These can be either global statistics, mobile data statistics, or statistics for a specific UID by means of the `getUid[R|T]xBytes(uid)` and `getUid[R|T]xPackets(uid)` methods. These methods return the received or transmitted data size in bytes or packets respectively. However, in Android 7.0 and above, the UID-specific methods return `UNSUPPORTED` for any UID that does not belong to the calling application, citing privacy reasons [34]. These methods have already been used

in the literature to conduct various attacks. Zhang et al. [84] have shown that it was possible to infer when a user has left home by exploiting such application specific data usage statistics of two different IoT surveillance solutions. Spreitzer et al. [77] have demonstrated that it was possible to infer visited websites with UID-specific data usage statistics as was provided by `TrafficStats` and via the procfs. Although these UID-specific information leaks have been removed, it is still possible to use the global and mobile data statistics to infer events like visited websites with high accuracy as shown in this work.

| **API methods requiring no/normal permission** | **Accuracy** |
|---|---|
| `android.net.TrafficStats.getTotalTxBytes()` | 90.6% |
| `android.net.TrafficStats.getMobileTxBytes()` | 90.0% |
| `android.net.TrafficStats.getTotalTxPackets()` | 87.5% |
| `android.net.TrafficStats.getMobileTxPackets()` | 86.9% |
| `android.net.TrafficStats.getTotalRxPackets()` | 86.2% |
| `android.net.TrafficStats.getMobileRxPackets()` | 83.8% |
| `android.net.TrafficStats.getTotalRxBytes()` | 83.1% |
| `android.net.TrafficStats.getMobileRxBytes()` | 80.0% |
| `android.app.usage.StorageStatsManager.getFreeBytes(java.util.UUID)` | 45.6% |
| `java.io.File.getUsableSpace()` | 43.8% |
| `java.io.File.getFreeSpace()` | 41.9% |
| `android.os.storage.StorageManager.getAllocatableBytes(java.util.UUID)` | 38.8% |
| `android.os.Process.getElapsedCpuTime()` | 16.9% |

Table 4.3.: Accuracies of the identified API methods when inferring website launches on Android 8.1. All of these methods can be invoked by zero-permission applications.

We have also found several storage-related methods to be leaking. These methods include `getFreeBytes(java.util.UUID)` of the class `android.app.usage.StorageStatsManager` and the method `getAllocatableBytes(java.util.UUID)` of the class `android.os.storage.StorageManager`. Both methods have been invoked with the parameter set to `UUID_DEFAULT`. This `UUID` represents the default

internal storage of the device. The objects of the class `StorageManager` and `StorageStatsManager` have been retrieved by invoking the method `Context.getSystemService(String)` with the parameters `"storage"` and `"storagestats"` respectively. These parameters have been added to the parameters file, as detailed in Section 3.3, during our experiments. The values have been chosen by evaluating the log messages and consulting the Android Developers documentation for more information about the constants. While the exact semantics of the return values do not matter to the automatic analysis process of SCAnDroid, we still give a short description of the methods for completeness and a better understanding of the identified information leaks.

The method `getFreeBytes(java.util.UUID)` reports the free space in bytes on the requested storage volume [32], while the method `getAllocatableBytes(java.util.UUID)` returns the number of bytes an application can allocate on the storage volume [31]. Both methods have been added in Android 8.0 and, therefore, constitute a new side channel added in the latest stable version of Android. While the inference accuracy is lower than for the methods in the class `TrafficStats`, it still significantly outperforms random guessing. For example, the method `getFreeBytes(java.util.UUID)` classifies more than 45% of the events correctly, while randomly guessing the right event would result in an inference accuracy of just 5%. The methods `getFreeSpace()` and `getUsableSpace()` of the class `java.io.File` are leaking in a similar fashion. The method `getUsableSpace()` returns the number of bytes available to non-root users, *i.e.*, the method takes write permissions and other restrictions into account when calculating the free space [25]. The method `getFreeSpace()`, on the other hand, returns the number of free bytes available to a root user, *i.e.*, any restrictions applying to the current user are ignored.

Sample traces of one storage-related method and one method returning network statistics are shown in Figure 4.1. These traces have been created while triggering the two websites `imgur.com` and `google.com`. We subtracted the value of the first data point of each trace from each data point of the trace for better comparability. As can be seen in the Figure, the website `google.com` does not change a lot between different visits and, therefore, produces a similar graph for both methods. There are only slight variations in the timing which can be successfully detected by our DTW approach. The

content of the website `imgur.com`, on the other hand, may change between different visits. This website shows user-generated content such as images and videos which are regularly updated. Still, our classifier was able to identify most of the `imgur.com` traces correctly.

### Top N Results

To show how the inference accuracies behave when taking more than one guess into account, we compare the top 10 guesses in Figure 4.2. For example, when two events are mistaken by our classifier due to their similarity, the second guess may increase stronger than random guessing. For reference, we also show the accuracy of guessing an event randomly. As the storage-related methods and the network statistics achieve a similar accuracy respectively, some of the methods have been omitted from the Figure for the sake of clarity. As shown in the comparison, the identified methods significantly outperform random guessing. Furthermore, when observing the top 5 guesses of the storage-related methods, the accuracy already reaches close to 80%.

### Dangerous and System-Level Permissions

For the sake of completeness, we have also applied our framework with dangerous and system-level permissions turned on. The results are shown in Table 4.4. We have identified the method `getDataActivity()` of the class `android.telephony.TelephonyManager` as leaking. The return value of the method indicates the type of activity of the cellular data connection [33]. The method distinguishes four different states. These states indicate either no data activity, in which direction the data is flowing, or if the connection is currently dormant. The method requires the `READ_PHONE_STATE` permission, which is a dangerous permission. To obtain an object of the class `TelephonyManager`, the framework has called the method `getSystemService(String)` of the class `Context` with the predefined parameter `"phone"`. Similar to previous constants, this value has been chosen during our experiments based on an evaluation of the log messages and by consulting the Android Developers documentation.

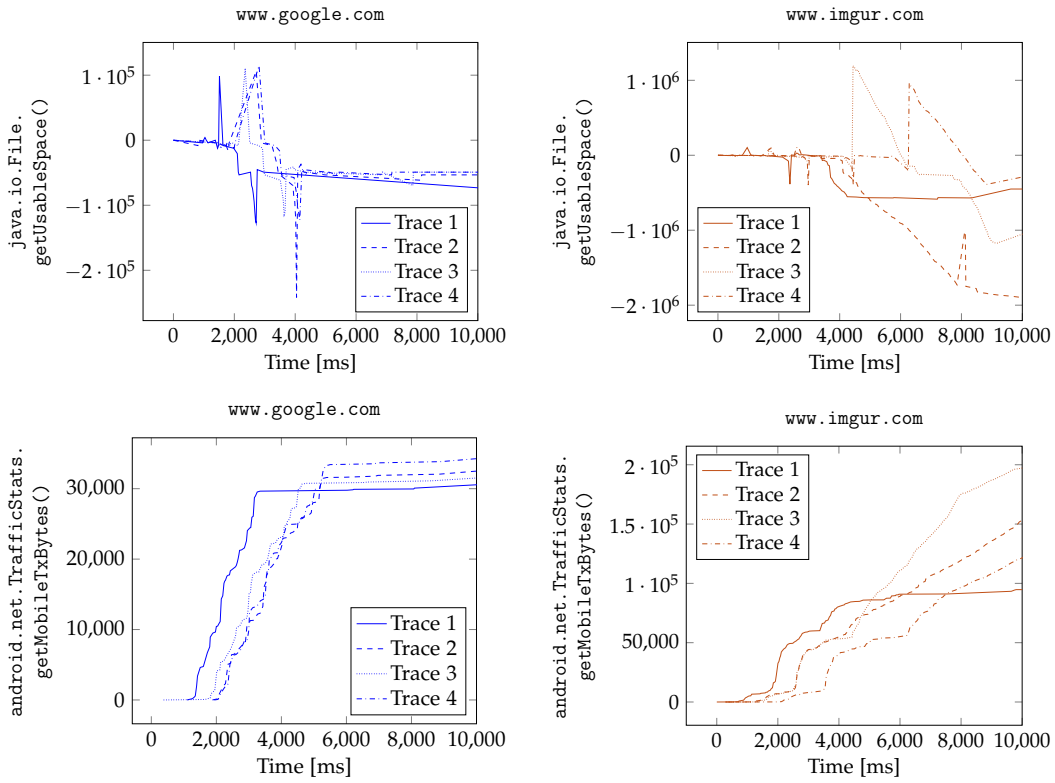


Figure 4.1.: Sample traces of `java.io.File.getUsableSpace()` and `android.net.TrafficStats.getMobileTxBytes()` when profiling the two websites `google.com` and `imgur.com` respectively.

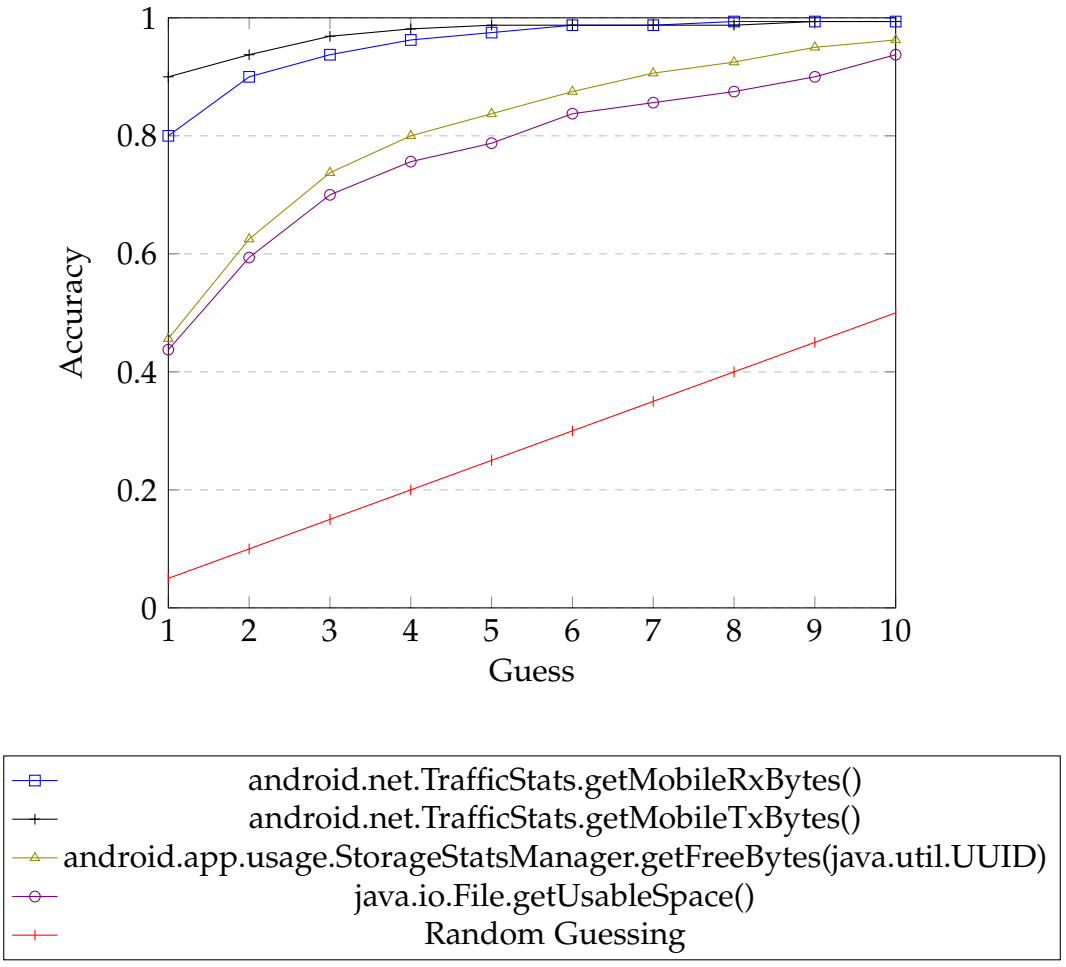


Figure 4.2.: Accuracies of the top 10 guesses when inferring website launches.

| **API method requiring dangerous permission (READ_PHONE_STATE)** | **Accuracy** |
|---|---|
| android.telephony.TelephonyManager.getDataActivity() | 20.6% |
| **API methods requiring system-level permission (PACKAGE_USAGE_STATS)** | **Accuracy** |
| `android.app.usage.NetworkStats.Bucket.getTxBytes()` | 90.6% |
| `android.app.usage.NetworkStats.Bucket.getTxPackets()` | 87.5% |
| `android.app.usage.NetworkStats.Bucket.getRxPackets()` | 85.0% |
| `android.app.usage.NetworkStats.Bucket.getRxBytes()` | 82.5% |

Table 4.4.: Accuracies of the identified API methods when inferring website launches on Android 8.1. These methods require dangerous or system-level permissions.

Furthermore, we have also identified the class `android.app.usage.NetworkStats.Bucket` to be leaking information similar to `TrafficStats`. However, compared to `TrafficStats`, retrieving the `NetworkStats.Bucket` object requires the system-level permission `PACKAGE_USAGE_STATS`. Similar to the `TelephonyManager` object, the framework first retrieved an `android.app.usage.NetworkStatsManager` object by calling the `getSystemService(String)` method with the parameter `"netstats"`. To obtain the actual `NetworkStats.Bucket` object, the framework then calls `querySummaryForDevice(int networkType, String subscriberId, long startTime, long endTime)`. Compared to the methods in the `TrafficStats` class, which return the global network statistics since device boot, this API allows specifying a time range. The start and end time and the other two parameters have been predefined. For example, the parameter `networkType` has been set to `ConnectivityManager.TYPE_MOBILE`, the `subscriberId` to `TelephonyManager.getSubscriberId()`, the `startTime` to `java.lang.System.currentTimeMillis()` and the `endTime` to `System.currentTimeMillis() + 10000`. The method does not throw an error or misbehaves if the `endTime` is in the future. Therefore, setting an offset such as adding 10 seconds to the current time is possible. Furthermore, the method `getSubscriberId()` requires the permission `READ_PHONE_STATE`. However, this parameter is only needed for mobile connections, and not for, e.g., WLAN. This example shows that SCAnDroid can invoke methods which require multiple permissions and sophisticated parameters.

We verified the results by manually launching the websites to rule out any inference caused by the automatic triggering of events. The manual investigation revealed the same side channels as the automatic examination. For the sake of completeness, the results of manually triggering website launches are available in Appendix B.

### 4.2.2. Application Inference

Inferring currently executed applications can be used by an adversary to launch targeted attacks like phishing attacks on banking applications [11]. Therefore, access to currently running applications is restricted on Android. Prior to Android 5.0, an application had to request the permission `GET_TASKS` to access this sensitive information. With the introduction of Android 5.0, the access to running applications has been restricted further, citing privacy improvements [14]. Effectively, the permission to access the whole list of running applications is now only granted to system applications via the `REAL_GET_TASKS` permission.

Irrespective of these restrictions, UID-specific network statistics, as were available in older versions of Android, allowed an attacker to infer running applications without a permission. Similar to application inference, Zhou et al. [87] have shown that these methods can be exploited to deduce specific activities inside applications like WebMD and Twitter. Furthermore, information provided by the procfs can be leveraged to infer running applications or application starts. Diao et al. [36] have shown that interrupt timings from the Display Sub-System released via the procfs can be used to derive currently running applications as each application has unique UI refreshing patterns. More side channels which allow unprivileged applications to infer application starts on Android 7 and 8 have been uncovered by Spreitzer et al. [78] with the ProcHarvester tool.

We show that the Android API also allows inferring application starts with high accuracy. A list of the 20 profiled applications is shown in Table 4.5. Each application was launched eight times with a profiling duration of eight seconds. The duration has again been chosen so that each application has enough time to start completely. The experiment detailed in the following

| com.airbnb.android |
|---|
| com.duckduckgo.mobile.android |
| com.fsck.k9 |
| com.instagram.android |
| com.isis_papyrus.raiffeisen_pay_eyewdg |
| com.moshbit.studo |
| com.paypal.android.p2pmobile |
| com.tripadvisor.tripadvisor |
| com.twitter.android |
| com.waze |
| com.whatsapp |
| de.mcdonalds.mcdonaldsinfoapp |
| de.pilot.newyorker.android |
| de.prosiebensat1digital.prosieben |
| de.spiegel.android.app.spon |
| de.zalando.mobile |
| org.chromium.chrome |
| org.indywidualni.fblite |
| org.mozilla.firefox |
| org.zwanoo.android.speedtest |

Table 4.5.: The package names of the 20 profiled applications.

sections was conducted on Android 8.1. Results for Android 7.1 are available in Appendix A.

### Normal Permissions

The results for the methods requiring normal permissions are shown in Table 4.6. The methods from the class `TrafficStats` again allow us to infer application starts with high accuracy. The storage-related methods allow us to deduce the launched applications with an accuracy of more than 60%. The `UUID` parameter for the methods `getAllocatableBytes(UUID)` and `getFreeBytes(UUID)` was again set to `UUID_DEFAULT`. The class objects used

| API methods requiring no/normal permission | Accuracy |
|---|---|
| android.net.TrafficStats.getTotalRxBytes() | 86.2% |
| android.net.TrafficStats.getMobileRxBytes() | 86.2% |
| android.net.TrafficStats.getTotalTxPackets() | 81.9% |
| android.net.TrafficStats.getTotalRxPackets() | 80.6% |
| android.net.TrafficStats.getMobileTxPackets() | 80.6% |
| android.net.TrafficStats.getMobileTxBytes() | 78.8% |
| android.net.TrafficStats.getMobileRxPackets() | 78.8% |
| android.net.TrafficStats.getTotalTxBytes() | 76.2% |
| java.io.File.getUsableSpace() | 61.9% |
| java.io.File.getFreeSpace() | 61.9% |
| android.os.storage.StorageManager. getAllocatableBytes(java.util.UUID) | 61.2% |
| android.app.usage.StorageStatsManager. getFreeBytes(java.util.UUID) | 61.2% |
| android.os.Process.getElapsedCpuTime() | 30.6% |

Table 4.6.: Accuracies of the identified API methods when inferring application starts on Android 8.1. All of these methods can be invoked by zero-permission applications.

for invocation of the methods have been retrieved as described in Section 4.2.1.

Figure 4.3 shows sample traces of the method `getTotalRxBytes()` and `getAllocatableBytes(java.util.UUID)` while launching the applications Waze (`com.waze`) and Spiegel Online (`de.spiegel.android.app.spon`). We subtracted the value of the first data point of each trace from each data point of the trace for better comparability. The traces for the same application look similar, while traces of different applications look different and can, therefore, be distinguished visually as well as by our classifier. However, there is one outlier in each of the depicted traces. For both methods and applications, this is the first trace recorded. We assume that this is the case because applications check for new data on startup and retrieve more updated information if the application has not been started for a longer time. Hence, the application retrieves much more data in the first trace of `getTotalRxBytes()` and reduces the allocatable size as reported by

`getAllocatableBytes(java.util.UUID)` more than subsequent launches by caching more data on the storage volume.

### Top N Results

An evaluation of the top 10 guesses compared to random guessing is illustrated in Figure 4.4. As the storage-related methods and the network statistics achieve a similar accuracy respectively, some of the methods have been omitted from the Figure for the sake of clarity. As in the case of website inference, the identified methods significantly outperform random guessing. Furthermore, when observing the top 3 guesses of the storage-related methods, the accuracy already reaches around 75 to 85%.

### Dangerous and System-Level Permissions

The accuracy of the methods requiring system-level permissions are shown in Table 4.7. These methods allow us to infer events with high accuracy, but they cannot be exploited by zero-permission applications. The `NetworkStats.Bucket` object was created as described in Section 4.2.1. As some application starts did not lead to a change in the data activity as reported by `android.telephony.TelephonyManager.getDataActivity()`, the method has been omitted from the investigation.

| **API methods requiring system-level permission (PACKAGE_USAGE_STATS)** | **Accuracy** |
|---|---|
| `android.app.usage.NetworkStats.Bucket.getRxBytes()` | 86.2% |
| `android.app.usage.NetworkStats.Bucket.getRxPackets()` | 80.6% |
| `android.app.usage.NetworkStats.Bucket.getTxPackets()` | 80.0% |
| `android.app.usage.NetworkStats.Bucket.getTxBytes()` | 77.5% |

Table 4.7.: Accuracies of the identified API methods when inferring application starts on Android 8.1. These methods require a system-level permission.

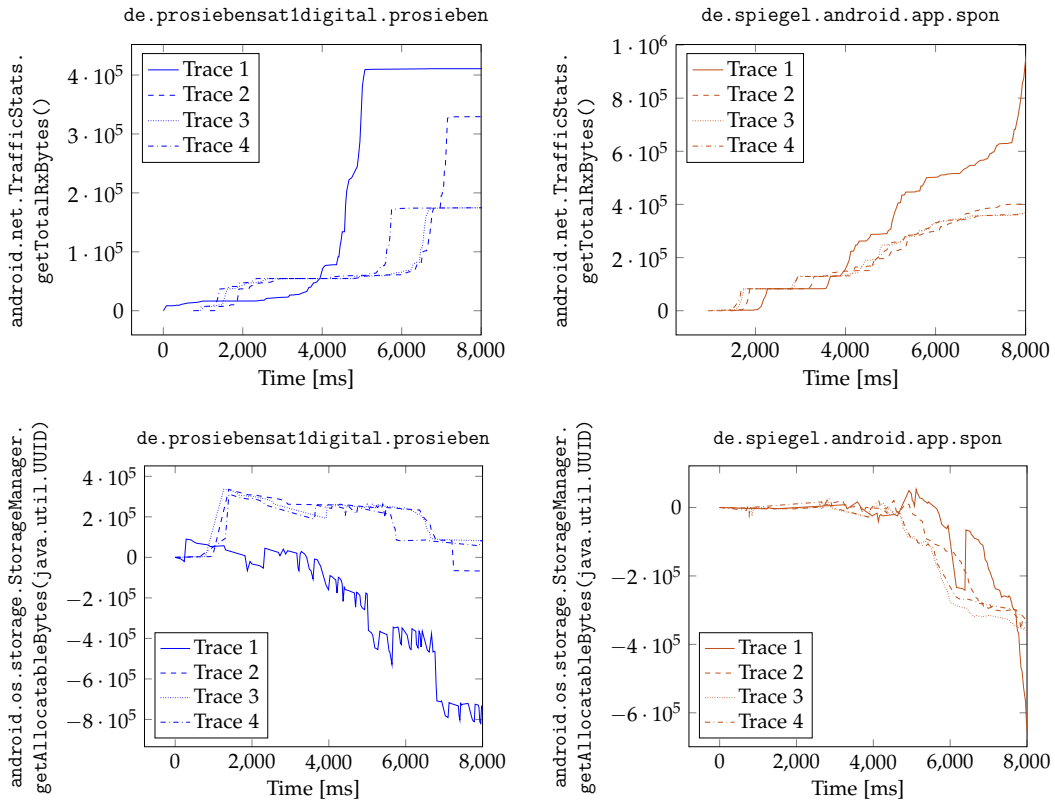


Figure 4.3.: Sample traces of `getAllocatableBytes(java.util.UUID)` and `getTotalRxBytes()` when profiling starts of the applications Pro 7 (`de.prosiebensat1digital.prosieben`) and Spiegel Online (`de.spiegel.android.app.spon`) respectively.

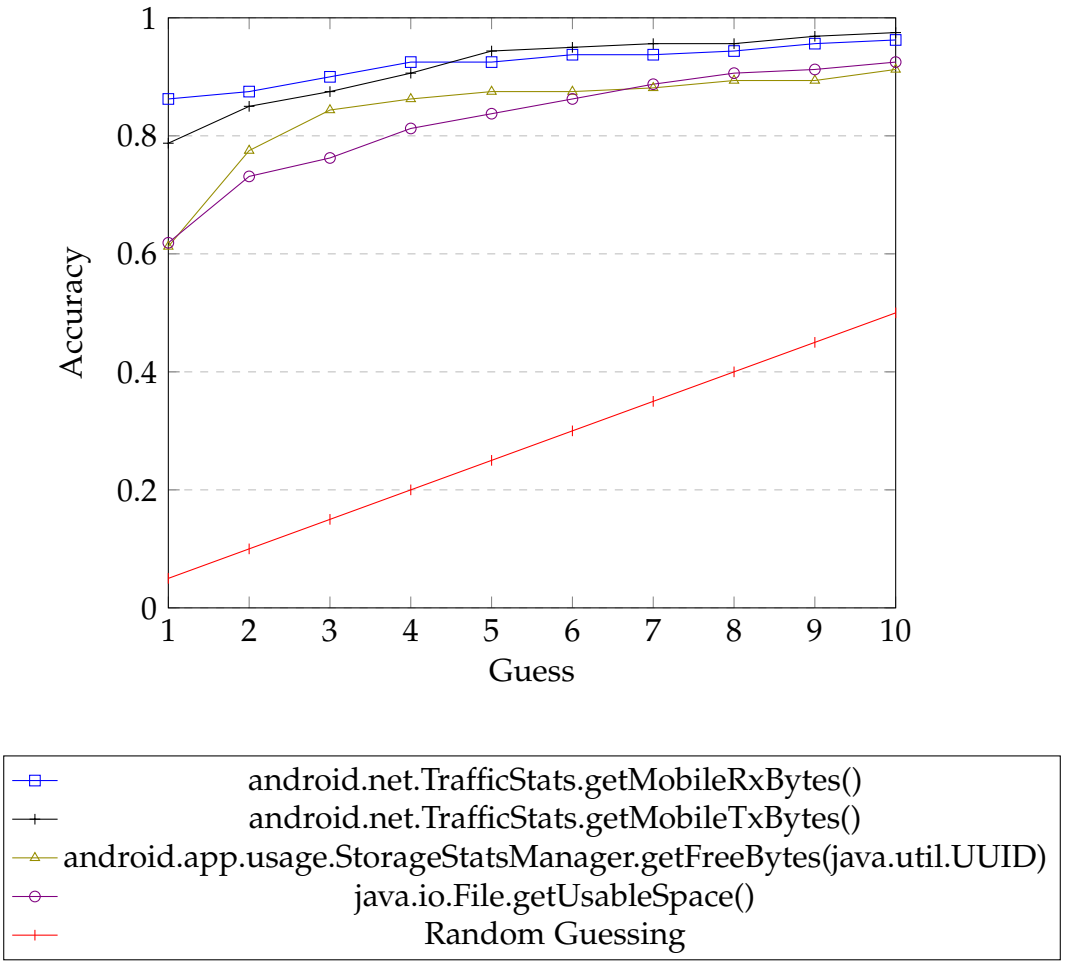


Figure 4.4.: Accuracies of the top 10 guesses when inferring application starts.

### 4.2.3. Google Maps Search Query Inference

In our third case study, we show that it is possible to infer search queries in Google Maps. The identified side channels allow an attacker to deduce places the user is planning to travel to or is otherwise interested in. Similar to Zhang et al. [86], who showed that it was possible to infer searches for POIs in Apple Maps on iOS, we show that the same attack is possible in the default navigation application on Google Android phones. For the evaluation, we picked 20 different points of interest (POIs) from around the world as our search queries. The complete list of search queries is shown in Table 4.8. We recorded eight samples per event with a profiling duration of 15 seconds. The duration has been chosen so that Google Maps has enough time to load the search result and the surrounding area. The experiments were again conducted on Android 8.1. Results for Android 7.1 are available in Appendix A.

| | |
|---|---|
| Acropolis of Athens | Petronas Towers |
| Big Ben | Pyeongchang |
| Burj Khalifa | Pyongyang |
| Cape Town | Pyramids of Giza |
| Colosseum Rome | Singapore |
| Eiffel Tower | Sydney Harbour |
| Empire State Building | Taipei 101 |
| Mirabell Gardens | The Great Wall of China |
| Mt. Everest | Toronto Canada |
| Peking | Wencelas Square Prague |

Table 4.8.: The names of the 20 profiled points of interest.

#### Normal Permissions

The inference accuracy of methods requiring only normal permissions is depicted in Table 4.9. Global network statistics published through the `TrafficStats` API again allow us to infer POI searches with high accuracy. However, there is a difference in accuracy between methods which report

| API methods requiring no/normal permission | Accuracy |
|---|---|
| `android.net.TrafficStats.getTotalRxBytes()` | 87.5% |
| `android.net.TrafficStats.getMobileRxBytes()` | 86.2% |
| `android.net.TrafficStats.getMobileRxPackets()` | 75.6% |
| `android.net.TrafficStats.getTotalRxPackets()` | 73.8% |
| `android.net.TrafficStats.getMobileTxPackets()` | 65.0% |
| `android.net.TrafficStats.getTotalTxPackets()` | 63.1% |
| `android.net.TrafficStats.getTotalTxBytes()` | 48.1% |
| `android.net.TrafficStats.getMobileTxBytes()` | 46.9% |
| `android.app.usage.StorageStatsManager.getFreeBytes(java.util.UUID)` | 18.1% |
| `android.os.storage.StorageManager.getAllocatableBytes(java.util.UUID)` | 17.5% |
| `java.io.File.getUsableSpace()` | 14.4% |
| `java.io.File.getFreeSpace()` | 12.5% |
| `android.os.Process.getElapsedCpuTime()` | 8.1% |

Table 4.9.: Accuracies of the identified API methods when inferring Google Maps search queries on Android 8.1. All of these methods can be invoked by zero-permission applications.

transmitted bytes or packets compared to the methods returning received bytes or packets. We attribute this to the fact that the transmitted search queries are only marginally different regarding the length of the query. The received data, on the other hand, can vary more significantly in terms of features, like the number and shape of nearby roads, buildings, and other POIs which are rendered by Google Maps when showing the search result. The storage-based methods achieve a lower accuracy compared to the network statistics. However, with 12.5%, even the lowest accuracy as achieved by the storage-based `java.io.File.getFreeSpace()` API still exceeds random guessing with an accuracy of 5%.

Figure 4.5 contains traces of the two methods `getMobileRxBytes()` and `getMobileTxBytes()` returning received and transmitted network data respectively. The traces were recorded while either triggering the search query `Pyongyang` or `Empire State Building`. We subtracted the value of the first

data point of each trace from each data point of the trace for better comparability. While the traces of the method `getMobileRxBytes()` share some similarities at first sight, the data usage when searching for `Empire State Building` is much higher than for `Pyongyang`. We attribute this to the fact that Google Maps renders much more points of interest and features in the vicinity of the Empire State Building compared to Pyongyang. The traces of the method `getMobileTxBytes()` look similar and, furthermore, also produce a similar data usage. Therefore, our classifier achieves a lower accuracy for the transmitted bytes than for the received bytes as argued in the previous paragraph.

#### Top N Results

An evaluation of the top 10 guesses compared to random guessing is illustrated in Figure 4.6. As the storage-related methods and the network statistics achieve a similar accuracy, some of the methods have been omitted from the Figure for the sake of clarity. As shown in the comparison, the identified methods again outperform random guessing. The accuracy of the methods returning transmitted bytes or packages improves by around 20% points when taking the second guess into account.

#### Dangerous and System-Level Permissions

The inference accuracy of the methods requiring dangerous or system-level permissions are shown in Table 4.10. Again, these methods cannot be exploited by zero-permission applications. The `TelephonyManager` object for the `getDataActivity()` method and the `NetworkStats.Bucket` object have been created as described in Section 4.2.1.

### 4.2.4. Keyboard Gesture Inference

In addition to the previous three events, we also profiled keyboard gestures using the default AOSP Android Keyboard. Inferring touch input, such as keyboard swipes, allows an attacker to reconstruct entered text on a gesture

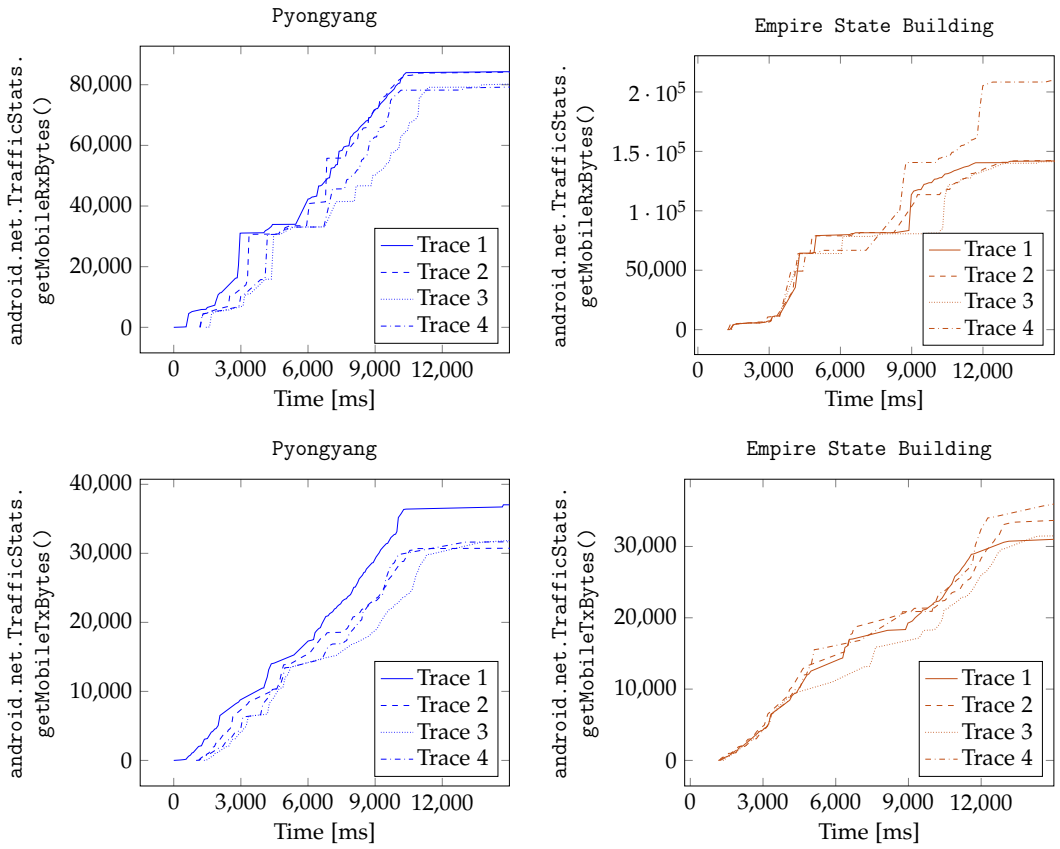


Figure 4.5.: Sample traces of `getMobileRxBytes()` and `getMobileTxBytes()` when profiling the search for the POIs `Pyongyang` and `Empire State Building` in Google Maps.

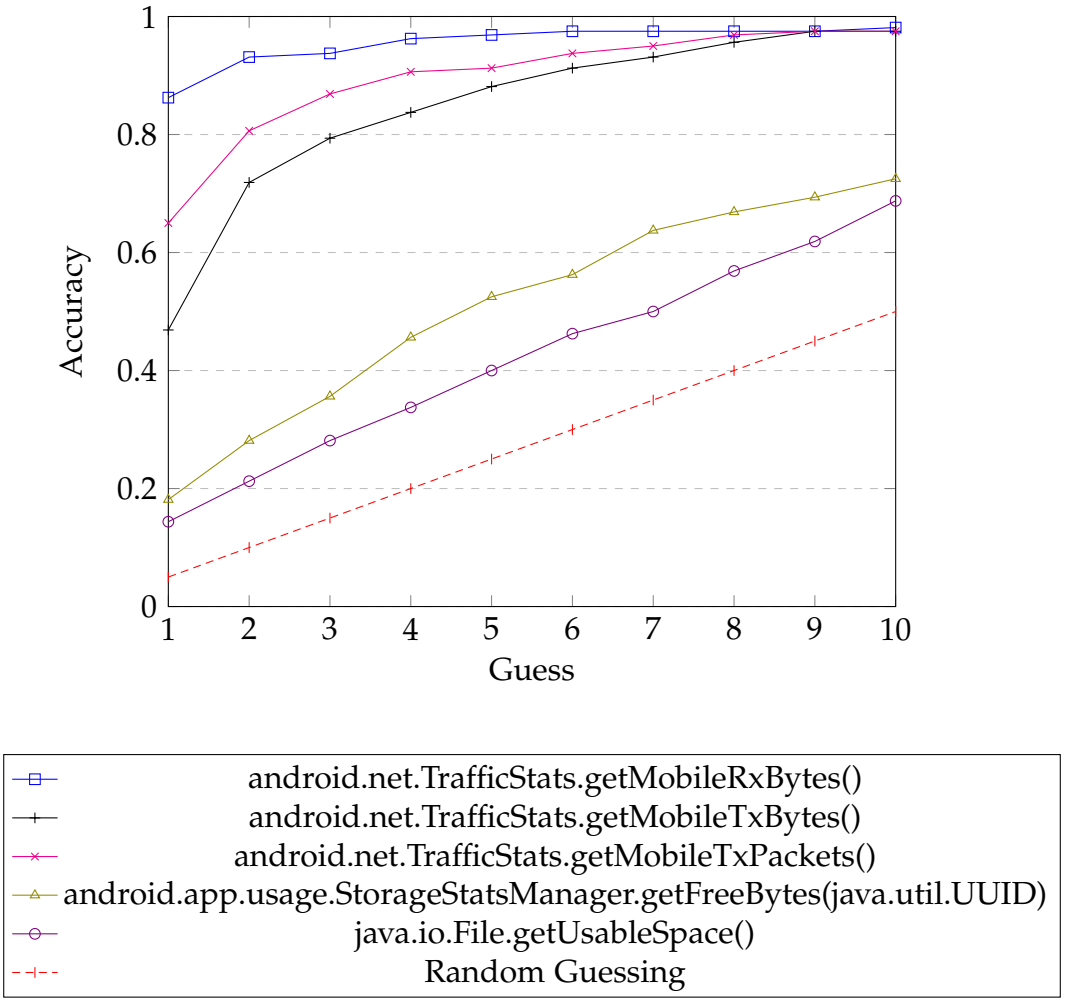


Figure 4.6.: Accuracies of the top 10 guesses when inferring Google Maps search queries.

| **API method requiring dangerous permission (READ_PHONE_STATE)** | **Accuracy** |
|---|---|
| android.telephony.TelephonyManager.getDataActivity() | 11.9% |
| **API methods requiring system-level permission (PACKAGE_USAGE_STATS)** | **Accuracy** |
| `android.app.usage.NetworkStats.Bucket.getRxBytes()` | 84.4% |
| `android.app.usage.NetworkStats.Bucket.getRxPackets()` | 74.4% |
| `android.app.usage.NetworkStats.Bucket.getTxPackets()` | 63.8% |
| `android.app.usage.NetworkStats.Bucket.getTxBytes()` | 41.2% |

Table 4.10.: Accuracies of the identified API methods when inferring Google Maps search queries on Android 8.1. These methods require dangerous or system-level permissions.

keyboard, as shown by Simon et al. [73]. By comparing the inferred text with sentences posted on an anonymous public message board, they were even able to identify the users. For our evaluation, we profiled five different keyboard gestures and gathered ten samples per event. The list of gestures is depicted in Table 4.11. Every event has been profiled for 1.8 seconds. Furthermore, the shift and regular buttons were pressed twice, as these events are very short compared to the other three. Therefore, we assume that a user would able to conduct two key presses in the time of 1.8 seconds. The experiments have been conducted on Android 8.1. Results for Android 7.1 are shown in Appendix A.

| Long-Press a Character Button |
|---|
| Tap a Character Button |
| Tap Shift Button |
| Short Swipe |
| Long Swipe |

Table 4.11.: A list of the five profiled keyboard gestures.

### Normal Permissions

The inference accuracy of methods which require normal permissions can be found in Table 4.12. Unlike the previous three events, touching and swiping on the keyboard does not produce any distinguishable network traffic or storage activity. Therefore, the only method which leaks information about the triggered gesture is `getElapsedCpuTime()` of the class `android.os.Process`. The method returns the time in milliseconds the current process has run [30]. As the SCAnDroid process records the profiled methods as fast as possible, we believe that other processes which need CPU time indirectly affect the elapsed CPU time of SCAnDroid.

| API method requiring no/normal permission | Accuracy |
|---|---|
| `android.os.Process.getElapsedCpuTime()` | 55.0% |

Table 4.12.: Accuracy of the identified API method when inferring keyboard gestures on Android 8.1. The method can be invoked by zero-permission applications.

### Top N Results

A comparison between random guessing and the top 5 guesses of the identified method is depicted in Figure 4.7. The `getElapsedCpuTime()` method constantly outperforms random guessing, especially when considering the first guess.

### Dangerous and System-Level Permissions

In contrast to the previous three events, SCAnDroid did not detect any leaking methods requiring dangerous or system-level permissions. All of the discovered methods in the previous experiments which require dangerous or system-level permissions leaked the event based on the network activity, which the AOSP Android Keyboard did not generate during the execution of the chosen gestures.

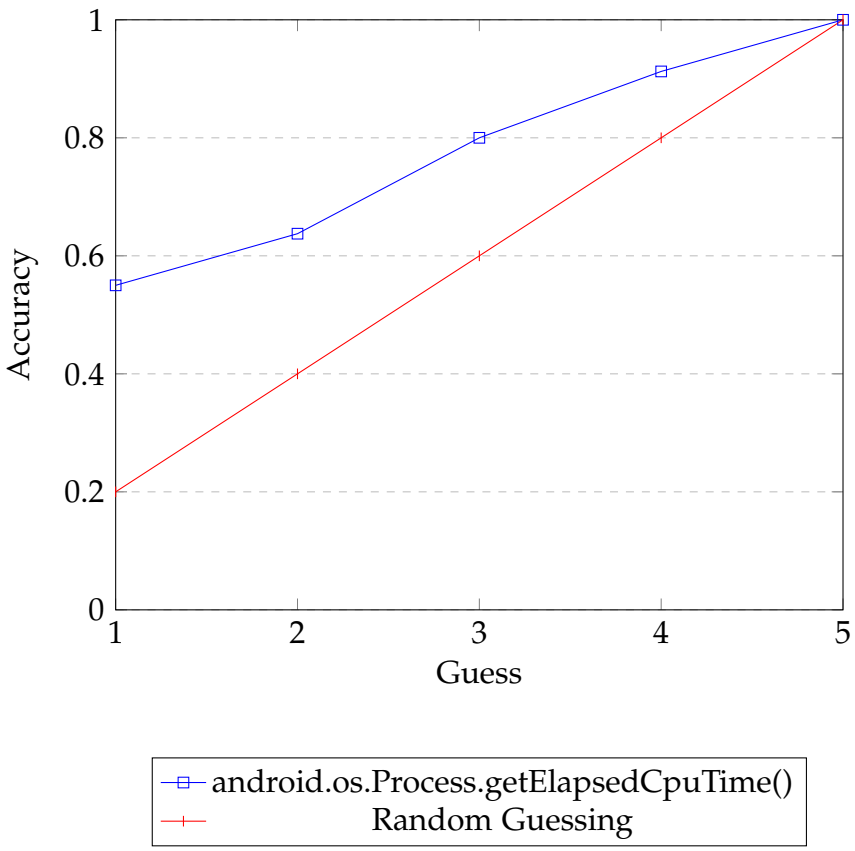


Figure 4.7.: Accuracies of the top 5 guesses when inferring keyboard gestures.

# 5. Discussion

In the following sections, we will discuss possible countermeasures, limitations of the framework, and future work.

## 5.1. Countermeasures

In this section, we show possible countermeasures like real-time attack detection applications and restrictions in the Android API. We also discuss the applicability of automatic side-channel detection tools like SCAnDroid in the process of evaluating such mitigations.

### 5.1.1. App Guardian

Zhang et al. [84] have proposed a tool called App Guardian to protect the user from side-channel attacks on Android. The application is designed to constantly run in the background and assess other background processes for malicious activity. The tool is based on the assumption that the attacking application needs to run in the background while the user is performing sensitive activities. To protect the user, App Guardian pauses background processes which it detects as possibly recording data during delicate tasks. It uses heuristics such as CPU usage, names of threads, and used permissions to detect possibly malicious applications, *i.e.*, the application itself relies on side-channel information to detect these malicious applications. Zhang et al. [84] have presented the effectiveness of their approach on a Nexus 5 running some version of Android 4 in videos on their website [85]. However, some of the information used by the tool has been restricted in more recent

versions of Android. For example, the application uses several process-related statistics in the `/proc/<pid>/` folder and the `getRunningTasks()` method of the `ActivityManager` API. Similar to `getRecentTasks()`, the method `getRunningTasks()` has been restricted in Android 5.0 and only returns the application's own tasks and possibly some non-sensitive tasks such as the home Activity [15]. Diao et al. [36] have shown that App Guardian did not detect and prevent their attack using information from `/proc/interrupts` on Android 5.1. Therefore, such a tool would need to be constantly updated to keep up with changes in the operating system and new attack methods.

### 5.1.2. Hardening the API

We believe that the best countermeasure is to restrict the leaking methods on the API level. Leaking methods can either be removed from the API or restricted by permissions, like in the case of `NetworkStats`. Furthermore, automatic side-channel detection tools like ProcHarvester [78] or SCAnDroid can be used by API developers to harden new methods before they are released and, therefore, mitigate information leaks before they can affect end users.

Another option to harden useful but leaking methods and resources is to decrease their accuracy or sampling rate. For example, the accuracy of battery statistics in the sysfs has been reduced in order to prevent attacks inferring the user's position based on battery usage, as detailed by Michalevsky et al. [55]. Automatic tools can help in the process of finding a tradeoff between accuracy and exploitability.

## 5.2. Limitations

SCAnDroid was able to find various side channels in the Android API. Still, profiling the whole Android API is a complex task. Therefore, some programming patterns are currently not supported by SCAnDroid. Supporting

these patterns could improve the coverage in future work. We will explore a few possibilities on how to improve the coverage.

The framework currently does not evaluate changes in the parameters used for method and constructor invocation. This pattern is, for example, used by the method `boolean getNextBucket(NetworkStats.Bucket)` [27] of the class `android.app.usage.NetworkStats`. Instead of returning a new `NetworkStats.Bucket` object each time the method is called, the method takes an existing `NetworkStats.Bucket` object and fills it with new values. It returns `true` if the `NetworkStats.Bucket` has been filled successfully. Supporting this pattern by exploring the method parameters could uncover more API methods leaking information.

In addition, SCAnDroid only calls methods with the specified prefixes `get`, `has`, `is`, and `query`. During the profiling phase, it polls these methods as fast as possible to record changes in the return values. However, for some frequently changing resources such as sensor readings, Android uses callback interfaces. An application using such a sensor registers an event listener, which gets called if the value of the sensor changes. These callback interfaces are not supported by SCAnDroid in its current version.

Furthermore, when invoked with the wrong parameters or on objects with a wrong internal state, some methods or constructors cause segmentation faults induced by null pointer dereferences or dereferences of invalid parameters. These segmentation faults cannot be caught as easily as Java exceptions and lead to the termination of SCAnDroid. Therefore, crashing methods had to be removed from the target set and were not profiled in this work. To profile these methods, a recovery strategy from segmentation faults would be needed.

Another limitation faced during our experiments is the amount of available RAM. To alleviate this issue, we had to split up the profiling into four parts. Furthermore, we reduced the recursion depth for the exploration of returned objects to 2. Higher RAM capacities or an even more fine-grained segmentation of the API may allow setting a higher exploration depth. To keep the memory usage manageable, SCAnDroid furthermore only invokes methods defined in the class itself, *i.e.*, it does not call any methods defined in superclasses.

Similar to other automatic side-channel detection tools such as ProcHarvester [78], SCAnDroid also suffers from false negatives, *i.e.*, even if SCAnDroid does not detect a method as leaking, it cannot prove that the method is in fact not leaking any information. For example, a method may only leak if the parameters have been chosen in a specific range, or if the object has been constructed with the appropriate parameters. Furthermore, a user may behave differently than what can be simulated by automatically triggering events of interest via ADB. For example, a user holding the phone while handling it unwittingly creates movements of the device which can be detected by the sensors available in modern smartphones. These sensor readings have been successfully used to infer, e.g., PIN codes as outlined in Section 2.1.2. In the automatic approach SCAnDroid is using, the phone lies on a table and, thus, no movement of the device occurs which may have leaked the executed event.

## 5.3. Future Work

Besides improving the coverage by alleviating the limitations outlined in the previous section, there are a few other approaches future work could focus on. These include a change in the parsing strategy, the automatic detection of timing side channels in the Android API, and the automatic exploration of native libraries for side channels.

### 5.3.1. Parsing Strategy

In this work, we have chosen to parse parameter names of methods and constructors from the Android Developers documentation. This parsing strategy has the advantage that the parameter names of new methods can already be parsed while the API is still in the developer preview stage. However, the documentation does not contain all methods available in the API, like private and protected methods. As Android is open source, another approach to identify parameter names could be to parse the source code directly. Pursuing this approach would allow SCAnDroid to associate the manually predefined parameters with the parameters of these

undocumented methods. However, the source code may not be available during development of new versions of Android. According to the FAQ of the Android Open Source Project [67], the source code of new APIs may only be released when the next platform version is ready. Furthermore, our experiments with private and protected methods have shown that some of these methods cause segmentation faults. Due to their protection status, they are not meant to be invoked with random parameters. These methods are only expected to be called by methods in the same class or package. They may omit sanity checks which are present in the public methods. Therefore, a recovery strategy from these faults would be required to enable the automatic profiling of such undocumented methods.

### 5.3.2. Timing Side Channels

As Zhang et al. [86] have shown in their experiments on Apple iOS, API methods may also leak internal states through timing differences. They have demonstrated that the `fileExistsAtPath` method can be used to infer installed applications. If the current process has access to this file, this method returns whether or not a file specified by a path exists on the device. In case the process does not have access, the method always returns false to avoid leaking sensitive information. However, by comparing the execution time of the method, Zhang et al. were able to deduce whether an application was installed or not. More precisely, they observed the difference between the execution time of the method when choosing a path that only exists if a particular application is installed compared to a random path string. Similar methods exist on Android, such as `java.io.File.exists()` or `java.nio.file.Files.isReadable(Path)`. Therefore, future work should focus on assessing these and similar methods in the Android API for possible leakage. As a manual investigation faces the same drawbacks as manually evaluating return values, automatic tools should be developed to investigate these timing side channels. For example, SCAnDroid could be extended to profile such timing differences to detect timing side channels on Android automatically.

### 5.3.3. Native Libraries

SCAnDroid profiles the Android API, while ProcHarvester [78] is designed to investigate the procfs. However, to our knowledge, no automatic analysis tools exist for the native libraries [21] of Android. An automatic tool for these native libraries may help to reveal side channels and assist in the hardening of these methods. Other operating systems, such as Apple iOS, may also benefit from automatic side channel detection and analysis tools to harden their API.

# 6. Conclusion

In this work, we have introduced SCAnDroid, a framework for the automatic investigation of side-channel-based information leaks in the Android API. We were able to overcome several challenges to enable the automatic exploration of the Android API. SCAnDroid drastically reduces the amount of manual effort required to find and evaluate side channels on Android. Furthermore, SCAnDroid has the potential to assist developers of the Android operating system to detect and mitigate possible side channels in the Android API before new versions are released to the public.

We have successfully demonstrated the applicability of SCAnDroid by identifying information leaks in different parts of the Android API. We were able to infer four different event types by exploiting the discovered methods. Our experiments have been conducted on both Android 7 and Android 8. As many applications communicate with external servers, the global network traffic statistics provided by the `TrafficStats` API allowed us to deduce events with high accuracy. Thus, this class breaks the claim of application isolation and allows an attacker to deduce highly sensitive information, such as the opened websites, which can leak, for example, personal preferences, political views, or medical conditions. Therefore, publishing such usage data to unprivileged applications should be rethought, especially as similar protected methods already exist for applications that need access to network statistics. Furthermore, filesystem usage statistics also allow inferring events with a non-negligible accuracy. Therefore, the precision of this information and the release to unprivileged applications, in general, should also be reevaluated.

While mitigating side channels, in general, remains a hard to reach goal, we believe that automatic frameworks such as SCAnDroid can help discover existing side channels and prevent them in new Android versions before these are released to developers and end users.

# Appendix

# Appendix A.

# Results on Android 7.1

Table A.1, A.2, A.3, and A.4 contain the results for inferring website launches, application starts, Google Maps search queries, and keyboard gestures respectively. The analysis has been executed on Android 7.1.

| **API methods requiring no/normal permission** | **Accuracy** |
|---|---|
| `android.net.TrafficStats.getTotalTxBytes()` | 87.5% |
| `android.net.TrafficStats.getTotalRxBytes()` | 87.5% |
| `android.net.TrafficStats.getMobileTxBytes()` | 87.5% |
| `android.net.TrafficStats.getMobileRxBytes()` | 86.9% |
| `android.net.TrafficStats.getTotalTxPackets()` | 83.8% |
| `android.net.TrafficStats.getTotalRxPackets()` | 83.8% |
| `android.net.TrafficStats.getMobileRxPackets()` | 83.1% |
| `android.net.TrafficStats.getMobileTxPackets()` | 82.5% |
| `java.io.File.getUsableSpace()` | 27.5% |
| `java.io.File.getFreeSpace()` | 26.2% |
| `android.os.Process.getElapsedCpuTime()` | 13.8% |
| **API method requiring dangerous permission** | **Accuracy** |
| `android.telephony.TelephonyManager.getDataActivity()` | 23.8% |
| **API methods requiring system-level permission** | **Accuracy** |
| `android.app.usage.NetworkStats.Bucket.getTxBytes()` | 86.2% |
| `android.app.usage.NetworkStats.Bucket.getRxBytes()` | 85.6% |
| `android.app.usage.NetworkStats.Bucket.getTxPackets()` | 81.2% |
| `android.app.usage.NetworkStats.Bucket.getRxPackets()` | 81.2% |

Table A.1.: Accuracies of the identified API methods when inferring website launches on Android 7.1.

| API methods requiring no/normal permission | Accuracy |
|---|---|
| `android.net.TrafficStats.getTotalRxBytes()` | 85.0% |
| `android.net.TrafficStats.getMobileRxBytes()` | 83.1% |
| `android.net.TrafficStats.getTotalTxBytes()` | 80.0% |
| `android.net.TrafficStats.getMobileTxBytes()` | 80.0% |
| `android.net.TrafficStats.getMobileTxPackets()` | 78.1% |
| `android.net.TrafficStats.getTotalTxPackets()` | 77.5% |
| `android.net.TrafficStats.getTotalRxPackets()` | 75.6% |
| `android.net.TrafficStats.getMobileRxPackets()` | 75.0% |
| `java.io.File.getUsableSpace()` | 53.8% |
| `java.io.File.getFreeSpace()` | 50.6% |
| `android.os.Process.getElapsedCpuTime()` | 14.4% |
| **API methods requiring system-level permission** | **Accuracy** |
| `android.app.usage.NetworkStats.Bucket.getRxBytes()` | 80.0% |
| `android.app.usage.NetworkStats.Bucket.getTxBytes()` | 77.5% |
| `android.app.usage.NetworkStats.Bucket.getTxPackets()` | 75.6% |
| `android.app.usage.NetworkStats.Bucket.getRxPackets()` | 73.8% |

Table A.2.: Accuracies of the identified API methods when inferring application starts on Android 7.1.

| API methods requiring no/normal permission | Accuracy |
|---|---|
| `android.net.TrafficStats.getMobileRxBytes()` | 83.1% |
| `android.net.TrafficStats.getTotalRxBytes()` | 80.0% |
| `android.net.TrafficStats.getTotalRxPackets()` | 77.5% |
| `android.net.TrafficStats.getMobileRxPackets()` | 75.6% |
| `android.net.TrafficStats.getTotalTxPackets()` | 66.2% |
| `android.net.TrafficStats.getMobileTxPackets()` | 63.8% |
| `android.net.TrafficStats.getTotalTxBytes()` | 48.8% |
| `android.net.TrafficStats.getMobileTxBytes()` | 45.0% |
| `java.io.File.getFreeSpace()` | 13.8% |
| `java.io.File.getUsableSpace()` | 9.4% |
| `android.os.Process.getElapsedCpuTime()` | 9.4% |
| **API methods requiring system-level permission** | **Accuracy** |
| `android.app.usage.NetworkStats.Bucket.getRxBytes()` | 88.1% |
| `android.app.usage.NetworkStats.Bucket.getRxPackets()` | 75.0% |
| `android.app.usage.NetworkStats.Bucket.getTxPackets()` | 63.1% |
| `android.app.usage.NetworkStats.Bucket.getTxBytes()` | 52.5% |

Table A.3.: Accuracies of the identified API methods when inferring Google Maps search queries on Android 7.1.

| API method requiring no/normal permission | Accuracy |
|---|---|
| `android.os.Process.getElapsedCpuTime()` | 47.5% |

Table A.4.: Accuracy of the identified API method when inferring keyboard gestures on Android 7.1.

# Appendix B.

# Results for Manually Triggered Website Launches

Table B.1 contains the inference accuracies for inferring manually launched websites. We profiled the same 20 websites as detailed in Section 4.2.1 with each of them launched eight times. This analysis verifies that the automatically identified methods also leak while manually triggering the events.

| **API methods requiring no/normal permission** | **Accuracy** |
|---|---|
| `android.net.TrafficStats.getTotalTxPackets()` | 51.9% |
| `android.net.TrafficStats.getTotalTxBytes()` | 46.9% |
| `android.net.TrafficStats.getTotalRxBytes()` | 46.9% |
| `android.net.TrafficStats.getMobileTxBytes()` | 45.6% |
| `android.net.TrafficStats.getMobileRxPackets()` | 45.0% |
| `android.net.TrafficStats.getMobileRxBytes()` | 45.0% |
| `android.net.TrafficStats.getMobileTxPackets()` | 43.1% |
| `android.net.TrafficStats.getTotalRxPackets()` | 41.2% |
| `java.io.File.getUsableSpace()` | 18.8% |
| `java.io.File.getFreeSpace()` | 16.2% |
| `android.app.usage.StorageStatsManager.getFreeBytes(java.util.UUID)` | 16.2% |
| `android.os.storage.StorageManager.getAllocatableBytes(java.util.UUID)` | 13.8% |
| `android.os.Process.getElapsedCpuTime()` | 13.8% |
| **API method requiring dangerous permission** | **Accuracy** |
| `android.telephony.TelephonyManager.getDataActivity()` | 7.5% |
| **API methods requiring system-level permission** | **Accuracy** |
| `android.app.usage.NetworkStats.Bucket.getTxPackets()` | 46.9% |
| `android.app.usage.NetworkStats.Bucket.getRxPackets()` | 46.9% |
| `android.app.usage.NetworkStats.Bucket.getTxBytes()` | 46.2% |
| `android.app.usage.NetworkStats.Bucket.getRxBytes()` | 41.2% |

Table B.1.: Accuracies of the identified API methods when inferring manually launched websites on Android 8.1.

# Bibliography


[1] Martín Abadi et al. 'TensorFlow: A System for Large-Scale Machine Learning'. In: *Operating Systems Design and Implementation – OSDI 2016*. USENIX Association, 2016, pp. 265–283 (cit. on p. 27).

[2] Dakshi Agrawal et al. 'The EM Side-Channel(s)'. In: *Cryptographic Hardware and Embedded Systems – CHES 2002*. Vol. 2523. LNCS. Springer, 2002, pp. 29–45. ISBN: 3-540-00409-2 (cit. on p. 5).

[3] Mike Bowler et al. *HtmlUnit*. URL: `http://htmlunit.sourceforge.net/` (visited on 21/03/2018) (cit. on p. 13).

[4] Adam J Aviv. 'Side channels enabled by smartphone interaction'. PhD thesis. 2012 (cit. on pp. 1, 10).

[5] Adam J. Aviv et al. 'Practicality of accelerometer side channels on smartphones'. In: *Annual Computer Security Applications Conference – ACSAC 2012*. ACM, 2012, pp. 41–50. ISBN: 978-1-4503-1312-4 (cit. on pp. 1, 10).

[6] Alessandro Barenghi et al. 'Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures'. In: *Proceedings of the IEEE* 100 (2012), pp. 3056–3076 (cit. on p. 5).

[7] Liang Cai and Hao Chen. 'On the Practicality of Motion Based Keystroke Inference Attack'. In: *Trust and Trustworthy Computing – TRUST 2012*. Vol. 7344. LNCS. Springer, 2012, pp. 273–290. ISBN: 978-3-642-30920-5 (cit. on pp. 1, 10).

[8] Liang Cai and Hao Chen. 'TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion'. In: *USENIX Workshop on Hot Topics in Security – HotSec*. USENIX Association, 2011 (cit. on pp. 1, 9).

[9] Liang Cai, Sridhar Machiraju and Hao Chen. 'Defending against sensor-sniffing attacks on mobile phones'. In: *Workshop on Networking, Systems, and Applications for Mobile Handhelds – MobiHeld*. ACM, 2009, pp. 31–36. ISBN: 978-1-60558-444-7 (cit. on p. 9).

[10] Peter Chapman and David Evans. 'Automated black-box detection of side-channel vulnerabilities in web applications'. In: *Conference on Computer and Communications Security – CCS 2011*. ACM, 2011, pp. 263–274. ISBN: 978-1-4503-0948-6 (cit. on pp. 12, 13).

[11] Qi Alfred Chen, Zhiyun Qian and Zhuoqing Morley Mao. 'Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks'. In: *USENIX Security Symposium 2014*. USENIX Association, 2014, pp. 1037–1052 (cit. on pp. 1, 6, 10, 44).

[12] Shuo Chen et al. 'Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow'. In: *IEEE Symposium on Security and Privacy – S&P 2010*. IEEE Computer Society, 2010, pp. 191–206. ISBN: 978-0-7695-4035-1 (cit. on p. 5).

[13] Maximilian Christ, Andreas W. Kempa-Liehr and Michael Feindt. 'Distributed and parallel time series feature extraction for industrial big data applications'. In: *CoRR* abs/1610.07717 (2016) (cit. on p. 27).

[14] Android Developers. *Android 5.0 Behavior Changes - getRecentTasks()*. 2014. URL: `https://developer.android.com/about/versions/android-5.0-changes.html#BehaviorGetRecentTasks` (visited on 08/03/2018) (cit. on p. 44).

[15] Android Developers. *Android 5.0 Behavior Changes - getRunningTasks(int)*. URL: `https://developer.android.com/reference/android/app/ActivityManager.html#getRunningTasks(int)` (visited on 30/03/2018) (cit. on p. 59).

[16] Android Developers. *Android 6.0 Changes - Browser Bookmark Changes*. 2015. URL: `https://developer.android.com/about/versions/marshmallow/android-6.0-changes.html#behavior-bookmark-browser` (visited on 08/03/2018) (cit. on p. 36).

[17] Android Developers. *Android 6.0 Changes - Runtime Permissions*. URL: https://developer.android.com/about/versions/marshmallow/android-6.0-changes.html#behavior-runtime-permissions (visited on 19/03/2018) (cit. on p. 16).

[18] Android Developers. *Android 8.0 Features and APIs - Updated Java language support*. 2017. URL: https://developer.android.com/about/versions/oreo/android-8.0.html#java (visited on 08/03/2018) (cit. on p. 23).

[19] Android Developers. *Android Debug Bridge (ADB)*. URL: https://developer.android.com/studio/command-line/adb.html (visited on 08/03/2018) (cit. on p. 25).

[20] Android Developers. *Android Developers Documentation*. URL: https://developer.android.com/reference/packages.html (visited on 28/02/2018) (cit. on p. 22).

[21] Android Developers. *Android NDK Native APIs*. URL: https://developer.android.com/ndk/guides/stable_apis.html) (visited on 17/03/2018) (cit. on p. 63).

[22] Android Developers. *App Manifest Overview*. URL: https://developer.android.com/guide/topics/manifest/manifest-intro (visited on 07/05/2018) (cit. on p. 24).

[23] Android Developers. *Context*. URL: https://developer.android.com/reference/android/content/Context (visited on 07/05/2018) (cit. on p. 30).

[24] Android Developers. *Context - getSystemService*. URL: https://developer.android.com/reference/android/content/Context.html#getSystemService(java.lang.String) (visited on 08/03/2018) (cit. on p. 35).

[25] Android Developers. *File - getUsableSpace()*. URL: https://developer.android.com/reference/java/io/File.html#getUsableSpace() (visited on 15/03/2018) (cit. on p. 39).

[26] Android Developers. *Manifest.permission - PACKAGE_USAGE_STATS*. URL: https://developer.android.com/reference/android/Manifest.permission.html#PACKAGE_USAGE_STATS (visited on 08/03/2018) (cit. on p. 17).

[27] Android Developers. *NetworkStats - getNextBucket*. URL: https://developer.android.com/reference/android/app/usage/NetworkStats.html#getNextBucket(android.app.usage.NetworkStats.Bucket) (visited on 16/03/2018) (cit. on p. 60).

[28] Android Developers. *Parameter - getName*. 2017. URL: https://developer.android.com/reference/java/lang/reflect/Parameter.html#getName() (visited on 08/03/2018) (cit. on p. 23).

[29] Android Developers. *Permissions Overview - Protection levels*. URL: https://developer.android.com/guide/topics/permissions/overview.html#normal-dangerous (visited on 08/03/2018) (cit. on p. 16).

[30] Android Developers. *Process - getElapsedCpuTime*. URL: https://developer.android.com/reference/android/os/Process.html#getElapsedCpuTime() (visited on 18/04/2018) (cit. on p. 56).

[31] Android Developers. *StorageManager - getAllocatableBytes*. URL: https://developer.android.com/reference/android/os/storage/StorageManager.html#getAllocatableBytes(java.util.UUID) (visited on 18/04/2018) (cit. on p. 39).

[32] Android Developers. *StorageStatsManager - getFreeBytes*. URL: https://developer.android.com/reference/android/app/usage/StorageStatsManager.html#getFreeBytes(java.util.UUID) (visited on 18/04/2018) (cit. on p. 39).

[33] Android Developers. *TelephonyManager - getDataActivity*. 2017. URL: https://developer.android.com/reference/android/telephony/TelephonyManager.html#getDataActivity() (visited on 10/03/2018) (cit. on p. 40).

[34] Android Developers. *TrafficStats - getUidRxBytes*. URL: https://developer.android.com/reference/android/net/TrafficStats.html#getUidRxBytes(int) (visited on 14/03/2018) (cit. on p. 37).

[35] Android Developers. *UI/Application Exerciser Monkey*. URL: https://developer.android.com/studio/test/monkey (visited on 04/05/2018) (cit. on p. 35).

[36] Wenrui Diao et al. 'No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis'. In: *IEEE Symposium on Security and Privacy – S&P 2016*. IEEE Computer Society, 2016, pp. 414–432. ISBN: 978-1-5090-0824-7 (cit. on pp. 1, 6, 44, 59).

[37] Hui Ding et al. 'Querying and mining of time series data: experimental comparison of representations and distance measures'. In: *PVLDB* 1 (2008), pp. 1542–1552 (cit. on p. 20).

[38] Ira R Forman, Nate Forman and John Vlissides Ibm. 'Java reflection in action'. In: (2004) (cit. on p. 18).

[39] Jeffrey Friedman. 'TEMPEST: A signal problem'. In: *NSA Cryptologic Spectrum* 35 (1972), p. 76 (cit. on p. 4).

[40] Qian Ge et al. 'A survey of microarchitectural timing attacks and countermeasures on contemporary hardware'. In: *J. Cryptographic Engineering* 8 (2018), pp. 1–27 (cit. on p. 5).

[41] Google Git. *AndroidManifest.xml*. 2017. URL: `https://android.googlesource.com/platform/frameworks/base.git/+/android-8.1.0_r18/core/res/AndroidManifest.xml` (visited on 08/03/2018) (cit. on p. 17).

[42] Aliaksei Halachkin. *Dynamic Time Warping in C and Python*. URL: `https://github.com/honeyext/cdtw/` (visited on 11/03/2018) (cit. on pp. 20, 26).

[43] Samuli Hemminki, Petteri Nurmi and Sasu Tarkoma. 'Accelerometer-based transportation mode detection on smartphones'. In: *Conference on Embedded Network Sensor Systems – SenSys 2013*. ACM, 2013, 13:1–13:14. ISBN: 978-1-4503-2027-6 (cit. on p. 11).

[44] Filip Hermans and Elena Tsiporkova. 'Merging microarray cell synchronization experiments through curve alignment'. In: *Bioinformatics* 23 (2007), pp. 64–70 (cit. on p. 18).

[45] Bo-Jhang Ho et al. 'From Pressure to Path: Barometer-based Vehicle Tracking'. In: *Embedded Systems for Energy-Efficient Built Environments – BuildSys*. ACM, 2015, pp. 65–74. ISBN: 978-1-4503-3981-0 (cit. on pp. 1, 12).

[46] Ralf Hund, Carsten Willems and Thorsten Holz. 'Practical Timing Side Channel Attacks against Kernel Space ASLR'. In: *IEEE Symposium on Security and Privacy – S&P 2013*. IEEE Computer Society, 2013, pp. 191–205. ISBN: 978-1-4673-6166-8 (cit. on p. 5).

[47] *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society. ISBN: 978-1-5090-0824-7.

[48] Alexa Internet Inc. *Alexa Top 500 Global Sites*. URL: `https://www.alexa.com/topsites/` (visited on 14/03/2018) (cit. on p. 37).

[49] Suman Jana and Vitaly Shmatikov. 'Memento: Learning Secrets from Process Footprints'. In: *IEEE Symposium on Security and Privacy – S&P 2012*. IEEE Computer Society, 2012, pp. 143–157. ISBN: 978-0-7695-4681-0 (cit. on pp. 1, 5).

[50] Thanh-Ha Le, Cécile Canovas and Jessy Clédière. 'An overview of side channel analysis attacks'. In: *Asia Conference on Computer and Communications Security – AsiaCCS*. ACM, 2008, pp. 33–43. ISBN: 978-1-59593-979-1 (cit. on p. 5).

[51] Jiayang Liu et al. 'uWave: Accelerometer-based Personalized Gesture Recognition and Its Applications'. In: *Pervasive Computing and Communication Workshops – PerCom 2009*. IEEE Computer Society, 2009, pp. 1–9. ISBN: 978-1-4244-3304-9 (cit. on p. 18).

[52] Jacques Malenfant, Marco Jacques and Franois Nicolas Demers. 'A tutorial on behavioral reflection and its implementation'. In: *Proceedings of the Reflection*. Vol. 96. 1996, pp. 1–20 (cit. on p. 17).

[53] Ali Mesbah, Engin Bozdag and Arie van Deursen. 'Crawling AJAX by Inferring User Interface State Changes'. In: *International Conference on Web Engineering – ICWE 2008*. IEEE Computer Society, 2008, pp. 122–134. ISBN: 978-0-7695-3261-5 (cit. on p. 13).

[54] Yan Michalevsky, Dan Boneh and Gabi Nakibly. 'Gyrophone: Recognizing Speech from Gyroscope Signals'. In: *USENIX Security Symposium 2014*. USENIX Association, 2014, pp. 1053–1067 (cit. on p. 10).

[55] Yan Michalevsky et al. 'PowerSpy: Location Tracking Using Mobile Device Power Analysis'. In: *USENIX Security Symposium 2015*. USENIX Association, 2015, pp. 785–800 (cit. on pp. 8, 59).

[56] Emiliano Miluzzo et al. 'Tapprints: your finger taps have fingerprints'. In: *Mobile Systems – MobiSys 2012*. ACM, 2012, pp. 323–336. ISBN: 978-1-4503-1301-8 (cit. on pp. 7, 10).

[57] Meinard Müller. *Dynamic time warping*. Springer, 2007, pp. 69–84. URL: https://doi.org/10.1007/978-3-540-74048-3 (cit. on p. 18).

[58] Sashank Narain et al. 'Inferring User Routes and Locations Using Zero-Permission Mobile Sensors'. In: *IEEE Symposium on Security and Privacy – S&P 2016*. IEEE Computer Society, 2016, pp. 397–413. ISBN: 978-1-5090-0824-7 (cit. on pp. 1, 11).

[59] Sarfraz Nawaz and Cecilia Mascolo. 'Mining users' significant driving routes with low-power sensors'. In: *Conference on Embedded Network Sensor Systems – SenSys 2014*. ACM, 2014, pp. 236–250. ISBN: 978-1-4503-3143-2 (cit. on pp. 1, 11).

[60] Pat Niemeyer. *BeanShell - Lightweight Scripting for Java*. URL: http://beanshell.org/ (visited on 11/03/2018) (cit. on p. 30).

[61] Oracle. *The Java$^{TM}$ Tutorials - Trail: The Reflection API*. 2017. URL: https://docs.oracle.com/javase/tutorial/reflect/ (visited on 08/03/2018) (cit. on pp. 17, 18).

[62] Emmanuel Owusu et al. 'ACCessory: password inference using accelerometers on smartphones'. In: *Mobile Computing Systems and Applications – HotMobile 2012*. ACM, 2012, p. 9. ISBN: 978-1-4503-1207-3 (cit. on pp. 1, 7, 9).

[63] Fabian Pedregosa et al. 'Scikit-learn: Machine Learning in Python'. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on pp. 20, 26).

[64] Dan Ping, Xin Sun and Bing Mao. 'TextLogger: inferring longer inputs on touch screen using motion sensors'. In: *Security and Privacy in Wireless and Mobile Networks – WISEC 2015*. ACM, 2015, 24:1–24:12. ISBN: 978-1-4503-3623-9 (cit. on pp. 7, 10).

[65] *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems, SenSys '14, Memphis, Tennessee, USA, November 3-6, 2014*. ACM. ISBN: 978-1-4503-3143-2.

[66] *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. USENIX Association.

[67] Android Open Source Project. *Frequently Asked Questions - When are source code releases made?* URL: https://source.android.com/setup/start/faqs#when-are-source-code-releases-made (visited on 30/03/2018) (cit. on p. 62).

[68] Android Open Source Project. *Security-Enhanced Linux in Android*. URL: https://source.android.com/security/selinux/ (visited on 19/03/2018) (cit. on p. 16).

[69] Selenium Project. *Android Developers Documentation*. URL: https://www.seleniumhq.org/ (visited on 21/03/2018) (cit. on p. 13).

[70] Andrew Raij et al. 'Privacy risks emerging from the adoption of innocuous wearable sensors in the mobile environment'. In: *Conference on Human Factors in Computing Systems – CHI 2011*. ACM, 2011, pp. 11–20. ISBN: 978-1-4503-0228-9 (cit. on p. 9).

[71] H. Sakoe and S. Chiba. 'Dynamic programming algorithm optimization for spoken word recognition'. In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 26.1 (Feb. 1978), pp. 43–49. ISSN: 0096-3518. DOI: 10.1109/TASSP.1978.1163055 (cit. on p. 18).

[72] Kartik Sankaran et al. 'Using mobile phone barometer for low-power transportation context detection'. In: *Conference on Embedded Network Sensor Systems – SenSys 2014*. ACM, 2014, pp. 191–205. ISBN: 978-1-4503-3143-2 (cit. on p. 11).

[73] Laurent Simon, Wenduan Xu and Ross J. Anderson. 'Don't Interrupt Me While I Type: Inferring Text Entered Through Gesture Typing on Android Keyboards'. In: *PoPETs* 2016 (2016), pp. 136–154 (cit. on pp. 6, 55).

[74] Android Source. *Interface Requirements - Filesystems - procfs*. URL: https://source.android.com/devices/architecture/kernel/reqs-interfaces#procfs (visited on 08/03/2018) (cit. on pp. 1, 5, 14).

[75] Android Source. *Interface Requirements - Filesystems - sysfs*. URL: https://source.android.com/devices/architecture/kernel/reqs-interfaces#sysfs (visited on 30/04/2018) (cit. on p. 5).

[76] Raphael Spreitzer. 'PIN Skimming: Exploiting the Ambient-Light Sensor in Mobile Devices'. In: *Security and Privacy in Smartphones & Mobile Devices – SPSM@CCS*. ACM, 2014, pp. 51–62. ISBN: 978-1-4503-3155-5 (cit. on pp. 1, 11).

[77] Raphael Spreitzer et al. 'Exploiting Data-Usage Statistics for Website Fingerprinting Attacks on Android'. In: *Security and Privacy in Wireless and Mobile Networks – WISEC 2016*. ACM, 2016, pp. 49–60. ISBN: 978-1-4503-4270-4 (cit. on pp. 1, 7, 38).

[78] Raphael Spreitzer et al. 'ProcHarvester: Fully Automated Analysis of Procfs Side-Channel Leaks on Android'. In: *Asia Conference on Computer and Communications Security – AsiaCCS 2018* (cit. on pp. 1, 2, 12, 14, 22, 36, 44, 59, 61, 63).

[79] Raphael Spreitzer et al. 'Systematic Classification of Side-Channel Attacks: A Case Study for Mobile Devices'. In: *IEEE Communications Surveys and Tutorials* 20 (2018), pp. 465–488 (cit. on pp. 4, 5).

[80] Eran Tromer, Dag Arne Osvik and Adi Shamir. 'Efficient Cache Attacks on AES, and Countermeasures'. In: *J. Cryptology* 23 (2010), pp. 37–71 (cit. on p. 4).

[81] Zhi Xu, Kun Bai and Sencun Zhu. 'TapLogger: inferring user inputs on smartphone touchscreens using on-board motion sensors'. In: *Security and Privacy in Wireless and Mobile Networks – WISEC 2012*. ACM, 2012, pp. 113–124. ISBN: 978-1-4503-1265-3 (cit. on pp. 1, 9).

[82] Lin Yan et al. 'A Study on Power Side Channels on Mobile Devices'. In: *Symposium of Internetware – Internetware 2015*. ACM, 2015, pp. 30–38. ISBN: 978-1-4503-3641-3 (cit. on pp. 1, 8).

[83] Kehuan Zhang et al. 'Sidebuster: automated detection and quantification of side-channel leaks in web application development'. In: *Conference on Computer and Communications Security – CCS 2010*. ACM, 2010, pp. 595–606. ISBN: 978-1-4503-0245-6 (cit. on pp. 12, 13).

[84] Nan Zhang et al. 'Leave Me Alone: App-Level Protection against Runtime Information Gathering on Android'. In: *IEEE Symposium on Security and Privacy – S&P 2015*. IEEE Computer Society, 2015, pp. 915–930. ISBN: 978-1-4673-6949-7 (cit. on pp. 38, 58).

[85] N. Zhang et al. *Leave Me Alone: App-Level Protection against Runtime Information Gathering on Android*. 2015. URL: https://sites.google.com/site/appguaridan/ (visited on 16/03/2018) (cit. on p. 58).

[86] Xiaokuan Zhang et al. 'OS-level Side Channels without Procfs: Exploring Cross-App Information Leakage on iOS'. In: (2018). URL: http://dx.doi.org/10.14722/ndss.2018.23260 (cit. on pp. 50, 62).

[87] Xiao-yong Zhou et al. 'Identity, location, disease and more: inferring your secrets from android public resources'. In: *Conference on Computer and Communications Security – CCS 2013*. ACM, 2013, pp. 1017–1028. ISBN: 978-1-4503-2477-9 (cit. on pp. 7, 8, 44).