Universidade Estadual de Campinas
Instituto de Computação

Vitor Monte Afonso

Improving Android Security with Malware Detection and Automatic Security Policy Generation

Aprimorando a Segurança do Android Através de Detecção de Malware e Geração Automática de Políticas

CAMPINAS
2016

Vitor Monte Afonso

Improving Android Security with Malware Detection and Automatic Security Policy Generation

Aprimorando a Segurança do Android Através de Detecção de Malware e Geração Automática de Políticas

Tese apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Supervisor/Orientador: Prof. Dr. Paulo Lício de Geus
Co-supervisor/Coorientador: Prof. Dr. André Ricardo Abed Grégio

Este exemplar corresponde à versão final da Tese defendida por Vitor Monte Afonso e orientada pelo Prof. Dr. Paulo Lício de Geus.

CAMPINAS
2016

Na versão final, esta página será substituída pela ficha catalográfica.

De acordo com o padrão da CCPG: “Quando se tratar de Teses e Dissertações financiadas por agências de fomento, os beneficiados deverão fazer referência ao apoio recebido e inserir esta informação na ficha catalográfica, além do nome da agência, o número do processo pelo qual recebeu o auxílio.”
e
“caso a tese de doutorado seja feita em Cotutela, será necessário informar na ficha catalográfica o fato, a Universidade convenente, o país e o nome do orientador.”

Universidade Estadual de Campinas
Instituto de Computação

Vitor Monte Afonso

Improving Android Security with Malware Detection and Automatic Security Policy Generation

Aprimorando a Segurança do Android Através de Detecção de Malware e Geração Automática de Políticas

Campinas, 19 de dezembro de 2016

Acknowledgements

I would like to begin thanking University of Campinas, Samsung and CAPES for the financial support I received during these years.

I also thank my advisor Paulo Lício de Geus, who has been advising me since 2008, even before I became a Master's student. His recommendations and ideas were very important for my development as a researcher and for my work.

I thank all friends and colleagues that helped me during this journey, specially André, Dario and Furuse, who worked with me for many years. All the chit-chatting and laughter certainly made this time very pleasant.

I would also like to thank Giovanni Vigna and Christopher Kruegel, for advising me while I was in Santa Barbara, and also Antonio and Yanick, for the many discussions that resulted in our NDSS paper.

Last but not least, I thank my family, my sister, mom and dad, and my girlfriend Mai for the love and support I have always received.

Resumo

Dispositivos móveis têm evoluído constantemente, recebendo novas funcionalidades e se tornando cada vez mais ubíquos. Assim, eles se tornaram alvos lucrativos para criminosos. Como Android é a plataforma líder em dispositivos móveis, ele se tornou o alvo principal de desenvolvedores de *malware*. Além disso, a quantidade de apps maliciosas encontradas por empresas de segurança que têm esse sistema operacional como alvo cresceu rapidamente nos últimos anos.

Esta tese aborda o problema da segurança de tais dispositivos por dois lados: (i) analisando e identificando apps maliciosas e (ii) desenvolvendo uma política de segurança que pode restringir a superfície de ataque disponível para código nativo. Para tanto, foi desenvolvido um sistema para analisar apps dinamicamente, monitorando chamadas de API e chamadas de sistema. Destes traços de comportamento extraiu-se atributos, que são utilizados por um algoritmo de aprendizado de máquina para classificar apps como maliciosas ou benignas. Um dos problemas principais de sistemas de análise dinâmica é que eles possuem muitas diferenças em relação a dispositivos reais, e exemplares de *malware* podem usar essas características para identificar se estão sendo analisados, impedindo assim que as ações maliciosas sejam observadas. Para identificar apps maliciosas de Android que evadem análises, desenvolveu-se uma técnica que compara o comportamento de uma app em um dispositivo real e em um emulador. Identificou-se as ações que foram executadas apenas no sistema real e se a divergência foi causada por caminhos de código diferentes serem explorados ou por algum erro não relacionado. Por fim, realizou-se uma análise em larga escala de apps que utilizam código nativo, a fim de se identificar como este é usado por apps legítimas e também para se criar uma política de segurança que restrinja as ações de *malware* que usam este tipo de código.

Abstract

Mobile devices have been constantly evolving, receiving new functionalities and becoming increasingly ubiquitous. Thus, they became lucrative targets for miscreants. Since Android is the leading platform for mobile devices, it became the most popular choice for malware developers. Moreover, the amount of malicious apps, found by security companies, that target this platform rapidly increased in the last few years.

This thesis approaches the security problem of such devices in two ways: (i) by analyzing and identifying malicious apps, and (ii) by developing a sandboxing policy that can restrict the attack surface available to native code. A system was developed to dynamically analyze apps, monitoring API calls and system calls. From these behavior traces attributes were extracted, which are used by a machine learning algorithm to classify apps as malicious or benign. One of the main problems of dynamic analysis systems is that they have many differences compared to real devices, and malware can leverage these characteristics to identify whether they are being analyzed or not, thus being able to prevent the malicious actions from being observed. To identify Android malware that evades analyses, a technique was developed to compare the behavior of an app on a real device and on an emulator. Actions that were only executed in the bare metal system were identified, recognizing whether the divergence was caused by different code paths being explored or by some unrelated error. Finally, a large-scale analysis of apps that use native code was performed, in order to identify how native code is used by benign apps and also to generate a sandboxing policy to restrict malware that use such code.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Motivation

Since their creation, mobile devices have been evolving steadily, acquiring new features and more resources. This led to them being used in many different activities, including areas involving critical information, such as banking credentials and credit card information. Because of that, mobile devices became very attractive targets. A study by Unucheck and Chebyshev [79] shows that Kaspersky Lab detected 884,774 new malicious mobile programs in 2015.

As Android is the mobile platform that has the highest number of users (87.6% of market share in the second quarter of 2016, according to IDC [43]), it is also the main target of attacks. According to PulseSecure [65], in 2014 97% of all mobile malware was developed for the Android platform.

Malicious apps are disseminated mainly through phishing, drive-by attacks, and app stores. Phishing messages may contain links to malicious apps and are sent over SMS or via some messaging app, such as WhatsApp. Drive-by attacks are carried out by exploits deployed in Web pages. When a vulnerable browser accesses it, the exploit is able to execute code in the victim's system. To infect users through app stores, malware are submitted to them disguising as some legitimate app, such as a game. In fact, in many cases miscreants modify some popular app to include malicious actions while keeping the app's main functionalities, in a process known as repackaging [97].

To protect users from these threats, there are mechanisms employed in devices and in app stores. On the device side, users can install anti-malware software, which will search for insecure configurations and also inspect other apps installed on the system to check for known malicious patterns. Furthermore, there is a series of security mechanisms employed by the Android operating system to restrict what apps can do. These mechanisms include, for instance, the permission enforcement, which forces apps to declare the use of certain functionalities in the manifest to be able to use them, and the app sandboxing, which restricts the access of apps to the filesystem and to the memory of other apps.

On the app store side, static and dynamic analysis is used to identify if apps contain malicious behavior or not. Malicious apps are then removed from the store, preventing them from infecting users. These types of analyses are also performed by security companies that produce signatures for anti-malware software, in order to identify whether unknown apps are malicious or not.

Static analysis techniques work by inspecting apps without the need to execute them. Several techniques of this type have been proposed by researchers to analyze Android apps [9, 10, 14, 33, 35, 39, 40, 54, 56, 85, 88, 92, 94, 98, 99]. They obtain information about the app by inspecting the dalvik code, the manifest and other files deployed within apps. One of the main weaknesses of static analysis techniques for Android apps is only being able to handle the Java part of them. Android apps can also contain native code components developed in C or C++, which are able to modify the Java code at runtime and may also directly perform malicious actions. Thus, static analysis systems that only look at Java code may miss malicious behavior of apps.

To address this issue, researchers [11, 72, 77] have proposed isolating native code from Java code and applying restrictions to native code, preventing malicious actions. However, the lack of data regarding the use of native code by benign apps makes designing security policies that do not affect many benign apps a difficult challenge. One of the contributions of this thesis is a study on the use of native code by benign apps and the proposal of a methodology to create a sandboxing policy to restrict this type of code, reducing the attack surface available. To accomplish this we developed a dynamic analysis system to analyze native code components of apps and performed a large-scale analysis of benign Android apps. More precisely, we statically inspected 1,208,476 Android apps to see if they use native code, then we dynamically analyzed the 446,562 that were found to use it. We provide insights into how native code is used by real-world Android apps. Moreover, our system can be used to create a native code sandboxing policy that allows for normal execution of the native code behaviors observed during the dynamic analysis of a predefined threshold of apps, while reducing the attack surface and thus limiting many malicious behaviors (e.g. root exploits).

Differently from its static counterpart, dynamic analysis is carried out by executing apps in a controlled environment and monitoring its behavior [18, 26, 28, 53, 75, 76, 91]. One of the main issues of this type of approach is related to malware that employ anti-analysis features. Analysis environments have several differences in relation to devices of real users, and some malware, sometimes referred to as evasive malware, exploit these differences to identify if they are being analyzed. When this happens, they can simply stop executing or perform only harmless actions, preventing the analysis system from observing the malicious behavior.

Another contribution of this thesis is a novel technique to identify evasive Android malware using dynamic analysis. We analyze apps in a baremetal and in an emulated environment, identifying the actions that are only executed in baremetal. For each action identified, we identify why it was not performed in the emulated environment, differentiating when there was an evasion and when there was an analysis problem that prevented the app from executing all its actions. We compare our approach to existing approaches that identify evasive Windows malware and demonstrate that ours is more appropriate to the Android context.

Researchers have proposed several approaches to use information obtained from dynamic analysis, static analysis or a combination of both to identify malicious apps [9, 70, 76, 88, 96, 98]. Improving these techniques is an important area of research to make devices more secure. Another contribution of this thesis is a system that analyses Android apps dynamically and classifies them as malicious or not, improving on known techniques. Our system monitors the use of Android APIs and system calls, extract features from these traces and uses machine learning for the classification. We trained it with 3,780 applications and tested it using 3,740 samples, obtaining an accuracy of 96.82%.

The aforementioned contributions were included in papers published or submitted for publication during the development of this research. Moreover, this thesis is organized as a collection of these papers.

1.2 Objectives

The main goal of this thesis is to contribute to improving Android security. We do this by focusing on two aspects: dynamic analysis and classification of malware, and restriction of native code. To improve the dynamic analysis and classification aspect we developed a system that obtained better results than similar systems by using different attributes in the classification. We also developed a novel technique to identify if an Android malware sample has anti-analysis features. To improve on the restriction of native code, we performed a large-scale analysis of apps to study how they use native code. We provide several insights on the use of native code by benign apps and we also create a security policy to restrict it, reducing the attack surface available to malicious code.

1.3 Contributions

The contributions of this thesis are the contributions included in the papers that comprise it. To make this information more easily identifiable, we present all the contributions summarized in this section.

The contributions related to the dynamic analysis and classification of Android malware were presented in the paper published in 2015 in the Journal of Computer Virology and Hacking Techniques and are the following [6]:

- We develop a dynamic analysis system to monitor Android API function calls and system calls, in order to gather information about apps. Currently available systems are tied to Android OS versions or to the SDK-provided emulator, whereas our approach is independent of the emulator and more portable, as it does not modify the Android OS;
- Our system is also able to classify apps as benign or malicious. We tested it with thousands of apps, correctly classifying 96.66% of them. To accomplish better results than similar systems we extract novel features, showing that those based on API function calls greatly increase the detection rate.

The contributions related to the study and restriction of native code in Android apps were published in 2016 at the Network and Distributed System Security Symposium and are the following [5]:

- We develop a tool to monitor the execution of native components in Android apps and we use this tool to perform the largest (in terms of number of apps and detail of information acquired) study of native code usage in Android;
- We systematically analyze the collected data, providing actionable insights into how benign apps use native code;
- Our results show that completely eliminating permissions of native code is not ideal, as this policy would break, as a lower bound, 3,669 of the apps in our dataset. However, we propose that our dynamic analysis system can be used to derive a native code sandboxing policy that limits many malicious behaviors, while allowing the normal execution of the native code behaviors observed during the dynamic analysis of a predefined threshold of apps (99.77% in our experiment).

The contributions related to the identification of Android malware with anti-analysis features were included in a paper submitted to the 2017 International Conference on Dependable Systems and Networks and are the following:

- We present a novel technique to identify evasive Android malware by comparing traces obtained in a baremetal environment with traces obtained in an emulated environment. Our technique identifies the cause of each action performed only in baremetal, filtering out those that were not executed due to some problem during analysis;
- We compared our approach to the detection techniques that focus on Windows malware, demonstrating that our technique is more appropriate for the Android context;
- We tested our technique with 1,470 samples, identifying 192 that employ evasive techniques, and discuss the techniques used by a subset of them to evade analysis.

1.4 Related Work

For ease of reading, we summarize in this chapter all background and related work of the articles included in this thesis, and we also include new research published after them.

1.4.1 Android

Android is an operating system for mobile devices that uses a customized version of the Linux kernel. To every app is assigned a unique user identifier (`uid`), at installation time, and group identifiers (`gids`), according to the requested permissions. Every app is executed in a separate Linux process, which is a child of `Zygote`, a process started when the system is initialized.

Apps are written mainly in Java, then compiled to Dalvik bytecode, but they can also contain native code components, developed in C/C++ and compiled to executable files or shared libraries. The interaction between native code components and Java code is defined by the Java Native Interface (JNI) specification.

Besides code, apps can contain resources, such as images, information related to the apps' certificates and the file `AndroidManifest.xml`. The manifest defines several information related to the app, such as permissions needed, activities, services, broadcast receivers, content providers, the minimum system version necessary for the app to run properly and the shared libraries referenced by the app.

Interactions between apps are performed through Intents, which are messages defining one or more receptors and possibly some data. This type of communication can also be used intra-application. Moreover, all Intents flow through an Android system-level process called Binder [22].

On Android, some operations and resources are protected by permissions. Apps must declare the permissions needed in the manifest. Before version 6, permissions had to be authorized by the user at install time and the user only had the option to continue with all permissions or cancel the installation. Starting on version 6, the user can revoke specific groups of permissions for apps installed. Permissions are enforced app-wise using Linux access-control mechanisms and by system services that check if the app is allowed to access certain resources or perform some requested operation [30].

1.4.2 Malicious Android Apps

Researchers have described the key behavior characteristics of Android malware found in the wild [31, 74, 97], which are the following: user information stealing; premium calls and SMS messages, which generate costs to the user; SPAM SMS messages; search engine optimization; ransom; privilege escalation; remote control of the device. Furthermore, they identified the following vectors of infection: repackaging—modifying a legitimate app to include malicious code and redistribution of the modified app to app stores; update—an app seemingly legitimate downloads and executes malicious code; drive-by-download—malicious Web pages can exploit the Web browser to infect the system.

To prevent analysis systems from obtaining information about them, many malicious apps employ evasion techniques. Several works in the literature describe such techniques [45, 58, 62, 74, 80]. Spreitzenbarth [74] details the analysis of two Android malware families, namely Bmaster and FakeRegSMS, that use several anti-analysis techniques, such as waiting for a long period before executing the malicious actions. Matenaar and Schulz [58] present a method for an app to identify if it is executing inside Qemu, which is the basis of the Android emulator. Vidas and Christin [80] present anti-analysis techniques based on Android APIs, system properties, network information, Qemu characteristics, performance, hardware components, and software components. Another similar work is presented by Petsas et al. [62]; they demonstrate anti-analysis techniques based on Android APIs, system properties, sensors, and Qemu characteristics.

Instead of manually identifying differences between real and emulated devices, Jing et al. [45] developed Morpheus, a framework that automatically generates heuristics that can identify, based on files, system properties and Android APIs, whether a sample is running on an emulated environment or not.

1.4.3 Analysis and Detection of Android Malware

Researchers have proposed several systems to analyze Android malware and obtain information about them. Enck et al. [28] propose TaintDroid, a dynamic taint analysis system, which tracks sensitive data flow to detect when it is sent over the network. Sun et al. [78] propose TaintART, an approach similar to TaintDroid that works with the most recent Android runtime, ART. DroidBox [26] builds upon TaintDroid and monitors API calls, network data, and data leaks. Spreitzenbarth et al. presented Mobile-Sandbox, a system that uses DroidBox and Taintdroid to track the behavior of apps, and includes the use of *ltrace* tool to monitor native code [75]. Yan and Yin propose DroidScope [91], a virtual machine introspection-based analysis system that bridges the semantic gap reconstructing OS-level and Java-level semantic views from outside the emulator. AASandbox [18] monitors system calls using a kernel module. Harvester [67] combines program slicing with code generation and dynamic execution to extract runtime values, such as URLs and destination numbers of SMS messages, from obfuscated malware. Bichsel et al. [16] present an approach for deobfuscating apps based on probabilistic learning of large code bases. It learns a probabilistic model over thousands of non-obfuscated apps and use it to deobfuscate new ones. TriggerScope [32] uses static analysis to detect logic bombs, i.e., application logic that is only executed under certain (often narrow) circumstances. TriggerScope is capable of identifying time-, location-, and SMS-related triggers.

One of the main drawbacks of dynamic analysis is only being able to observe behavior that is actually executed. This means that the analysis system needs to provide the correct inputs so that the malicious behavior is triggered. Researchers have proposed systems that inspect the analyzed app in order to identify the inputs and paths that lead to the execution of suspicious code and then provide these at runtime [15, 87, 95].

Other systems leverage information obtained from dynamic or static analysis to classify apps as malicious or benign. Zhou et al. propose DroidRanger, a two-scheme system based on signatures and heuristics [98]. The signature-based scheme relies on common permissions and behavioral footprints to identify samples from known families. the heuristics-based filtering scheme identifies suspicious behaviors (e.g., downloading and executing code from the Web and dynamic loading of native code). Zheng et al. propose DroidAnalytics, a system to automatically collect, analyze and detect Android malware that makes use of repackaging, code obfuscation, or dynamic payloads [96]. It disassembles apps to obtain Android API calls. These are used within a three-level signature generation process, which extracts malware features at the opcode level to identify variants. Elish et al. propose a tool to determine whether unknown applications are malicious or not based on static data dependence analysis [27], which correlates user inputs with critical function calls. Malisa et al. [55] presents an approach to detect app impersonation attacks by extracting user interfaces from apps and analyzing the extracted screenshots.

Researchers have proposed several systems to identify Android malware using machine learning and different feature sets. PUMA [70] uses information obtained from apps' permissions [70]. DroidMat [88] uses clustering techniques applied to features statically extracted from apps' manifest file (permission, component, and intent information) and permission-related Android API calls from apps' bytecode. Another system that uses features obtained statically is DREBIN [9]. It extracts features from the manifest and dex code. Su et al. [76] present a framework that dynamically analyzes new apps to collect two sets of features: one related to tracing 15 system calls and the other related to network traffic statistics. StormDroid [21] also uses dynamic analysis and combines features related to permissions and sensitive API calls in a framework that can process large sets of apps. Zhu and Dumitras [100] present FeatureSmith, a system that generates a feature set by analyzing the contents of papers published in security conferences.

Since evasive malware is one of the main problems of analysis systems, independently of the target operating system, the automatic identification of this type of threat is an important research topic. Although this topic has not been explored in the Android context, systems to identify evasive Windows malware have been proposed [13,47,48,50,52]. Balzarotti et al. [13] propose a system that records the system calls executed by a sample on a reference environment and replay the monitored system calls on an emulator to identify if the observed behavior is different. Lindorfer et al. [52] analyze malware samples in different environments and identify differences on the observed actions, recognizing techniques that malware apply to detect the analysis environment or analysis system. BareCloud [48] dynamically analyzes malware in four different environments, including a baremetal one, and detects evasive samples by comparing the reports provided by these systems in a hierarchical approach. Kolbitsch et al. [50] detects and mitigates malicious programs that wait for some time (stall) before executing their malicious behavior. Malgene [47] analyzes evasive malware and uses sequence alignment on the system call traces obtained from a baremetal and an emulated environment to identify evasion signatures.

1.4.4 Protection mechanisms

Several approaches have been proposed to increase the security of Android, focusing on different issues. Dietz et al. [25] presents modifications to the Android system that allow an app to know the complete path taken by Intents received and to encrypt data transmitted trough Intents. To prevent apps from having to request more permissions than necessary, just to be able to use ad libraries, Yagemann and Du [90] propose changing the logic of access control of Intents to an app, which will work as an Intent firewall, allowing, blocking, or even modifying Intents. Shekhar et al. [71] present a technique to execute libraries in a separate process, with its own set of permissions.

Portokalidis et al. [64] present a security model in which a synchronized replica of the user's phone is executed in a server, which has much more resources and can use several attack identification techniques that would consume too much resources to be executed on a mobile device. Batyuk et al. [14] introduce a system that looks for malicious patterns in apps and patch them according to some policies. Rewriting bytecodes is also the approach used by the framework presented by Davis and Chen [23]. In this case, calls to certain methods are replaced by calls to methods inserted in the app by the framework; these methods use security policies to restrict the behavior of apps. Xu et al. [89] present a system that applies security policies by intercepting calls to *libc*.

Focused on preventing root exploits, Fedler et al. [29] propose a protection system that prevents apps from giving execution permission for custom executable files and by introducing a permission related to the use of the `System` class. PREC [42] tries to prevent root exploits by learning the normal behavior of apps during an analysis phase and then preventing deviations from the normal behavior.

Another way to protect the system is by isolating native code. The challenge of isolating native code components used by managed languages has been studied before. Klinkoff et al. [49] focus on the isolation of *.NET* applications, whereas Robusta [72] focuses on the isolation of native code used by Java applications. NativeGuard [77] and NaClDroid [11] are security frameworks for Android that places native code components in a separate app, and therefore a separate process as well.

Marforio et al. [57] propose a scheme to securely setup security indicators in the presence of malware on the users' devices. These indicators can help users identify malicious apps that pose as legitimate ones to perform phishing attacks. Ying et al. [93] study attacks using free floating windows and propose a priority framework to protect users against these threats.

1.4.5 Large Measurement Studies

Some researchers have analyzed large datasets of Android apps. Viennot et al. [81] did a large measurement study on 1,100,000 applications crawled from the Google Play app store. In particular, they collected meta-data and statistics taken from the Google Play store itself. Another important measurement study has been performed by Lindorfer et al. [53]. In their work, they analyzed over one million apps, of which 40% are malware, and discuss the trends of Android malware behavior observed.

1.5 Thesis outline

This thesis is organized as a collection of three papers, which are presented as they were published. The only modifications made to them are related to adjusting them to the thesis format. The remainder of this thesis is organized as follows. Chapter 2 contains three sections, one for each paper. Chapter 3 summarizes the results of all papers, facilitating their identification by the reader. Finally, Chapter 4 presents the conclusions and future work.

Chapter 2

Published Documents

2.1 Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy

Publication: This paper was published in the Proceedings of the Symposium on Network and Distributed System Security (NDSS) 2016

Vitor Afonso[1], Antonio Bianchi[2], Yanick Fratantonio[2], Adam Doupé[3], Mario Polino[4], Paulo de Geus[5], Christopher Kruegel[2], and Giovanni Vigna[2]
(1) University of Campinas
Email: vitor@lasca.ic.unicamp.br
(2) UC Santa Barbara
Email: {antoniob, yanick, chris, vigna}@cs.ucsb.edu
(3) Arizona State University
Email: doupe@asu.edu
(4) Politecnico di Milano
Email: mario.polino@polimi.it
(5) University of Campinas
Email: paulo@lasca.ic.unicamp.br

Abstract

Current static analysis techniques for Android applications operate at the Java level—that is, they analyze either the Java source code or the Dalvik bytecode. However, Android allows developers to write code in C or C++ that is cross-compiled to multiple binary architectures. Furthermore, the Java-written components and the native code components (C or C++) can interact.

Native code can access all of the Android APIs that the Java code can access, as well as alter the Dalvik Virtual Machine, thus rendering static analysis techniques for Java unsound or misleading. In addition, malicious apps frequently hide their malicious functionality in native code or use native code to launch kernel exploits.

It is because of these security concerns that previous research has proposed native code sandboxing, as well as mechanisms to enforce security policies in the sandbox. However, it is not clear whether the large-scale adoption of these mechanisms is practical: is it possible to define a meaningful security policy that can be imposed by a native code sandbox without breaking app functionality?

In this paper, we perform an extensive analysis of the native code usage in 1.2 million Android apps. We first used static analysis to identify a set of 446k apps potentially using native code, and we then analyzed this set using dynamic analysis. This analysis demonstrates that sandboxing native code with no permissions is not ideal, as apps' native code components perform activities that require Android permissions. However, our analysis provided very encouraging insights that make us believe that sandboxing native code can be feasible and useful in practice. In fact, it was possible to automatically generate a native code sandboxing policy, which is derived from our analysis, that limits many malicious behaviors while still allowing the correct execution of the behavior witnessed during dynamic analysis for 99.77% of the benign apps in our dataset. The usage of our system to generate policies would reduce the attack surface available to native code and, as a further benefit, it would also enable more reliable static analysis of Java code.

2.1.1 Introduction

Mobile operating systems allow third-party developers to create applications (hereafter referred to as *apps*) that extend the functionality of the mobile device. Apps span across all categories of use: banking, socializing, entertainment, news, health, sports, and travel.

Google's Android operating system currently enjoys the largest market share, currently at 84.7%, of all current smartphone operating systems [44]. The official app market for Android, the Google Play Store, has around 1.4 million available apps [7] (according to AppBrain, a third-party Google Play Store tracking site) with over 50 billion app downloads [84].

Android apps are typically written in Java, and then compiled to bytecode that runs on an Android-specific Java virtual machine, called the Dalvik Virtual Machine (DVM).[1] These apps can interact with the filesystem, the Android APIs (to access phone features such as GPS location, call history, microphone, or SMS messages), and even other apps.

The wealth of information stored on smartphones attracts miscreants who want to steal the user's information, send out premium SMS messages, or even have the user's device join a botnet [20].

Static analysis of Android applications has been proposed by various researchers to check the security properties of the apps that the user installs [10, 14, 33, 35, 39, 40, 54, 56, 85, 92, 94, 98, 99].

All the proposed static analysis techniques for Android apps have operated at the Java level—that is, these techniques process either the Java source code or the Dalvik bytecode. However, Android apps can also contain components written in *native code* (C or C++) using the Android NDK [36]. Some of the reasons why developers might use this feature, as stated by the NDK documentation [36], are:

> For certain types of apps, [native code] can be helpful so you can reuse existing code libraries written in these languages, but most apps do not need the Android NDK.
>
> Typical good candidates for the NDK are CPU-intensive workloads such as game engines, signal processing, physics simulation, and so on.

Using the NDK, the C or C++ code will be compiled and packaged with the app. Android provides an interface (JNI) for Java code to call functions of native code and vice versa.

While attempting to allow native code in Android apps is noble, there are serious security implications of allowing apps to execute code outside the Java ecosystem.

The existence of native code severely complicates static analysis of Android apps. First, to our knowledge, no static analysis of Android apps attempts to statically analyze the native code included in the app. Thus, malware authors can include the malicious payload/behavior in a native code component to evade detection. Furthermore, the native

[1] In recent versions, the bytecode is instead compiled and executed by a new runtime, called ART. For simplicity, in the rest of the paper we will only refer to the DVM. However, everything we describe conceptually applies to ART as well.

code in an Android app has *more capabilities* than the Java code. This is because the native code has direct access to the memory of the running process, and, because of this access, can read and modify the Dalvik Virtual Machine and its data.[2] Effectively, this means that the native code can completely modify and change the behavior of the Java code—*rendering all static analysis of the Java code unsound.*

In light of these security problems with native code usage in Android applications, researchers have turned to sandboxing mechanisms, which limit the interaction between the native code and the Java code [17, 72, 77]. This follows the least-privilege principle: The native code does not need full access to the Java code and thus should be sandboxed.

A native code sandbox should be *security-relevant* and *usable* with benign, real-world apps. These requirements result in the following properties:

- **Least-Privilege**: The native code of the app should have access only to what is strictly required, thus reducing the chances the native component could extensively damage the system.
- **Compartmentalization**: The native code of the app should communicate with the Java part only using specific, limited channels, so that the native component cannot modify, interact with, or otherwise alter the Java runtime and code in unexpected ways.
- **Usability**: The restrictions enforced by the sandbox must not prevent a significant portion of benign apps from functioning.
- **Performance**: The sandbox implementation must not impose a substantial performance overhead on apps.

Even though previous research has focused on the *mechanism* of native code sandbox enforcement [72, 77], to this point no research has focused on *how to generate a security policy* that a sandbox can enforce so that the policy is both practical (i.e., it would not break benign apps) and useful (i.e., it would limit malicious behaviors).

Sun and Tan [77], in their paper presenting the native code sandboxing mechanism NativeGuard, state:

> We decide to follow a heuristic approach and by default grant no permission to the [sandboxed native code] in NativeGuard. The approach is motivated by the observation that it is rare for legal native code to perform privileged operations, as it is a "bad practice" according to the NDK.

Sun and Tan are correct that the NDK considers native code performing privileged operations to be bad practice, however, *we need data to confirm this intuition.* We must know: what is the native code in real-world apps doing? How do real-world apps use native code? For instance, what if native code is used to perform exactly the same actions as Java code? In this case, it would not be possible to meaningfully constrain the permission

[2]Even if the Dalvik Virtual Machine memory is initially mapped as read-only, a native code component can change the memory permission by using the `mprotect` syscall.

of native code components, and enforcing the least-privilege principle would not grant any security benefits. We also need clarification as to how tightly coupled the communication is between the native code and the Java code. Enforcing compartmentalization might break or negatively affect tightly-coupled apps.

To answer these questions, we perform a large-scale analysis of real-world Android apps. Specifically, we look at how apps use native code, both statically and dynamically. We statically analyze 1,208,476 Android apps to see if they use native code, then we dynamically analyze the 446,562 that were determined to use native code. Our system is able to monitor the dynamic execution of an app, while recording activities performed by its native code components (e.g., invoked system calls, interactions between native and Java components). From this analysis, we shed light on *how real-world Android apps use native code.*

In addition, our dynamic analysis system can be used to generate a native code sandboxing policy that allows for normal execution of the native code behaviors observed during the dynamic analysis of a set threshold of apps, while reducing the attack surface and thus limiting many malicious behaviors (e.g., root exploits) of malicious apps.

The main contributions of this paper are the following:

- We develop a tool to monitor the execution of native components in Android applications and we use this tool to perform the largest (in terms of number of apps and detail of information acquired) study of native code usage in Android.
- We systematically analyze the collected data, providing actionable insights into how benign apps use native code. Moreover, we release the full raw data and we make it available to the community [4].
- Our results show that completely eliminating permissions of native code is not ideal, as this policy would break, as a lower bound, 3,669 of the apps in our dataset. However, we propose that our dynamic analysis system can be used to derive a native code sandboxing policy that limits many malicious behaviors, while allowing the normal execution of the native code behaviors observed during the dynamic analysis of a set threshold of apps (99.77% in our experiment).

2.1.2 Background

To understand the analysis that we perform on Android applications and our proposed policy, it is necessary to review the Android security mechanisms, how native code is used in Android, the damage that malicious native code can cause, and the previously proposed native code sandboxing mechanisms.

Android Security Mechanisms

When apps are installed on an Android phone, they are assigned a new user (`UID`) and groups (`GIDs`) based on the permissions requested by the app in its manifest. Every app is executed in a separate process, which is a child of `Zygote`, a process started when the system is initialized. Moreover, inter-process communication is done using intents which all flow through an Android system-level process called Binder [22].

On Android, some operations and resources are protected by permissions. Apps must declare the permissions needed in the manifest, and at installation time the requested permissions are presented to the user, who decides to continue or cancel the installation. Permissions are enforced app-wise using Linux access-control mechanisms and by system services that check if the app is allowed to access certain resources or perform the requested operation [30].

Native Code

Native code in Android apps is deployed in the app as ELF files, either executable files or shared libraries. There are four ways in which the Java code of an Android app can execute native code: Exec methods, Load methods, Native methods, and Native activity.
Exec methods. Executable files can be called from Java by two methods, namely `Runtime.exec` and `ProcessBuilder.start`. Hereinafter we refer to these methods as *Exec methods.*
Load methods. Native code in shared libraries can be loaded by the framework when a NativeActivity is declared in the manifest, along with its library name, or by the app through the following Java methods, which are hereinafter referred to as *Load methods:* `System.load`, `System.loadLibrary`, `Runtime.load`, and `Runtime.loadLibrary`. Native code in shared libraries can be invoked at loading time, through calls to native methods and through callbacks in native activities. When a library is loaded, its `_init` and `JNI_OnLoad` functions are called.
Native methods. Native methods are implemented in shared libraries and declared in Java. When the Java method is called, the framework executes the corresponding function in the native component. This mapping is done by the Java Native Interface (JNI) [38]. JNI also allows native code to interact with the Java part to perform actions such as calling Java methods and modifying Java fields.
Native activity. Native code is invoked in native activities using activities' callback functions, (e.g., `onCreate` and `onResume`), if defined in a native library.

Malicious Native Code

Malicious apps can use native code to hide malicious actions from static analysis of the Java portion of the app. These actions can be calls to methods in Java libraries, such as sending SMS messages, or complex attacks that involve exploiting the kernel or privileged processes to compromise the entire OS. These root exploits are possible because native code is allowed to directly call system calls. Another possible way that attackers can directly call system calls to execute root exploits is by exploiting vulnerabilities in native code used by benign apps.

As previous research has shown [77], because native code shares the same memory address space as the Dalvik Virtual Machine, it can completely modify the behavior of the Java code, rendering static analysis of the Java code fundamentally unsound. For instance, malicious code can use functions exported by *libDVM.so* to identify where the bytecode implementing a specific Java method is placed in memory. At this point, the native code can dynamically replace the method at run time.

Native Code Sandboxing Mechanisms

Several approaches have been proposed to sandbox native code execution. For instance, NativeGuard [77] and Robusta [72] move the execution of native code to a separate process. Two complementary goals are obtained: (1) the native code cannot tamper with the execution of the Java code and (2) different security constraints can be applied to the execution of the native code.

Communication between the Java code and the native code is then ensured by modifying the JNI interface to make the two processes communicate through an OS-provided communication channel (e.g., network sockets).

While moving native code to a separate process is a natural mechanism to achieve the aforementioned goals (because it relies on OS-provided security mechanisms, such as process memory separation or process permissions), other solutions are possible. For instance, thread-level memory protection (as proposed in Wedge [17]). However, applying this solution in Android would require significant modifications to the underlying Linux kernel.

2.1.3 Analysis Infrastructure

We designed and implemented a system that dynamically analyzes Android applications to study how native code is used and to automatically generate a native code sandboxing policy. Our analysis consists of an instrumented emulator, and it records all events and operations executed from within native code, such as invoked syscalls and native-to-Java communication. The dynamic instrumentation is completely generic, and it allows the usage of any manual or automatic instrumentation tool. The version of the Android system used was 4.3.

Since our goal was to obtain a comprehensive characterization of native code usage in real world applications, we used a corpus of 1,208,476 distinct—different package names and APK hashes—free Android apps that we have continuously downloaded from the Google Play store from May 2012–August 2014. The age of the apps varies throughout the time-frame, as we currently do not download new versions of apps.

Static Prefiltering

Performing dynamic analysis of all 1,208,476 apps by running each app would take a considerable amount of time; therefore, by using static analysis, we filtered the apps that had some indication of using native code. The characteristics we looked for in the apps are the following: having a native method, having a native activity, having a call to an Exec method, having a call to a Load method, or having an ELF file inside the APK.

We used the Androguard tool [24] as a basis for the static analysis. To identify native methods we searched for methods declared in the Dalvik bytecode with the modifier[3]

[3]Modifier here is an attribute of a method, similar to `public`. An example Dalvik method signature

Table 2.1: Results of the static analysis.

Apps	Type
267,158	Native method
42,086	Native activity
288,493	Exec methods
242,380	Load methods
221,515	ELF file
446,562	At least one of the above

"native." Native activities were identified by two means: (1) looking for a NativeActivity in the manifest and (2) looking for classes declared in the Dalvik bytecode that extend NativeActivity. Finally, calls to Exec and Load methods were identified by investigating method invocations in the bytecode.

Of the 1,208,476 apps statically analyzed, 446,562 apps (37.0%) used at least one of the previously mentioned ways of executing native code. Table 2.1 presents the number of apps that use each of these characteristics.

Dynamic Analysis System

Now that we have identified *which* Android apps use native code, we now want to understand *how* apps use native code. During the dynamic analysis we monitor several types of actions performed by the analyzed apps, including system calls, JNI calls, Binder transactions, calls to Exec methods, loading of third-party libraries, calls to native activities' native callbacks, and calls to native methods. The system calls were captured using the `strace` tool, and the other information we obtained through instrumentation.

To monitor JNI calls, calls to native methods, and library loading, we modified `libdvm`. However, we do not want to monitor all JNI calls, just JNI calls to the app's native code, rather than calls to native code in the standard libraries that Android includes. To avoid monitoring JNI calls in standard libraries and calls to native methods in standard libraries, we modified the "Method" structure to include a property indicating whether it belongs to a third-party library or not. When a third-party library is loaded, this property is set accordingly.

We modified `libbinder` to track and monitor Binder transactions. We record the class of the remote function being called and the number that identifies the function. To map the identifiers to function names, we parse the AIDL (Android Interface Definition Language) files and source files that define Binder interfaces. To find files that have such definitions, we search for uses of the macros `DECLARE_META_INTERFACE` and `IMPLEMENT_META_INTERFACE` and classes that extend "IInterface." Furthermore, to match identification numbers to names, we search in ".cpp" files for enumerations that use `IBinder::FIRST_CALL_TRANSACTION` and, in ".java" files, for variables defined using `IBinder.FIRST_CALL_TRANSACTION`. We use the names assigned `FIRST_CALL_TRANSACTION` as the functions with identifier 1, the ones assigned `FIRST_CALL_TRANSACTION + NUM` as

would be: `.method public native example()`.

the functions with identifier 1+NUM and, for the enumerations that only use `FIRST_-CALL_TRANSACTION` to define the first element, we consider they are increasing the identifier one by one.

Calls to Exec methods are identified by instrumenting `libjavacore`. Finally, to monitor the use of native callbacks in native activities, we modified `libandroid_runtime`.

We determine which actions were performed by native code and which by Java code after the dynamic analysis. To make this determination, we observe when threads change execution context from Java to native and from native to Java. Thus, we process all system calls, keeping a list of threads that are executing native code. We add a thread to this list when one of the following happens: Exec method is executed—we add the child process, which is then used to call `execve`, a custom (third-party) shared library is loaded, a native method is executed, or a callback in the native component of a native activity is executed. When these actions are completed and the execution control changes back to Java, the thread is removed from the list.

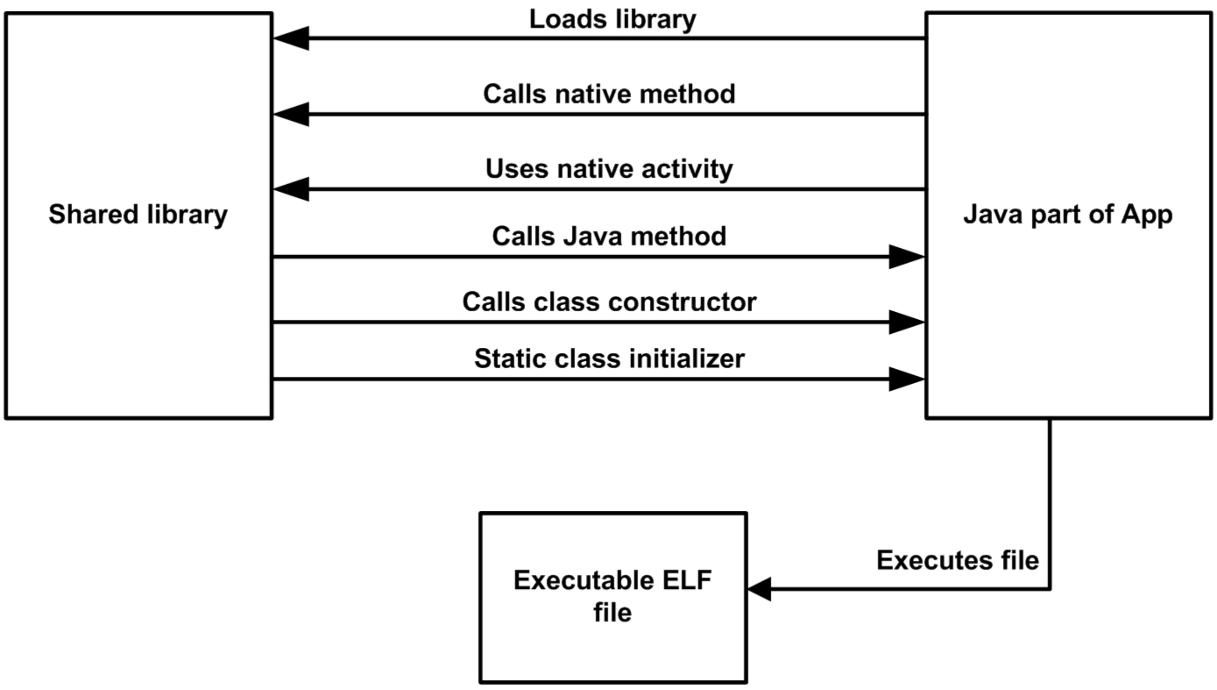


Figure 2.1: Possible transitions between native code and Java.

We also remove a thread from the list when one of the JNI methods in Table 2.2 is executed. The `Call*<TYPE>` functions are used to call Java methods, and the `NewObject*` functions are used to create instances of classes, which results in the execution of Java constructors. When these methods return, the thread is placed back on the list. Additionally, we remove a thread from the list when the `clinit` method, which is the static initialization block of a class, is executed. Figure 2.1 presents all mentioned transitions.

To understand how isolating the native code from the Java code would impact the performance of the apps, we also monitor the amount of data exchanged between native and Java code. We measured the amount of data passed in parameters of calls from native

Table 2.2: JNI methods that cause a transition from native to Java. <TYPE> can be the following: Object; Boolean; Byte; Char; Short; Int; Long; Float; Double; Void.

Call<TYPE>Method
CallNonVirtual<TYPE>Method
Call<TYPE>MethodA
CallNonVirtual<TYPE>MethodA
Call<TYPE>MethodV
CallNonVirtual<TYPE>MethodV
CallStatic<TYPE>Method
CallStatic<TYPE>MethodA
CallStatic<TYPE>MethodV
NewObject
NewObjectV
NewObjectA

code to Java methods and vice versa, as well as the size of the returned value. We also capture the size of data used to set fields in Java objects. The results of this analysis are presented in Section 2.1.5.

2.1.4 Evaluation and Insights

We ran both the static pre-filter and dynamic analysis across numerous physical machines and private-cloud virtual machines. In total, we used 100 cores and 444 GB of memory. Moreover, the analysis was run in parallel.

The dynamic analysis was performed using an instrumented Android emulator (as described in the previous section), and to keep the analysis time feasible we limited the analysis to two minutes for each app. To dynamically exercise each application, we followed an approach similar to what is used in Andrubis [86]: we used the Google Monkey [37] to stimulate the app with random events, and we then automatically generated a series of targeted *events* (by means of sending properly-crafted intents) to stimulate all activities, services, and broadcast receivers defined in the application.

Ideally, it would have been possible to use more sophisticated dynamic instrumentation systems. However, the large scale of our analysis motivated our choice to use a simpler approach, as it would have required a prohibitive amount of resources to run on hundreds of thousand of apps. While our dynamic instrumentation system is acceptable for the purposes of understanding the lower bound on what behaviors native code performs, the incompleteness inherent in dynamic analysis can affect the native code policies generated by our system. However, if Google or another large company were to adopt the idea of using a dynamic analysis system to automatically generate a native code security policy, they could use substantial resources to run the applications for longer periods of time, use sophisticated dynamic analysis approaches [68], or even introduce the instrumentation into the Android operating system and sample the behaviors from real-world devices.

During dynamic analysis, 33.6% (149,949) of the apps identified by static analysis as potentially having native code actually executed the native code. Table 2.3 presents the number of apps that executed each type of native code. These numbers constitute a lower bound of the apps that could actually execute native code.

In order to understand, for our study, why the native code was not reached during dynamic analysis, we manually analyzed, statically and dynamically, 20 random apps that were statically determined to have native code. For 40% (8) of them, we established through analysis of the decompiled code that the native code was unreachable from Java code (also known as deadcode). The remaining applications were too complex to be manually inspected, and we were not able to ascertain whether the native code components were not reached due to deadcode. For this reason, we dynamically analyzed and manually interacted with them and we did not find any path that led to the execution of the native code. Thus, we believe that also in this case the native code component was not reached due to deadcode, even if we were not able to be completely certain, due to the incomplete nature of manual analysis.

We further investigated *why* there was deadcode in these apps. In each case, the native code was deadcode in third-party libraries. In fact, in our experience, it often happens that an app includes a third-party library, to then actively use only a (sometimes very limited) subset of its functionality, thus leading to deadcode. Hence, we expect this to be the case for many apps where our analysis did not reach native code. As an additional experiment, we also manually and extensively dynamically exercised another 20 random apps. We observed no cases of significant changes in the results compared to the Google Monkey automated analysis (neither additional native code components were reached nor more syscalls were called).

To further understand the coverage of our dynamic analysis system we performed two additional experiments, one measuring the Java method coverage and one measuring the native code coverage. Section 2.1.8 discusses these experiments in depth.

Table 2.3: The number of apps that executed each type of native code.

Apps	Type
72,768	Native method
19,164	Native activity
132,843	Load library
27,701	Call executable file (27,599 standard, 148 custom and 46 both)
149,949	At least one of the above

2.1.5 Native Code Behavior—An Overview

We present in this section an overview of the actions performed by native code on Android. We split the actions into those performed by shared libraries (including those performed during library loading, native methods, and native activities) and those that are the result of invoking custom, executable, and binaries through Exec methods. We also present the actions performed using standard binaries (i.e., not created by the app), but in this case based on their names and parameters, instead of looking at the system calls.

Table 2.4: Overview of actions performed by custom shared libraries in native code.

Writing log messages
Performing memory management system calls, such as `mmap` and `mprotect`
Reading files in the application directory
Calling JNI functions
Performing general multiprocess and multithread related system calls, such as `fork`, `clone`, `setpriority`, and `futex`
Reading common files, such as system libraries, font files, and "/dev/random"
Performing other operations on files or file descriptors, such as `lseek`, `dup`, and `readlink`
Performing operations to read information about the system, such as `uname`, `getrlimit`, and reading special files (e.g., "/proc/cpuinfo" and "/sys/devices/system/cpu/possible")
Performing system calls to read information about the process or the user, such as `getuid32`, `getppid`, and `gettid`
Performing system calls related to signal handling
Performing `cacheflush` or `set_tls` system calls or performing `nanosleep` system call
Reading files under "/proc/self/" or "/proc/<PID>/", where PID is the process' pid
Creating directories

94.2% (125,192) of the apps that used custom shared libraries executed only a set of common actions in native code, and Table 2.4 contains the common actions.

Table 2.5: Top five most common actions performed by apps in native code, through shared libraries (SL) and custom binaries (CB). For the interested reader, we report the full version of this table in [4].

SL	CB	Description
3,261	72	`ioctl` system call
1,929	39	Write file in the app's directory
1,814	35	Operations on sockets
1,594	5	Create network socket
1,242	144	Terminate process or thread group

The top five most common actions performed by apps in native methods, native activities, and custom binaries called through Exec are presented in Table 2.5. Table 2.6 presents the top five most common actions performed by the apps that used Exec to call standard (system) binaries.

By analyzing the system calls and the Java methods called from native code, we identified 3,669 apps that perform an action requiring Android permissions from native code. Table 2.7 presents the top five most popular permissions used, how many apps use them, and how we detected its use. We used PScout [12] to compute the permissions

Table 2.6: Top five most common actions performed by apps that called standard binaries in the system. For the interested reader, we report the full version of this table in [4].

Apps	Description
19,749	Read system information
3,384	Write file in the app's directory or in the sdcard
3,362	Read logcat
1,041	List running processes
861	Read system property

Table 2.7: The five most common (by number of apps) actions in native code that require Android permission. For the interested reader, we report the full version of this table in [4].

Apps	Permission	Description
1,818	INTERNET	Open network socket or call method `java.net.URL.openConnection`
1,211	WRITE_EXTERNAL_STORAGE	Write files to the sdcard
1,211	READ_EXTERNAL_STORAGE	Read files from the sdcard
132	READ_PHONE_STATE	Call methods `getSubscriberId`, `getDeviceSoftwareVersion`, `getSimSerialNumber` or `getDeviceId` from class `android.telephony.TelephonyManager` or Binder transaction to call `com.android.internal.telephony.IPhoneSubInfo.getDeviceId`
79	ACCESS_NETWORK_STATE	Call method `android.net.ConnectivityManager.getNetworkInfo`

required by each Java method. Comparing the permissions used in native code with the permissions requested by the app, we found that only 81 apps use, in native code, *all the permissions requested by the app.*

In addition to this being the first concrete look into how many apps use native code and what that native code does, we can draw two important conclusions: (1) if the native code is separated in a different process, it is necessary to give some permissions to the native code and (2) the permissions of the native code can be more strict (less permissive) than the permissions of the Java code.

It is interesting to note how conclusion (1) shows that the drastic measure adopted in NativeGuard [77], which does not grant any permissions to the native code, would break 3,669 of apps. This observation reinforces even more our belief that security policies should be generated following a data-driven approach. For instance, a reasonable tradeoff would be to allow to the native code only the `INTERNET`, `WRITE_EXTERNAL_STORAGE`, and `READ_EXTERNAL_STORAGE` permissions (the three most commonly used in native code), thus blocking only 152 applications.

Java—Native Code Interactions

To better understand the performance implications of separating the native code from the Java code of the apps, we measured the number of interactions per millisecond between Java and native code, i.e., the number of calls to JNI functions, calls to native methods, and Binder transactions.

The mean of interactions per millisecond is 0.00142, whereas the variance is 0.00003 and the maximum value is 0.22. NativeGuard's [77] performance evaluation with the Zlib benchmark shows a 34.36% runtime overhead for 9.81 interactions per millisecond and 26.64% for 3.96 interactions per millisecond. Therefore, our experiment shows that isolating native code in a different process should not have a substantial performance impact on average.

Additionally, we measure the number of bytes exchanged between the Java code and native code per second. The mean of bytes exchanged per second is 1,956.55 (1.91 KB/s) and the maximum value is 6,561,053.27 (6.26 MB/s). Only 11 apps exchanged more than 1 MB/s. We believe the amount of data exchanged between Java and native code would not incur a significant overhead, although it could vary greatly depending on the specific app.

Usage of the `su` Binary

Unlike common Linux distributions, in Android, users do not have access to a super user account and, therefore, are prevented from performing certain actions, such as uninstalling pre-installed apps. Thus, to have greater control over the system, many users perform a process known as "rooting," to be able to perform actions as the "root" user. Usually, during this process, a suid executable file called `su` is installed, as well as a manager app that restricts which apps can use this binary to perform actions as root. Because this process is so common among users, there are many apps that provide functionality that can only be performed by the root user, such as changing the fonts of the system or changing the DNS configuration.

Table 2.8: Top five most common types of command passed with the "-c" argument to `su`, separated between the apps that mention they need root privileges in their description or name and the ones that do not mention it. For the interested reader, we report the full version of this table in [4].

Does not Mention Root	Does Mention Root	Description
12	10	Custom executable (e.g., su -c sh /data/data/com.test.etd062.ct/files/occt.sh)
1	13	Reboot
2	12	Read system information
1	8	Change permission of file in app's directory
1	7	Remove file in app's directory

Our analysis identified 1,137 apps that try to run `su`. Surprisingly, 28.23% (321) of these apps *do not mention in their description or in their name that they need root privileges.*

Some of these apps use the "-c" argument of `su` to specify a command to be executed as root. Table 2.8 presents the top five most common types of actions that these apps tried to execute using `su`, along with the number of apps that attempt to execute that command, and if the app mentioned that it requires root or not. This table gives insights into what the app is trying to accomplish as root. The table shows that the most common action used with the "-c" argument of `su` is calling a custom executable. Because apps cannot use `su` in the emulator, these actions did not work properly during dynamic analysis, so we cannot obtain more information on their behavior.

JNI Calls Statistics

Understanding the JNI functions called by native code can reveal how the native components of apps interact with the app and the Android framework. Table 2.9 presents the types of JNI functions that were used by the apps and how many apps used them. The most relevant actions for security considerations in this table are: (1) calling Java methods and (2) modifying fields of objects. Calling methods in Java libraries from native code can be used to avoid detection by static analysis. Moreover, modifying fields of Java objects can change the execution of the Java code in ways that static analysis cannot foresee.

Calling Java methods, both from the Android framework and from the app can be performed by some of the methods presented in Table 2.2, more precisely the ones whose name starts with "Call." As Table 2.9 shows, we identified 35,231 apps that have native code which calls Java methods. More specifically, 24,386 apps used these functions to call Java methods from the app and 25,618 apps used them to call Java methods from the framework. Table 2.10 presents what groups of methods from the framework were called, along with the amount of apps that called methods in each group.

Binder Transactions

1.64% (2,457) of the apps that reached native code during dynamic analysis performed Binder transactions. Table 2.11 presents the top five most commonly invoked classes of the remote methods. The most common class remotely invoked by this process is `IServiceManager`, which can be used to list services, add a service, and get an object to a Binder interface. All apps that used this class obtained an object to a Binder interface and two apps also used it to list services. This data shows that using Binder transactions from native code is not common. From a security perspective this is good as the use of Binder transactions represent a way in which native code can perform critical actions while staying undetected by static analysis.

Table 2.9: Groups of JNI calls used from native code.

Apps	Description
94,543	Get class or method identifier and class reference
71,470	Get or destroy JavaVM, and Get JNIEnv
53,219	Manipulation of String objects
49,321	Register native method
45,773	Manipulate object reference
41,892	Thread manipulation
35,231	Call Java method
19,372	Manipulate arrays
18,601	Manipulate exceptions
14,330	Create object instance
6,918	Modify field of an object
2,203	Manipulate direct buffers
47	Memory allocation
37	Enter or exit monitor

Table 2.10: Top 10 groups of Java methods from the Android framework called from native code.

Apps	Description
7,423	Get path to the Android package associated with the context of the caller
6,896	Get class name
5,499	Manipulate data structures
4,082	Methods related to cryptography
3,817	Manipulate native types
3,769	Read system information
3,018	Audio related methods
2,070	Read app information
1,192	String manipulation and encoding
575	Input/output related methods
483	Reflection

Table 2.11: Top five most common classes of the methods invoked through Binder transactions. For the interested reader, we report the full version of this table in [4].

Apps	Class
2,427	android.os.IServiceManager
740	android.media.IAudioFlinger
725	android.media.IAudioPolicyService
327	android.gui.IGraphicBufferProducer
303	android.gui.SensorServer

Usage of External Libraries

Understanding the libraries used by the apps in native code can help us comprehend their purpose. Table 2.12 presents the top 10 most used system libraries and Table 2.13

presents the top 10 must used custom libraries by apps in native code. It demonstrates that apart from the bitmap manipulation library, which was used by 16.6% (24,942) of the apps that reached native code, no standard library was used by a great number of apps. On the other hand, several custom libraries were used by more than 7.5% of the apps that executed native code.

Table 2.12: Top 10 most used standard libraries.

Apps	Name	Description
24,942	libjnigraphics.so	Manipulate Java bitmap objects
2,646	libOpenSLES.so	Audio input and output
2,645	libwilhelm.so	Multimedia output and audio input
349	libpixelflinger.so	Graphics rendering
347	libGLES_android.so	Graphics rendering
183	libGLESv1_enc.so	Encoder for GLES 1.1 commands
183	gralloc.goldfish.so	Memory allocation for graphics
182	libOpenglSystemCommon.so	Common functions used by OpenGL
182	libGLESv2_enc.so	Encoder for GLES 2.0 commands
181	lib_renderControl_enc.so	Encoder for rendering control commands

Table 2.13: Top 10 most used custom libraries.

Apps	Name	Description
19,158	libopenal.so	Rendering audio
17,343	libCore.so	Used by Adobe AIR
16,450	libmain.so	Common name
13,556	libstlport_shared.so	C++ standard libraries
11,486	libcorona.so	Part of the Corona SDK, a development platform for mobile apps
11,480	libalmixer.so	Audio API of the Corona SDK
11,458	libmpg123.so	Audio library
11,090	libmono.so	Mono library, used to run .NET on Android
10,857	liblua.so	Lua interpreter
10,408	libjnlua5.1.so	Lua interpreter

2.1.6 Security Policy Generation

One step to limit the possible damage that native code can do is to isolate it from the Java code using the native code sandboxing mechanisms discussed in Section 2.1.2. These mechanisms prevent native code from modifying Java code, which allows static analysis of the Java part to produce more reliable results. However, this is not enough, considering that the app can still perform dangerous actions—that is, by interacting with the Android framework/libraries and by using system calls to execute root exploits.

Our goal here is to *reduce the attack surface* available to native code, by restricting the system calls and Java methods that native code can access. In particular, we propose to use our dynamic analysis system to generate *security policies.* A security policy represents the *normal* behavior, which can be seen as a sort of whitelist that represents the syscalls and Java methods that are *normally* executed from within native code components of benign applications. These policies also implicitly identify which syscalls and Java methods should be considered as *unusual* or *suspicious* (as they do not belong to the *common* syscalls), such as the ones used to mount root exploits.

One aspect to be considered is what action is taken when an unusual syscall is executed. Similar to the design choice adopted by SELinux, we envision two modes: permissive and enforcing. In permissive mode, the system would log and report the usage of unusual behavior, while in enforcing mode the system would block the execution of such unusual behavior and stop the application. Depending on the context, it might make sense to use permissive or the more aggressive enforcing mode. As an alternative, one could selectively pick permissive or enforcing mode depending on whether the unusual syscall is well-known to be used by root exploits. The policy generation process for syscalls is described in Section 2.1.6, while the one for Java methods is described in Section 2.1.6. We discuss the possibilities and the implications of this choice in Section 2.1.7.

It is worth noting that while this will not guarantee perfect protection from attacks, by applying the security principle of *least privilege* to the native code, we gain the dual security benefits of (1) increasing the precision of Java static analysis and (2) reducing the impact of malicious native code.

System Calls

Based on the system calls performed by the apps in native methods, in native activities, during libraries loading, and by programs executed by Exec methods, our system can automatically generate a security policy of allowed system calls. To compile this list, we first normalize the parameters of the system calls and later iterate over them, selecting the ones performed by most apps, until the list of selected system calls is comprehensive enough to allow at least a (variable threshold) percentage of the apps that executed native code to run properly. In Android, inter-process communication is done through Binder. Native code can directly use Binder transactions to call methods implemented by system services. At the system call level, these calls are performed by the `ioctl` system call. To consider these actions in our automatically generated whitelist, we substitute `ioctl` calls to Binder with the Binder transactions performed by the apps.

To understand the possible policies that could be generated, we performed this process using a threshold (the percentage of apps that use native code whose dynamically-executed behavior would function properly when enforcing this policy) of 99%. Tables 2.14 and 2.15 present the actions obtained by this procedure. The system call arguments that were normalized were replaced by symbols in the form <*> and * (meaning anything). Some of the arguments that are file descriptors were changed to a file path representation of it. All arguments that were not normalized represent a numeric value or a constant value that was converted by `strace` to a string representation. For the system calls that do not have the arguments next to it in the policies, the policy accepts calls with any arguments. Table 2.16 presents more details about the symbols used.

To better understand which types of apps would be blocked by our example policy (when in enforcing mode), we studied them and manually analyzed a subset of them. The findings of this analysis are presented in Section 2.1.7.

The policies restrict the possible actions of native code, thus following the principle of *least privilege* and making it harder for malicious apps to function. Previously, malicious code could easily hide in native code to evade static analysis. With our example policies enforced by a sandboxing mechanism, the native code does not (depending on the exact threshold) have the ability to perform any malicious actions in native code, and therefore attackers will have to move the malicious behavior to the Java code, where it can be found by existing Java static analysis tools. Furthermore, the policies do not prevent the correct execution of the dynamically-executed behavior of many benign apps. Using the rules generated with the 99% threshold, only 1,483 apps (0.12% of the total apps in our dataset) would be affected. Of course, as the dynamic analysis performed by our system is incomplete (in that it can not execute all possible app code), this number is a lower bound. This can be alleviated by an organization wishing to use our system in one of two ways: (1) increase the completeness of the dynamic analysis or (2) deploying the sandboxing enforcement mechanism in reporting mode. Both choices will reveal more app behaviors.

Another benefit of enforcing a native code sandboxing policy is that it would prevent the correct execution of several root exploits. For this work, we considered the 13 root exploits reported in Table 2.17. These exploits require native code to be successful. Our example security policy would hinder the execution of 10 of them. This follows because the policies attempt to reduce the attack surface of the OS for native code, while at the same time maintaining backward compatibility. Table 2.17 presents which of the considered exploits are successfully blocked, along with which entry of the policy they violate.

The root exploits that are prevented by our example security policy are blocked due to rules related to four system calls, namely `socket`, `perf_event_open`, `symlink`, and `ioctl`. More precisely, two exploits need to create sockets with `PF_NETLINK` domain and `NETLINK_KOBJECT_UEVENT` (15) protocol, however, the rules only allow `PF_NETLINK` sockets with protocol 0. One of the exploits needs the `perf_event_open` system call, which is not allowed by the policy. Two exploits need to create symbolic links that target system files or directories, but the policy only allows symbolic links to target "USER-PATH," which means files or directories in the app's directory or in the SD Card. Finally, five exploits use `ioctl` to communicate with a device. One of the rules allows `ioctl` calls to any device, namely `ioctl(<NON STD FD>,SNDCTL_TMR_TIMEBASE or TCGETS,*)`.

However, this rule specifies the valid request value (the second parameter), whereas the exploits use different values, therefore they would be blocked.

The table also reports the details about the three exploits that would not be currently blocked. In one case (CVE-2011-1149), the exploit would still work because our example policy allows the invocation of the `mprotect` syscall, since it is used by benign applications. In the two remaining cases (RATC and Zimperlinch), the exploits rely on repeatedly invoking the `fork` syscall to exhaust the number of available processes. The `fork` syscall is allowed by our policy as some benign applications do use it. However, note that this kind of exploit could be blocked by a security policy that would take into account the frequency of invocations of a given syscall: In fact, no benign application would ever invoke the `fork` syscall so frequently. We believe that considering this additional aspect of native code behavior is a very interesting direction for future work.

Although our example security policy does not block all exploits, we believe the adoption of native sandboxing to be useful. In fact, it does sensibly reduce the attack surface available to native code components, and it is able to successfully block a number of root exploits. Similarly, we believe that useful policies can be generated by our dynamic analysis system that will be able to block future exploits.

Java Methods

Even with the system call restrictions, native code can still perform dangerous actions by invoking Java methods. This can be accomplished by using certain JNI functions, as discussed in Section 2.1.3. Static analysis of the Java component of apps cannot identify these calls, therefore, the possibility of apps calling methods in Java libraries poses a threat to the system and can be abused by malicious apps.

We performed the same process presented in Section 2.1.6 to automatically generate policies that restrict the use of methods in Java libraries. Table 2.18 presents these policies, using different values as the minimum percentage of allowed apps that reached native code during dynamic analysis. We used 97%, 98%, and 99% as the values for the minimum. The methods authorized for each threshold include the ones associated with lower thresholds.

Using the list of apps associated with a minimum of allowed apps of 99% (the most permissive of our thresholds), we would block 1,414 apps (0.12%). The method `java.lang.ClassLoader.loadClass`, which is allowed when using 99% as a threshold, causes the invocation of the static initialization block (<`clinit`>) of a class. Therefore, it could be used to execute the static initialization block of classes in Java libraries. However, as far as we know, these blocks do not contain important operations that need to be contained.

2.1.7 Impact of Security Policies

Considering both our policies—Java methods and system calls—, and the 99% threshold, we would block 0.23% (2,730) of all the apps in our dataset. To understand what the impact of implementing (and enforcing with the strictest enforcement mechanism) these policies would be on users, we analyzed the popularity (lower number of installations) of

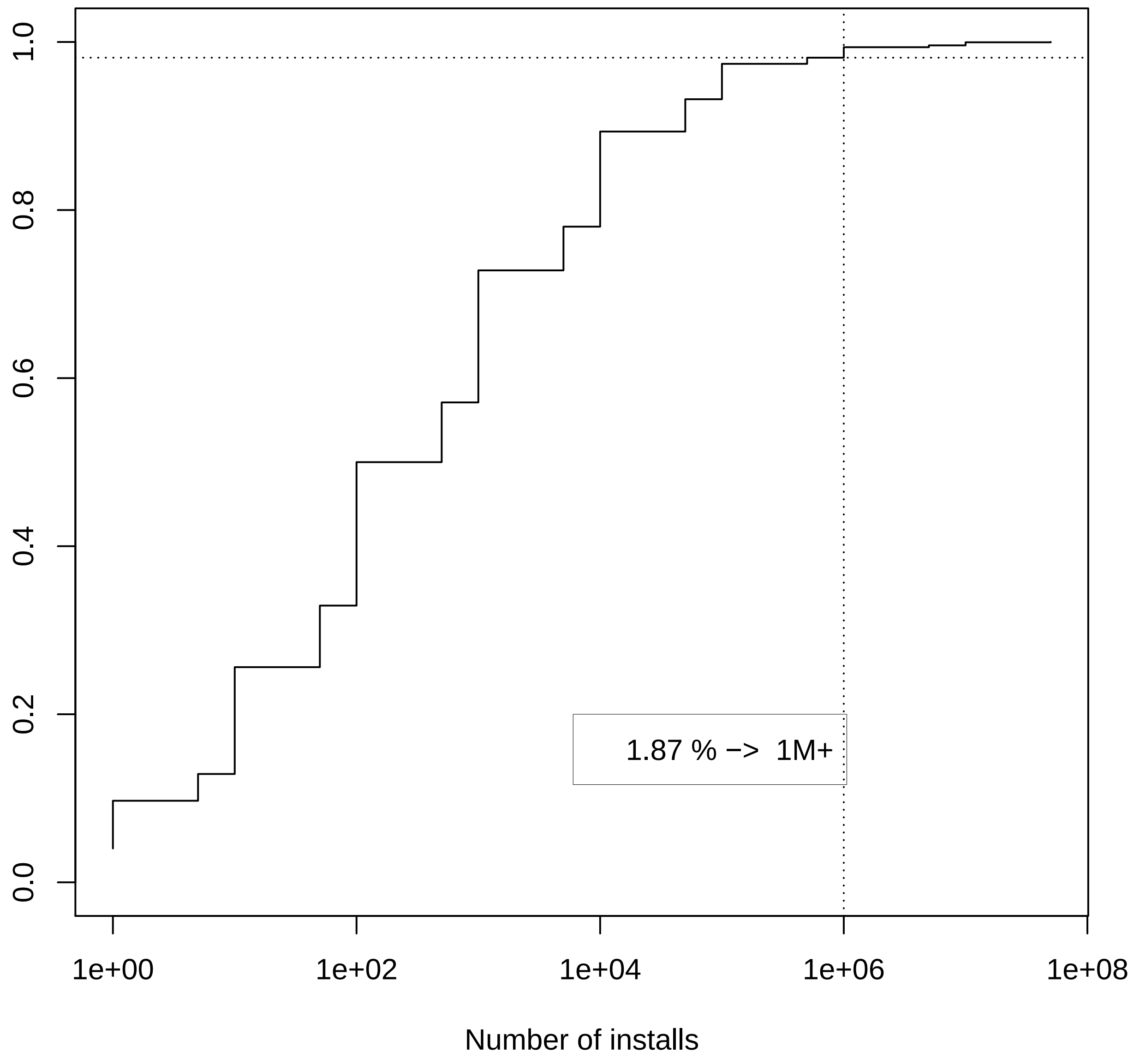


Figure 2.2: Popularity of apps that would be blocked by enforcing our policy. X-axis is in logarithmic scale, and the Y-axis is the percentage of apps that would be blocked.

the apps whose behavior seen during the dynamic analysis would be blocked. Figure 2.2 presents the cumulative distribution of the popularity of the apps that would be blocked. As the figure shows, among the applications for which our policy would block at least one behavior that has been executed at runtime, 1.87% (51) of them have more than 1 million installations.

Because manual analysis is very time-consuming, we did not perform it on all blocked apps. However, we did a general investigation of the blocked apps and manually analyzed the ones that showed traces of suspicious behavior. We identified three types of suspicious activities among these apps, and we discuss them here.

Ptrace. Overall, 280 apps used `ptrace`. 276 of these only call `ptrace` to trace itself without checking the result. We assume that the developers do this as a defensive measure to prevent the analysis of the app, because an app cannot be traced by another process if there is already a process tracing it. Therefore, for these 276 apps we believe that the app's functionality would remain intact with our policy. Four apps, on the other hand, create a child process, which try to attach `ptrace` to the parent, checking the result of the call and changing behavior if the call failed.

Modifying Java code. We identified 7 apps that modify the Java section of the app from native code. All these apps perform this action from the library `libAPKProtect.so` [8]. This library is provided by an obfuscation service, thus making it harder for reverse engineering tools to decompile the app. This functionality can also be used by malicious apps and illustrates the importance of isolating native code.

Fork and `inotify`. We identified 57 apps that create a child process in native code and use `inotify` to monitor the apps' directory, in order to identify when they are uninstalled. In fact, the spawned child process uses `inotify` to detect when the app is uninstalled and, when this happens, it opens a survey in the browser. This behavior is not a malicious action; however, executing code after being uninstalled is suspicious, as the user does not expect the app to be running after being uninstalled.

2.1.8 Dynamic Coverage

Dynamic analysis is inherently incomplete, and in this section we attempt to measure the code coverage of the dynamic analysis that we used, using function coverage of the Java code and function coverage of the native code. Both code coverage methods have large overhead, so we were only able to analyze a subset of the apps.

Java Method Code Coverage

To measure the code coverage based on the Java methods executed, we instrumented the DVM. The instrumented code records the execution of every method of the app under analysis. Since this instrumentation introduces more overhead and slows the emulator, we did the experiment with 25,000 apps randomly selected and used a kernel driver, instead of `strace`, to record the system calls executed. The code coverage obtained was 8.31%

Native Code Coverage

While code coverage of the Java methods allows us to gain insight into the high level code coverage of our dynamic analysis system, it does not shed light on the core issue we are interested in: *how much of an app's native code is the dynamic analysis able to execute*? To answer this question, we modified both the Android emulator and the Android framework to support measuring function coverage of the native code.

One technical challenge here is that the native code coverage must understand not only which native libraries are loaded by an app, but also which part of the native library is actually executed. Thus we need to: (1) trace the executed native functions and (2) statically determine the total number of native functions. This will allow us to calculate the function coverage of the native code.

To the best of our knowledge, there is no previously released tool to trace the execution of the native code of an app. Android Open Source Project implements a tracing mechanism since version 4.4. This tracing mechanism is implemented using a kernel device called qemutrace that is part of the goldfish kernel. The kernel send information to assist the emulator to trace correctly the execution, e.g., the PID of the running process each time there is a context switch, a message that notifies that a fork or an execve is executed, etc. The whole tracing system significantly slows down the performance of the emulator. However, this tracing system is too general: we are interested only in the execution of the native code of a specific app. We need to trace only functions of loaded libraries of the app under analysis.

For this reason, we created two ways to limit the tracing to the interesting part only. First, we only want to trace processes with a specific UID because each app in Android is executed with it own UID. In addition, we are interested only in portions of the executable memory where the native libraries have been loaded.

To inform the emulator about the UID of the currently executing process we leverage the existing qemutrace device. We added the UID into the message sent for each context switch. To send the information about the map of the memory to the emulator we cannot use the qemutrace device, since it can only pass 32 bit integers as messages. Moreover, we also need a mechanism to extract the libraries from the emulated system. To solve both problems we instrumented the Android framework. We found that the function `java.lang.Runtime.doLoad` is able to intercept all the loading operations. Our hook inside the `doLoad` function blocks the loading (and the app) while syncing all the gathered data to the external emulator. The mapping of the memory and the PID are read from `/proc/self/`. The path of the loaded library is one of the parameters of the `doLoad` function. Hence, when `doLoad` returns, the emulator knows the address space reserved for the new library, and the content of the native library.

After the dynamic execution, we compute the code coverage using all the data gathered during the execution. We use IDA Pro to find all functions boundaries of libraries. Then, we use the map of the memory to translate the virtual addresses traced by the emulator. Next, we flag all the functions whose boundaries include at least one address of the trace. The code coverage is then calculated.

Our tracing system slows down the execution of the apps by around 10 times. Therefore, we only ran it on a small subset of the apps, more specifically, we analyzed 177. The code coverage of most libraries is less that 1%. Some small libraries, on the other hand, were covered by 100%. Furthermore, the average coverage was 7%. More details about executed libraries and coverage can be seen in Figure 2.3.

2.1.9 Threats to Validity

Our study is affected by a few limitations, which we discuss in this section. An intrinsic limitation of the automatically-generated security policies is that we base their automatic generation on data and insights obtained by means of dynamic analysis, which is well-known to be incomplete and affected by code coverage issues. In fact, dynamic analysis does not ensure that all native code is exercised in the apps that actually use it, and for those apps that used native code, dynamic analysis may not have exercised all code paths in the native code. Consequently, the policies that our tool generated might not

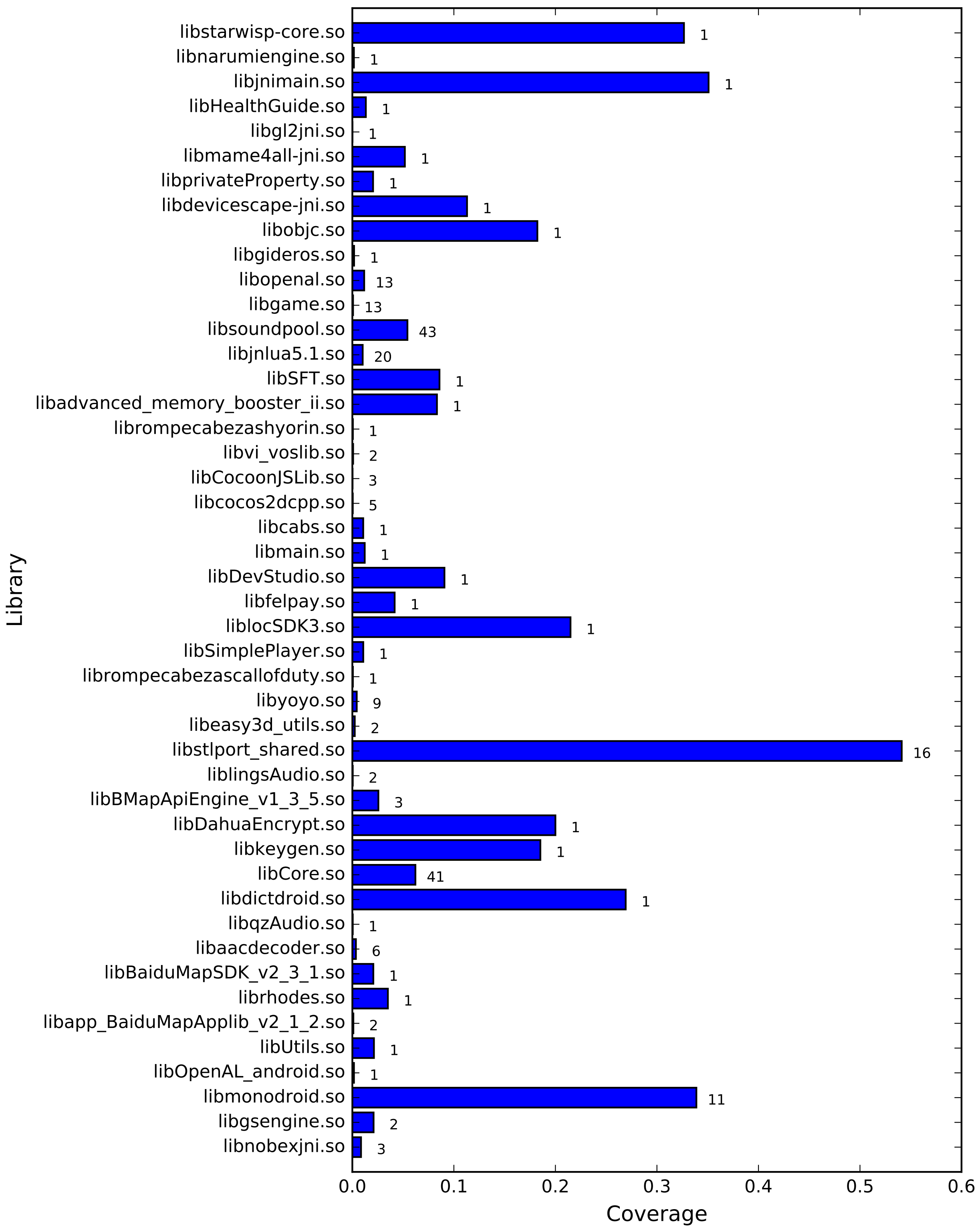


Figure 2.3: Per library coverage of executed functions. Horizontal axis contains libraries name, vertical, instead contains the function coverage. For each bar we also show the number of libraries that has been found in all executed applications

be complete, they might block more applications when adopted at large-scale, and the performance overhead of isolating native code could be higher. However, using a more-sophisticated instrumentation tool could possibly improve the amount of native code behavior that our system observes, or deploying the automatically generated policies in a native sandbox with reporting mode would help to observe the behaviors that the policies would block.

Nonetheless, we believe this work to be a significant first step in a very important direction. In fact, to the best of our knowledge, this work is the first, largest, and most comprehensive study on how real-world applications use native code. Our results demonstrate that it is infeasible to adopt a completely restrictive sandboxing policy. In addition, we propose a system to automatically generate a native code sandboxing policy following a data-driven approach. This system could be used by large organizations that are interested in automatically generating a native code sandboxing policy. Furthermore, the completeness issues could possibly be addressed by increasing the fidelity of the dynamic analysis, either through more sophisticated analysis techniques or increased resources, or by obtaining the actual behavior of native code in the wild, by instrumenting real-world Android devices.

Another limitation is that our approach restricts access to permissions from native code, but it still allows the native code to invoke (some) Java methods. This aspect would make, in principle, Java-only analysis more precise, but still not completely sound, as a malicious application could introduce *hidden* execution paths by invoking a native method, which, in turn, could invoke a Java method. However, we note that our automatically-generated policy only allows native code to invoke a very narrow subset of Java methods defined in the Android framework (Table 2.18), through which it is virtually impossible to perform any security-sensitive operation. Thus, our policy, although not perfect, would drastically reduce the possibility of introducing malicious behaviors.

Lastly, we consider all the apps we obtained from Google Play as benign, but we cannot be completely certain that there are no malicious apps among them. The effects of having malicious apps in our dataset vary depending on how the malware works. In the worst case it could cause our policies to allow some malicious actions.

2.1.10 Related Work

In this section we relate our work to the vast amount of research published in the field of Android security.

Large Measurement Studies. Several works have analyzed large datasets of Android apps, but with goals that differ from ours. Viennot et al. [81] did a large measurement study on 1,100,000 applications crawled from the Google Play app store. In particular, they collected meta-data and statistics taken from the Google Play store itself. As part of their study, they measured the frequency with which Android applications make use of native code components. Another important measurement study has been performed by Lindorfer et al. [53]. In their work, they analyzed 1,000,000 apps, of which 40% are malware. To perform the analysis, the authors used Andrubis, a publicly-available analysis system for Android apps that combines static and dynamic analysis. When focusing on native code, our work significantly extends their study.

Application Analysis Systems. Several systems have been proposed to perform be-

havioral analysis of Android applications based on dynamic analysis [26, 28, 64, 66, 69, 91]. Moreover, several other works have been proposed to identify malicious Android apps [9, 19, 40]. Our analysis complements all these research efforts by performing a large scale study, based on dynamic analysis, specifically focused on native code usage.
Protection Systems. Fedler et al. [29] proposes a protection system from root exploits by preventing apps from giving execution permission for custom executable files and by introducing a permission related to the use of the `System` class. PREC [42] is a framework intended to protect Android systems from root exploits. PREC uses two steps, learning and enforcement. During the learning phase, the analysis generates a model of the normal behavior for a given app. Then, during the enforcement phase, the system makes sure that the app does not deviate from the *normal* behavior. Our work has the advantage that the generated policies can be applied to all apps, whereas PREC generates per-app models. Hence, our results are more general. Moreover, our analysis also monitors, in addition to system calls, JNI function calls, Binder transactions and calls from Java to native methods.
Native Code Isolation. Another way to protect the system is by isolating native code. The challenge of isolating native code components used by managed languages has been extensively studied. For instance, Klinkoff et al. [49] focus on the isolation of *.NET* applications, whereas Robusta [72] focuses on the isolation of native code used by Java applications. Recently, NativeGuard [77] proposed a similar mechanism to isolate native code in the context of Android. Our work is complementary to these sandboxing mechanisms and fills the knowledge gap necessary to define security policies on the execution of native code in Android that are both usable in real-world applications and effective in blocking malicious behavior of native components.

2.1.11 Conclusion

While allowing developers to mix Java code and native code enables developers to fully harness the computing power of mobile devices, we believe that, in the current state, this feature does more harm than good and that native code sandboxing is the correct approach to properly limit its potentially malicious side-effects. However, a native code sandboxing mechanism without a proper policy will never be feasible. We hope that, in addition to shedding light on the previously unknown native code usage of Android apps, this paper demonstrates an approach to automatically generate an effective and practical native code sandboxing policy.

Acknowledgment

This material is based upon work supported by CAPES Foundation under Award No. BEX 12269/13-1, by NSF under Award No. CNS-1408632, by DHS under Award No. 2009-ST-061-CI0001, and by Secure Business Austria. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of CAPES Foundation, NSF, DHS, or Secure Business Austria.

This material is also based on research sponsored by DARPA under agreement number FA8750-12-2-0101. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

2.2 Identifying Android malware using dynamically obtained features

Publication: This paper was published in the Journal of Computer Virology and Hacking Techniques, volume 11, 2015

Vitor Monte Afonso[1], Matheus Favero de Amorim[1], André Ricardo Abed Grégio[1], Paulo Lício de Geus[1], Glauco Barroso Junquera[2]
(1) University of Campinas
Email: {vitor,matheus,gregio,paulo}@lasca.ic.unicamp.br
(2) Samsung Institute for Informatics Development (SIDI)
Email: glauco.b@samsung.com

Table 2.14: Allowed system calls automatically generated using a threshold of 99% apps unaffected by the policy (part 1).

accept(*,*,*)	access(<SYS-PATH>, F_OK)	access(<SYS-PATH>,R_OK)
access(<SYS-PATH>, W_OK)	access(<SYS-PATH>,X_OK)	access(<USER-PATH>, F_OK)
access(<USER-PATH>,R_OK)	access(<USER-PATH>, R_OK|W_OK|X_OK)	bind
BINDER(android.os.IServiceManager. CHECK_SERVICE_ TRANSACTION)	brk	cacheflush(*,*,0,*,*)
cacheflush(*,*,0,0,*)	chdir	chmod(<USER-PATH>,*)
clone(child_stack=*,flags=CLONE_VM|CLONE_FS| CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM)		
connect(*, {sa_family=AF_UNIX, path=@"jdwp-control"},*)	connect(*, {sa_family= AF_INET,*,*},*)	connect(*,{sa_family= AF_UNIX, path= @"android:debuggerd"},*)
connect(*, {sa_family=AF_UNIX, path=<SYS-PATH>},*)	dup	dup2
epoll_create(*)	epoll_ctl(*,*,*,*)	epoll_wait
execve	exit(<NEG INT>)	exit(0)
exit_group(<POS INT>)	exit_group(0)	fcntl64(<NON STD FD>, F_DUPFD,*)
fcntl64(<NON STD FD> ,F_GETFD)	fcntl64(*,F_GETFL)	fcntl64(<NON STD FD>, F_SETFD,*)
fcntl64(<NON STD FD> ,F_SETFL,*)	fcntl64(<NON STD FD>, F_SETLK,*)	fdatasync(*)
fork	fstat64	fsync(*)
ftruncate(*,*)	futex	getcwd
getegid32	geteuid32	getgid32
getpeername	getpgid(0)	getpid
getppid	getpriority(PRIO_PROCESS,*)	getrlimit(RLIMIT_DATA,*)
getrlimit(RLIMIT_NOFILE,*)	getrlimit(RLIMIT_STACK,*)	getrusage(RUSAGE_CHILDREN,*)
getrusage(RUSAGE_SELF,*)	getsockname	getsockopt(*, SOL_SOCKET, SO_ERROR,*,*)
getsockopt(*,SOL_SOCKET, SO_PEERCRED,*,*)	getsockopt(*, SOL_SOCKET, SO_RCVBUF,*,*)	gettid
getuid32	ioctl(<ASHMEM-DEV>,*,*)	ioctl(*,FIONBIO,*)
ioctl(<LOG-DEV>,*,*)	ioctl(*,SIOCGIFADDR,*)	ioctl(*, SIOCGIFBRDADDR,*)
ioctl(*,SIOCGIFCONF,*)	ioctl(*,SIOCGIFFLAGS,*)	ioctl(*, SIOCGIFHWADDR,*)

Table 2.15: Allowed system calls automatically generated using a threshold of 99% apps unaffected by the policy (part 2).

ioctl(*,SIOCGIFINDEX,*)	ioctl(*,SIOCGIFNETMASK,*)	ioctl(<STD IN/OUT/ERR>, SNDCTL_TMR_TIMEBASE or TCGETS, *)
ioctl(*, SNDCTL_TMR_TIMEBASE or TCGETS,*)	ioctl(<URANDOM-DEV>, SNDCTL_TMR_TIMEBASE or TCGETS,*)	listen
lseek(*,*,SEEK_CUR)	lseek(*,*,SEEK_END)	lseek(*,*,SEEK_SET)
lstat64	madvise(*,*, MADV_DONTNEED)	madvise(*,*, MADV_NORMAL)
madvise(*,*, MADV_RANDOM)	mkdir(<SYS-PATH>,*)	mkdir(<USER-PATH>,*)
mmap2	mprotect	mremap(*,*,*, MREMAP_MAYMOVE)
munmap	nanosleep	open(<SYS-PATH>,*,*)
open(<SYS-PATH>,*)	open(<USER-PATH>,*,*)	open(<USER-PATH>,*)
pipe	poll	prctl(PR_GET_NAME, ,0,0,0)
prctl(PR_SET_NAME, *,*,*,*)	prctl(PR_SET_NAME,*,*,*,0)	prctl(PR_SET_NAME, ,0,0,0)
ptrace(PTRACE_TRACEME, *,0,0)	readlink(<USER-PATH>,*,*)	recvfrom
recvmsg	rename(<USER-PATH>, <USER-PATH>)	rmdir(<USER-PATH>)
rt_sigprocmask(SIG_BLOCK,*,*,*)	rt_sigprocmask(SIG_SETMASK,*,*,*)	rt_sigreturn(*)
rt_sigtimedwait([QUITUSR1], NULL, NULL, 8)	sched_getparam	sched_getscheduler
sched_yield	select	sendmsg
sendto	setitimer(ITIMER_REAL,*,*)	setpriority(PRIO_PROCESS ,*,<POS INT>)
setpriority(PRIO_PROCESS,*,0)	setrlimit(RLIMIT_NOFILE,*)	setsockopt(*,SOL_IP,*,*,*)
setsockopt(*, SOL_SOCKET,*,*,*)	set_tls(*,*,*,*,*)	set_tls(*,*,0,*,*)
sigaction	sigprocmask(SIG_BLOCK,*,*)	sigprocmask(SIG_SETMASK,*,*)
sigprocmask(SIG_UNBLOCK,*,*)	sigreturn	sigsuspend([])
socket(PF_INET, SOCK_DGRAM, IPPROTO_ICMP)	socket(PF_INET, SOCK_DGRAM, IPPROTO_IP)	socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) SOCK_RAW, 0)	socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)	socket(PF_NETLINK,
socket(PF_UNIX, SOCK_STREAM, 0)	stat64	statfs64(<SYS-PATH>,*)
statfs64(<USER-PATH>,*)	symlink(<USER-PATH>, <USER-PATH>)	tgkill(*,*,SIGTRAP)
umask	uname	unlink(<USER-PATH>)
utimes	vfork	wait4

Table 2.16: Symbols used to replace the arguments of system calls.

<USER-PATH>	A file path in the apps' directory or in the sdcard
<SYS-PATH>	A file path different than the ones represented by <USER-PATH>
<URANDOM-DEV>	"/dev/random" or "/dev/urandom"
<ASHMEM-DEV>	"/dev/ashmem"
<LOG-DEV>	"/dev/log/system", "/dev/log/main", "/dev/log/events" or "/dev/log/radio"
<NEG INT>	A negative number
<STD IN/OUT/ERR>	A file descriptor equal 0, 1, or 2
<NON STD FD>	A file descriptor different than 0, 1, or 2
<POS INT>	An integer greater than 0

Table 2.17: This table shows the list of considered root exploits, on which syscall-level behavior they rely, and which exploits are successfully blocked by our policy.

Name / CVE	Description	Blocked
Exploid (CVE-2009-1185)	Needs a `NETLINK` socket with `NETLINK_KOBJECT_UEVENT` protocol	Yes
GingerBreak (CVE-2011-1823)	Needs a `NETLINK` socket with `NETLINK_KOBJECT_UEVENT` protocol	Yes
CVE-2013-2094	Uses `perf_event_open` system call	Yes
Vold/ASEC [73]	Creates symbolic link to a system directory	Yes
RATC (CVE-2010-EASY)	Relies on invoking many times the `fork` syscall	No
CVE-2013-6124	Creates symbolic links to system files	Yes
CVE-2011-1350	`ioctl` call used violates our rules	Yes
Zimperlinch	Relies on invoking many times the `fork` syscall	No
CVE-2011-1352	`ioctl` call used violates our rules	Yes
CVE-2011-1149	It relies on the `mprotect` syscall	No
CVE-2012-4220	`ioctl` call used violates our rules	Yes
CVE-2012-4221	`ioctl` call used violates our rules	Yes
CVE-2012-4222	`ioctl` call used violates our rules	Yes

Table 2.18: List of allowed methods (Java methods called from native code) automatically generated for allowing a minimum of 97%, 98% and 99% of apps that reached native code.

Allowed apps (%)	Method
97	java.lang.Integer.doubleValue
97	android.content.ContextWrapper.getPackageName
97	java.lang.String.getBytes
98	java.lang.Double.doubleValue
98	android.content.ContextWrapper.getClassLoader
98	android.content.ContextWrapper.getFilesDir
98	java.io.File.getPath
98	android.content.ContextWrapper.getExternalFilesDir
98	android.view.WindowManagerImpl.getDefaultDisplay
98	java.lang.String.toLowerCase
98	android.app.Activity.getWindowManager
98	java.util.ArrayList.add
98	android.view.Display.getMetrics
98	android.app.Activity.getWindow
98	android.view.View.getWindowVisibleDisplayFrame
98	java.util.Calendar.getInstance
98	android.view.View.getDrawingRect
99	java.util.Calendar.get
99	android.os.Bundle.getByteArray
99	android.content.ContextWrapper.getPackageManager
99	android.content.res.AssetManager$AssetInputStream.read
99	java.lang.Long.doubleValue
99	java.lang.ClassLoader.loadClass
99	android.app.ApplicationPackageManager.getPackageInfo
99	android.content.res.AssetManager$AssetInputStream.close
99	java.lang.Float.doubleValue
99	java.lang.Class.getClassLoader

Abstract

The constant evolution of mobile devices' resources and features turned ordinary phones into powerful and portable computers, leading their users to perform payments, store sensitive information and even to access other accounts on remote machines. This scenario has contributed to the rapid rise of new malware samples targeting mobile platforms. Given that Android is the most widespread mobile operating system and that it provides more options regarding application markets (official and alternative stores), it has been the main target for mobile malware. As such, markets that publish Android applications have been used as a point of infection for many users, who unknowingly download some popular applications that are in fact disguised malware. Hence, there is an urge for techniques to analyze and identify malicious applications before they are published and able to harm users. In this article, we present a system to dynamically identify whether an Android application is malicious or not, based on machine learning and features extracted from Android API calls and system call traces. We evaluated our system with 7,520 apps, 3,780 for training and 3,740 for testing, and obtained a detection rate of 96.66%.

2.2.1 Introduction

Mobile devices have been ubiquitously widespread as personal and professional tools whose computing power is approaching that of ordinary desktop computers. Consequently, smartphone users are able to do more complex tasks with their devices, such as producing documents and spreadsheets, making video conferences and managing their Internet Banking accounts. These users are now storing all sorts of sensitive information on their devices (e.g., bank credentials, corporate documents), effectively creating an interesting and potentially lucrative scenario for cybercriminals. To take advantage of this situation, attackers are ramping up the creation of malicious applications that affect mobile devices.

Since Android is the most widespread operating system for mobile devices [34], it is the main target of mobile malware. According to Juniper [46], the amount of malicious applications discovered between March 2012 and March 2013 has increased 614%, considering all mobile platforms. In addition, the same report states that 92% of every malware that affects mobile devices targets the Android operating system. Users obtain Android applications mostly from markets, including Google Play—Google's official market—and others known as "alternatives". In order to infect users' devices, attackers submit to markets malware that look like legitimate applications, such as games. In fact, many of the available malware are repackaged versions of legitimate applications, i.e., applications modified to include malicious code and republished in the markets.

Some works in the literature refer to the presence of malicious applications both in the official market and in alternative ones [40, 98]. They show that the official market does a better job at filtering out malicious applications, but nonetheless is still used as a vector to infect users. Addressing this issue requires the development and deployment of improved techniques to analyze and identify malicious Android applications.

To that effect, several approaches based on static and dynamic analysis have been proposed to detect malicious Android applications [27, 70, 76, 96, 98], but all of them present shortcomings regarding their detection scope or ability. Firstly, approaches that rely on static analysis of the application's code have a hard time dealing with highly obfuscated samples [59] and only analyze code packed with the application file, missing code that can be downloaded and executed at runtime [63]. Secondly, although Android malware samples do not make use of obfuscation techniques as heavy as those affecting Windows desktops, the natural evolution of Android malware will inevitably lead to the improvement of obfuscation techniques currently used [74], turning static analysis into a difficult proposition. Moreover, dynamic analysis approaches usually suffer from not being able to observe the malicious behavior of some samples due to their ever growing awareness of the analysis environment, to the lack of appropriate stimulation under the analysis environment or else to the inability of the malware sample under analysis to obtain some required external data.

In general, detection techniques for Android malware use statically extracted data from the manifest file or from Android API function calls, as well as dynamically obtained information from network traffic and system call tracing. However, most articles available in the literature whose focus lies on malware identification either use small datasets or require manual steps at some stage of the process. In this paper, we present a system that identifies malicious Android applications based on a machine learning classifier, using dynamically obtained features. These features are extracted from Android API function calls and system call traces. We trained our classifier with 3,780 samples and tested it with 3,740 samples (with both datasets including malicious and benign applications), which was then able to correctly classify 96.66% of those samples. The results obtained were compared to the ones from other Android malware detection approaches and demonstrate the relevance of our system. Using a larger dataset, we obtained results similar to the state-of-the-art, including static and dynamic approaches. Furthermore, we show that features extracted from API function calls, which, as far as we know, were not used by other automatic and dynamic approaches, are very good for the identification of malware.

The main contributions of this paper are:

- We developed an analysis system to monitor Android API function calls as well as system calls, in order to gather information (features) required to detect malicious behavior. Currently available systems are tied to Android OS versions (some of them to older versions, such as 2.x) or to the SDK-provided emulator, whereas our approach is independent of the emulator and much more portable as it does not modify Android OS;
- From that, we developed a system that classifies applications as benign or malicious and tested it with thousands of apps, correctly classifying 96.66% of them. To accomplish better training and accuracy, we extract novel features showing that those based on API function calls greatly increase the detection rate.

The remainder of this paper is organized as follows. Section 2.2.2 provides a background about Android malware and presents related work. The developed system is introduced in Section 2.3.5, whereas evaluation results and discussion are presented in Section 2.2.4. Section 2.2.5 discusses some of the limitations and, in Section 2.2.6, we conclude this paper and discuss some follow-up work.

2.2.2 Background and Related Work

Based on reports from antivirus companies, the authors of [31] describe the behavior of 46 malware samples collected between January 2009 and June 2011. The malicious behaviors identified were the following: user information stealing; premium calls and SMS messages[4]; SPAM SMS messages; novelty and amusement[5]; user credential stealing; search engine optimization; and ransom.

A similar study is presented in [97], but in this case the authors analyzed the samples manually. They used a dataset of 1,260 Android malware samples, which were collected between August 2010 and October 2011 and were separated in 49 malware families. The authors describe the behavior of these samples and show information regarding their time of discovery in the official market and in alternative ones. For each family, they specify how the malware is installed, how the malicious behavior is activated and what

[4]These actions generate costs to the user.

[5]Some samples performed actions that seemed to be only useful for the amusement of the author.

the malicious payload is. Also, the authors indicate the events monitored by the malware and the exploits used by them for privilege escalation.

Android malware detection is a critical task towards protecting users of application markets and improving these markets' vetting processes. However, detection is intimately bound to analysis, since features must be extracted so as to generate signatures or behavioral profiles. There are systems proposed in the literature that aim to analyze apps from markets in order to detect the presence of malware among them, as well as others solely with the purpose of providing useful information about static and dynamic characteristics of an unknown application. We discuss some of these systems in the following sections.

Android Malware Analysis

Enck et al. [28] propose TaintDroid, a dynamic taint analysis system, which tracks sensitive data flow to detect when it is sent over the network. TaintDroid instruments the Android virtual machine interpreter and some APIs to accomplish system-wide taint tracking, but it does not handle native code. Their results show that seemingly unsupicious applications often disclose sensitive data, such as location, UUID and phone number. Although based on permissions granted by users, the data exposure monitoring process requires the application to be dynamically analyzed.

DroidBox [26] is a dynamic analysis system that builds upon TaintDroid and provides API calls, network data and data leaks, besides other important information.

Andrubis [53], which has a publicly available submission interface, is a system whose goal is to analyze Android applications using static and dynamic techniques. In the static analysis step, Andrubis collects information about required permissions, components to communicate with the operating system, intent-filters and URLs found in the bytecode. Dynamically-based information collection is accomplished through instrumentation of the Dalvik VM, taint tracking and network traffic capture. Andrubis is based on TaintDroid, DroidBox and other related projects. Yan and Yin propose DroidScope [91], a virtual machine introspection-based analysis system that bridges the semantic gap reconstructing OS-level and Java-level semantic views from outside. They also developed additional analysis tools to provide taint tracking and several levels of instruction tracing.

Spreitzenbarth et al. present Mobile-Sandbox, a system that combines static and dynamic analysis techniques to obtain Android applications' behavior [75]. Mobile-Sandbox's static analysis includes parsing the manifest file and the extracted bytecode, and aims to guide the dynamic analysis process, which is based on TaintDroid and DroidBox. In addition, Mobile-Sandbox monitors native code using the `ltrace` tool and analyzes network traffic captured during the application's execution. Another system that uses both static and dynamic analysis is AASandbox [18]. During static analysis, the system decompiles the application to Java code and look for suspicious patterns, such as the use of `Runtime.exec()` and functions related to reflection. During the dynamic step, AASandbox runs the application on a controlled environment and monitors system calls using a kernel module.

Android Malware Detection

Zhou et al. propose DroidRanger, a two-scheme system based on signatures and heuristics that intends to detect Android malware [98]. On the one hand, the signature-based scheme relies on common permissions and behavioral footprints to identify samples from known families. On the other hand, the heuristics-based filtering scheme identifies suspicious behaviors (e.g., downloading and executing code from Web and dynamic loading of native code). Applications identified as suspicious are manually analyzed and if they are indeed malicious, the information necessary to detect samples from the same family in the signature-based step are manually extracted.

Zheng et al. propose DroidAnalytics, a system to automatically collect, analyze and detect Android malware that makes use of repackaging, code obfuscation or dynamic payloads [96]. Collection is accomplished by an extensible application crawler that receives marketplaces (official and alternatives) or URLs as input. Collected applications are then disassembled so as to obtain Android API calls. These API calls are used within a three-level signature generation process, which extracts malware features at the opcode level to identify variants. The dynamic analysis step consists of running samples that present network behavior, inside an emulator, in order to download additional pieces of code.

Sanz et al. introduce PUMA, an Android malware detection method based on machine learning that uses information obtained from application's permissions [70]. To evaluate their method, they collected 1,811 supposedly benign applications of several categories from Android Market and 249 unique malicious samples from the VirusTotal database. The features used to represent each sample are based on the set of permissions and the device's features required by the application. Using this information, the authors evaluated eight algorithms available in the WEKA framework and concluded that RandomForest provided the best results.

Elish et al. propose a tool to determine whether unknown applications are malicious or not based on static data dependence analysis [27], aiming to identify software execution patterns related to the correlation of user inputs with critical function calls. They construct a data dependence graph for each analyzed application, which can then be used in comparisons to identify stealthy Android malware. Although their results show that the analyzed malware samples are distinguishable from the legitimate applications, since the former performed sensitive function calls without any user input.

Wu et al. propose DroidMat, a detection system based on clustering techniques applied to statically extracted features from the application's manifest file (permission, component and intent information) and permission-related Android API call traces from the application's bytecode [88]. The process for evaluating the system applied four combinations of clustering and classification algorithms to analyze a dataset of 1,500 benign applications (downloaded from GooglePlay) and 238 malicious ones, and resulted in 97.87% accuracy.

Another system that performs Android malware detection using features obtained statically is DREBIN [9]. This system uses machine learning and features extracted from the manifest and the dex code of applications. The authors performed experiments with 123,453 benign samples and 5,560 malicious samples and the system obtained 93% of accuracy.

Su et al. present a smartphone dual defense protection framework to perform detection of malicious applications, using machine learning, as they are submitted for release on Android markets [76]. Their approach consists of dynamically analyzing a new application to collect two sets of features: one related to system call tracing and the other related to network traffic statistics. A system call monitoring process makes use of the Linux's `strace` tool and restricts itself to 15 (of almost 300) of them that are related to process, memory and I/O activities. The `tcpdump` tool is used to capture network traffic, from which TCP/IP flows are extracted. The training of the system calls classifier involved 200 benign and 180 malicious applications, whereas the training of the network classifier involved 60 benign and 49 malicious applications. Both classifiers are based on WEKA's implementation of J.48 and RandomForest algorithms. The authors selected 70 benign and 50 malicious applications to evaluate their classifiers and obtained an accuracy rate of 94.2% and 99.2% for J.48 and RandomForest, respectively.

2.2.3 System Overview

Figure 2.4 presents the system overview. To identify malicious applications, the developed system obtains information about the application's behavior using dynamic analysis. This process is explained in Section 2.2.3. The obtained information is comprised by Android API function calls and system calls, and is fed to a processor, which extracts features from the information. These features are composed by the frequency of use of API functions and system calls, and are used by a classifier to categorize the application as malicious or benign. The feature extraction and classification processes are explained in Section 2.2.3.

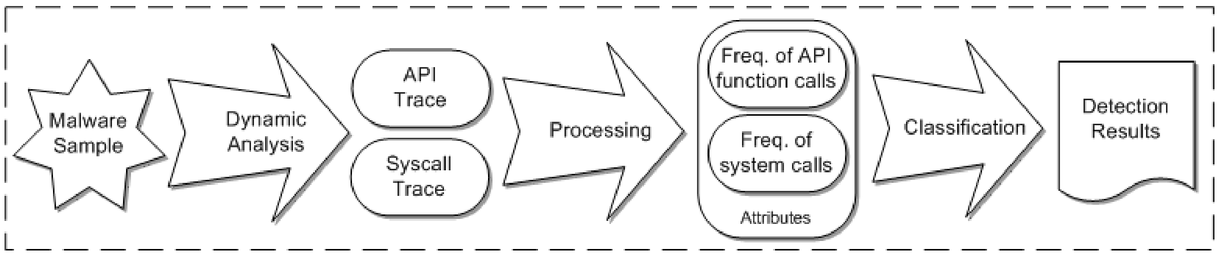


Figure 2.4: System overview

Data Extraction

To obtain its behavior, the application is first instrumented by APIMonitor[6], a tool that modifies the application so that calls to certain functions are registered, along with the parameters passed and the return value. We modified the `default_api_collection`[7] file, used by APIMonitor, to include methods related to network access, process execution, string manipulation, file manipulation and information reading. The instrumented version of the application is executed for five minutes in the standard Android emulator—distributed with the Android SDK.

[6]`https://code.google.com/p/droidbox/wiki/APIMonitor`.
[7]This file defines the functions that are monitored.

The analysis of Android API function calls is important because it allows the extraction of high-level information about the behavior of applications. However, some applications use native code instead of Android API functions. Thus, through the `strace` tool, we also monitor the system calls executed by the application. Section 2.2.3 presents examples of registered API function calls and system calls.

The advantages of our monitoring process are not needing to modify the Android code and also being independent of the virtualization platform. Analysis systems that use a modified version of Android, such as TaintDroid [28], Andrubis [53] and Mobile-Sandbox [75], need to be constantly updated to the newest version of the Android system, a task that is quite time-consuming, so that they are able to analyze samples that target that particular version of the operating system. Moreover, systems that use virtual machine introspection, such as Droidscope [91], are dependent on the virtualization platform (e.g., Qemu) and cannot be used on a different virtualization platform or on a bare-metal one.

On the negative side, the disadvantages of our monitoring system are the use of a monitoring tool inside the analysis environment and the modification of the analyzed sample. These actions make the system more detectable by malware, which can stop the execution or execute benign actions when it becomes aware of the analysis. The previously mentioned systems that use virtual machine introspection or a modified version of Android do not suffer from that. Although in these cases the monitoring tool cannot be detected, the malware sample can still detect the virtual or emulated environment, if it is not a bare-metal platform.

Log Examples.

Listings 2.1 and 2.2 present examples of API function calls registered by the instrumented application. Listing 2.1 presents a call to a function that sends an SMS message. In this case the destination number is "7132" and the message is "846978". Listing 2.2 presents a call to a function the executes a process. The executed process is `/data/data/org.zenth.oughtflashrec/cache/asroot` and the parameters are `/data/data/org.zenthought. flashrec/cache/explXXXXXX`, `/data/data/org. zenthought.flashrec/cache/dump_image`, `recovery` and `/mnt/sdcard/recovery-backup.img`.

Listing 2.1: Call to send an SMS message

```
Landroid/telephony/SmsManager;->sendTextMessage(Ljava/lang/String;=7132
| Ljava/lang/String;=null
| Ljava/lang/String;=846978
| Landroid/app/PendingIntent;=null
| Landroid/app/PendingIntent;=null)V
```

Listing 2.2: Call to execute a process

```
Ljava/lang/Runtime;->exec([Ljava/lang/String;={
/data/data/org.zenthought.flashrec/cache/asroot,
/data/data/org.zenthought.flashrec/cache/explXXXXXX,
/data/data/org.zenthought.flashrec/cache/dump_image,
recovery,
/mnt/sdcard/recovery-backup.img})Ljava/lang/Process;=Process[id=541]
```

Listing 2.3 presents two calls to the `execve` system call. They were both used to obtain information about the device, one focusing on CPU information and the other on memory information.

Listing 2.3: Examples of registered system calls

```
execve(``/system/bin/cat'', [``/system/bin/cat'', ``/proc/cpuinfo''],
[``ANDROID_SOCKET_zygote=9'', ``ANDROID_BOOTLOGO=1'',
   ``EXTERNAL_STORAGE=/mnt/sdcard'', ``ANDROID_ASSETS=/system/app'',
   ``PATH=/sbin:/vendor/bin:/system/s''...,
   ``ASEC_MOUNTPOINT=/mnt/asec'', ``LOOP_MOUNTPOINT=/mnt/obb'',
   ``BOOTCLASSPATH=/system/framework/''..., ``ANDROID_DATA=/data'',
   ``LD_LIBRARY_PATH=/vendor/lib:/sys''..., ``ANDROID_ROOT=/system'',
   ``ANDROID_PROPERTY_WORKSPACE=8,327''...]) = 0

execve(``/system/bin/cat'', [``/system/bin/cat'', ``/proc/meminfo''],
[``ANDROID_SOCKET_zygote=9'', ``ANDROID_BOOTLOGO=1'',
   ``EXTERNAL_STORAGE=/mnt/sdcard'', ``ANDROID_ASSETS=/system/app'',
   ``PATH=/sbin:/vendor/bin:/system/s''...,
   ``ASEC_MOUNTPOINT=/mnt/asec'', ``LOOP_MOUNTPOINT=/mnt/obb'',
   ``BOOTCLASSPATH=/system/framework/''..., ``ANDROID_DATA=/data'',
   ``LD_LIBRARY_PATH=/vendor/lib:/sys''..., ``ANDROID_ROOT=/system'',
   ``ANDROID_PROPERTY_WORKSPACE=8,327''...]) = 0
```

Analysis Stimulation

Some actions of the malware are only carried out if certain events are observed or if certain interactions with the graphic interface are performed. To stimulate these actions we automatically generate random events with the MonkeyRunner tool, which is distributed with the Android SDK, and create some events related to phone calls, SMS messages, geographic location and battery state, using the emulator.

As the events that interact with the graphic interface are generated randomly, they may not lead the application to execute the malicious code. One way to solve that is manually interacting with the applications during the analyzes, but when analyzing a large number of applications, it becomes too time-consuming. Another way to do this is by statically identifying which interactions are necessary to reach the relevant portions of the code and provide these interactions during the analysis. This approach is used by [95], but their system requires a modified version of the Android OS, which may be a problem, as discussed earlier. Another way to do this would be to identify the necessary interactions as done by [95], but generate them without needing a modified version of the OS. We leave this as a future work.

To make the analysis system more similar to the system of a real user, making it harder for malware to identify it is being analyzed, we changed the IMEI and phone number of the device [83]. Moreover, we added some contact information.

Malware Identification

The attributes used to classify the applications as malicious or benign are extracted from the data obtained during dynamic analysis. More precisely, we extract the amount of calls to each one of the 74 monitored Android API functions and the amount of calls to each one of 90 system calls[8].

For example, after an analysis, if the API function calls log produced the results illustrated in Listings 2.1 and 2.2, and the system call trace produced Listing 2.3, the attributes of the evaluated sample would be the following array:
`1,1,0,...,0,2,0,...,0`, in which the first two "1" refer to the `android/telephony/SmsManager;-> sendTextMessage` and the `java/lang/Runtime;->exec` API function calls, and the following "zeroes" refer to the frequency of the other API function calls, whereas the "2" refer to the `execve` system call, followed by a sequence of "zeroes" related to the remaining system calls' frequencies.

To create the classifier we first evaluated several algorithms, using the Weka [41] framework, and the one that performed best was RandomForest (with 100 trees). This experiment is detailed in Section 2.2.4.

2.2.4 Evaluation

This section describes the datasets used in the evaluations, the comparison of algorithms performed to select which one would compose the classifier and the test carried out to evaluate the classification system, including a comparison with other systems.

Datasets

The malicious application dataset is composed by samples from the "Malgenome Project" [97] and from a torrent file acquired from VirusShare (`http://tracker.virusshare.com:6969/`), totalling 4,552 samples. To compose the benign dataset we developed a crawler to collect applications from the AndroidPIT market (`http://www.androidpit.com/`). Through it, we gathered 3,831 applications to compose the benign dataset. These applications were submitted to VirusTotal, a system that uses more than 40 antivirus systems to scan the submitted file, and the ones that were detected by at least one antivirus were removed. Hence, the benign dataset contains 2,968 applications. In order to compose the training and testing datasets, we randomly split the malicious and benign datasets. Table 2.19 shows the amount of malicious and benign samples in the training and testing datasets[9].

Evaluation of classification algorithms

In order to identify which algorithm to use in the classifier, we compared the results obtained using several machine learning algorithms (the same ones used in [70]). For this test we used the training dataset mentioned before. Table 2.20 presents the algorithms and configurations used in the comparison. Furthermore, Table 2.21 presents the accuracy yielded by the 10-fold validation performed using each algorithm. The accuracy was

[8]The lists of API functions and system calls used are presented in `http://pastebin.com/T7Yfbksq` and `http://pastebin.com/5Xyjh8GS`.

[9]The lists with the SHA-1 hash values of the samples used can be found at `http://pastebin.com/OK9Xxj7U` (training/malicious), `http://pastebin.com/FCp9pCsK` (training/benign), `http://pastebin.com/ZwLnDPJd` (testing/malicious) and `http://pastebin.com/apV32ywX` (testing/benign)

Table 2.19: The amount of malicious and benign samples in the training and testing datasets

	Training	Testing	Total
Malicious	2,295	2,257	4,552
Benign	1,485	1,483	2,968
Total	3,780	3,740	7,520

calculated as $Accuracy = \frac{(TP+TN)}{(TP+TN+FP+FN)}$, with FP being false-positive, FN being false-negative, TP being true-positive and TN being true-negative. The algorithm that achieved th best results was RandomForest with 100 trees. The RandomForest algorithm generates several decision trees and chooses the one with the best results.

Table 2.20: The algorithms and configurations used in the evaluation to select the algorithm to be used by our classifier

Algorithm	Configurations
RandomForest	Number of trees {10, 50, 100}
J.48	Default
SimpleLogistic	Default
NaiveBayes	Default
BayesNet	Search algorithm {K2, TAN}
SMO	Kernel {PolyKernel, NormalizedPolyKernel}
IBk	Value of k {1, 3, 5, 10}

Table 2.21: Comparison of the detection using several classification algorithms over the training dataset with 10-fold validation

Algorithm	Accuracy (%)
RandomForest 10	93.20
RandomForest 50	95.65
RandomForest 100	95.96
J.48	93.04
NaiveBayes	82.39
SimpleLogistic	67.92
BayesNet TAN	74.53
BayesNet K2	89.92
SMO PolyKernel	75.03
SMO NPolyKernel	85.45
IBk 1	89.92
IBk 3	87.60
IBk 5	86.85
IBk 10	83.70

Detection evaluation

As the RandomForest algorithm (with 100 trees) yielded the best results in the previous experiment, we used it to evaluate our detection system. We trained the classifier using the training dataset and used it to classify the testing dataset. Table 2.22 presents the confusion matrix with the results of this test. From the 2,257 malicious applications used for testing, 2,168 were correctly classified and 89 were false-negatives, i.e., malicious applications classified as benign. From the 1,483 benign applications, 1,447 were classified as such, whereas 36 were considered malicious, comprising the false-positives. The values of false-positive, false-negative, true-positive, true-negative, accuracy (A), recall (R), precision (P), harmonic mean (F-measure) and the amount of correctly classified samples are shown in Table 2.26. The recall was calculated as $Recall = \frac{(TP)}{(TP+FN)}$, the precision was calculated as $Precision = \frac{(TP)}{(TP+FP)}$ and the harmonic mean was calculated as $F - measure = \frac{(2*R*P)}{(R+P)}$.

Table 2.22: Confusion matrix with the detection results using the RandomForest (100) algorithm

		Correct class		
		Malicious	**Benign**	*Total*
Results	**Malicious**	2,168	36	2,204
	Benign	89	1,447	1,536
	Total	2,257	1,483	3,740

Table 2.23: Values of false-positive (FP), false-negative (FN), true-positive (TP), true-negative (TN), accuracy (A), recall (R), precision (P), harmonic mean (F-measure) and correctly classified samples (CC) obtained in the system evaluation

FP	**FN**	**TP**	**TN**	**A**	**R**	**P**	**F-measure**	**CC**
2.43%	3.94%	96.06%	97.57%	96.82%	96.06%	97.53%	96.79%	96.66%

Discussion

Table 2.24 presents the comparison of the results obtained by our system with the results presented in [70], [76], [9] and [88]. The PUMA [70], DREBIN [9] and DroidMat [88] systems statically extract features, whereas our system and the one presented in [76] do it dynamically. Though the results obtained by DroidMat are a little better than ours, systems that rely on static analysis to obtain information from the code may fail when dealing with highly obfuscated samples and samples that download and execute code at runtime, as mentioned before. Moreover, our evaluation used a significantly larger number of malicious samples than PUMA and DroidMat.

The features used by our system and the one presented by Su et al. [76] have some elements in common. Their system uses the frequency of use of 15 system calls and 9 features from network traffic, whereas our system uses the frequency of use of 90 system calls and also the frequency of use of 74 Android API functions. We argue that the API calls provide important information for the classification. To corroborate that assertion, we performed another experiment, using the same datasets presented before, for training and testing, but this time we used three additional sets of features: the frequency of the 15 syscalls used by Su et al.; the frequency of the 15 syscalls used by Su et al. plus the frequency of API calls; the frequency of API calls. The results of this test along with the detection rate obtained by the previous test (the evaluation of our system) are presented in Table 2.25 and show that using the features related to API calls greatly improved the detection rate.

Besides the classification using 15 system calls, the system presented by Su et al. has also a classifier that uses features extracted from network traffic. This classifier is used to detect malicious samples that were not identified by the first classifier and that match a certain heuristic. From the 191 malicious samples incorrectly labeled by the classifier that used 15 system calls, 150 matched the heuristic used in their work. Considering the best scenario, in which these 150 samples are correctly identified using their classifier that uses network features, the accuracy would be 93.02%, which is still considerably lower than the value of 96.82% obtained by our system. This is another evidence of the benefits obtained using the features related to Android API function calls. A possible reason for the accuracy obtained by Su et al. being greater in the evaluation test presented in their work is the use of too few samples.

Table 2.24: Comparison of the results obtained by our system with the results presented in related work, showing the number of malicious and benign samples used in the evaluation test, the accuracy obtained and whether the system extracts features statically or dynamically

System	**Samples (Mal./Ben.)**	**Accuracy**	**Type**
DroidMat [88]	238 / 1,500	97.87%	static
PUMA [70]	249 / 1,811	86.41%	static
DREBIN [9]	5,560 / 123,453 [a]	93%	static
Su et al. [76]	50 / 70	99,20%	dynamic
Our system	2,257 / 1,483	96,82%	dynamic

[a]This is the total dataset used by them, including testing and training. They randomly split the dataset into training (66%) and testing (33%), 10 times, and average the results.

2.2.5 Limitations

The main limitations of the developed system are related to shortcomings inherent to dynamic analysis approaches. The analysis system may fail to observe the malicious behavior of samples in some situations, due to problems when gathering resources, to the lack of the necessary stimulation or to the detection of the analysis environment. If, for example, the malware tries to obtain some piece of code from the Internet or tries to connect to a command and control server to get instructions, but the connection fails,

Table 2.25: Comparison of the features used by our system with the features used by Su et al. [76]

Feature set	FP	FN	Accuracy
Freq. of API function calls + Freq. of system calls	36	89	96.82%
Freq. of API function calls + Freq. of 15 system calls	39	93	96.62%
Freq. of 15 system calls	180	191	89.70%
Freq. of API function calls	73	190	93.33%

the sample may stop executing without performing malicious actions. In addition, the malware sample may execute malicious actions only when certain interactions with the user interface are performed or when certain events, such as receiving an SMS message, occur. If the system fails to simulate these events, the malicious behavior will not be shown. Lastly, malware may detect the analysis environment and stop executing or execute innocuous actions, so the system will not obtain information about them. This detection can be carried out by the identification of virtualized or emulated environment, or the identification of monitoring tools.

2.2.6 Conclusions and Future Work

In this paper we presented a system that uses machine learning to classify Android applications as malicious or benign using information about the use of Android API functions and system calls. To gather the information needed by the detection system, we implemented a dynamic analysis system. To evaluate the capabilities of the detection system, we trained it with 3,780 applications and tested it using 3,740 samples, obtaining an accuracy of 96.82%. This result was compared to other detection systems, which demonstrated the relevance of our approach.

Future work includes the following: using attributes obtained from the network traffic and attributes obtained statically to enhance the detection capabilities of our system; detecting the evasion of sensitive information using signatures; making a public submission interface available to other researchers and common users, so they can check whether a given application is malicious; developing a non-random way to stimulate the malware using information obtained from the code without the need to modify the Android OS.

2.2.7 Acknowledgment

Part of the results presented in this paper were obtained through the project "Evaluation and prevention of security vulnerabilities in smartphones and tablets", sponsored by Samsung Electronics da Amazônia Ltda., in the framework of law No. 8,248/91.

2.3 Dynamically Identifying Evasive Android Malware

Publication: This paper was submitted to the 47th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) 2017

Vitor Monte Afonso[1], André Ricardo Abed Grégio[2] and Paulo Lício de Geus[1]
(1) University of Campinas
(2) Federal University of Parana

Abstract

Dynamic analysis of Android malware suffers from anti-analysis techniques that identify the analysis environment and prevents the malicious behavior from being observed. Researchers have proposed systems that identify evasive malware that affect Windows. However, due to differences between Windows and Android, applying these techniques directly to Android malware does not yield results just as good. In this paper, we present a novel technique to identify evasive Android malware. Our technique compares the execution of malware in baremetal and emulated environments, taking into account problems of dynamic analysis that are more common in Android and leveraging information more easily obtained in Android. We analyzed 1,470 samples using our approach, detecting 192 as evasive. Furthermore, we compared our results with the existing approaches using a subset of these samples and obtained better results. We also discuss which information is used by some of the detected samples to evade analysis.

2.3.1 Introduction

Mobile devices are increasingly becoming prevalent and being used to store several types of sensitive data, such as banking credentials and corporate information. This scenario makes attacks against users of such devices more rewarding. Therefore, more malicious applications (hereafter referred to as *apps*) that affect mobile devices are created and spread. Moreover, to overcome defense measures, these threats are constantly being improved.

Since Android is the most widespread operating system for mobile devices [43], it became the main target of mobile malware. According to PulseSecure [65], in 2014 97% of all mobile malware was developed for the Android platform. Furthermore, a study by Unucheck and Chebyshev [79] shows that Kaspersky Lab detected 884,774 new malicious mobile programs in 2015.

In order to identify the actions that apps may execute in infected systems, they need to be analyzed. The results of such analysis can be used to vet apps from app stores, as input to defense mechanisms or for incident response efforts. Several approaches have been proposed to analyze Android apps, obtaining information about them ([40, 53, 69, 75]) and classifying them as malicious or benign ([9, 27, 70, 76, 88, 96, 98]).

Analysis techniques can be static or dynamic. Approaches that rely on static analysis of code become less effective when dealing with highly obfuscated samples [59] or samples that obtain and execute code at run time [63]. Dynamic approaches, on the one hand, usually do not present problems when analyzing samples with obfuscated code, because they observe the actions performed during execution, which is not affected by the obfuscation. On the other hand, dynamic analysis may fail to observe the malicious actions of samples that employ anti-analysis techniques. These techniques are used by malware to identify when they are being analyzed, changing their behavior to prevent analysis systems from obtaining information about them.

Researchers have identified several anti-analysis techniques that can be employed by Android apps to differentiate between a real and an emulated environment ([45, 58, 62, 74, 80]), which is used by most dynamic analysis systems as its base due to its scalability advantage. One possible alternative to analyze apps without being evaded by most anti-analysis techniques is to use a real device instead of the emulator (such technique is used by BareDroid [60]). However, identifying which apps have evasive features is also useful: by studying these samples researchers may identify ways to make systems that use emulation more resilient against evasive malware.

In this paper, we present a technique to identify Android malware that exhibit evasive behavior. To do so, we compare the behavior of samples in an actual mobile device as well as in an emulator. Moreover, we use information obtained from the Android runtime, such as what methods were executed, and information from system call traces. Systems that identify evasive Windows malware by comparing their behavior in real systems and in emulated environments have been proposed in the literature [48, 52]. To accomplish this, these systems use information obtained from system call traces. However, differences between Windows and Android make a simple direct application of these techniques to the Android context less likely to succeed. To demonstrate this, we created detectors based on the techniques used by Disarm [52] and Barecloud [48], and compared their results with our approach. Our technique obtained better results, demonstrating that it is more appropriate for this context.

We analyzed 1,470 Android malware samples selected from different families and identified 192 samples with evasive behavior among them. We manually inspected a subset of the detected ones in order to identify how they evade dynamic analysis systems. The comparison with other techniques was performed using 50 randomly selected samples, all from different families. We performed manual analysis of these to validate the results of our approach.

The main contributions of our work are the following:

- We present a novel technique to identify evasive Android malware by comparing the results obtained from a baremetal system and an emulated system;
- We compared our approach to the detection techniques that focus on Windows malware, demonstrating that our technique is more appropriate for the Android context;
- We tested our technique with 1,470 samples, identifying 192 that employ evasive techniques, and discuss the techniques used by a subset of them to evade analysis.

The remainder of this paper is organized as follows: Section 2.3.2 presents the motivation and an overview of our technique.;Section 2.3.3 describes how we represent the behavior of apps; Section 2.3.4 presents our approach to identify evasive Android malware; Section 2.3.5 presents the system we developed to dynamically analyze apps and monitor their behavior; Section 2.3.6 presents the experiments we performed to demonstrate the effectiveness of our technique; Section 2.3.7 explains the limitations of our system and our technique; Section 2.3.8 presents related work and finally, Section 2.3.9 presents conclusions and future work.

2.3.2 Motivation and approach

Approaches to detect evasive malware through the comparison of their behavior in emulated and real environments have already been proposed in the literature about "traditional" systems [52] [48]. Disarm [52] analyses samples in real and emulated environments, and compare their behavior profile to calculate an evasion score based on the Jaccard index. Barecloud [48] is a similar approach, but it organizes the observed behavior in a hierarchical structure, which is used to compute the similarity of samples in different levels of abstraction. These systems use as input data obtained from system call traces and focus on Windows malware. We believe that simply applying the same techniques for Android apps does not yield just as good results, because of differences between the operating systems and differences among malware that affect each system.

Android malware in general execute much fewer actions than Windows malware. Disarm [52] uses 150 actions as the minimum to consider a sample executed a normal amount of actions. In our experiments, however, Android malware executed on average less than 10 actions. Furthermore, many Android malware are repackaged apps, which include benign behavior. Therefore, depending on the malware family, the actions protected by anti-analysis features in Android malware can be a very small subset of all possible behaviors, but might just as well comprise most of the app's behavior.

Another important difference is that Android is more event-driven, so failure to provide exactly the same events to both emulated and real environments may result in very different traces. One possible cause of the difference in event generation is the emulator being much slower than real devices. For instance, an Intent may take too long to be sent to its destination and the analysis may end before the Intent can cause its effects. It is also possible that some unforeseen event affects the real device, such as the Wifi network getting disconnected and reconnected. Another possible source of divergence is the "App Not Responding" verification of Android. A sample may execute successfully in a real device and during its execution on the emulator, Android may kill the sample's process and present the "App Not Responding" message, causing the behavior to differ between the emulated and baremetal environments.

Because of the presented reasons, we believe approaches that work by comparing the amount of actions executed in the baremetal and in the emulated environments, without considering that the different behavior may be due to reasons other than anti-analysis features, are less likely to succeed in identifying Android malware. To verify this, we compared our technique with the techniques presented by Disarm [52] and Barecloud [48], and our approach yielded better results. This comparison is presented in Section 2.3.6.

To overcome the aforementioned problems, we try to identify the cause of each of the different actions observed. This is accomplished by leveraging information that is easily obtained in Android, but not for Windows programs. More precisely, we track the methods of the app under analysis that are executed, the methods from the framework called by them, the system calls used by the app and the interaction of apps with functionalities that create threads or indirectly change their execution flow. We also monitor information provided by the system regarding events that stop the execution of apps and information about external stimuli used. With all this information, we can trace back the call sequence that led to the behavior that was only observed in baremetal, identifying the entry point that originated this sequence and possibly the external stimulus that caused it. By comparing the sequence obtained from the baremetal system to the behavior observed in the emulator we can identify if the divergence happened due to some event not being generated, due to a difference in some method's execution, due to the analysis time ending in one of the environments, or due to the system stopping the app.

2.3.3 Behavior representation

We represent the behavior of a sample in a given analysis environment as a set of actions observed during its execution. Each action is a tuple and is represented as follows.

$a = (action\_type, operation, argument)$.

action_type is one of {*Network*, *File*, *Intent*, *Exec*, *Phone*, *Dex*, *Billing*, *Multimedia*}.

Each action type and the associated operations and arguments are presented below.

Network. For network related actions, operation is one of {*INET*, *UNIX*, *NETLINK*, *BLUETOOTH*}. *INET* operations represent TCP and UDP connections and *argument* is the destination that the sample connected to. Since multiple resolutions of the same DNS name may result in different IP addresses, we consider two actions the same if they use the same IP address or the same DNS name as destination. *UNIX* operations represent connections to UNIX sockets and the argument is the filesystem path used by the socket. *BLUETOOTH* operations represent the use of the Bluetooth device and the argument, in this case, is the operation performed with this device. Lastly, *NETLINK* operations represent connections using NETLINK sockets and the argument used is the protocol parameter passed to the socket.

File. The monitored operations on files are *WRITE* and *DELETE*, and the argument of both operations is the file path.

Intent. The operations related to Intents are *ACTIVITY*, *SERVICE*, *BROADCAST* and *ALARM*. The argument for all these operations is the "action" argument of the Intent or the destination class of the Intent. *ALARM* operations refer to the use of *AlarmManager* to send Intents.

Exec. This action type represent the use of the `execve` system call, which is used by the API methods *ProcessBuilder.start* and *Runtime.exec*. The argument used is the name of the executable file being called.

Phone. This action represents the use of phone capabilities. We currently only consider one operation of this type, namely sending SMS messages and the argument is the destination number of the message.

Dex. This action type represents the use of dynamic code loading and its argument is the path of the file being loaded.

Billing. This action represents the use of the billing functionality; the argument is the type of action performed.

Multimedia. The operations included in this action type are *CAMERA*, *AUDIO* and *WAKELOCK*. The argument in these cases is the type of action being performed, which includes taking pictures, recording videos, recording audio and acquiring wake locks, which allow the app to keep the CPU running and the screen on.

Behavior normalization

File names written by apps may be randomly generated, making that multiple executions of the same sample may incorrectly result in different behavior profiles. To overcome this problem we take the same approach used by Disarm [52]. For each sandbox, which can be emulated or baremetal, we identify files that were written in only one instance of this sandbox. We consider these as possibly random files. Possibly random files in multiple instances of a sandbox that have the same directory and extension are considered as random. We keep the directory name and extension of these actions but replace the file name by *<RANDOM>*. We also normalize file paths related to the sdcard, as it can be accessed in different ways.

Malware may randomly select contacts registered in the system as destinations of SMS messages. Therefore, we inspect actions related to sending SMS messages and, if some destination is a contact registered in the system, we replace it by *<CONTACT>*. This also prevents the list of actions from growing as large as the list of contacts if the malware sends messages to all of them.

We also remove simple actions that are common to most apps, such as writing to the shared memory device or to the logging device. Another group of actions we filter is related to Webview. Because of libraries available solely in the baremetal system, when Webview is used, some actions are only performed in the baremetal system. To avoid this problem, we trace the source of these actions and, if the Webview library is their source and they were executed by a specific method used in Webview's initialization, we filter them. It is worth to notice that we do not filter every action performed by Webview, which would also include important actions for understanding the app's behavior. We only filter few actions that are always performed when Webview is loaded, and only if the action was generated from a specific method.

One common problem that systems face when running analyses multiple times is that unless one runs them all in a small time frame, it is possible that the hosts accessed through the network are not available for all executions (they may be down due to incident reporting, for example). This may lead to certain network behavior only being observed in some of the analyses. To avoid this problem, when some host is accessed in baremetal and the same DNS name is requested in the emulated context, but this request fails, we also add it to the emulated analysis.

2.3.4 Evasive behavior identification

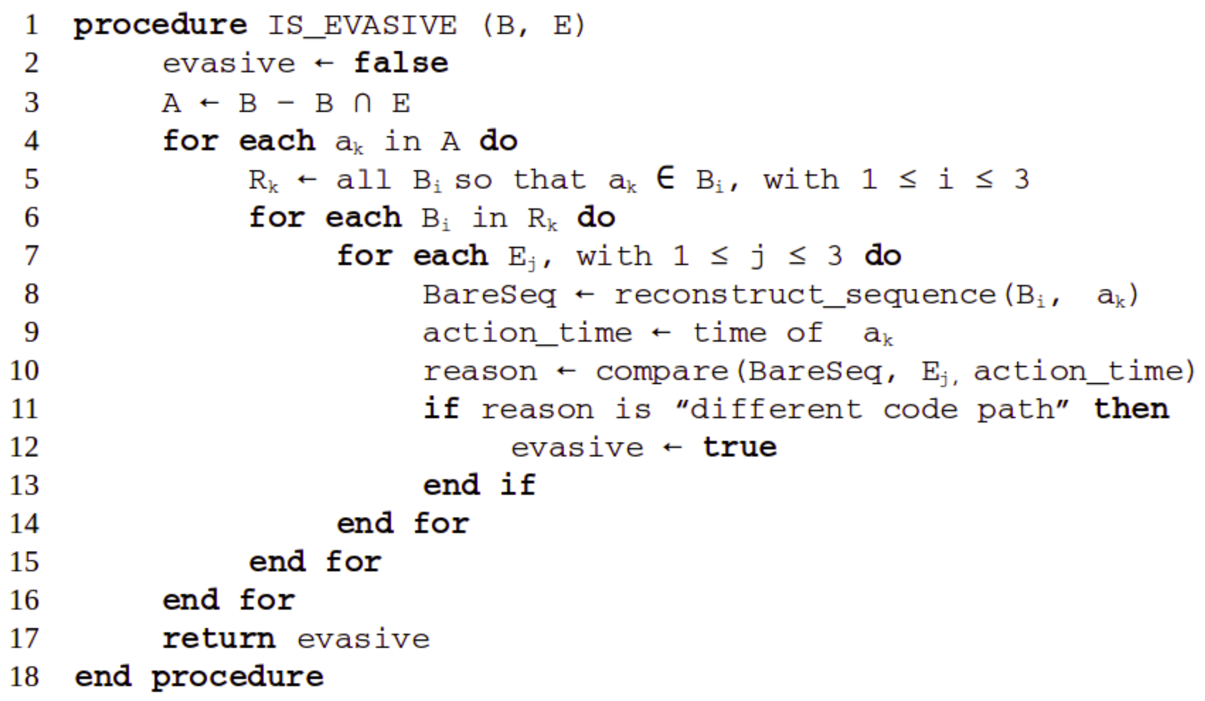
Overview

To identify whether an app is evasive or not, we analyze it in a baremetal environment and in an emulated environment, and then compare the monitored behavior to identify differences. If they are different, we identify the root cause of the divergence, which can be a variation in the code path executed or something that prevented the app from continue executing in the emulated environment. If we identify a divergence in the code path executed, we consider it as an evasive sample. To increase the amount of code executed during dynamic analysis, we generate stimuli in the form of GUI interactions and Intents, which can be used to start activities or receivers. We provide the same stimuli for both baremetal and emulated environments. Moreover, to identify non-deterministic behavior we execute each sample three times in each environment.

Let B_i and E_j be the set of actions monitored in the baremetal environment for the ith time and in the emulated environment for the jth time, respectively, with $1 \leq i \leq 3$ and $1 \leq j \leq 3$. Also, let $B = \bigcup_{i=1}^{3} B_i$ and $E = \bigcup_{j=1}^{3} E_j$ be the set of all actions executed in baremetal and in emulated environments, respectively. Since we are interested in finding apps that hide their actions when being analyzed, we first select the set A of actions that were only executed in a real device. Thus, $A = B - B \cap E$.

For each action a_k in A, we construct R_k, a set with the instances of baremetal analysis that contain this action. We compare each B_l in R_k to every E_j to identify why a_k was not executed in the emulated analyses.

Since we track when methods begin and end, we can identify the app's method that executed the action we are interested in. We trace back the sequence of method calls that



```
procedure IS_EVASIVE (B, E)
    evasive ← false
    A ← B − B ∩ E
    for each $a_k$ in A do
        $R_k$ ← all $B_i$ so that $a_k \in B_i$, with $1 \leq i \leq 3$
        for each $B_i$ in $R_k$ do
            for each $E_j$, with $1 \leq j \leq 3$ do
                BareSeq ← reconstruct_sequence($B_i$, $a_k$)
                action_time ← time of $a_k$
                reason ← compare(BareSeq, $E_j$, action_time)
                if reason is "different code path" then
                    evasive ← true
                end if
            end for
        end for
    end for
    return evasive
end procedure
```

Figure 2.5: Algorithm to determine if a sample is evasive.

led to this action, going back until an external stimulus is found, such as calling the main Activity of the app or until we find an entry point whose origin we cannot determine. We track the use of Intents and thread related API methods, such as *Timer.schedule*, so we can identify indirect calls for the most common entry points of Android classes, such as *onCreate*, *run* and *handleMessage*. The process to reconstruct call sequences is explained in more detail in Section 2.3.4.

At this point, we know the main entry point for a baremetal analysis instance B_l—called from now on of M_k—that led to the execution of action a_k, and possibly the external stimulus that caused it. We find the occurrences of M_k in E_j. If we know the stimulus that originated it, we only consider the M_k instances that were also caused by the same stimulus, otherwise we consider all executions of it. For each M_k instance in E_j, we compare it with the B_l call sequence that led to a_k. With this comparison we identify where the path that should lead the emulated system to also execute the action diverged. More precisely, we identify which of the following is the cause of the divergence: different execution path; app not responding; analysis ended; fatal exception; entry point not reached. If the reason for the diversion is a different code path being executed, we consider this as an evasive sample, otherwise we consider the divergence as an execution problem. Figure 2.5 illustrates the entire process. The algorithm could return after setting *evasive* as true for the first time, reducing the time that it takes for the algorithm to run. However, the comparison also reveals information that is useful for an analyst to identify where in the app the divergence happens, thus we do not stop the comparisons when the first sign of evasion is found.

For the samples considered evasive, we are not able to automatically identify precisely why different code paths were taken in the emulated and baremetal systems. For instance, the divergence can be caused by a verification of the IMEI value or because of some information not available in the emulated environment. The identification of the precise information that caused the divergence can be performed through taint analysis, as done by TaintDroid [28] and Malgene [47], and we leave it as future work. However, our results can assist analysts in manually locating the cause of the divergence, because we identify in which method and after which method calls the divergence happened

Call sequence reconstruction

Since we record when each thread of the analyzed app enters and leaves its methods, we can identify which method performed a given action. We also log the methods called, so we can look back in the analysis trace, starting from the method that executed the action, and identify the sequence of method calls that led to this action.

This is enough to track the execution back to an entry point, but it cannot go further. As the Android framework is responsible for calling these methods, we will not be able to observe any direct calls to them. Android classes may have several entry points, which may be executed because of several reasons, the most common being related to creating activities (e.g., *onCreate*), starting services (e.g., *onStart*), starting receivers (e.g., *onReceive*), running tasks (e.g., *run*) and handling received messages (e.g., *handleMessage*). One possibility would be to compare all executions of the entry point method in the baremetal and emulated systems, but this could lead to wrong results, due to the previously mentioned uncertainty. This can happen, for instance, if an activity handles different functionalities, all executed through the same entry point. Thus, we try to identify the source methods from which the execution changed to the entry points, in order to have a more precise comparison of the executions. To accomplish this, we investigate Intents sent by the app, the use of several methods that cause indirect changes in the execution flow and the use of external stimuli.

To identify Intents that may have resulted in a specific entry point method being executed, we look for Intents sent by the app that match this specific method. For instance, if the method we are analyzing is *ClassA.onStartCommand(Intent, int, int)*, we assume *ClassA* is a service, since *onStartCommand(Intent, int, int)* is one of the entry points of the class *android.app.Service* that can be overwritten. As the Android documentation states, this method is called by the system when a client explicitly starts the service by calling *startService(Intent)*. Thus, to find the source that directed the execution to this method we look for actions that start services using *ClassA* as an argument. Finding sources of entry points to activities is similar to services. To find the sources that led to the execution of receivers, however, we need to inspect the intent filters used by the class and find out which Intents sent match it. More details about how we monitor Intents are presented in Section 2.3.5.

Another source of control flow changes that are performed by the framework is the use of the following groups of methods: methods that schedule a class to be invoked after some delay or periodically, such as *Timer.schedule* and *ScheduledThreadPoolExecutor.schedule*, which result in the execution of the method *run()* or the method *call()* of the destination class; methods that send messages to its UI thread, such as *Han-*

dler.sendMessageDelayed, which result in the execution of *handleMessage(Message)*; and methods that start a new thread, such as *Thread.start()*, which result in the execution of *run()* and *AsyncTask.execute(Params...)*, which in turn may result in the execution of different methods, the main one being *doInBackground(Params...)*. In order to be able to track these control flow changes, we instrument the Android framework to assign labels to the messages or tasks sent or to the threads created. We log them when the source method is executed and also when the destination methods are executed. More details on the implementation of this process are presented in Section 2.3.5. With this information, we can track the source call of any of these methods. It works even in cases when there are several destination methods, such as the source methods that invoke a class periodically, because they have only one source and we are tracking the sequence call backwards.

The last type of interaction that can cause the execution of entry points is external stimuli. As we mentioned before, these are important to increase the code coverage of the analyzed apps. To be able to correlate their use with the behavior of the app, we instrument the tools used to create the stimuli, identifying when Intents are sent, when keys are pressed and when GUI interactions are performed. These Intents are identified as the source of some entry point in the same way we do for Intents sent by the app, which we explained before. Key pressing and GUI interactions are handled in a similar way, but they are source of different entry points. Some of the entry points executed by key strokes are *onKey(DialogInterface, int, KeyEvent)* and *onKeyDown(int, KeyEvent)*. For GUI interactions, some common entry points executed are *onClick(DialogInterface, int)*, *onTouchEvent(MotionEvent)* and *onItemClick(AdapterView, View, int, long)*.

When tracing the sequence of calls that led to some action, we create a list of subsequences. The first subsequence goes from an external stimulus or from an entry point whose source we did not identify, until the call that created the next subsequence. Each one of the following subsequences represent the call sequence from one entry point until the call that resulted in the creation of the next one, while the last subsequence ends in the execution of the action. Along with each subsequence we keep the time of the call that created the next subsequence or that executed the action.

Comparing sequences

After identifying the list of subsequences of method calls that led to the execution of some action in the baremetal environment, we need to compare this with the results of the emulated system to identify the cause of divergence. We hereafter refer to this list of subsequences as *BareSeq* and to this instance of results from the emulated environment as *ResEmu*.

We iterate over each subsequence $SubBare_i$ of BareSeq, comparing it to its counterpart in ResEmu. For each iteration, suppose *action_time* is the time when the call that created the next subsequence was executed or the time when the action was performed. Also, let EP_i be the entry point of $SubBare_i$. We find all occurrences of EP_i in ResEmu that have the same origin as in BareSeq. We then proceed to compare EP_i from BareSeq with each instance of EP_i identified in the emulated results.

Given two entry point methods, we find where they begin and where they end, obtain-

```
procedure COMPARE_SEQUENCES(BareSeq, EmuRes, action_time)
    for all subsection SubBare in BareSeq do
        EPBare ← entry point of SubBare
        EPEmu ← entry point of EmuRes that matches EPBare
        if did not find EPEmu then
            return "entry point not reached"
        end if
        RawSeqBare ← find seq from start to end of EPBare
        RawSeqEmu ← find seq from start to end of EPEmu
        aligned ← align RawSeqBare and RawSeqEmu
        if SubBare is last subsequence of BareSeq then
            return compare_aligned(aligned, action_time)
        else
            CallNext ← action that created next subsequence
            if not emulated executed CallNext then
                return compare_aligned(aligned, CallNext)
            else
                continue
            end if
        end if
    end for
end procedure
```

Figure 2.6: Algorithm to compare a call sequence obtained in baremetal to its equivalent obtained from the emulated system.

ing the call sequence in this interval. We align these two sequences, one from BareSeq and the other from ResEmu, using a global alignment algorithm described in Section 2.3.4. If $SubBare_i$ is the last subsequence of BareSeq, we compare the aligned sequences to determine the divergence that prevented ResEmu from reaching the call at *action_time*. Otherwise, let *CallNext* be the method call in $SubBare_i$ that created the next subsequence of BareSeq. If the app did not reach *CallNext* in ResEmu, we compare the aligned sequences to determine the divergence that prevented ResEmu from reaching *CallNext*. However, if the app did reach *CallNext* in ResEmu, we get the next subsequence of BareSeq, with entry point EP_{i+1}, and find this entry point in ResEmu, by checking for possible destinations of *CallNext* in ResEmu. If we are not able to find an equivalent of EP_{i+1} in ResEmu, it is likely that the execution was interrupted before the call performed at *CallNext* could take effect, so we do not consider that an evasion happened. This algorithm is presented in Figure 2.6.

When comparing two aligned sequences, we want to identify the reason for their divergence in regards to some action executed at time t_i. This action is either a behavior only observed in BareSeq or some call that created the next subsequence of BareSeq and that was not executed in ResEmu.

We iterate over the calls in the aligned sequences and when we are past t_i, considering the time of the baremetal calls, we check what was the last call in the emulated sequence. There are three possibilities: i) a tag indicating the analysis process ended; ii) a tag

```
[78] BARE: 'com.adobe.flashplayer_.AdobeFlashCore.onCreate()' -> 'com.adobe.flashplayer_.
     AdobeFlashCore.writeConfig(java.lang.String, java.lang.String)'
      EMU: 'com.adobe.flashplayer_.AdobeFlashCore.onCreate()' -> 'com.adobe.flashplayer_.
           AdobeFlashCore.writeConfig(java.lang.String, java.lang.String)'
...
[85] BARE: 'com.adobe.flashplayer_.AdobeFlashCore.onCreate()' -> 'java.lang.String.indexOf(java.
     lang.String)'
      EMU: 'com.adobe.flashplayer_.AdobeFlashCore.onCreate()' -> 'java.lang.String.indexOf(java.
           lang.String)'
[86] BARE: 'None'
      EMU: 'com.adobe.flashplayer_.AdobeFlashCore.onCreate()' -> 'java.lang.System.exit(int)'
[87] BARE: 'com.adobe.flashplayer_.AdobeFlashCore.onCreate()' -> 'com.adobe.flashplayer_.
     AdobeFlashCore.isOnline()'
      EMU: 'None'
...
[102] BARE: 'com.adobe.flashplayer_.AdobeFlashCore.onCreate()' -> 'com.adobe.flashplayer_.
     FlashVars.<init>()'
       EMU: 'None'
```

Listing 2.4: Excerpt of the alignment of an evasive sample.

indicating the system killed the app for not responding or for some other error; iii) a call to some method of the app or the framework.

If we identify case iii, we consider that there was a divergence in code path taken and, therefore, that we found an evasive behavior. We print the aligned sequences, in order to help an analyst that needs to manually identify what caused the executions to follow different code paths. Conversely, if the last identified call matches cases i or ii, we consider that there was an execution error and not an evasion. Listing 2.4 presents an excerpt of the output generated for one sample that is evasive. This example shows that during the execution of method *AdobeFlashCore.onCreate*, the app called *AdobeFlashCore.writeConfig* and *String.indexOf* in both analysis systems. Then, in the emulated system the app called *System.exit*, whereas in the baremetal it called *AdobeFlashCore.isOnline*. An analyst can use this information to locate the exact location in the malware code where the evasion happens and what information is used. This information is presented in Listing 2.5. Invocations in lines 4 and 6 are the ones that were executed by both systems. The conditional instruction in line 9 is responsible for redirecting the execution flow if an analysis system is not detected, while the information used for this detection is the result of calling *String.indexOf*, which was used to verify if the device id contains the string "000000000000000".

Sequence alignment

To perform sequence alignment, we use the global alignment algorithm provided by the *swalign*[10] library. We chose a global alignment algorithm because we need to have a global understanding of the sequences, as our analysis depends on the alignment reaching the point in the baremetal sequence where the target action happened. If the aligned sequence does not reach this point, we are unable to identify the cause of divergence.

We only modified the code of *swalign* to work with the same data structure as our scripts, instead of the default, which is a string where each char is an element. We provide

[10] `https://github.com/mbreese/swalign/`

```
invoke−virtual/range {v17 .. v17}, Landroid/telephony/TelephonyManager;−>getDeviceId()Ljava/lang
    /String;
move−result−object v7
...
invoke−direct {v0, v1, v15}, Lcom/adobe/flashplayer_/AdobeFlashCore;−>writeConfig(Ljava/lang/
    String;Ljava/lang/String;)V
const−string v1, “000000000000000”
invoke−virtual {v7, v1}, Ljava/lang/String;−>indexOf(Ljava/lang/String;)I
move−result v1
const/4 v2, −0x1
if−eq v1, v2, :cond_5
const/4 v1, 0x0
invoke−static {v1}, Ljava/lang/System;−>exit(I)V
:cond_5
invoke−virtual/range {p0 .. p0}, Lcom/adobe/flashplayer_/AdobeFlashCore;−>isOnline()Z
```

Listing 2.5: Excerpt of the disassembled app where the evasion happens.

our own scoring method to *swalign*. The scoring method is responsible for comparing two elements and identifying if they match or not. Our method matches calls that are equal but also any call to special tags that represent special events. These events include the end of the analysis and the app being killed by the system.

Selecting the arguments is an important step in using alignment algorithms. The arguments we need to define are the following:

- m for matches, with $m > 0$;
- mi for mismatches, with $mi < 0$;
- g_o for opening gaps, with $g_o < 0$;
- g_e for extending gaps, with $g_e < 0$.

We do not want any mismatches, as they may mislead the analysis, so we used a high value for $|mismatch|$. We also believe that beginnings and endings of methods of the analyzed app are more important in the alignment than other types of calls, so we assign $2 * m$ for matches of this type. Furthermore, since we are investigating evasive behavior, we want to prioritize gap extensions over gap openings, so $|g_o| > |g_e|$. In the end, we have the following inequality to guide the definition of arguments:

$|mismatch| > m > |g_o| > |g_e|$.

2.3.5 Monitoring system

Behavior monitoring

To track the behavior of the analyzed apps, we monitor which apps' methods were executed, which methods were called from them and which system calls were executed.

To monitor system calls, we used a kernel driver that intercepts them. When a system call is executed, the driver registers its arguments and calls the original system call. In order to obtain information related to the use of Intents, the driver inspects *ioctl* calls that target the Binder device. If the operation performed is a BC_TRANSACTION, we log the destination class, method id and arguments passed. To identify which actual method is represented by the method id, we examine the corresponding AIDL file in the Android source code.

To monitor the executed methods, we leverage the "method trace" functionality of the Android runtime (ART) and instrument *libart*. Every time the execution goes in and out of a method, we register it. Also, when some method is called, we log the source and destination of such action. This allows us to also identify Java methods called from native code. To avoid registering too much information, we focus on new UIDs, so we do not track apps that are already installed in the system when it is in a clean state.

When trying to identify the method call that resulted in the execution of some receiver, we need to identify which intent filters are used by this receiver and look for broadcasts sent that match them. Parsing the app's manifest is not enough to obtain all intent filters that were used, since the app can register others at runtime. To overcome this limitation, we also track all calls to *android.content.Context.registerReceiver*.

As mentioned in Section 2.3.4, the use of threads, tasks and message passing between them introduces a level of indirection that prevents us from tracking the execution flow by just looking at method invocations. To be able to reconstruct the call sequence even in these cases, we track the use of threads, tasks and messages by assigning (and logging) a randomly generated number to them when they are created, scheduled or sent. We also log this identifier when they are actually used or executed, allowing a parser to match each use or execution of these types to their creation. This matching allows us to track the call sequence in these cases. So, for instance, when the method *java.util.Timer.schedule(TimerTask task, long delay, long period)*, which schedules a task for repeated fixed-delay execution, is executed, the system generates a random number, logs it and assigns it to the task. Every time this task is executed, the identification number is logged.

In order to correlate the external stimuli, like clicks and broadcasts, with the behavior of the app, we need to know the time at which each stimulus was provided. To achieve this, we instrument both tools used to create these actions, which are `am` and `input`.

Analysis environments

When analyzing malware, it is important to make sure that the environment is not infected before each analysis. Performing an analysis in an infected system may result in wrong results, as one malware can influence the behavior of others analyzed afterwards. To analyze samples in the emulator, we take advantage of the snapshot functionality, which allows us to restore the system to a clean state after every analysis and avoid waiting the time of booting the system. Analyzing malware in real devices, though, is a rather more complicated task, as they do not have an easy snapshot functionality in the same way as the emulator does. One possible way to overcome this problem is to restore the state of the device's partitions after every analysis, as in the Baredroid's approach [60]. However, this is very time consuming, as the system needs to reboot every time it is restored.

We chose a different approach to maintain the system clean after each analysis. In Android, apps can only write to a very limited set of directories, which includes mainly the app's dir (*/data/data/<PACKAGE-NAME>/*) and the sdcard. By uninstalling every app after it is analyzed, its own directory is removed by the framework. Files written to the sdcard can affect the behavior of other apps that interact with such files. Because of that, we delete every file from the sdcard after every analysis. To be able to overcome the system's restrictions and modify important files, some malware exploit vulnerabilities in the kernel or privileged processes, obtaining root privileges. However, since the */system* partition is mounted as read only by default, even apps that are able to obtain root privilege first need to remount this partition. To prevent this from happening, our kernel driver blocks all system calls that try to remount the */system* partition in a mode that allows writing. This protection can be bypassed if the malware manages to access the original `mount` system call. However, we only used the system to test our proposed technique to identify evasive malware. If one wants to use a similar system to receive submissions or analyze apps that could potentially target the system, a better protection or restoration process would be necessary. Furthermore, during our experiments our driver did not actually have to block any calls to `mount`, so we believe this was not a problem.

For our experiments we used the standard Android emulator deployed with the sdk and an LG G2Mini device. Both systems had our modified version of Android 5.1. Each analysis in the baremetal environment was executed for at most 3 minutes. As for the emulated environment, since it is much slower, we executed each analysis for at most 10 minutes. In small experiments we found that this time was necessary for the emulator to execute in a similar way as the baremetal in 3 minutes. Since we can identify when a divergence in behavior is caused by one analysis system finishing before the other, this difference in execution time does not negatively affects our technique.

When dynamically analyzing Android apps, it is important to provide GUI interactions and to cause activities, services and receivers to execute in order to increase code coverage. However, as we are comparing multiple executions of apps, it is also important that we provide exactly the same interactions, so the same code paths are exercised, at least until evasive code is reached or some problem stops the app execution. In order to accomplish this, we use the Droidbot [51] tool to interact with apps. Droidbot generates random events, including GUI interactions, broadcasts and specific activities. It also registers the exact events generated and is able to replay them from a file instead of randomly generating them. Thus, in our first baremetal execution of each sample, we randomly generate events and save them to a file. In the following baremetal executions and in the emulated analyses, we make Droidbot read the events from the saved file.

One way that malware can identify the analysis environment is by checking which apps are installed on the system. The lack of the Google Play app, for instance, is a strong indication that it is not a real device. Therefore, we installed in the baremetal environment Open GAPPS [2], a set of basic apps present in all Android systems. Furthermore, we also installed a few very popular apps and created fake contact information. These apps and contact information make the baremetal and emulated systems different, which could, therefore, cause some apps to behave differently, but not because they intend to evade analysis systems. This could possibly lead our technique to identify such samples as evasive, increasing the number of false-positives. However, not doing so may result in false-negatives since, as we mentioned, the baremetal system could also be detected as an analysis system. We chose to use these techniques and risk increasing false-positives instead of risking increasing false-negatives.

2.3.6 Experiments

To evaluate our technique, we used our dynamic analysis system to analyze a subset of the samples in our malware dataset, which consists of samples obtained from VirusShare [82], Malgenome [97], contagio mobile [61], AndroMalShare [1] and Drebin [9]. To select this subset we first obtained their detection by antivirus software, using Virustotal [3]. We separated them by families, using the results of the ESET-NOD32 antivirus, and selected at most 5 samples from each family, resulting in a set of 1,470 samples.

We analyzed these samples with our system to obtain their behavior and used our proposed technique to identify which ones have evasive behavior. Since we do not have a ground truth with information about all these samples, we randomly selected 50 samples, all from different families, and manually inspected their results to identify possible false-negatives and false-positives.

Our technique detected 7 out of 50 samples in the subset as evasive. Using manual analysis we identified no false-negatives and 3 false-positives. We consider as false-negatives the samples that did evade analysis but our approach did not identify them as evasive, and we consider as false-positives the ones that we considered evasive in cases the diverging behavior is similar to some action performed in the emulated environment and it does not stem from an identification of the analysis system. Note that we consider evasive those samples that execute some action only in the baremetal system, without executing some similar action in the emulated system, even if this divergence is not caused by a clear identification of the analysis system. For instance, if some sample tries to send SMS messages to contacts stored in the phone and it only shows this behavior in the baremetal because there is no contact registered in the emulated environment, we consider it as evasive. We do this because, despite not being a clear sign of anti-analysis behavior, it is successful in preventing some action from being observed in the emulator and so could be employed as an anti-analysis technique.

We discuss below the samples that our technique identified as evasive, explaining why we consider them as a true-positive or false-positive. For the true-positive ones, we discuss which extra behavior was observed due to the diversion and what difference between the baremetal and emulated environments was its cause.

Sample 1. It changes its behavior if */system/xbin/busybox*, */system/bin/busybox* or */bin/busybox* is present in the system. This deviation resulted in the malware writing to the file *shared_prefs/config.xml* and many files in the dir */sdcard/LuckyPatcher/*. This may not have been intended as an anti-analysis technique, since most user systems do not have these files. However, it does prevent some of the malware behavior from being observed in the emulated environment, so we consider this as a true-positive.

Sample 2. It Identifies if the phone number starts with “15555”, if the value of IMEI starts with “00000000” or if the value of IMSI starts with “31026”. Upon detection, this sample calls *System.exit(0)*. This is a clear case of evasive malware, so we consider it as a true-positive. The behavior resulting from the divergence is composed of starting a service and starting two alarms that send Intents.

Sample 3. It copies the icons of the apps installed in the system to the dir */data/-data/com.pintudog/files/icons/*. Since the list of apps installed in the emulator and in the baremetal environments is not the same, the monitored actions ended up being different. However, at a higher level it is still the same behavior, so we consider this as a false-positive.

Sample 4. It verifies if the IMEI contains the string "000000000000000". If so, the malware calls *System.exit(0)*. Similarly to Sample 2, this is a clear example of anti-analysis, so we consider it as a true-positive. The actions resulting from the divergence are the following: starting a service, creating a wake lock and connecting to the *dnsproxyd* device to make a DNS request.

Sample 5. The different actions in the behavior of this sample are related to a file associated with the graphical interface. This happened because the graphical libraries used in the baremetal and emulated systems are different. Since this is not actually related to the behavior of the malware, we considered this sample as a false-positive.

Sample 6. During its execution this sample verifies which Wifi networks are available to the device. In the emulated system it does not identify any Wifi network, so it takes a different execution path. The behavior that is executed only in baremetal, as a result of this difference, is writing a file in the sdcard. Since this behavior is related to the malware execution and cannot be observed in the emulator unless some update is made to it, we consider this sample as a true-positive.

Sample 7. This sample randomly chooses the domain name to access from a list of predefined names. This resulted in one domain used in baremetal not being used in the emulated analysis. Except for the domain difference, their behavior is the same, so we consider this as a false-positive.

Comparison

To test our intuition that the existing techniques to identify evasive Windows malware would not present as good results if applied to Android malware, we implemented detectors based on the techniques proposed by Disarm [52] and Barecloud [48]. Since the behavior of Android and Windows malware are different in various aspects, we used our behavior model when implementing these techniques. To make the comparison fairer, we used the threshold that would yield the best results to each of these techniques, instead of the threshold they found for Windows malware. Table 2.26 presents the comparison of our results with the results obtained from the other approaches. Our technique performed better than the others, demonstrating that it is more appropriate to this context.

Other evasion techniques

Out of the complete dataset of 1,470 samples, our technique identified 192 as evasive. We manually inspected the results of some of these samples to understand how they evade

Table 2.26: Comparison with other approaches to identify evasive malware

	TP	TN	FP	FN	A
Our approach	100%	93.5%	6.5%	0.0%	96.7%
Disarm (t=0.12)	100%	78.3%	21.7%	0.0%	89.1%
Barecloud (t=0.36)	100%	67.4%	32.6%	0.0%	83.7%

analysis. Bellow we describe the anti-analysis techniques we identified that are different from the ones explained before.

GSF package. Malware can verify if the package *com.google.android.gsf* is installed in the system. This package provides Google Service Framework and is present in most Android devices, but not in the emulator.

Location updates. When using the method *LocationManager.requestLocationUpdates* in the emulator, an exception is generated, which can change the execution path of the sample. This type of evasion is harder to recognize statically, because there is not a clear *if-then-else* situation.

GCM. Samples can use the Google Cloud Messaging (GCM) service to identify the emulator. For instance, apps can try to register an id, which will fail because the emulator does not have a registered Google account.

Build properties. Apps can verify the value of the following properties of the *android.os.Build* class: if *Build.PRODUCT* or *Build.MODEL* equals "google_sdk"; if *Build.BRAND* or *Build.DEVICE* starts with "generic"; if *Build.HARDWARE* equals "goldfish".

App store. Apps can try to use Google Play by starting an activity with the category *android.intent.category.APP_MARKET*. Since the Google Play app is not installed in the emulator, this causes an exception, which can change the app execution path.

su binary. Samples that target rooted devices can try to execute "su", which allows for the execution of actions with root privileges. Since the emulator does not have such file, this action fails and the execution path changes.

2.3.7 Limitations

Our detection approach relies on identifying differences between the execution of samples in a baremetal and in an emulated environment. Therefore, if we are not able to reach the code that has anti-analysis functionalities or some sample is able to detect the baremetal environment as an analysis system, we will not be able to observe the differences. Not reaching the desired code path is a problem with all dynamic analysis systems, as they can only observe behavior that is actually executed. To exploit this, malware can wait for a period of time longer than the analysis is executed for, before activating the malicious behavior, or can only execute it after a series of complex GUI interactions that automatic interaction tools are unlikely to reach. Some malware may be able to detect both environments as analysis systems, because despite the baremetal environment being more similar to a real device, there are still differences that can be exploited, such as information about the user's behavior, e.g., browsing history and SMS history.

When tracing back the origin of some behavior executed in baremetal, we may find some entry point whose source we cannot identify. In these cases we compare it with all instances of the same entry point in the emulated environment. In some cases this may lead to wrong conclusions. Furthermore, differences in the systems may lead to the execution of different actions that are not related to evading analysis or to the execution of equivalent actions in both systems, but that are considered different in our behavior model. This is the general problem that resulted in our technique incorrectly detecting Sample 3 and Sample 5 as evasive. Also, sources of non-determinism that we do not currently handle may lead to the execution of the same high-level behavior, but different actions according to our model. This is the problem that resulted in our technique misidentifying Sample 7 as evasive. This malware randomly selects the domain name to access from a predefined list, so the domain accessed in baremetal and emulator were different, but the same code path was executed in both cases.

2.3.8 Related work

Android malware analysis

Researchers have proposed several systems to analyze Android malware, obtaining information about them. Enck et al. [28] propose TaintDroid, a dynamic taint analysis system, which tracks sensitive data flow to detect when it is sent over the network. Sun et al. [78] propose TaintART, an approach similar to TaintDroid that works with the most recent Android runtime, ART. DroidBox [26] builds upon TaintDroid and adds tracking of API calls and network data. Spreitzenbarth et al. [75] present Mobile-Sandbox, a system that uses DroidBox and Taintdroid and also includes the use of the *ltrace* tool to monitor native code. Yan and Yin [91] propose DroidScope, a virtual machine introspection-based analysis system that bridges the semantic gap reconstructing OS-level and Java-level semantic views from outside the emulator. AASandbox [18] monitors system calls using a kernel module. Harvester [67] combines program slicing with code generation and dynamic execution to extract runtime values, such as URLs and destination numbers of SMS messages, from obfuscated malware. Bichsel et al. [16] present an approach for deobfuscating apps based on probabilistic learning of large code bases. It learns a probabilistic model over thousands of non-obfuscated apps and use it to deobfuscate new ones. TriggerScope [32] uses static analysis to detect logic bombs, i.e., application logic that is only executed under certain (often narrow) circumstances. TriggerScope is capable of identifying time-, location- and SMS-related triggers.

One of the main drawbacks of dynamic analysis is only being able to observe behavior that is actually executed. This means that the analysis system needs to provide the correct inputs so that the malicious behavior is triggered. Researchers have proposed systems that inspect the analyzed app in order to identify the inputs and paths that lead to the execution of suspicious code and then provide these at runtime [15, 87, 95].

Anti-analysis techniques for Android

Researchers [45, 58, 62, 74, 80] have presented several techniques that may be used by Android malware to evade detection by making static analysis harder or by evading dynamic analysis. Spreitzenbarth [74] details the analysis of two Android malware families, namely Bmaster and FakeRegSMS, that use several anti-analysis techniques, such as waiting some time before executing the malicious actions. Matenaar and Schulz [58] present a method for an app to identify if it is executing inside Qemu, which is the basis of the Android emulator. Vidas and Christin [80] present anti-analysis techniques based on Android APIs, system properties, network information, Qemu characteristics, performance, hardware and software components. Another similar work is presented by Petsas et al. [62]: they demonstrate anti-analysis techniques based on Android APIs, system properties, sensors and Qemu characteristics. Instead of manually identifying differences between real and emulated devices, Jing et al. [45] developed Morpheus, a framework that automatically generates heuristics that can identify, based on files, system properties and Android APIs, whether a sample is running in an emulated environment or not.

Detection of evasive malware

Systems that automatically identify malware samples that employ anti-analysis techniques have been developed for the Windows context [13, 47, 48, 50, 52]. Balzarotti et al. [13] propose a system that records the system calls executed by a sample on a reference environment and replay the monitored system calls on an emulator to identify if the observed behavior is different. Lindorfer et al. [52] analyze malware samples in different environments and identify differences on the observed actions, recognizing techniques that malware apply to detect the analysis environment or analysis tools. Barecloud [48] is a system that dynamically analyzes malware in four different environments, including a baremetal one, and detects evasive malware by comparing the reports provided by these systems in a hierarchical approach. Kolbitsch et al. [50] detect and mitigate malicious programs that wait for some time (stall) before executing their malicious behavior. Malgene [47] combines sequence alignment of system call traces, obtained from a baremetal and an emulated environment, with taint tracking to identify evasion signatures of evasive malware.

2.3.9 Conclusions

In this paper we presented a novel approach to identify evasive Android malware by comparing its execution on a baremetal analysis system and on an emulated analysis system. For each action executed only in baremetal, we identified the reason why it was not successfully performed in the emulated environment, differentiating the cases in which there was evasion from the cases in which there was some analysis problem. Our experiments showed that our approach provides promising results and is more suited for detecting Android malware with anti-analysis features than trying to directly apply existing approaches used to detect evasive Windows malware. We analyzed 1,470 malware samples, from which our technique identified 192 as evasive. We manually analyzed some of the detected ones and discuss the information used by them to evade analysis.

Future work to reduce the false-positive rate includes improving the identification of non-deterministic actions, such as the use of randomly selected domain names. In addition, to automatically identify which information is used to evade analysis, we plan on applying taint-analysis techniques. Code related to anti-analysis might only be triggered right before the malicious actions are executed. Being able to reach this code path will therefore increase our chances of detecting evasive behavior. Finally, we plan on applying some of the techniques proposed in the literature to force apps to follow code paths that lead to suspicious actions, further improving the identification of more types of evasive mobile malware.

Chapter 3

Discussion

The security of Android users is influenced by several layers of protection, which are employed in devices and in app stores. In this thesis we present three papers that demonstrate advances to two different aspects of Android security, namely, malware analysis and native code restriction, contributing to the overall improvement of the users' security. In this chapter we summarize the results presented by these papers, making it easier for the reader to find this information.

In Android, native code components are executed in the same process as the Java part. As such, they can modify at runtime the code that was developed in Java, rendering the results of most static analysis tools for Android unsound, as they can only inspect the original code. Moreover, native code has more capabilities, since it has direct access to system calls, and can be used to launch privilege escalation attacks against the kernel or other processes. To address this problem researchers have proposed separating native code components in a different process and applying restrictions to it. However, the lack of data on the use of native code by benign apps makes the creation of policies that can block attacks but not affect many benign apps a difficult challenge.

To overcome this problem we developed a system capable of monitoring the behavior of native code components deployed in Android apps. We use this system to perform a large-scale analysis of apps and provide several insights on how real-world apps use native code. These insights can help other researchers guide their decisions in regards to restricting native code. For instance, we show that the approach taken by NativeGuard [77] to remove all permissions from native code would negatively affect at least 3,669 apps in our dataset. Furthermore, we provide an approach to automatically generate a native code sandboxing policy in a way that is effective and practical. The policy generated by our approach affects at most a predefined threshold of apps (1% in our experiments), while at the same time blocking actions used by several known root exploits. These results were published in the Network and Distributed System Security Symposium 2016 and presented in Section 2.1 of this thesis.

Another important problem that affects Android security is how to effectively analyze and identify malware. App stores, antivirus companies and security researchers need to analyze large amounts of apps, extracting information about their behavior and identifying the malicious ones. This can be used to vet apps submitted to app stores, to create signatures that can detect malware and to build better analysis tools.

In the paper published in 2015 in the Journal of Computer Virology and Hacking Techniques, Section 2.2 of this thesis, we present a system developed by us that dynamically analyzes Android apps to obtain API function and system calls. Currently available systems are tied to a specific Android OS version or to the SDK-provided emulator, whereas our approach is independent of the emulator and much more portable as it does not modify the Android OS. By using the information gathered from dynamic analysis, our system is able to extract features related to the frequency of use of each API function and each system call, using machine learning to classify apps as malicious or benign. In our experiments, we obtained an accuracy of 96.82%. By comparing our approach to the system presented by Su et al. [76], the most similar in the literature, we demonstrate that the frequency of API calls are good features to detect malicious Android apps.

Dynamic malware analysis systems are typically developed on top of an emulator, because it provides scalability and ease in restoring the analysis environment to a clean state. Consequently, these systems have many differences as compared to real devices; these discrepancies, in turn, may be leveraged by malware to identify when they are being analyzed and prevent the malicious behavior from being observed. One of the main challenges of dynamic analysis is how to create analysis systems transparent to these malware; one important step in doing so is to identify them and the techniques used for their evasion.

In the paper submitted to the International Conference on Dependable Systems and Networks 2017, Section 2.3 of this thesis, we present a novel approach to identify evasive Android malware. We achieve this by comparing the analysis results of a sample in a baremetal environment and in an emulated environment. For each action executed only in the baremetal system, we identify if it was not executed in the emulator due to evasive behavior or due to some analysis problem. We compare our technique with existing approaches that identify evasive Windows malware, demonstrating that ours is more appropriate to the Android context. Moreover, we analyzed 1,470 samples with our technique, identifying 192 with evasive behavior. We manually inspected a subset of them and discuss how they identify the analysis environment.

Chapter 4

Conclusions

The security of Android users is influenced by several layers of protection, which are employed in devices and in app stores. In this thesis we present advances to three different aspects, which overall contribute to improving Android security. Our study on native code use, presented in Section 2.1, can help systems that restrict native code to make devices more robust against malware. Our system to analyze and detect Android malware, presented in Section 2.2, can aid app stores and antivirus companies in identifying malicious apps. Finally, our technique to identify evasive malware, presented in Section 2.3, can help security researchers identify malicious apps that use anti-analysis features.

Future work related to the native code analysis includes expanding the policy to consider the number of calls to certain system calls and providing a way to enforce the policy, which could be done by constructing a new system or by integrating the policy to an existing mechanism, such as SELinux. For the identification of malicious apps, an interesting follow up is to include in the classification network related and statically obtained features. Finally, the identification of evasive malware might benefit from improving the identification of non-deterministic behavior and using taint analysis to automatically identify which information is used by malware to identify that they are being analyzed.

Bibliography

[1] Andromalshare. [Online] Available: http://sanddroid.xjtu.edu.cn:8080/. Accessed on January 25 2017.

[2] Open gapps. [Online] Available: http://opengapps.org/. Accessed on January 25 2017.

[3] Virustotal - free online virus, malware and url scanner. [Online] Available: https://www.virustotal.com/en/. Accessed on January 25 2017.

[4] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. Full version of Tables 5, 6, 7, 8, and 11. [Online] Available: https://github.com/ucsb-seclab/android_going_native. Accessed on January 25 2017.

[5] Vitor M. Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Krue gel, and Giovanni Vigna. Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, February 2016.

[6] Vitor Monte Afonso, Matheus Favero de Amorim, André Ricardo Abed Grégio, Glauco Barroso Junquera, and Paulo Lício de Geus. Identifying android malware using dynamically obtained features. *Journal of Computer Virology and Hacking Techniques*, 11(1):9–17, 2015.

[7] AppBrain. Number of Available Android Applications. [Online] Available: http://www.appbrain.com/stats/number-of-android-apps. Accessed on January 25 2017.

[8] Axelle Apvrille and Ruchna Nigam. Obfuscation in Android Malware, and How to Fight Back. In *Virus Bulletin*, 2014.

[9] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[10] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid:

Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2014.

[11] Elias Athanasopoulos, Vasileios P Kemerlis, Georgios Portokalidis, and Angelos D Keromytis. Nacldroid: Native code isolation for android applications. In *European Symposium on Research in Computer Security*, pages 422–439. Springer, 2016.

[12] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM conference on Computer and Communications Security (CCS)*, 2012.

[13] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient detection of split personalities in malware. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2010.

[14] Leonid Batyuk, Markus Herpich, Seyit Ahmet Camtepe, Karsten Raddatz, A Schmidt, and Sahin Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, pages 66–72. IEEE, 2011.

[15] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. Brahmastra: Driving apps to test the security of third-party components. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*, pages 1021–1036, 2014.

[16] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 343–355. ACM, 2016.

[17] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[18] Thomas Blasing, Leonid Batyuk, A-D Schmidt, Seyit A Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 55–62. IEEE, 2010.

[19] I. Burguera, U. Zurutuza, and S. 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.

[20] Victor Chebyshev and Roman Unuchek. Mobile Malware Evolution: 2013. [Online] Available: `http://securelist.com/analysis/kaspersky-security-bulletin/`

58335/mobile-malware-evolution-2013/. Accessed on January 25 2017, February 2014.

[21] Sen Chen, Minhui Xue, Zhushou Tang, Lihua Xu, and Haojin Zhu. Stormdroid: A streaminglized machine learning-based system for detecting android malware. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 377–388. ACM, 2016.

[22] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services (MobiSys)*, 2011.

[23] Benjamin Davis and Hao Chen. “retroskeleton: Retrofitting android apps”. In *Proceedings of the 11th International Conference on Mobile Systems, Applications and Services (Mobisys)*, pages 25–28, 2013.

[24] Anthony Desnos. Androguard: Reverse Engineering, Malware and Goodware Analysis of Android Applications... and More (Ninja!). [Online] Available: `https://code.google.com/p/androguard/`. Accessed on January 25 2017.

[25] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D.S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Security Symposium*, San Francisco, CA, 2011.

[26] DroidBox. Android application sandbox. Available at `https://code.google.com/p/droidbox/`. Accessed on July 7th 2013.

[27] Karim O Elish, D Yao, and Barbara G Ryder. User-centric dependence analysis for identifying malicious mobile apps. In *Workshop on Mobile Security Technologies*, 2012.

[28] W. Enck, P. Gilbert, B.G. Chun, L.P. Cox, J. Jung, P. McDaniel, and A.N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–6. USENIX Association, 2010.

[29] Rafael Fedler, Marcel Kulicke, and Julian Schütte. Native Code Execution Control for Attack Mitigation on Android. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices (SPSM)*, 2013.

[30] 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 (CCS)*, 2011.

[31] A.P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2011.

[32] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. TriggerScope: Towards Detecting Logic Bombs in Android Apps. In *Proceedings of the IEEE Symposium on Security and Privacy (SSP)*, San Jose, CA, May 2016.

[33] A.P. Fuchs, A. Chaudhuri, and J.S. Foster. Scandroid: Automated security certification of android applications. *Technical report, University of Maryland, `http://www.cs.umd.edu/~avik/projects/scandroidascaa`*, 2009.

[34] Gartner. Gartner says smartphone sales surpassed one billion units in 2014. Available at `http://www.gartner.com/newsroom/id/2996817`. Accessed on August 15 2015., 2015.

[35] Clint Gibler, Jonathan Crussel, Jeremy Erickson, and Hao Chen. AndroidLeaks: Detecting Privacy Leaks in Android Applications. Technical report, Tech. rep., UC Davis, 2011.

[36] Google. Android NDK. [Online] Available: `https://developer.android.com/tools/sdk/ndk/index.html`.

[37] Google. UI/Application Exerciser Monkey | Android Developers. [Online] Available: `http://developer.android.com/tools/help/monkey.html`. Accessed on January 25 2017.

[38] Rob Gordon. *Essential JNI: Java Native Interface.* Prentice-Hall, Inc., 1998.

[39] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, 2012.

[40] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 281–294. ACM, 2012.

[41] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[42] Tsung-Hsuan Ho, Daniel Dean, Xiaohui Gu, and William Enck. PREC: Practical Root Exploit Containment for Android Devices. In *Proceedings of the 4th ACM conference on Data and application security and privacy (CODASPY)*, 2014.

[43] IDC. Smartphone os market share, 2016 q2. [Online] Available: `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`. Accessed on January 25 2017.

[44] IDC Corporate. IDC: Smartphone OS Market Share 2014, 2013, 2012, and 2011. [Online] Available: `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`. Accessed on January 25 2017.

[45] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. Morpheus: automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 216–225. ACM, 2014.

[46] JuniperNetworks. Juniper networks mobile threat center third annual mobile threats report: March 2012 through march 2013. Available at `http://www.juniper.net/us/en/local/pdf/additional-resources/3rd-jnpr-mobile-threats-report-exec-summary.pdf`. Accessed on 20 August 2013., 2013.

[47] Dhilung Kirat and Giovanni Vigna. Malgene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 769–780. ACM, 2015.

[48] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: bare-metal analysis-based evasive malware detection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 287–301, 2014.

[49] Patrick Klinkoff, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Extending .NET Security to Unmanaged Code. *International Journal of Information Security*, 2007.

[50] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 285–296. ACM, 2011.

[51] Yuanchun Li. Droidbot. [Online] Available: `http://honeynet.github.io/droidbot/`. Accessed on January 25 2017.

[52] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting Environment-Sensitive Malware. In *Recent Advances in Intrusion Detection (RAID) Symposium*, 2011.

[53] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proceedings of the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[54] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, 2012.

[55] Luka Malisa, Kari Kostiainen, Michael Och, and Srdjan Capkun. Mobile application impersonation detection using dynamic user interface extraction. In *European Symposium on Research in Computer Security*, pages 217–237. Springer, 2016.

[56] Christopher Mann and Artem Starostin. A Framework for Static Detection of Privacy Leaks in Android Applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC)*, 2012.

[57] Claudio Marforio, Ramya Jayaram Masti, Claudio Soriente, Kari Kostiainen, and Srdjan Capkun. Hardened setup of personalized security indicators to counter phishing attacks in mobile banking. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 83–92. ACM, 2016.

[58] Felix Matenaar and Patrick Schulz. Detecting android sandboxes. Available at `http://www.dexlabs.org/blog/btdetect`. Accessed on 20 August 2013., 2012.

[59] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 421–430. IEEE, 2007.

[60] Simone Mutti, Yanick Fratantonio, Antonio Bianchi, Luca Invernizzi, Jacopo Corbetta, Dhilung Kirat, Christopher Kruegel, and Giovanni Vigna. Baredroid: Large-scale analysis of android apps on real devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 71–80. ACM, 2015.

[61] Mila Parkour. Contagio mobile. [Online] Available: `http://contagiominidump.blogspot.com/`. Accessed on January 25 2017.

[62] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, page 5. ACM, 2014.

[63] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2014.

[64] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid Android: Versatile Protection for Smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, pages 347–356, 2010.

[65] PulseSecure. 2015 mobile threat report. [Online] Available: `https://www.pulsesecure.net/lp/mobile-threat-report-2014/`. Accessed on January 25 2017.

[66] Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin TS Chan. On Tracking Information Flows through JNI in Android Applications. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[67] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[68] Vaibhav Rastogi, Yan Chen, and William Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proceedings of the third ACM conference on Data and application security and privacy (CODASPY)*, 2013.

[69] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of the 6th European Workshop on System Security (EUROSEC)*, Prague, Czech Republic, April 2013.

[70] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo Garcia Bringas, and Gonzalo Alvarez. Puma: Permission usage to detect malware in android. In *CISIS/ICEUTE/SOCO Special Sessions*, pages 289–298, 2012.

[71] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. Adsplit: separating smartphone advertising from applications. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, Berkeley, CA, USA, 2012. USENIX Association.

[72] Joseph Siefers, Gang Tan, and Greg Morrisett. Robusta: Taming the Native Beast of the JVM. In *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS)*, 2010.

[73] AIRBUS Defense & Space. Local Root Vulnerability in Android 4.4.2. [Online] Available: `http://blog.cassidiancybersecurity.com/post/2014/06/Android-4.4.3,-or-fixing-an-old-local-root`. Accessed on January 25 2017.

[74] Michael Spreitzenbarth. The Evil Inside a Droid - Android Malware: Past, Present and Future. In Estonian Forensic Science Institute, editor, *Proceedings of the 1st Baltic Conference on Network Security & Forensics*, pages 41–59, 2012.

[75] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1808–1815. ACM, 2013.

[76] X. Su, M. Chuah, and G. Tan. Smartphone dual defense protection framework: Detecting malicious applications in android markets. In *Mobile Ad-hoc and Sensor Networks (MSN), 2012 Eighth International Conference on*, pages 153–160, 2012.

[77] Mengtao Sun and Gang Tan. NativeGuard: Protecting Android Applications from Third-Party Native Libraries. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks (WiSec)*, 2014.

[78] Mingshen Sun, Tao Wei, and John Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 331–342. ACM, 2016.

[79] Roman Unuchek and Victor Chebyshev. Mobile malware evolution 2015. [Online] Available: `https://securelist.com/analysis/kaspersky-security-bulletin/73839/mobile-malware-evolution-2015/`. Accessed on January 25 2017.

[80] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458. ACM, 2014.

[81] Nicolas Viennot, Edward Garcia, and Jason Nieh. A Measurement Study of Google Play. In *Proceedings of the 2014 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2014.

[82] VirusShare. Virusshare bittorrent client tracker. [Online] Available: `http://tracker.virusshare.com:6969`. Accessed on January 25 2017.

[83] VRT. Changing the imei, provider, model, and phone number in the android emulator. Available at `http://vrt-blog.snort.org/2013/04/changing-imei-provider-model-and-phone.html`. Accessed on July 07 2013., 2013.

[84] Christina Warren. Google Play Hits 1 Million Apps. [Online] Available: `http://mashable.com/2013/07/24/google-play-1-million/`. Accessed on January 25 2017, July 2013.

[85] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[86] Lukas Weichselbaum, Matthias Neugschwandtner, Martina Lindorfer, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. ANDRUBIS: Android Malware Under The Magnifying Glass. Technical Report TR-ISECLAB-0414-001, Vienna University of Technology, 2014.

[87] Michelle Y Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[88] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Seventh Asia Joint Conference on Information Security (Asia JCIS)*, 2012.

[89] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[90] Carter Yagemann and Wenliang Du. Intentio ex machina: Android intent access control via an extensible application hook. In *European Symposium on Research in Computer Security*, pages 383–400. Springer, 2016.

[91] Lok Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX conference on Security symposium*. USENIX Association, 2012.

[92] Zhemin Yang and Min Yang. Leakminer: Detect Information Leakage on Android with Static Taint Analysis. In *Proceedings of the 2012 Third World Congress on Software Engineering (WCSE)*, 2012.

[93] Lingyun Ying, Yao Cheng, Yemian Lu, Yacong Gu, Purui Su, and Dengguo Feng. Attacks and defence on android free floating windows. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 759–770. ACM, 2016.

[94] Zhibo Zhao and Fernando C Colon Osono. “TrustDroid$^{\text{TM}}$”: Preventing the Use of SmartPhones for Information Leaking in Corporate Networks Through the Use of Static Analysis Taint Tracking. In *Proceedings of the 2012 7th International Conference on Malicious and Unwanted Software (MALWARE)*, 2012.

[95] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104. ACM, 2012.

[96] Min Zheng, Mingshen Sun, and John CS Lui. Droidanalytics: A signature based analytic system to collect, extract, analyze and associate android malware. In *Proceedings of The 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 13)*, 2013.

[97] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[98] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, 2012.

[99] Yajin Zhou and Xuxian Jiang. Detecting Passive Content Leaks and Pollution in Android Applications. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, 2013.

[100] Ziyun Zhu and Tudor Dumitras. Featuresmith: Automatically engineering features for malware detection by mining the security literature. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 767–778. ACM, 2016.