Analysis and Classification of Android Malware

Kimberly Kim-Chi Tam

Information Security Department

Royal Holloway, University of London

A thesis submitted for the degree of

Doctor of Philosophy

2016

Acknowledgements

First I would like to acknowledge my supervisor and advisor, Dr. Lorenzo Cavallaro and Professor Cid Carlos, for the amazing opportunity to do my PhD at Royal Holloway University of London. It would not have been possible without you two.

I would also like to acknowledge my fellow Royal Holloway University of London lab members and external collaborators for their partnership in collaborative work. In particular I would like to thank Salahuddin Khan for his technical expertise, his professionalism, and his friendship.

Similarly, I feel much gratitude towards Professor Kevin Jones and Dr. Warrick Cooke for their extensive advice for both fine-tuning this thesis in detail and the overall story and flow.

Parts of this research was funded by UK EPSRC grant EP/L022710/1. I would also like to thank Hewlett-Packard in Bristol for the internship opportunity during my PhD as well as all of their support since then.

Finally, I want to acknowledge my family and friends scattered across the globe. Organizing opportunities to connect has been a real challenge over the last few years, but we always managed it. So thank you all for the fun and help throughout the good times and bad. I could not have gotten this far without your awesomeness.

"One does not discover new lands without consenting to lose sight of the shore for a very long time."

Andre Gide

Abstract

With the integration of mobile devices into our daily lives, smartphones are privy to increasing amounts of sensitive information. As of 2016, Android is the leading smartphone in popularity with sophisticated mobile malware targeting its data and services. Thus this thesis attempts to determine how accurate and scalable Android malware analysis and classification methods can be developed to robustly withstand frequent, and substantial, changes within the Android device and in the Android malware ecosystem.

First, the author presents a comprehensive survey on leading Android malware analysis and detection techniques, and their effectiveness against evolving malware. Through the systematized survey, the author identifies underdeveloped areas of research which lead to the development of the novel Android malware analysis and classification solutions within in this thesis.

This thesis considers the usefulness and feasibility of reconstructing high-level behaviours via system calls intercepted while running Android apps. Previously, this method had only been rudimentarily implemented. However, the author was able to remedy this and developed a robust, novel, framework, to automatically and completely reconstructs all Android malware behaviours by thoroughly analysing dynamically captured system calls.

Next, the author investigates the efficacy of using our reconstructed behavioural profiles, at different levels of abstractions, to classify Android malware into families. Experiments in this thesis show our reconstructed behaviours to be more effective, and efficient, than raw system call traces. To classify malware, we utilized support vector machines to achieve high accuracy, precision and recall. Deviating from previous methods, we further apply statistical classification to achieve near-perfect accuracies.

Finally, the author explores an alternative Android malware analysis method using memory forensics. By extrapolating from these experiments, the author theorizes how to use this method to assist in capturing behaviours our previous methods could not, and how they could assist classification.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

This chapter introduces Android malware threats and solution traits that this thesis aims to encompass, as well as the distribution of any collaborative work. The essence of the author's observations and contributions illustrated within this thesis, toward the analysis and classification of Android malware, can be reduced to the following thesis statement:

Low-level system data produced by Android applications can be used to accurately, and scalably, characterize malware whilst remaining agnostic to significant device changes.

1.1 Android Malware Threat

Smartphones, tablets, and other mobile platforms have become ubiquitous due to their highly personal and powerful attributes. As the current dominating personal computing device — mobile device shipments surpassed PCs in 2010 [146] — smartphones have spurred an increase of mobile malware. As of Q4 2014, over six million mobile malware samples have been accumulated by McAfee. This represents a 14% increase from the previous quarter, with over 98% samples targeting the prevalent Android platform [143]. Furthermore, with more than a billion Android-activated devices and monthly-active users [83, 120], the majority of smartphone users are vulnerable to Android malware.

Application marketplaces, such as Google Play, drive the entire economy of mobile applications (apps). For instance, with more than 50 billion downloaded apps [194], Google Play has generated revenues exceeding 5 billion USD [96] in 2013. Such a wealthy and unique ecosystem, with high turnovers and access to sensitive data, further spurs the alarming growth of Android malware; most of which are also growing in sophistication. Privacy breaches (e.g., access to address book or GPS coordinates) [250], monetization through premium SMS and calls [250], and colluding malware to bypass 2-factor authentication schemes [56], have become real threats. Recent studies also report how easily mobile marketplaces have been abused to host malware or seemingly legitimate apps embedding malicious components (e.g., DroidMoss [248]).

1.2 Desirable Solution Traits

Mobile hardware and the Android operating system (OS) are still in a state of considerable growth and change, exemplifiedby the complete replacement of the Dalvik runtime with the new ART runtime in Android 5.0, released in November 2014 [218]. Despite such a significant modification, the frameworks presented in this thesis have remained robust to such changes. In contrast, majority of related approaches are more dependent on certain aspects of the Android architecture and are, hence, less applicable to a wide range of Android versions. This is undesirable, as Android is continuously releasing new, open-source, versions (Table 1.1). Furthermore, the Android OS is normally modified or customized before being shipped and sold by different providers within different physical devices. To remain effective over the range of available Android OS versions, our methods have been intentionally designed to be robust against version changes.

With the rapid growth of mobile malware (and the steady introduction of newer, more sophisticated, Android malware), another important aspect of new analysis and classification techniques must be scalability. By achieving high scalability, our methods can process large amounts of malware samples while retaining high accuracy, coverage, and detail. We have found that the author's methods for high level abstraction, discussed in this thesis, also greatly increase the scalability of analysis and classification. Although many simplification methods have been used by other methods to increase scalability, our method of abstraction is novel and unique to our frameworks.

Table 1.1: Frequency of Android OS version releases (x.y.[0,1] translates to version x.y and x.y.1).

Android Version	**2008**	**2009**	**2010**	**2011**	**2012**	**2013**	**2014**	**2015**
	1.0	1.1						
Cupcake		1.5						
Doughnut		1.6						
Eclair		2.0, 2.0.1	2.1					
Froyo			2.2	2.2.[1-3]				
Gingerbread			2.3.[0,1]	2.3.[2-7]				
Honeycomb				3.[0-2], 3.2.[1-4]	3.2.[5,6]			
Icecream Sandwich				4.0.[0-3]				
Jelly Bean					4.1.[0-2], 4.2, 4.2.1	4.2.2, 4.3.[0,1]		
KitKat						4.4.[0,-2]	4.4.[3,4]	
Lollipop							5.0.[0-2]	5.1, 5.1.1
Marshmallow								6.0, 6.0.1

1.3 Organization of the Thesis

The intent of these brief introductions to the following chapters is to illustrate the connections between the bodies of work produced by the author. In any case, to better supplement the descriptions in this section, each of the following chapters will contain its own detailed introduction with addition relevant information, research goals, motivation, in-depth descriptions of contributions, related works, and results.

In Chapter 2 the author presents a thorough survey on the state of existing studies relating to Android analysis, detection, and classification. This survey gives an introduction to the Android device, key words and concepts, and spans several years in order to evaluate technique advancements, overall progression, and any remaining weaknesses.

The author's contributions are then presented in the following chapters. In Chapter 3 the author presents a new enhanced version of CopperDroid, a framework to automatically implement unified dynamic analysis on Android malware, collect the resulting system calls, and fully reconstruct all behaviours. The work presented in this chapter, primarily the author's key effort in fully adapting traditional system call analysis to Android and recreating behaviours from system calls, appeared in the conference NDSS 2015 with the author listed first on this publication [202].

In Chapter 4, we address the multi-class classification problem using CopperDroid's reconstructed behaviours. There are two components to this framework: (1) a traditional support vector machine classifier, and (2) a conformal predictor that is used in conjunction to create a novel hybrid classification tool. Furthermore, by analysing CopperDroid's reconstructed behaviours (developed by the author) instead of raw system calls, we were able to improve performance without sacrificing accuracy.

In Chapter 5 we present a supplementary or alternative analysis method that may be further applied to Android malware detection and classification. Although the memory forensics approach differs from the content of previous chapters, it shares the same fundamental goals of accuracy, robustness, and scalability. As an alternative method explored, the available feature set is not always as detailed as our previous methods, but can theoretically provide essential features and behaviours difficult or impossible to gain via CopperDroid. In particular, such a collaboration may be necessary for detecting evasive malware such as bootkits, which only reside in certain partitions of memory [130]. Similarly, the added behavioural information can aid in malware classification.

In the final chapter, Chapter 6, we conclude by summarizing the author's research work, contributions, and achievements, and by listing various areas of future research that may be built on the work achieved in this thesis.

1.4 Declaration of Authorship for Co-Authored Work

This section serves as an overview of the distribution of work that went into this thesis. This is necessary as some of the author's work is built on tools primarily developed by collaborators and some work has been done in collaboration. All division of labour will be reiterated throughout the body of the thesis, but the purpose of this section is to provide a quick overview and explanation. Further details of the work distribution may be found in Table 1.2 which is organized by chapters and sections.

The background and survey chapter, Chapter 2, is primarily the work of the author with the exception of a few collaborative subsections.

Chapter 3, depicts the work of the author recreating malware behaviours from low-level events. This research was built on a pre-existing tool that collects system calls. Performance measurements of this tool were performed by its developers, while the performance and results on the author's contribution were performed by the author.

In Chapter 4 the author maps reconstructed behaviours, created by CopperDroid, into a vector space to classify malware samples. This builds on the author's work in the previous chapter, and the results are considerably better than baseline results by a collaborator. The purpose of using machine learning was not primarily to develop novel classification techniques, but to demonstrate the usefulness of the author's feature set.

In order to improve our traditional support vector machine classifier (SVM), set up by a collaborator, the author and collaborator built a conformal predictor (CP) to assist with difficult choices. Unlike the traditional SVM, this hybrid solution is novel. Specifically, the collaborator calculated several constants and the collaborator's machine generated the images shown in Figures 4.6 - 4.11. However, the statistics that went into these figures were primarily generated with the author's work. The author's tools calculated misclassification statistics, feature statistics, SVM precision, and SVM recall per sample, family and overall dataset. Furthermore, the author implemented the core decision components of the novel conformal predictor (i.e., prediction sets) and performed calculations for the improved accuracy, precision, and recall accordingly. Implementation details will be further explained in the following chapters.

The content of the second to last chapter, Chapter 5, was developed purely by the author. Apart from using two, pre-existing, memory forensics tools, clearly stated in the chapter, all research was conceived and executed solely by the author. As this content is not directly tied to that of the previous chapters, this should be considered as an alternate method developed that theoretically has attractive complementary applications to the work in previous chapters.

Table 1.2: Declaration of authorship for co-authored work.

Section	Contributors
Ch 2	Thesis author (⌣93%)
2.1.4 (p. 24-27)	Malware statistics were created with Ali Feizollah (wrote a script) and Lorenzo Cavallaro (only individual with access to all malware samples). The author created tables and made extrapolations from the data.
2.2.4.10 (p. 41)	App metadata section written by Ali Feizollah, modified by the author.
2.2.5 (p. 42)	Feature selection content laid down by Ali F., simplified by the author.
2.4.1.4 (p. 50)	Dataset discussion written by author, assisted by Ali Feizollah.
Ch 3	Thesis author (⌣88%)
3.3.2 (p. 65)	System call tracking was initially created by Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro and then re-done by Salahuddin Khan.
3.4.1 (p. 66)	Pre-analysis AIDL parser written by Salahuddin Khan.
3.5 (p. 76-78)	Simple behaviour reconstruction (e.g., based on one system call) primarily developed by Aristide/Alessandro. More complex behaviours and analyses solely developed by the author.
3.5.2 (p. 79-83)	Full or no app stimulation written by Alessandro/Aristide. Fine-grained analysis of stimuli effects primarily led by the author with the help of Salahuddin Khan (see Table 3.4).
3.6.2 (p. 84)	Evaluation of system call collection tool done by Aristide Fattori, while evaluations of the Oracle were done by the author.
Ch 4	Thesis author (⌣85%)
4.4.2 (p. 99)	Santanu Dash: 2 modes dealing with raw system calls for a baseline. Author: 4 operational modes dealing with reconstructed behaviours.
4.4.3 (p. 100)	Santanu Dash extract some data from CopperDroid files. Author extracted remaining from data and analysed to make behaviour sets/subsets, and analysed additional sample files, e.g. recreated files for file type.
4.5.1 (p. 103-104)	Author mapped features to vectors, Santanu set up a standard SVM classifier (not one of the claimed contributions).
4.5.3 (p. 105-106)	Santanu calculated P-values. The author used these values for conformal prediction (set of choices) and developed tools to calculate precision/recall/accuracy before and after conformal prediction.
4.6.2 (p. 108-116)	Author developed tools to calculate statistics. Experiment numbers and graphs mainly generated on collaborator's machine and Santanu set up system call baseline to demonstrate improvements with author's work on CopperDroid behaviours and the unique application of CP.
Ch 5	Thesis author (100%)
5.1-5.6 (p. 124-157)	All work, minus referenced related works and two pre-existing tools used (repeatedly declared in chapter), developed by author.

Chapter 2

Background

Contents

2.1 Introduction

The goal of this survey [1] is to understand analysis, detection, and classification methods for the Android operating system (OS) on Android-specific devices. First, the author provides some background information on mobile devices, their evolution, and their characteristics. Secondly, the author presents a comprehensive study on an extensive, and diverse, set of Android malware analysis frameworks including method, year, and outcome. The author then compares similar studies to identify evolving state-of-the-art techniques and attempts to determine their general strengths, weaknesses, performance, and uses. This was essential in determining novel areas of research, several of which the author explores in the following chapters. The author further discusses the effectiveness of techniques against major changes in Android malware and the Android system.

Next this chapter addresses several Android malware countermeasures used to obstruct, or evade, analysis. It classifies and describes transformation attacks and examine advanced malware obfuscation techniques, such as encryption, native exploits, and VM-awareness. With that knowledge, we determine effective techniques implemented by both malware and analysis methods by comparing malware strengths to common analysis weaknesses. We enhance these findings with malware statistics we gathered from several available datasets along with statistics from previous studies.

Lastly, the author supports and justifies several directions of future research, including those pursued in this thesis, and highlight issues that may not be apparent when looking at individual studies. This chapter does not focus on general mobile attack vectors [28, 71, 196], but focuses on Android. Furthermore, we primarily focus on the aspects of malware that have the most negative affect on analysis, detection, and classification (i.e., hindrances, sabotage), although we still discuss aspects like market infections. By doing so, we can provide in-depth studies on both sides of the race-for-arms.

This chapter is constructed in the following manner. In the remainder of the section, we will analyse the history of mobiles and mobile malware (with an emphasis on Android), give a short background on Android itself, and present malware statistics we have gathered. Section 2.2 discusses the taxonomy of mobile malware analyses divided into static, dynamic, and hybrid methods (Sections 2.2.1-2.2.2). Sections 2.2.4 and 2.2.5 evaluate the wide range of state-of-the-art analysis techniques and their feature selection methods. Section 2.2.6 then describes detection, classification, and security systems that can be built on analysis frameworks. In Section 2.3, we delve into the evolution of malware countermeasures, elaborating on methods to evade detection systems.

[1] accepted for publication in ACM Computing Surveys (early 2016) pending minor revisions

This leads to an extensive discussion presented in Section 2.4 to systematically analyse the state of current research and help shape the future of malware detection. Here, we conclude by addressing several questions on what the growing malware threat is, how we successfully address those problems today, and what future research directions we should pursue in order to develop new methods for improving Android malware analysis, classification, and detection. To highlight several key works in Section 2.4 and illustrate trends in literature spanning 2011 to 2015, we also present Appendix A.

2.1.1 Evolution of Malware

Initially, when computing systems were primarily understood by a few experts, malware development was a test of one's technical skill and knowledge. For example, the PC Internet worm known as Creeper displayed taunting messages, but the threat risk (e.g., stolen data, damaged systems) was considerably low. However, as time progressed from the 1980's, the drive to create malware became less recreational and more profit-driven as hackers actively sought sensitive, personal, and enterprise information. Today, malware development is much more lucrative and often aided by malware developing tools. This, in part, resulted in over a million PC malware samples, well before smartphones had even taken off; as of 2009, less than 1,000 mobile malware were known [67]. Since 2009, however, the rise of mobile malware has been explosive, with new technologies providing new access points for profitable exploitations [142, 144].

In 2013 a report showed that attackers can earn up to 12,000 USD per month via mobile malware [173]. Moreover, an increase in black markets (i.e., markets to sell stolen data, system vulnerabilities, malware source code, malware developer tools) has provided more incentive for profit-driven malware [106]. Although we may borrow and adapt traditional PC solutions, the basic principles of mobile security differs due to inherently different computing systems. Furthermore, despite improvements to their computing power and capabilities, mobile devices still possess relatively limited resources (e.g., battery) which limits on-device analysis.

2.1.1.1 Mobile versus Traditional Devices, Malware, and Analysis

There are several key differences between mobile and traditional devices as well as application acquisition that contribute to the variances in malware and, hence, the analysis of that malware. As mobiles are constantly crossing physical and network domains, they are exposed to more infection venues than traditional PCs. For example, by exploiting

their host's physical movements, mobile worms are capable of propagating across network domains more easily [177]. Additionally, with over a million available apps and near instantaneous installation, mobile devices are subjected to a high turnover of potentially malicious software [123]. Third-party app markets with less protection also increase infection probability in mobile devices compared to traditional devices. This may be due to the difficulties mobile markets face, as Android malware are often repackaged versions of legitimate apps (86% of them in [250]) and therefore harder to identify.

Smartphones also accept a wide set of touch commands, such as swipe and tap, which is unlike the traditional mouse and keyboard input. This added complexity can complicate analysis, as it is harder to autonomously traverse execution paths (see Section 2.3). Mobile devices are also accessible, and vulnerable, through multiple (sometimes simultaneous) connections, such as email, WiFi, GPRS, HSCSD, 3G, LTE, Bluetooth, SMS, MMS, and web browsers. Smartphones also utilize a complex plethora of technologies such as camera, compass, Bluetooth, and accelerometers, which are also vulnerable, e.g. via drivers [239]. New exploits and variants of traditional malware are thus possible by adapting to mobile technologies. Furthermore, as battery life is a larger concern for mobile devices, anti-virus and analysis software face more user scrutiny.

As an attack, an alarming number of Android mobile malware send background SMS messages to premium rate numbers to generate revenue (similar malware still affect PCs via phone lines). Although attempts to mitigate this have been made in Android OS 4.3, released in 2012, more robust solutions such as AirBag [230] are still necessary. This is evident as background SMS are still considered a high risk event by users [80], and since a considerable number of malware targeting Android still exhibit this behaviour [144].

For example, it was estimated that over a thousand devices were affected with one, particular, malicious version of the Angry Birds game. Once installed, the malware secretly sent premium SMS each time the game was started, costing roughly 15 GBP per text [191]. This is just one example of how, since 2010, the number of profit-driven malware for both for traditional and mobile devices has surpassed the number of non-profit driven malware [76]. Furthermore, this the gap continues to grow steadily.

2.1.1.2 Android Popularity and Malware

Based on a report from F-Secure, Android contributed 79% of all mobile malware in 2012, compared to 66.7% in 2011 and 11.25% in 2010 [76]. In accordance with this pattern, Symantec determined that the period from April 2013 to June 2013 witnessed an Android malware increase of almost 200%. Furthermore, Android malware now represents over 95% of more than 12 million mobile malware samples as of Q4 2015 [144].

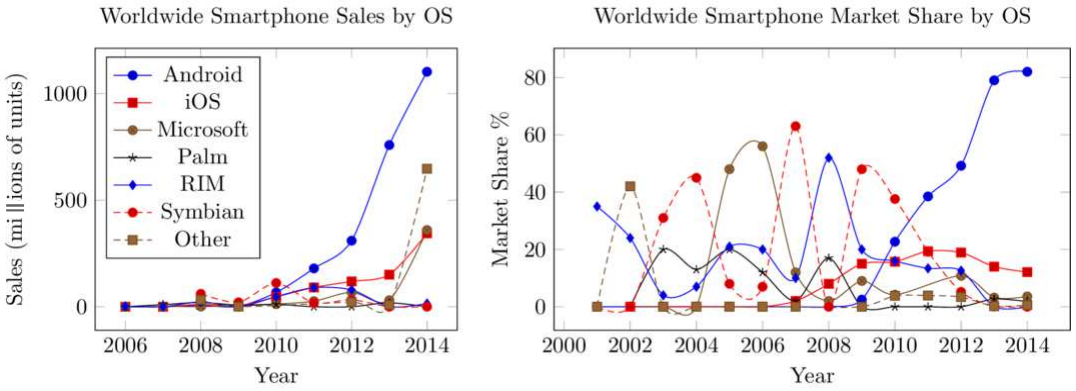


Figure 2.1: Worldwide Smartphone Sales by Operating System (OS) from 2006 to end of 2014.

Furthermore, in February 2014, Symantec stated that an average of 272 new malware and five new malware families targeting Android were discovered every month [199].

One of the prime contributing factors to this immense malware growth is Android's popularity (Figure 2.1), its open-source operating system [203], and its application markets. This includes the official Google Play, which has some vetting processes, as well as "unofficial" third party markets across the world (e.g., SlideME [188]). In general, third party markets have higher infection rates than Google Play, but not all countries have had access to the official market since its introduction (e.g., China). Looking towards 2015 and beyond, it is possible that Google will be adopting manual approaches for vetting apps in an attempt to lower malware existence on the Google Play [161].

Currently, the popularity of Android devices makes it a desirable target. However, its popularity is relatively recent, as illustrated in Figure 2.1. Its popularity begun roughly in 2010, as shown by the statistics provided by Canalys (2001-2004) and Gartner (2005-2014) [86]. Interestingly, this figure also depicts a sizeable dip in Symbian market shares in 2005, which may be the result of the first mobile worm, Cabir, discovered in 2004 and designed for Symbian [92]. Figure 2.1 also demonstrates why certain studies spanning 2000-2008 focus entirely on Symbian and Windows mobile malware threats; they were the most popular operating systems during that period [21, 70].

As general smartphone sales rose dramatically in 2010, several alternatives rose to compete with Symbian. Studies such as [125] and [81] reflected this shift by including emerging OSs such as Android and iOS, but by 2012 Android began to clearly dominate. Studies then began to focus purely on Android as Android malware sky-rocketed [200, 250]. Furthermore, just as the sophisticated Cabir worm targeted Symbian when it was the most popular in 2004, the Trojan Obad, considered one of the most sophisticated mobile Trojans today, was discovered in 2013 and targets Android [209].

In general, nearly half of all mobile malware today are Trojans, and are tailored to

target specific demographics. Together, Russia, India, and Vietnam account for over 50% of all unique users attacked worldwide [183], while USA infections, as determined with three months of DNS traffic, is less than 0.0009% [127]. However, this method indirectly measured domain-name resolution traces and may not be entirely accurate. At the end of 2014, McAfee also analysed regional infections rates of devices running their security products. They found the infection rates in Africa and Asia were roughly 10%, while Europe and the Americas had rates of 6-8%. Further discussions on varying infection rates due to geological and virtual market factors can be found in Section 2.4.

2.1.2 Android Architecture

As the most popular and predominant mobile operating system, this thesis focuses on Android as opposed to the alternative mobile platforms shown in Figure 2.1 (e.g., iOS). The open-source Android OS was initially released in 2008, runs on top of a modified Linux kernel, and runs all Java written applications in isolation. Normally, this means all apps are run separately within their own Dalvik virtual machines, but with the release of Android 5.0 in 2014, the Dalvik just-in-time compiler was replaced with an ahead-of-time compiler, ART. As we will discuss further on, this change has negatively affected many current, state-of-the-art analysis frameworks, but not those in this thesis.

The Android hardware consists of a baseband ARM processor [15] (future tablets may use the Intel x86 Atom), a separate application processor, and devices such as GPS and Bluetooth. ARM is the standard CPU for embedded systems, i.e. smart phones. The appeal of these CPUs are low-power consumptions, high-code density, good performance, small chip size, and low-cost solutions. In detail, ARM has a 32-bit load-store architecture with 4-bytes instruction length and 18 active registers (i.e., 16 data registers and 2 processor status registers). These are important for system call interception (see Chapter 3). Each processor mode has its own banked registers (i.e., a subset of the active registers) which get replaced during mode changes. Specifically, there is one non-privileged mode, `user`, and six privileged modes `abort`, `fast interrupt request`, `interrupt request`, `supervisor`, `system` and `undefined`.

In order to access the system, all apps must be granted permissions by the Android Permission System during installation. Several studies evaluating the effectiveness of Android permissions (more in Section 2.2) can be found in [19, 20, 79, 223]. Once installed, i.e. permissions granted, apps can interact with each other and the system through well-defined, permission protected, API calls, which are enforced by the kernel. Unfortunately, this also applies to anti-virus apps, preventing these products from introspecting other apps. Hence, most anti-virus solutions are signature-based and may

be more viable implemented in markets instead of on-device (e.g., [47,251]). Figure 2.2 gives an overview of the Android architecture.

Android apps themselves are comprised of a number of activity, broadcast receiver, service, and content providers components (see Figure 3.2). Content providers manage access to structured sets of data by encapsulating them for security mechanisms, while the other three are activated by `Intents`. The Android `Intent` is an abstract description of an operation, or task, one component requests another component to do, and is communicated with asynchronous messages. While broadcast receivers and services tend to run in the background, activities are the most visible component to the user, and is often what handles user interactions like button clicking.

2.1.3 Notable Android Malware

There have been many malware families discovered from 2011 to 2015, but there have been a few pivotal samples we wish to mention upfront. These sophisticated samples may exhibit characteristics already seen in traditional malware, but are new — perhaps even the first of its kind — in the mobile area. The majority of these samples have also been discovered between 2014 and 2015, showing that mobile malware are, in some ways, catching up to traditional malware.

The Android malware NotCompatible.C infected over 4 million Android devices to send spam emails, buy event tickets in bulk, and crack WordPress accounts [195]. Furthermore, this malware is self protecting through redundancy and encryption, making static analysis very difficult. Conversely, malware such as Dendroid, Android.hehe, and BrainTest are more difficult to analyse dynamically, as they are aware of emulated surroundings (details in Section 2.3), and have evaded Google Play's vetting processes. The

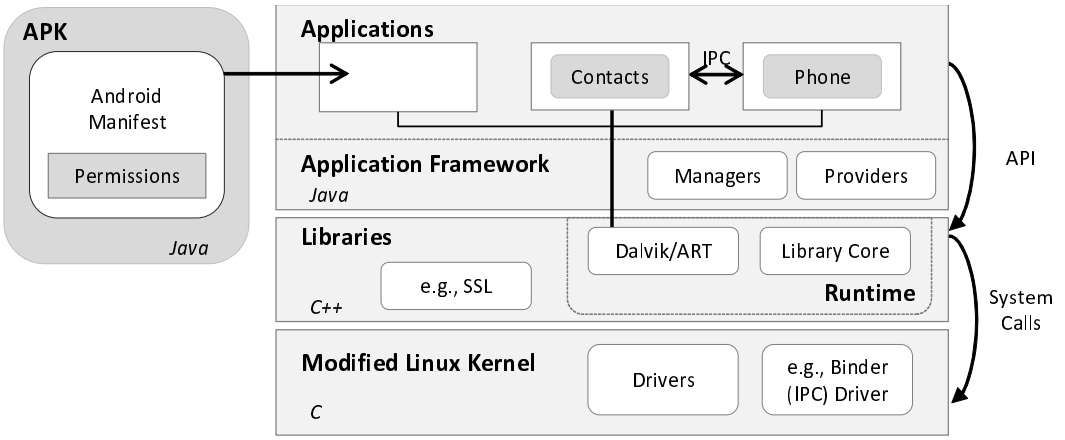


Figure 2.2: Overview of the Android Operating System (OS) Architecture.

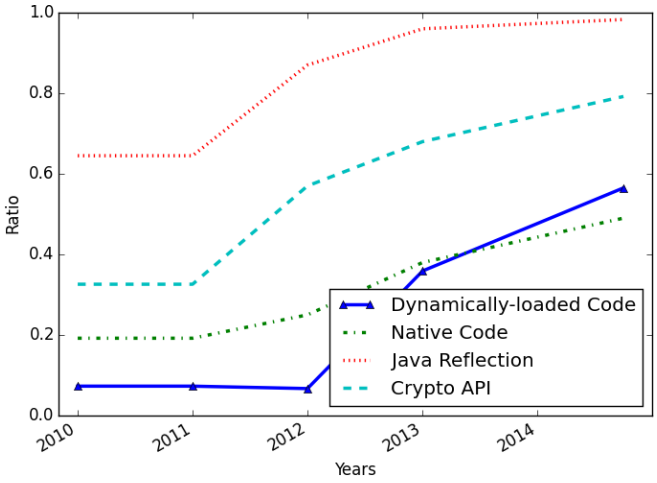


Figure 2.3: Evolution of Android malware using obfuscating techniques (e.g., cryptography APIs).

last notable Android malware we wish to mention is the first Android bootkit, which can evade anti-virus (AV) products as it only exists in the boot partition. Memory analysis may be necessary to analyse the malware, such as Oldboot, that can only be found in volatile memory [130]. Although we had no access to Oldboot, the author has explored using memory forensics for detecting Android malware in Chapter 5.

2.1.4 Statistics for Android Malware Evolution

As one of the contributions of this chapter, we demonstrate one aspect of the Android malware evolution from 2010 to 2015. Collaborators ran basic permission experiments on the malware and the author analysed the results. Our earliest dataset from 2010-2012 is made of 5,560 Android malware samples provided by the Drebin project [16], including those previously studied as part of the Android Malware Genome Project [249]. The latter dataset comes from a live telemetry of more than 3,800 Android malware—704 samples in 2012, 1,925 in 2013, and 1,265 in 2014—that were detected in the wild. [2].

Android Malware Obfuscation: Overall, we automatically analysed more than 9,300 Android malware samples to understand how the malware threat evolved in terms of used dynamically-loaded code (i.e., ability to, at run time, load and a library or binary into memory), Java reflection (i.e., a way for an object to inspect itself), native code invocation (i.e., code that is compiled to run with a particular processor), cryptography APIs, and top used permissions. Table 2.1 shows the permission rankings found in our analyses. We then examined the implication of such trends on the state-of-the-art techniques

[2]Due to confidentiality agreements, we cannot redistribute these McAfee Android malware samples.

and how it influences future research. To date, a great deal of static analysis methods have been created to understand, and mitigate, Android malware threats. However, trends show an increase in the usage of dynamically-loaded code and Java reflection, as depicted in Figure 2.3. Such features hinder the effectiveness of static analysis and call for further research on robust hybrid or dynamic analysis development [233, 241].

Although dynamic analysis is more robust against the use of dynamically-loaded code and Java reflection, its effectiveness is often limited by its limited code coverage. Recent works, such as [11, 87, 216], have begun to address this particular limitation, and it is clear that further research is needed to provide effective and efficient solutions (further discussions in Section 2.4). Similarly, Figure 2.3 shows a constant increase in the use of native code, which calls for further research in the development of techniques able to transparently analyse low-level semantics as well as high-level Android semantics seamlessly. Our work in this area can be found in Chapter 3.

Malware Threat: Shifting to permission usage, a reasonable indicator of the growing abilities (i.e., threat) of malware, within our dataset the INTERNET permission was the most requested, followed by READ_PHONE_STATE (e.g., phone number, IMSI, IMEI). For example, these device identifiers are useful for malware-based banking scams [42]. As seen in Table 2.1, their popularity initially fluctuated several positions but eventually stabilized. Furthermore, even though 82% of all apps read device ID and 50% collect physical locations, malware are even more likely (eight times more so [144]) to gather such data. There are many ways to misuse leaked user information, such as determining the user's location and differentiating between real devices and emulators. The author utilizes such behaviours for classifying Android malware, in Chapter 4.

To collect geographical data, the malware we analysed became increasingly interested in location based permissions (COARSE and FINE in Table 2.1). We also noted the prevalence of the SEND_SMS permission, although it lessened over the years due to Google's efforts and thus omitted from Table 2.1. Despite this, SMS malware have increased over three times since 2012, are a top concern in the US, Spain, and Taiwan, and can generate revenue for attackers or steal bank SMS tokens to hack accounts [144].

In Table 2.1, the number of Android malware requesting WRITE_SETTINGS permission was relatively low in 2010 (8.5%), but the number rocketed up to 20.38% in 2014. There was also a similar increase in READ_SETTINGS, and while benign apps only ask for this permission pair 0.2% of the time, malware do so 11.94% of the time [131]. Another drastic change was with the SYSTEM_ALERT_WINDOW permission (i.e., allows an app to open a window on top of other apps) being requested only by

Table 2.1: Rank variations of top 10 Android permission requests from 2010 to 2014.

Permission Ranking 2010-2011	%	Permission Ranking 2012	%	
1. INTERNET	96.6	INTERNET	97.0	
2. READ_PHONE_STATE	90.5	ACCESS_NETWORK_STATE	92.0	↑3
3. VIBRATE	67.0	VIBRATE	89.0	
4. WRITE_EXTERNAL_STORAGE	67.2	ACCESS_FINE_LOCATION	84.5	↑6
5. ACCESS_NETWORK_STATE	67.2	READ_PHONE_STATE	90.5	↓3
6. SEND_SMS	58.1	WAKE_LOCK	80.9	↑1
7. WAKE_LOCK	50.0	ACCESS_WIFI_STATE	59.0	↑2
8. RECEIVE_BOOT_COMPLETED	48.0	WRITE_EXTERNAL_STORAGE	67.2	↓4
9. ACCESS_WIFI_STATE	46.6	ACCESS_COARSE_LOCATION	48.0	↑5
10. ACCESS_FINE_LOCATION	43.0	FACTORY_TEST	40.9	↑8

Permission Ranking 2013	%		Permission Ranking 2014	%	
INTERNET	97.7		INTERNET	98.7	
ACCESS_NETWORK_STATE	95.6		ACCESS_NETWORK_STATE	98.3	
READ_PHONE_STATE	94.2	↑2	READ_PHONE_STATE	96.2	
VIBRATE	92.6	↓1	VIBRATE	93.7	
ACCESS_WIFI_STATE	88.6	↑2	WAKE_LOCK	92.5	↑1
WAKE_LOCK	85.9		ACCESS_WIFI_STATE	92.1	↓1
ACCESS_FINE_LOCATION	82.1	↓3	ACCESS_FINE_LOCATION	86.8	
WRITE_EXTERNAL_STORAGE	70.6		FACTORY_TEST	81.6	↑1
FACTORY_TEST	67.2	↑1	WRITE_EXTERNAL_STORAGE	78.8	↓1
ACCESS_COARSE_LOCATION	57.0	↓1	ACCESS_COARSE_LOCATION	63.7	

0.23% of malware in 2010, but 24.8% by 2014. Granting this permission can be very dangerous as malware can deny services to open apps and attempt to trick users into clicking ads, install software, visit vulnerable sites, etc.

We also witnessed several new permissions being requested across the years. As an example, the dangerous permission MOUNT_FORMAT_FILESYSTEMS (i.e., used to format an external memory card), was first used by three malware in 2011. Other permissions becoming popular due to malware include USE_CREDENTIALS and AUTHENTICATE_ACCOUNTS. These were categorized as dangerous by Google as they could greatly aid in privilege escalation.

Partly due to the introduction of more permissions, the percentage of our malware dataset requesting dangerous permissions increased from 69% in 2010 to 79% in 2014. This may be the result of malware seeking more control and access over their environment, but may also reflect precarious changes in the permission system. As discussed later on, other studies on the Android permission system have also shown it growing larger, more coarse grained, and with a higher percentage of dangerous permissions. However, to gain a more in-depth understanding of the malware's intent and purpose, a more detailed behavioural profile is necessary. While many of these permissions hint at an action, the following chapters explore each behaviour in detail and within context.

Apart from analysing the evolution of permissions, we also examined malware in terms of registered broadcast receivers (i.e., app components that listen to system events). Generally, malware have more broadcast receivers than benign apps. Specifically, [131] found that 85% of malware registered more than one broadcast receiver, while only 41.86% of goodware did so. This proved useful in triggering behaviours in Chapter 5.

2.2 Taxonomy of Mobile Malware Analysis Approaches

The risks introduced by mobile malware motivate the development of robust and accurate analysis methods. One method to counter or detect malware is with the use of anti-virus (AV) products. Unfortunately, as mentioned previously, on-device AV applications face difficulties as they are just as limited as everyday, user installed, applications. Hence cloud-based signature-based detection are popular AV services.

A malware signature is created by extracting binary patterns, or random snippets, from a sample. Therefore, any app encountered in the future with the same signature is considered a sample of that malware. However, this approach has at least two major drawbacks. Firstly, this method is ineffective for detecting unknown threats, i.e. zero-day attacks, as no previously-generated signature could exist. This is costly as additional methods are needed to detect the threat, create a new signature, and distribute it.

Secondly, malware can easily bypass signature-based identification by changing small pieces of its software without affecting the semantics [170]. Section 2.2 provides further details on obfuscation techniques including those that break signature-based detection. As a result of these issues, exemplified by the Google App Verification system released in 2012 [112], more efforts have been dedicated to implementing semantic signatures; signatures based on functions or methods [53, 248]. Alternatively, a wider set of available app features may be analysed statically or dynamically. In the remainder of this section, we examine such methods, their applications, and feature choice.

Although not discussed thoroughly within this thesis, it is natural that research on newer mobile environments builds upon decades of traditional static and dynamic malware research. For example, although decompiling and virtualization are traditional methods, the particulars of code packaging (i.e., Android APK dex files, AndroidMAnifest, versus Windows binary) and VM architectures (i.e., app VM) differ for Android. Furthermore, as discussed previously, mobile malware is beginning to match traditional malware in sophistication and construction. Thus, it is prudent to adapt and further develop traditional methods to deal with similar threats. Nonetheless, the nature of Android apps and the specifics of its architecture create divergent methods.

2.2.1 Static Analysis

Static analysis examines a program without executing any code. Although it could potentially reveal all possible paths of execution, there are several limitations. Furthermore, alternative code compilers mean traditional analyses and signature methods (e.g., Windows whole-file, section, and code hashing) are incompatible with Android and its unique APK layout and content. All forms of static analysis, for traditional and mobile devices, are vulnerable to obfuscations (e.g., encryption) that remove, or limit, access to the code. This has hindered Symbian malware analysis [182], and despite established static methods, obfuscation is still an open issue [192] (see Section 2.4). Similarly, the injection of non-Java code, network activity, and the modification of objects at run-time (e.g., reflection) are often outside the scope of static analysis as they are only visible during execution. This is a growing issue, as seen in our statistics in Section 2.1.4, as these dynamic actions are occurring more and more frequently in Android malware.

For static analysis, as Android app source code is rarely available, many frameworks choose to analyse various components of the application package (APK). Specifically, many analyse the app bytecode as it is the product of compiling the source code. APK contents are described as follows, including variations introduced with the ART runtime:

- **META-INF:** this directory holds the manifest file, app RSA software key, list resources, and all resource SHA-1 digests
- The **assets** directory holds: files the app can retrieve with the AssetManager
- **AndroidManifest.xml**: additional Android manifest file describing package name, permissions, version, referenced library files, and app components, i.e. activities, services, content providers, and broadcast receivers (see Figure 3.1)
- The **classes.dex** file: contains all Android classes compiled into dex file format for the Dalvik virtual machine (DVM). For ART (only runtime as of Android v.5), the Dalvik bytecode is stored in an .odex file, a preprocessed version of .dex
- The folder **lib**: holds compiled code in sub folders specific to the processor software layer and named after the processor (e.g., armeabi holds compiled code for all ARM-based processors)
- **resources.arsc:** this file contains all precompiled resources
- The folder **res**: holds resources not compiled into resources.arsc

The two most used APK components for static analysis are (1) the `AndroidManifest.xml`, and (2) `classes.dex`, as they hold the most meta data and possible app actions and `Intents`. We are unaware of any studies analysing `odex` files.

2.2.1.1 Permissions

Permissions, such as SEND_SMS, are an important feature for analysis as most actions (e.g., a series of APIs) require particular permissions in order to be invoked [228]. As an illustration, before an app can access the camera the Android system checks if the requesting app has the CAMERA permission [79]. Many other permissions were discussed in our analysis of malware in Section 2.1.4, all of which must be declared in the Android Manifest. As the manifest is easy to obtain statically, many frameworks, such as PScout [20], Whyper [158], and [79, 223], use static analysis to evaluate the risks of the Android permission system and individual apps. Although their methods vary, their conclusions agreed that the evolution of the Android permission system continues to introduce dangerous permissions and fails to deter malware from exploiting vulnerabilities and performing escalation. During our experiments on over nine thousand malware samples, we also found this to be true. Three primary reasons for why this may be so are poor documentation, poor developer habits, and malicious behaviours [79].

Two important studies have found a detrimental lack of documentation and comprehension concerning APIs and their required permissions, despite very little redundancy within the growing Android permissions system [20, 158]. Furthermore, [223] found that the number of permissions in Android releases from 2009 to 2011 had increased steadily, and mostly in dangerous categories. It has also been shown by other studies, and in Section 2.1.4, that malware request more permissions than benign apps. In the million apps Andrubis received from 2010 to 2014, malicious apps requested, on average, 12.99 permissions, while benign apps asked for an average of 4.5.

2.2.1.2 Intents

Within Android, `Intents` are abstract objects containing data on an operation to be performed for an app component. Based on the `Intent`, the appropriate action (e.g., taking a photo) is performed by the system and can therefore be useful for analysis. In one scenario, private data can be leaked to a malicious app that requested the data via `Intents` defined in its Android manifest file. Further discussion on the flow of `Intents` via IPC can be found in the following chapter (see Figure 3.2, page 62).

In DroidMat [228] `Intents`, permissions, component deployment, and APIs were extracted from the manifest and analysed with machine learning algorithms (e.g., k-means, k-nearest neighbours, naive Bayes) to develop malware detection systems. Similarly, DREBIN [16] collected intents, permissions, app components, APIs, and network addresses from malicious APKs, but used support vector machine learning instead.

2.2.1.3 Hardware Components

Another part of the Android Manifest that has been used for static analysis is the listed hardware components. DREBIN [16] utilized these components listed in the manifest in its analysis. This can be effective as apps must request all the hardware (e.g., camera, GPS) they require in order to function. Certain combinations of requested hardware can therefore imply maliciousness. For example, there is no apparent necessity for a basic calculator app to require 3G, GPS, and microphone access. Dynamic analysis can be used to analyse hardware usage, but these normally analyse hardware related API calls, or system calls, as it is easier than analysing the hardware directly.

An application's dex or classes.dex files can be found in the Android APK. For most, these files are not human-readable, and are often decompiled first into a more comprehensible format, such as `Soot`. There are many levels of formats, from low-level bytecode, to assembly code, to human-readable source code. See Table 2.2 for a brief comparison of disassembled formats and a non-exhaustive list of tools that use this format. For example, Dexpler enables `Soot` to convert a dex file to Dalvik bytecode, and then continued to convert it to `Jimple`, a *simplified* version of Java source code.

Both frameworks PScout [20] and AppSealer [240] use `Soot` directly on the dex, see Figure 2.4(a), to acquire Java bytecode, while [73] uses ded/DARE, and Pegasus created its own "translation tool" [49]. Alternatively, [79] decompiles dex into an assembly-like code with dedexer, while others choose to study Dalvik bytecode [93, 119, 240], smali [99, 242, 243, 246], or the source code [53, 58]. In general, more drastic decompiling methods have a higher fail rate or error rate, due to the significant change from the old format to the new. Some of which can be amended by post-processing.

From the decompiled format, static features (e.g., classes, APIs, methods), structure sequences, and program dependency graphs can be extracted and analysed. These

Table 2.2: Decompiled DEX formats and abilities based on existing tools (✓= yes, ✗, = no, ∼ = partial).

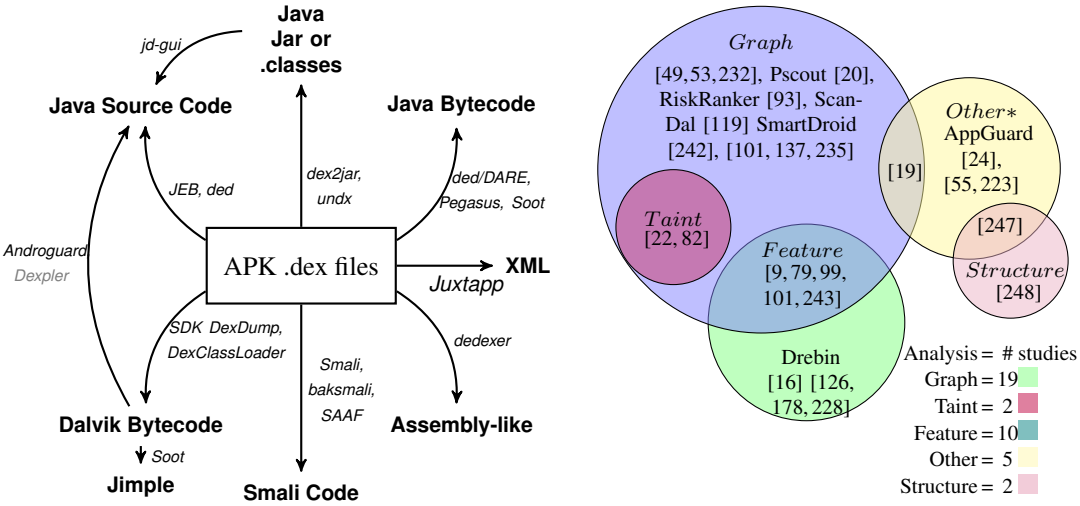
Format	Example Tool	Performance	Coverage
Dalvik Bytecode	dexdump [119]	false+ (15-18%) unknown (∼75%)	✗ dynamic code ✗ instruction change ∼ reflection ✗ JNI
Java Bytecode	Pegasus [49]	false+ (12.5%)	∼ reflection ∼ `Intents`
Source Code	ded [58]	accuracy (94)%	✗ dynamic code ✗ instruction change ∼ number recovery
Smali	SAAF [99]	accuracy (99.9%)	✗ obfuscation ✗ runtime
Assembly	dedexer [79]	false+ (4%)	∼ reflection ✓ `Intents`
Jar	dex2jar [88]	false+ (35%)	✗ ad libs, JNI, `Intents`, Java data structures
Jimple	FlowDroid [18]	93% recall, 86% precision	✗reflection

methods are all, to some degree, adaptations of traditional static methods to analyse and detect malware. An overview of analysis frameworks based on statically decompiled features can be found in Figure 2.4 (*"Other*"* encompasses static analysis on source code [19], bytecode [24,55], the Android Manifest [223], and module decoupling [247]).

2.2.1.4 Dex files

Dex files are particularly interesting for study, as they are unique to Android analysis and the versatile bytecode can be easily converted to a range of formats. In general, feature based analysis determines the presence, absence, or frequency of a set of features. Conversely, graph and taint based analysis are primarily concerned with the flow and relationships of features during the course of an app's execution. For example, dex files have been decompiled and analysed to track the flow of `Intents` in interprocess communications (IPC), i.e. inter-component communications (ICC) [129, 235]. Graph-based analysis has also been deployed to aid smart stimulation [137]. More details and comparisons of these static analysis techniques can be found Section 2.2.4.

Different types of static analysis, such as feature, graph, or structure-based, may also be combined for a richer, more robust, analysis. For example, as seen in Figure 2.4(b), [247] combines structural and feature analysis by decoupling modules and analysing extracted semantic feature vectors to detect destructive payloads. Other studies, such as [99], extract both feature and dependency graphs, via smali program slices in order to find method parameter values. Conversely, ADAM [243] tested if anti-malware products could detect apps repackaged by altering dependency graphs and obfuscated features.



(a) Dex decompile/disassemble tools and formats (b) Venn diagram of static analysis methods

Figure 2.4: Systematization of static analysis methods.

2.2.2 Dynamic Analysis

In contrast to static analysis, dynamic analysis executes a program and observes the results. This may provide limited code coverage, as only one path is shown per execution, but can be improved with stimulation. As Android apps are highly interactive, many behaviours need to be triggered via the interface, received `Intents`, or smart, automatic event injectors [22, 133, 137]. Another degree of complexity is also added, as the malware is "live" and able to view and interact with its environment. This has led to two different types of dynamic analysis: *in-the-box* analysis and *out-of-the-box* analysis.

If the analysis resides on the same permission level, or architectural layer, as the malicious software, malware can detect and tamper with the analysis. This is known as in-guest, or in-the-box, analysis as it relies on the Dalvik runtime (or the ART runtime) and/or the Android OS. The upside to this approach is easier access to certain OS-level data (see Figure 2.6). On the other hand, if the analysis was to reside in a lower layer, say the kernel, it would increase robustness and transparency, but make it more difficult to intercept app data and communications. To overcome this weakness, there are several methods to fill the semantic gap, i.e. recreating OS/app semantics from a lower observation point such as the emulator [85, 202]. Details of in-the-box, out-of-the-box, and virtualization can be found in further down in Sections 2.2.2.1-2.2.2.3.

To better understand the progression of dynamic analysis, we present Figure 2.5. Here we attempt to illustrate the number of different architectural layers (e.g., hardware, kernel, app, or OS) being studied dynamically from 1997–2015. One interesting trend is the increasing amount of multi-layered analyses, which increases the number of unique and analysable features. Details on these analysis methods can be found in Section 2.2.4. Different analysis environments are also represented in this figure, including emulators, real devices, and hybrids of both [216]. Again, because the malware is running during

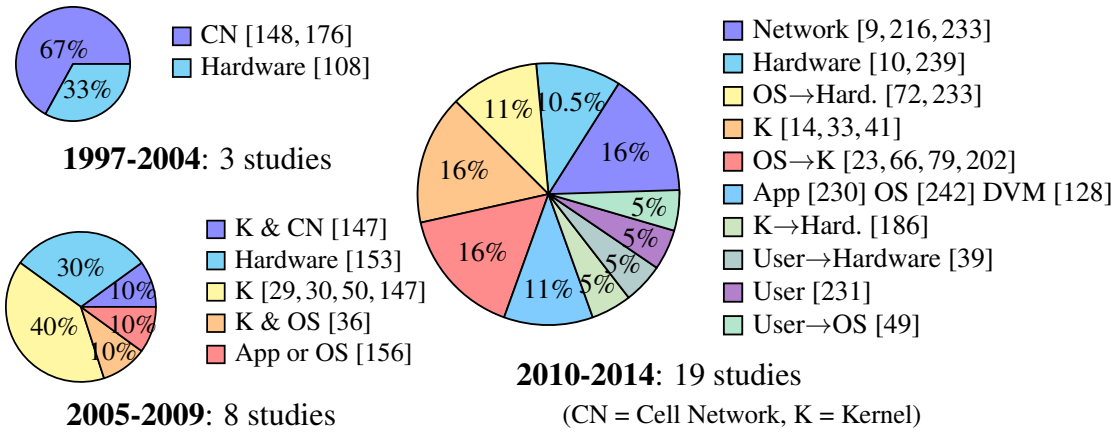


Figure 2.5: Dynamic studies analysing different architectural layers.

analysis, the choice of environment is more complicated. In 2013, Obad [209] was the first malware to detect emulated environments and could decide to not exhibit malicious behaviours, i.e. have a split-personality (discussed in Section 2.4).

To stimulate Android apps and find software bugs, the DynoDroid [133] system utilized real user interactions (e.g., tapping the screen, long pressing, dragging) for analysis. Alternatively, hybrid solutions like EvoDroid [137], use static and dynamic analysis to explore as much of the application code, in the fewest number of executions, possible. Besides increasing code coverage, user interactions with apps may also be analysed for malware detection. By crowdsourcing scenarios, PuppetDroid [87] captured user interactions as stimulation traces and reproduced the UI interactions to stimulate malicious behaviours during dynamic analysis. This is based on the assumption that similar user interaction patterns can be used to detect malicious apps, as malware are often repackaged code or variants of each other (i.e., a malware family).

2.2.2.1 In-the-box (in-guest) Analysis

In this method of analysis, the examination and/or gathering of data occurs on the same privilege level (e.g., architectural level) as the malware. This often requires modifying, or being finely tuned into, the OS or the runtime. For example, DIVILAR [246] inserts hooks into the Android internals, i.e. Dalvik VM, to run apps modified against repackaging. In contrast, Mockdroid [32] modified the OS permission checks to revoke system accesses at run-time. The advantage to these methods are that memory structures and high OS-level data are easily accessible. Access to libraries, methods, and APIs are also available, but not necessarily granted to applications because of permissions.

The downside of in-guest analysis, as mentioned previously, is that the "close proximity" to the app leaves the analysis open attacks or bypassing, e.g. with native code or reflection [231]. It is possible to increase analysis transparency by hiding processes or loaded libraries, but this is impossible to achieve from the user space alone. Additional downfalls to editing the Android OS or runtime are (1) necessary modifications to multiple OS versions, (2) potential software bugs, and (3) the replacement of Dalvik with ART [218]. While in-guest methods already require moderate to heavy modifications between OS versions, many of these methods need fundamental changes to adapt to a new runtime. Alternatively, a lower-level framework would hold a higher privilege level than user-level apps, increasing transparency and security, unless malware gain root privileges via a root exploit. Although high-level semantics are more difficult to analyse out-of-the box, this method can provide greater portability across different Android OS versions as there is more stability, i.e. less change, in lower architecture layers.

2.2.2.2 Out-of-the-box Analysis

VM-based analyses utilize emulators and virtual environments to provide increased security through isolation. While both emulated environments and virtualization achieve isolation by sandboxing dangerous software, emulators also provide complete control and oversight of the environment. Furthermore, full system emulation completely emulates a real device, which includes all system functionality and required peripherals. Traditionally, this includes CPU, memory, and devices such as network interface cards, but for smartphones this may also include cameras, GPS, and accelerometers.

While the mobile emulator MobileSandbox [29] works for both Windows and Android, most other systems like Andrubis [225], DroidScope [233], CopperDroid [202], and [84, 227], use purely Android emulators. In particular, these were built on top of QEMU, an open-source CPU emulator available for ARM hardware. Unfortunately, malware can, and have, counter emulation by detecting false, non-real, environments and stop or misdirect the analysis (e.g., split personality with different sets of behaviours depending on situation). There are many samples of traditional PC malware that do exactly this, and more mobile malware are now exhibiting similar levels of sophisticated VM-awareness (details in Section 2.3). While it was accepted that out-of-the-box analysis meant less available high-level semantic data, it was previously believed that fully recreating high-level behaviours, such as IPC/ICC, outside the box would be too challenging. The author's contribution to CopperDroid has proven this false. The author's out-of-the-box approach also allows the enhanced CopperDroid to switch between Android OS versions seamlessly, including versions running ART (see Chapter 3).

2.2.2.3 Virtualization

Analysis using virtualization assigns the system (e.g., hardware) a privileged state to prevent unrestricted access by sandboxed software. This partial emulation is lighter than full emulation, but, if implemented correctly, still provides robust security. Furthermore, in contrast to emulators, guest systems within VMs can execute non-privileged instructions directly on the hardware, greatly improving performance. Currently, Android app sandboxing is handled by the kernel, but despite this, malware can still compromise the system using privilege escalation. To improve isolation, or to host multiple phone images (e.g., Cells with lightweight OS virtualization [14]), additional virtualization can be introduced at the kernel or hypervisor levels. Highly privileged kernel-level or hypervisor-level (either bare-metal or hosted) sandboxing is less susceptible to corruption and, as seen in Figure 2.6, provide easier access to kernel data such as system

Layer*	Information of Interest
User	User interface inputs
Apps, OS	IPC/ RPC, APIs, security frameworks
Kernel, Hypervi-sor	Addr space, network, syscalls, virt registers, data types/constructors/fields
Hardware	Battery usage, file access, CPU

*Lower layers found lower on the table.

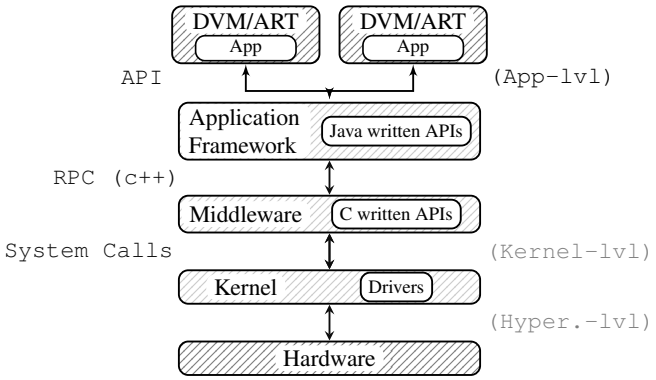


Figure 2.6: Data and sandboxing available at all Android architectural layers.

calls [30, 39]. The negatives of virtualization, and some emulators, is the isolation introduces a discontinuity between the data seen by the analysis, and high-level OS data. Such semantic gaps are reconstructible with virtual machine introspection (VMI). However, the Android Dalvik VM complicates VMI as two-level VMI might be necessary.

If implemented, an Android hypervisor would reside on top of the hardware (i.e., highest possible permission level) where it can provide the most isolation and security. Both desktops and server domains use this method for intrusion detection, isolation, and preventing rootkits. In 2008, [95] was one of the first to analyse the security benefits of hypervisors in embedded (e.g., mobile) devices. Unfortunately, the majority of on-shelf ARMs cannot currently support pure-virtualization[3], and so alternative solutions have relied on other methods such as para-virtualization or hardware extensions.

Para-virtualization simulates the underlying hardware with software and requires modifications to critical parts of the virtualized OS. Using para-virtualization and a Xen hypervisor, [104] successfully created a secure hypervisor, or virtual machine monitor (VMM), on an ARM processor. In contrast, *pure-virtualization* (i.e., hardware virtualization) utilizes built-in processor hardware to run unmodified virtual operating systems. This has the advantage of being able to host guest OS kernels without modification. Introducing hardware extensions can enhance the ARM processor in order to grant pure-virtualization capabilities, which is significantly less complex than para-virtualization [212]. In 2012, [84] used an ARM TrustZone processor extension to achieve effects similar to full virtualization, and in 2013, [189] implemented and evaluated a fully operational hypervisor that successfully ran multiple VMs on an ARM Cortex A9-based server. Besides added security, these studies have also demonstrated that hypervisors for mobiles often require an order of magnitude fewer lines of code than full OS hypervisors. This implies better performance and less software bugs introduced.

[3]The Cortex-A15 has full virtualization support, but has only been installed in a few selected devices.

2.2.3 Hybrid Analysis

By combining static and dynamic analysis, hybrid methods can increase robustness, code coverage, monitor apps, and find vulnerabilities. For example, [24] and [49] statically inserted hooks into functions (i.e., sensitive APIs) which provided run-time data for dynamic policy enforcement. Similarly, [156] governed static permission assignments and then dynamically analysed Android IPC, as dictated by its policies. Although unable to analyze ICC, Harvester [168] obtains important runtime data via a hybrid method.

Hybrid malware detectors like [33] have also used static analysis to asses an app's danger before dynamically logging its system calls with kernel-level sandboxing. Alternatively, to increase code coverage, SmartDroid [242], EvoDroid [137], and [193] use static analysis to find all possible activity paths to guide later dynamic analysis. A5 [216] employed a similar hybrid analysis for detection, triggering `Intents` found in the code in order to examine all paths for malicious behaviours. A5 also utilized both real devices and emulators (one or the other). Concolic testing, a mixture of static and dynamic, has also been used to uncover malicious information leaks in Android apps [11].

2.2.4 Analysis Techniques

This section describes various analysis techniques. While most are used statically or dynamically, e.g. APIs, several are unique to one or the other. These methods, and whether they are applicable statically and/or dynamically, are summarized in Table 2.3.

2.2.4.1 Network Traffic

As discovered in Section 2.1, most apps, normal and malicious, request permissions for network connectivity. In [250], 93% of collected Android malware samples made network connections to a malicious entity. Additionally, [179] analysed 150k Android apps in 2012 and found 93.38% of malicious apps required network access while only 68.50% of normal apps did so. Similarly, in [178], permissions of 2k apps were analysed to find that over 93% of malware requested network access in 2013. This demonstrates that network access is requested by most apps, but particularly by the malicious ones.

If granted permission to access networking, network payloads may contain malicious drive-by-downloads flowing into the device, or leaked data flowing out of the device. Network ports are therefore often sinks in taint analysis and lead to more thorough network packet analysis. Frameworks studying network communications have been implemented on both real and emulated devices [39, 186, 224], as well cell networks, which is computationally easier on individual devices but must protect communication

channels from attacks [41, 113, 127, 148, 176, 177]. It is unclear on how different the challenges are between mobile malware detection and traditional malware detection via network analysis. However, for bot-like behaviours and leaked data, network analysis seems an effective method both traditional PCs and mobiles devices.

2.2.4.2 Application Programming Interfaces

APIs are a set of coherent methods for apps to interact with the device. This includes app libraries in the Dalvik VM (same permissions as the app), and unrestricted API implementations running in system processes. For example, to modify a file, the API is proxied by the public library API to the correct system process API implementation. Pegasus [49], [242], and Aurasium [231] dynamically monitor these APIs for app policy enforcement and discovering UI triggers. If a private interface has no corresponding public API, it can still be invoked with reflection, i.e. the ability an object has to examine itself. Library and system APIs can also be studied in conjunction [233], and used as features to classify malware, as shown in [9, 57] (more in Chapter 4).

2.2.4.3 System Calls

System level APIs are highly dependent on the Android hardware, i.e. ARM. The ARM ISA provides the `swi` instruction for invoking system calls. This causes a user-to-kernel transition where a user-mode app accesses kernel-level system calls through local APIs. Once an API is proxied to a system call and the system has verified the app's permissions, the system switches to kernel mode and uses system calls to execute tasks on behalf of the app. As apps can only interact with the hardware via system calls, system call-centric analysis has been implemented for Windows devices [29, 104] and Android devices [41, 93, 202]. While these were based on low-level data, it is still possible to reconstruct high-level semantics via system call analysis (see Chapter 3).

2.2.4.4 Dependency Graphs

Dependency graphs provide a program method representation, with each node a statement, and each edge a dependency between two statements. The manner in which these edges are created determines the type of graph. For example, a data dependent edge exists if the value of a variable in one state depends on another state. Once created, dependency graphs can be analysed for similarities such as plagiarism [53].

In control dependency graphs, an edge exists if the execution trigger of one state depends on the value in another state. For example, ScanDal [119] builds, and analyses,

control flow graphs (CFG) based on sensitive data returned by APIs to discover information leaks. Similarly, [232] also uses CFGs to detect information leaks, but utilizes content providers instead of APIs. DroidSIFT [9], on the other hand, created weighted, contextual, API dependency graphs to construct feature sets. Using these features and graphs, DroidSIFT creates semantic signatures for classifying Android malware.

In comparison to feature API permission mapping, PScout [20] combines all call graphs from the Android framework components for a full, flow-sensitive analysis, and Pegasus [49] constructs permission event graphs to abstract the context in which events fired. Multiple flow analysis can also be used together to search for malicious background actions [79, 93]. To make these frameworks scalable, graphs must remove all redundancies to avoid path explosions since more paths require more computations.

2.2.4.5 Features

Feature-based analysis extracts and studies sets of features in order to enforce policies, understand API permissions, classify apps, and detect code reuse through feature hashing (e.g., Juxtapp [94]). To enforce security policies, hooks can be inserted at key points for later dynamic monitoring [24,55]. Conversely, to identify which permissions an API requires, [79] ran different combinations of extracted content providers and `Intents`.

Besides analysing the actual feature, like which APIs were triggered, feature frequency analysis is also often used to see how many times certain features are found, i.e. multiple executions of the same API. The primary downside of feature-based analysis, however, is it cannot reveal the context (e.g., when, how) something was triggered [79].

2.2.4.6 Function Call Monitoring

By dynamically intercepting function calls, such as library APIs, frameworks can analyse both single calls and sequences of calls to reconstruct behaviours for semantic representations, or monitor the function calls for misuse. Function hooks can also be used to trigger additional analyses. For example, if a function was hooked and triggered, parameter analysis could then be applied to retrieve the parameter values of when the function was invoked.

The analysis framework InDroid inserted function call stubs at the start of each opcode's interpretation code in order to monitor bytecode execution and analyse Android behaviours. While it does require modifications to the Dalvik VM and may not work on Android 5.0 (i.e., with ART), the method requires relatively light modifications and has been used on versions 4.0-4.2 [128].

2.2.4.7 Information Flow and Tainting

Information flow is an essential analysis technique that tracks the transfer of information throughout a system. While implemented for both traditional PCs and mobile devices, it is important to note that flow analysis for Android differs greatly from traditional control flow and data flow graphs. This is largely due to the fact that Android flow graphs are typically fragmented in real-world settings. This is inherently caused by Android app's component-based nature which allows components to be executed in an arbitrary order, depending on user interactions and system events.

The biggest challenge for any information flow analysis on Android, therefore, is to develop these graphs or data flows. One method to analyse information being moved, or copied, to new locations is taint analysis. Analysing tainted data allows one to track how data propagates throughout the program execution from a source (i.e., taint source) to a destination (i.e., taint sink). Taint sources create and attach taint labels to data leaving designated sources, such as phone contacts. The system can then implement different taint propagation rules, i.e. tainting data that come into contact with tainted data, during execution. Such rules include direct taint labels for assignments or arithmetic operations, memory address dependent taints, and control flow taint dependencies.

When tainted data arrives at a sink, different procedures can be run depending on the data, source, and sink. Typically, taint analysis method is used to detect leaked data, like in TaintDroid and AndroidLeaks [72, 88]. Specifically, TaintDroid performs dynamic taint analysis on app level messages and VM level variables, while AndroidLeaks uses a mapping of API methods and permissions as the sources and sinks of data-flow analysis.

Alternatively, FlowDroid [18], implemented both object and flow-sensitive taint analysis to consider the life-cycle of an Android app through control-flow graphs. While the graphs provided context for which each methods belonged to, FlowDroid is, however, computationally expensive and excludes network flow analysis. More recently, SUSI [167], built on Android v4.2, uses machine learning on used APIs, semantic features, and syntactic features, to provide more source and sink information than both TaintDroid (Android v2.1) and SCanDroid.

More broadly, information flows can be implicit, or explicit. In general, implicit information flows (IIF) are more difficult to track than explicit. As a result, malware often leverage IIF to evade detection while leaking data. In order to understand the types of IIFs within Android, [238] analysed application Dalvik bytecode to identify indirect control transfer instructions. By seeking various combinations of these instructions, the authors extrapolated five types of instruction-based IIF and used them to bypass detection frameworks such as TaintDroid.

Again, while these techniques have been implemented in traditional PCs, TaintDroid is one of the first attempts to apply them to Android. In another taint analysis framework, the tools `Dflow` and `DroidInfer` were used in a type-based taint analysis for both log flows and network flows [102]. Using the same static decompilation methods as FlowDroid (i.e., `Soot` and `Dexpler`), `Dflow` and `DroidInfer` were used to understand context sensitive information flows and type inference. By tainting data as safe, tainted, or poly (declared safe or tainted based on the context), the authors were able to detect multiple information leaks (including ICC leaks) faster than previous methods.

2.2.4.8 Inter-Process Communication Analysis

Within the Android OS, applications rely on inter-process communications (IPC) and remote procedure calls (RPC) to carry out most tasks. While RPC can be implemented on top of Binder, RPC across the network is not available on Android. These channels use Binder, a custom implementation of the OpenBinder protocol which allows Java processes (e.g., apps) to use remote objects methods (e.g., services) as if they were local methods. Analysing IPC/RPC can therefore provide essential Android-level insights, such as a malicious web browsing `Intents` or colluding applications.

As elaborated in Chapter 3, the author's work in CopperDroid reconstructs IPC communications dynamically. Conversely, most other methods use static methods to track the movement of `Intents` within IPC, i.e. ICC [129, 235]. As information can be passed though various communications such as IPC, they are often analysed for information flow analysis. Such communications occur fairly often during malware execution. As we discovered with CopperDroid, see Table 4.2 (page 107), IPC Binder methods were invoked in 70% of analysed malware samples and were 40% of seen behaviours.

One static study, Epicc [155] created and analysed a control-flow super graph to detect ICC information leaks. While Epicc relied on `soot` for majority of its needs, Amandroid used a modified version of dexdump (i.e., `dex2IR`) to vet apps by analysing inter-component data flows [222]. Furthermore, while Epicc built control flow graphs, Amandroid built data dependence graphs from each app's ICC data flow graph.

Amandroid is also capable of more in depth analyses (e.g., of libraries), which leads to a higher accuracy but at a performance cost. Particularly for Android, analysing ICC/IPC is essential for understanding and detecting stealth behaviours [101] and leaked information [129] as its IPC Binder protocol is unique, a key part of the Android system, and much more powerful and complex than most other IPC protocols. Furthermore, roughly 96% of 15,000 Android applications analysed by [129] used IPC. Further, the malware samples analysed noticeably leaked more data via IPC than benign apps.

2.2.4.9 Hardware Analysis

Several studies monitor the hardware status for abnormal behaviour through app power signatures [118] and power/CPU consumption [38, 108, 153]. Since 2010 (see Figure 2.5), most dynamic analyses that extracted hardware-based features also analysed additional layers and features. Furthermore, since devices like the camera can only be accessed by system calls, they are rarely analysed on a hardware level.

The framework STREAM [10] collects data regarding system components like cpuUser, cpuIdle, cpuSystem, cpuOther, memActive, and memMapped. STREAM gains this information via APIs from its own installed app, then subsequently uses machine learning algorithms to train the system to detect Android malware. As mentioned previously, hardware components can also be studied statically when analysing the Android Manifest of an APK.

Table 2.3: Analysis techniques used by static and dynamic methods.

	Network Traffic	**APIs**	**System Calls**	**Dependency Graphs**	**Features**
Dynamic	destination & packets [39, 186, 224]	hooks etc. [174, 231, 233]	[41, 93, 202]	[11, 22]	[24, 55]
Static	hard coded info	decompiling [6, 9, 16, 49, 57, 234, 242]	✗	[20, 101, 119, 137, 235]	[79, 94]

	Function Call Monitoring	**Taint**	**IPC**	**Hardware**
Dynamic	[29, 68, 128, 171, 231]	[72, 72, 88]	[39, 110, 156, 202, 231]	[10, 38, 108, 118, 153]
Static	✗	[18, 102, 167, 236, 238]	[129, 155, 222, 235]	Manifest [16]

2.2.4.10 Android Application Metadata

Application market metadata is the information users see prior to downloading and installing an app. Such data includes app description, requested permissions, rating, and developer data. Since app metadata is not a part of the APK itself, we do not categorize it as a static or dynamic feature (see Table 2.3). In WHYPER [158], app permissions were acquired through the market and Natural Language Processing (NLP) was implemented to determine why each permission was requested by parsing the app description. WHYPER achieved 82.8% precision for three sensitive-data related permissions.

Similarly, [203] used sophisticated knowledge discovery process and lean statistical methods to analyse Google Play metadata and detect malware. However, this study also stressed that metadata should only be used with other analyses, not alone. App metadata they fed to their machine learning algorithms included the last time modified, category (e.g., game), price, description, permissions, rating, number of downloads, creator (i.e., developer) ID, contact email and website, promotional videos, number of screenshots, promo texts, package name, installation size, version, and app title.

2.2.5 Feature Selection

Choosing an appropriate feature set is essential when conducting an analysis as it strongly determines the effectiveness and accuracy of the research. As Android apps have many features to choose from, there needs to be sound reasoning to why certain ones were chosen for specific experiments. The following approaches are sensible reasoning.

Selection Reasoning: As mentioned previously, Android apps must be granted permissions in order to perform specific actions. Therefore, many studies such as VetDroid [241] and DroidRanger [251] based their analyses on permission-use using this reasoning. Similarly, DREBIN [16] collected permissions as well as `Intents`, components, and APIs. This provided additional permission and usage based features, leading to more finer results. One method for feature selection, therefore, is that the authors hypothesized and proved that a set of features will provide the most reliable malware analysis or detection. Alternatively, researches may actively explore new, or largely unused, feature with the reasoning that they might discover new viable solutions.

Feature Ranking Algorithms: Feature ranking and selection involves choosing a subset of all available features that consistently provide accurate results. Choosing features can be done with pre-existing algorithms implementing various mathematical calculations to rank all the possible features in the dataset [109]. For example, the information gain algorithm has been widely used for feature selection, and is based on the entropy difference between the cases utilizing, and not utilizing, certain features [105].

One study, [185] used feature ranking algorithms to select feature subsets from 88 features (i.e., top 10, 20 and 50). Comparably, [186] analysed the network traffic of Android apps and used selection algorithms to study the most useful features. This step was essential due to the massive number of network traffic features. Similarly, in [237], the authors collected 2,285 apps and extracted over 22,000 features. They then chose the top 100, 200, 300, 500 and 800 features with selection algorithms for analysis.

2.2.6 Building on Analysis

Section 2.2 provided a study on a diverse set of Android analysis approaches to obtain detailed behavioural profiles [11, 20, 72, 202, 223, 224, 233, 242] and assess the malware threat [73, 79, 81, 90, 114, 170, 243, 250]. These analyses can be further developed to build classification or clustering [9, 93, 167, 181, 250], policy frameworks [24, 55, 63, 68, 156, 231], and detection (see tables in Appendix A for comparison).

With these frameworks to build on, it is possible to detect Android malware [30, 41, 177, 181, 186], policy violations such as information leaks [39, 119, 232], colluding apps [140], and even repackaged or plagiarized apps [53, 248]. Most detection methods are also either anomaly-based [186] (e.g., defining normal and abnormal attribute sets), misuse-based [232] (e.g., identifying specific malicious actions), or signature-based (e.g., semantic or bytecode) [53]. Furthermore, once detected, it is essential to classify the threat for proper mitigation, family identification, and so new malware (i.e., zero day malware [93]) can be brought to attention and further analysed for mitigation.

With the increasing amount of malware flooding markets scalable and automated classifying (or clustering) techniques are essential. In one study, it was shown that over 190 app markets host significant amounts of malware [214]. The primary difference between classification and clustering, is that classification generally has a set of predefined classes and the objective is to correctly label all samples with the correct class label. Conversely, clustering groups unlabelled objects together by seeking similarities. Traditionally, the output of a classifier is either binary (i.e., the sample is malicious or benign), or it is multi-class (i.e., a sample can belong to one of many malware families or types). Additionally, for input, classifiers normally compute vectorial data. Therefore, features for analysis must be mapped to a vector space that the classifier can compute.

One of the more popular classifiers that have been used for Android malware is support vector machines, but many more are available depending on the data and the objective [6, 57, 234]. Several general methods for feeding data to classifiers include a binary representation, feature frequency, and by representing graph states and/or transitions.

In terms of scalability, manual efforts [250] will not scale, and sometimes accuracy is sacrificed for scalability (see Section 2.4). To keep accuracy high but improve its scalability, different filters or simplification methods can be used. For instance, DNADroid [53] implemented several filters on their graphs to automatically reduce the search space and improve scalability with little cost. In Chapter 4, we will demonstrate how condensing system call traces into high-level behaviours (see Chapter 3) improves the scalability for classifying Android malware. This is due to filtering uninteresting system calls leading to a fewer number of features without loosing essential detail.

2.3 Malware Evolution to Counter Analysis

As mentioned throughout the chapter, there are several kinds of obfuscation and VM-detection methods used by both traditional and mobile malware to obstruct analysis. In this section, we shall discuss these techniques.

For organizational purposes, we categorize static obfuscation techniques into several tiers; trivial transformations, transformations that hinder static analysis, and transformations that can prevent static analysis.

2.3.1 Trivial Layout Transformations

Trivial transformations require no code or bytecode level changes and mainly deter signature-based analysis. One transformation, unique to the Android framework, is unzipping and repackaging APK files. This is a trivial form of obfuscation that does not modify any data in the manifest. Furthermore, by merely repackaging the new app, it is signed with custom keys instead of the original developer's keys. Thus, signatures created with the developer keys, or the original app's checksum, would be rendered ineffective, allowing an attacker to easily distribute seemingly legitimate applications with different signatures. This method does not add functionality to the repackaged app.

Android APK dex files may also be decompiled, as previously shown in Figure 2.4(a), and reassembled. We are unaware of any studies decompiling and analysing ART oat, or odex, files as of late 2015. Once disassembled, components may be rearranged, or their representations altered. Like repackaging, this obfuscation technique also changes the layout of the app, which primarily breaks signatures based on the order, or number, of items within the dex file in an APK.

2.3.2 Transformations that Complicate Static Analysis

While several static techniques are resilient to obfuscations, each technique is still vulnerable to a specific obfuscation method at this tier. Specifically, what we have classified as feature based, graph based, and structure based static analysis, can overcome some of these transformations, but be broken by others.

For example, feature based analysis is generally vulnerable against data obfuscation but may be robust against graph-based obfuscation. Depending on its construction, structural analysis is vulnerable to layout, data, and/or control obfuscation. We will now describe these transformations, which can complicate static analysis but might not completely prevent it.

2.3.2.1 Data Obfuscation

Data obfuscation methods alter APK data, such as the Android Manifest's package name. Unlike control flow obfuscation, this does not primarily alter the semantics of the application. Renaming app methods, classes, and field identifiers with tools like ProGuard is one method of data obfuscation. Instance variables, methods, payloads, native code, strings, and arrays can also be reordered and/or encrypted within the dex file, disrupting most signature methods and several static techniques as well. In Android, native code (i.e., code compiled for ARM) is normally accessed via the JNI, but native exploits can also be stored within the APK itself and encrypted to obfuscate data.

Furthermore, in the cases where the source code is available, the bytecode can be altered by changing variables from local to global, converting static data to procedural data, changing variable types, and splitting or merging data such as arrays and strings. These changes that are functionally neutral have been adopted from traditional obfuscation [51]. For an Android malware developer, this is a simple way to create malware variants with the same functionality but with different signatures to evade detection.

2.3.2.2 Control Flow Obfuscation

This method deters call-graph analysis with call indirections. This primarily entails changing execution patterns by moving method calls without altering semantics. For example, a method can be moved to a new method which then calls the original method. Alternatively, code reordering can obfuscates an application's flow.

Programming languages, such as Java, are often compiled into more expressive languages, such as virtual machine code. This is the case with Android applications, as Java bytecode possesses the `goto` instruction while normal Java does not. Bytecode instructions can then be scrambled and obfuscated with `goto` instructions inserted to preserve the runtime execution.

Other obfuscation transformations include injecting dead or irrelevant code sequences, adding arbitrary variable checks, loop transformations (i.e., unrolling), and function inlining/outlining, as they often add misdirecting graph states and edges. Function inlining (i.e., breaking functions into multiple functions) can be combined with call indirections to generate stronger obfuscation. Alternatively, functions can be joined (i.e., outlining) and Android class methods can be combined by merging their bodies, methods, and parameters (i.e., interweaving). Lastly, Android allows for a few more unique transformations by renaming or modifying non-code files and stripping away debug information, such as source file names, source file line numbers, and local parameters [170].

2.3.3 Transformations that Prevent Static Analysis

These transformations have long been the downfall of static analysis on traditional analysis [150] and mobile malware analysis [99, 170]. Unless also a hybrid solution, no static framework can fully analyse Android apps using full bytecode encryption or Java reflection. This has become more relevant, as Android version 4.1 added support for installing encrypted APKs. Bytecode encryption encrypts all relevant pieces of the app and is only decrypted at runtime: without the decryption routine, the app is unusable. This is popular with traditional polymorphic viruses that also heavily obfuscate the decryption routine. Additionally, run-time cryptography on Android uses crypto APIs. For Android APKs, the bulk of essential code is stored in an encrypted dex, or odex, file that can only be decrypted and loaded dynamically through a user-defined class loader.

Reflection for Android apps can also be used to access all of an API library's hidden and private classes, methods, and fields. Therefore, by converting any method call to a reflective call with the same function, it becomes difficult to discover exactly which method was called. Moreover, encrypting that method's name would make statically analysing it impossible. Similarly, dynamically-loaded code (i.e., the loading of a library into memory at run-time) and the resulting behaviours can only be analysed with some dynamic methods and is impossible to analyse statically.

2.3.4 Anti-Analysis and VM-Aware

With the rapid growth of Android malware, sophisticated anti-analysis RATs (i.e., remote access Trojans) such as Obad, Pincer, and DenDroid, are detecting and evading emulated environments by identifying missing hardware and phone identifiers. More sophisticated anti-analysis methods include app collusion (willingly or blindly), complex UI, and timing attacks on QEMU scheduling, implemented by [162] and BrainTest [48] to evade cutting-edge detection tools. DenDroid, a real-world Trojan discovered in 2014, is capable of many malicious behaviours, but will not exhibit them if it detects emulated environments such as Google Bouncer [65]. Another malware family, AnserverBot, detects and stops on-device anti-virus software by tampering with their processes. The malware Android.hehe also has split-personality and acts benignly when the device IDs (e.g., IMEI) and `Build` strings indicate that it is running in a VM [97].

Other ways to deter analysis, but not necessarily detect it, is to make the app UI intensive, execute at "odd" times (e.g., midnight, hours after installation), require network, or require the presence of another app. For example, the malware CrazyBirds will only execute if the AngryBirds app had also been installed and played at least once.

Additional obfuscation methods to deter dynamic analysis are data obfuscation (e.g., encryption), misleading information flows [238] mimicry, and function indirections.

2.4 Discussion

As we have already determined, smartphones are currently the top, personal, computing device with over 2.5 billion mobile shipments made by early 2015 [86]. Of these shipped smartphone OSs, Android is by far the most popular and has attracted a growing number of dangerous malware [143, 183]. To understand the current malware threat and give context to this thesis, the author surveyed Android malware analysis and detection methods to assess their effectiveness. This determined areas for future research and resulted in a few general observations. For example, it is clear from previous studies that the Android permission system is not becoming more fine-grained, and that the number of dangerous permissions is still increasing. Although it is also apparent that malware are taking advantage of this situation, it is not clear what needs to be improved.

While the permission system does provide flexibility and allow users to be more involved in security decisions, it has devolved the responsibility of securing Android and its users. Therefore, while it is important to create accurate and reliable malware analysis and detection, which we have discussed extensively, knowing which flaws need to be repaired by which party (e.g., users or manufacturer) is also essential.

2.4.1 Impact and Motivation

With developing mobile technologies and a shift towards profit-driven malware, the research community has striven to (1) understand, and improve, mobile security, (2) assess malware risks, and (3) evaluate existing frameworks and anti-virus solutions. By amassing and analysing various Android malware techniques and malware analysis frameworks, we have identified several risks to motivate research efforts in certain areas.

2.4.1.1 Malware Growth and Infections

Despite encouraging trends in Android malware detection and mitigation, we feel that mobile malware—Android in particular—is still growing in sophistication and more challenging problems lie ahead. We also believe that these threats and infections, albeit not spread evenly across countries, are a global threat. Even with low infection rates in some countries, if the right devices are compromised, a much larger number of individuals can still be negatively affected.

In detail, despite low Android malware infections in some geographical areas like the USA [127, 183], the overall global infection rate is concerning. For example, [208] estimated a 26-28% infection rate worldwide based on real device data, and McAfee estimated a 6%-10% infection rate using Android devices running their security products.

As with real infections, it is also dangerous to allow malware to develop in other physical and virtual regions, as they may eventually cross over. To reduce malware infections, malware markets need to both accurately vet submitted apps and remove available malware as soon as they have been identified by itself, or by a reliable third party. Ideally, users should be encouraged to download apps from a central, official market that rigorously checks its applications. However, third-party markets are sometimes the only source of apps in different countries (e.g., China 2014). Online and on-device malware detectors can then be used by users to lower infections rates in these cases.

Privilege escalating root exploits for Android are also easily available 74%-100% of a device's life time [81]. While only known one malware sample attacked rooted phones in 2011, by the following year, more than a third the malware analysed by [250] leveraged root exploits. Furthermore, more than 90% of rooted phones were surrendered to a botnet, which is a significant amount as 15%-20% of all Android devices were rooted at that time. Built-in support for background SMS to premium numbers was also found in 45%-50% [81, 125] of the s amples, and user information harvesting, a top security issue in 2011 [81], is still a current issue with 51% of malware samples exhibiting this behaviour [144].

2.4.1.2 Weaknesses in Analysis Frameworks

Many frameworks today are unable to analyse dynamically loaded code and are vulnerable to at least one kind of obfuscation (see Tables A.1 and A.2). This is significant and within our own experiments in Section 2.1, we have shown the growing correlation between current malware and the use of reflection, native code based attacks, and dynamically loaded code based attacks. Methods for dynamic code loading within Android include class loaders, package content, the runtime Java class, installing APKs (i.e., piggy-back attack, dangerous payload downloads), and native code. Malware often use these methods to run root exploits. Furthermore, even when used benignly, dynamically loaded code has caused widespread vulnerabilities [78, 164].

In 2014, an attack against the Android In-app Billing was launched using dynamically loaded code and was successful against 60% of the top 85 Android apps [151]. Native based attacks can also be used on at least 30% of the million apps Andrubis analysed as they were vulnerable to web-based attacks by exposing native Java objects [131].

Despite this, as can be seen in the Appendix A tables, many frameworks exclude native code and dynamically loaded code in their analyses. Similarly, as seen in these tables, static obfuscation is often the cause of incorrect static results and sometimes prevents the complete analysis of a subset of analysed malware (e.g., failed during decompiling).

2.4.1.3 Weaknesses in AV Products

To evaluate AV products, [250] tested four AV systems in 2012. The best and worst detection rates were 79.6% and 20% respectively, but the most current and advanced malware families were completely undetected. As shown in Section 2.3, signature-based AV products can be broken by the simplest transformations, and dynamic code can be used to evade dynamic systems, such as Google Bouncer [164]. Unfortunately, the inner workings of Google Bouncer [154] and similar systems are not available, but can still be evaluated. In 2013, DroidChameleon [170] submitted automatically obfuscated Android apps to ten popular AV products and found all ten were vulnerable to trivial transformations, the lowest of the three transformations "tiers". Approximately 86% of apps also use repackaging [250]. This is significant as at least 43% of the malware signatures are not based on code-level artefacts and can therefore be broken with trivial transformations on the APK or Manifest.

If malware alter class names, methods, files, or string/array data within the `dex` file (i.e., second tier obfuscation), they can deter 90% of popular AV products [170]. Half of Android apps also use Java reflection to access API calls, which is a top tier obfuscation method [79]. In 2012, ADAM [243] showed results similar to DroidChameleon even when analysing a different set of AV products. Specifically, ADAM stress tested their top 10 AV products by repackaging malware and found that the detection rate lowered by roughly 10%. Interestingly, middle tier obfuscation (e.g., renaming, altering control flow, string encryption) successfully lowered the detection rate further from 16.5% to 42.8%, implying that more sophisticated obfuscation methods are more successful. Furthermore, despite improvements in the AV products' detection rate due to consistent, rigorous, signature updating, as malware shift to stronger obfuscations, this cannot be sustained; one year later ADAM, [214] found AV detection rates fell to 0-32%.

The framework AndroTotal [135], can also be used to analyse a malware with multiple mobile AV products and compare their results. In 2014, Morpheus [116] used static and dynamic techniques to create a wide range of malware for benchmarking computational diversity in mobile malware. Although they have not yet tested them on any AV products or analysis frameworks, such an experiment could be very enlightening. In summary, multiple studies have tested the top AV systems and found them lacking at all

levels of transformations attacks. Furthermore, higher tiered transformations, namely Java reflection and native code (61% and 6.3% of apps studied by Stowaway [79]), are still more successful than lower tiers [139]. Similarly, besides being heavily obfuscated against static analysis, sophisticated malware are also bypassing dynamic analyses like Google Bouncer by detecting emulated environments.

2.4.1.4 A Lack of Representative Datasets

Every Android analysis, detection, and classification system should be evaluated on a dataset of Android app samples, benign and/or malicious. Initially, even a few years after the first Android malware was discovered in 2010 [132], researchers lacked a solid, standard dataset to work with. Many instead wrote their own malware to assess their systems [186]. Other collected and shared samples with website crawlers, such as Contagio [52]. These approaches, however, yielded limited datasets, limiting the ability to thoroughly evaluate new analysis, detection, and classification systems.

In 2012, the MalGenome project [250] attempted to fix this as it contained 1260 malware samples categorized into 49 different malware families and was collected from August 2010 to October 2011. Later that year, at least four notable research projects had used the MalGenome dataset, and in 2013 the number increased by three-fold.

However, based on the rapidly evolving nature of Android malware, it is essential to update the dataset with newer samples to continue testing systems effectively. This, in part, was satisfied with DREBIN [16], a collection of 5,560 malware from 179 different families collected between August 2010 and October 2012. Unfortunately, when considering the continuing increase in malware, 400% from 2012 to 2013 [200], and all the new sophisticated malware that have appeared since 2012 (e.g., Oldboot, Android.HeHe), a more complete and up to date dataset is necessary [142, 144]. For reasons we will explain in Section 2.4.2, it is also essential to have a diverse dataset, with samples from a range of years, app categories, popularity, markets, etc.

2.4.1.5 IoT

One interesting point of discussion is the Internet of things (IoT), the concept that everything from keys to kitchen appliances will be connected via the Internet. This poses many interesting possibilities, as well as security concerns, as there is a high likelihood that a growing IoT will adopt an open-source, popular, reasonably sized OS, such as Android. Hence portable Android analysis frameworks may be even more desirable. We are already beginning to see Android watches, i.e. Android Wear.

Efforts have also been made to adopt the Android operating system for satellites, espresso makers, game controllers, and refrigerators [211]. If the IoT were to adopt smaller, altered versions of the Android OS, it would give researchers an incentive to create portable analysis and detection tools so they may be usable across all Android OS versions no matter what device it powers. This would be even more effective, if done in conjunction with improved application market vetting methods.

2.4.2 Mobile Security Effectiveness

To evaluate the present status of Android malware analysis and detection frameworks, we analysed over 35 studies from the 2011-2015 time period. As seen in Figure 2.1, this is the time in which the Android OS dominates in popularity. In addition to our analyses and extrapolations in this section, all referenced studies have been compiled into Tables A.1 and A.2, found Appendix A due to their large sizes. These tables help provide details on framework methods (e.g., static or dynamic), sample selection process, scalability, accuracy, and sturdiness against obfuscation and changes between Android versions.

2.4.2.1 Analysed Datasets

As mentioned previously, sample selection is essential as different app markets and geographic locations are infected by dissimilar malware sets and in varying amounts [115, 183, 248]. Many studies, however, only use one app source and either choose several apps per category (e.g., games, business), or select apps that best test their system. For example, SmartDroid [242] chose a small set of malware triggered by UI to test its system for revealing UI-based triggers. In most cases, however, a diverse, representative dataset is desired. Interestingly, this may be more complicated than originally believed. For example, AppProfiler [175] discovered that popular Google Play apps exhibited more behaviours, and were more likely to monitor the hardware, than an average app. This is significant as many studies, e.g. [55, 119, 241], only analysed popular apps.

Similarly, while analysing free, popular apps may provide more malicious behaviours to analyse, the selection would not be a reasonable representation of the Android markets as a whole. A significant number of studies only analyse free apps, but as ProfileDroid established, paid apps behaved very differently than their free counterparts. For example, free apps processed an order of a magnitude more network traffic [175].

Furthermore, different app markets and geographical locations host different amounts of malware as well as different malware families [144], which should be considered while choosing a dataset for analysis or testing detection or classification frameworks.

2.4.2.2 Scalability, Accuracy, and Portability

Scalability when processing large amounts of malware and information is a vital trait as the body of malware grows and diversifies. This is due to the sheer number of samples that need to be analysed, but also so we can quickly identify new malware, flag them for further analysis, and notify others. While most systems scale well enough, some do trade scalability for accuracy, and visa versa; of course, improvements for both are being continuously developed. Despite developing faster or more accurate classifiers, finding different feature sets or ways to map the features into a vector space that the classifier can use have also improved accuracy and performance [9]. Our approach to scalable malware classification can be found in the following chapters.

Within our analyses and Appendix tables, we attempt to make note of any performance statistics or scalability information. We also attempt to base each framework's sturdiness on several key points, made in Section 2.4.1, concerning native code, Java reflection, vm-awareness, and different levels of obfuscation.

With Tables A.1 and A.2, we discovered that several systems were able to detect, but not analyse, samples with such traits. Furthermore, these traits often contributed to their false positives/negatives. An encouraging number of frameworks such as Apposcopy [82] are making efforts to overcome limitations like low levels of obfuscation, but are still vulnerable to higher ones. Portability is also essential, so that malware can be analysed on multiple Android OS versions, as they have different vulnerabilities, and to minimize the window of vulnerability whenever a new Android version is released.

2.4.2.3 Significant Changes in Android

With the significant changes in the Android runtime, it is important to see which frameworks can adapt to ART. Ideally, solutions should be agnostic to the Android runtime, however many static solutions rely on the Dalvik dex file, as opposed to the new odex files, and many dynamic solutions either modify or are very in-tuned with specific aspects of runtime internals. It is possible that no more drastic changes will be made to the Android OS, but ideally frameworks should be resilient or easily adaptable to changes within Android. Furthermore, given the constant release of Android versions seen in Table 1.1, it is highly likely that more changes are to come. The benefit of being resilient to these changes is high portability across all Android versions, Android variants in a future Internet Of Things, and possibly even other platforms. Changes in the Android hardware to support virtualization (i.e., ARM Cortex A-15) may also help determine further security against malware with less added complexity (e.g., heavy modifications).

2.5 General Areas for Future Research

In summary, we feel Android malware analysis is trending in the right direction. Many simple solutions and anti-virus products do provide protection against the bulk of malware, but methods such as bytecode signatures are weak against the growing amount of advanced malware [143, 183, 200, 214]. We therefore suggest the following areas for future research, including several addressed in this thesis.

2.5.1 Hybrid Analysis and Multi-Levelled Analysis

Static solutions are beginning to harden against trivial obfuscations [82], but many apps, and most malware, are already using higher levels of obfuscation [79, 183]. As current static systems are still effective and scalable, in the cases where obfuscation (e.g., native code, reflection, encryption) is detected, dynamic analysis can be used in conjunction for a more complete analysis. Alternatively, dynamic solutions inherently have less code coverage, but can use static analysis to guide analyses through more paths [137, 193, 242], or use apps exercisers like MonkeyRunner, manual input, or intelligent event injectors [22, 133, 137]. Hybrid solutions could therefore combine static and dynamic analysis in ways that their added strengths mitigate both weaknesses.

It also seems beneficial to develop multi-level systems, as it often provides more, richer, features. Furthermore, in a multi-level system analysis, it would be harder for malware to hide actions if multiple layers of the Android architecture are being monitored. Furthermore, while some layers may be tampered by malware or only to analyse some Android versions, monitors on the lower levels should be able to function securely across all OS versions. Parallel processing could also greatly enhance multi-level analyses and provide faster detection systems [66]. The downside of this multi-level method, however, is it can cause large additional overhead, decrease transparency, and be less portable. Hence, at this point in time, we believe that the most desirable techniques enable the analysis of multiple layers from a single low-point of observation.

2.5.2 Code Coverage

As mentioned previously, code coverage is essential for complete, robust malware analyses. Statically, this can be difficult when dealing with dynamically loaded code, native code, and network-based activity. Dynamically, this is challenging as only one path is shown per execution, user interactions are difficult to automate, and malware may have split-personality behaviours. There are several benefits to dynamic out-of-box solutions, considering the launch of ART [218], like being able to cope with available

Android versions, and to bar malware avoiding analyses with native code or reflection. For example, system-call centric analysis is out-of-the box but can still analyse Android-level behaviours and dynamic network behaviours (see Chapter 3) and can be used to stop certain root exploits [215]. While hybrid solutions and smarter stimulations would greatly increase code coverage, different approaches should be further researched based on malware trends. For example, while manual input is normally not scalable, crowd sourcing [87] may be an interesting approach. However, zero-day malware will introduce complications as time is needed to create and collect user input traces.

This also introduces an interesting question on whether malware tend to use "easily" chosen paths to execute more malicious behaviour, or harder paths to avoid detection. This would be an interesting area for future research, as it would help identify malware trends and, therein, increase the effectiveness of future analyses. Another topic of interest is identifying and understanding subsets of malware behaviour through path restrictions (e.g., remove permissions or triggers like user UI or system events), to see what behaviour equates to what permission(s) and/or trigger(s).

We also feel that there needs to be a better understanding of when an event is user-triggered or performed in the background and how. To increase code coverage, apps should also be run on several different Android OS versions as different versions have different sets of vulnerabilities (several root exploit examples found in Chapter 5). This would be much more difficult to implement if any modifications were made to the runtime or the OS to accommodate for high-level analyses.

2.5.3 Hybrid Devices and Virtualization

In addition to smart stimuli, modifying emulators for increased transparency (e.g., realistic GPS, realistic phone identifiers) or using emulators with access to real physical hardware (e.g., sensors, accelerometer) to fool VM-aware malware may prove useful [239]. Newer, more sophisticated, malware from 2014 and 2015 are becoming increasingly aware of emulated environments, but achieving a perfect emulator is, unfortunately, infeasible. Things such as a timing attack, where certain operations are timed for discrepancies, are still open problems for traditional malware as well. Furthermore, such malware (e.g., DenDroid, Android.HeHe) do not just detect their emulated environments, but often hide their malicious behaviours or tamper with the environment.

Based on a previous study, malware can check on several device features to detect emulators. This includes, but does not stop at, the device IMEI, routing table, timing attacks, realistic sensory output, and device serial number [162]. It is also possible to

fingerprint and identify specific emulated environments, e.g. dynamic analysis frameworks, via the aforementioned device performance features [138]. One solution would be to use real devices in all dynamic experiments. However, this makes analysing large malware sets a laborious and expensive task, as many devices would be needed as well as a way to restore a device to a clean state for quick, efficient, and reliable analysis.

Thus it would be interesting to combine real devices and emulators as a hybrid solution, where real devices pass necessary values to emulators to enhance their transparency. Real device data can also be slightly or randomly altered and fed to multiple emulators. This would ideally reduce the cost and speed of experiments while revealing more malicious behaviours. This hybrid method has proved to be effective for analysing embedded systems' firmware [239], and it would be interesting to see if it could work for detection and analysis, and how effective it would be against VM-aware malware.

Alternatively to virtualization, it would be interesting to split the Android kernel so the untrusted system calls are directed to the hardened kernel code. This method has only been applied to a traditional Linux kernel, and it would be interesting if regular application system calls can be redirected to, and monitored by, the hardened part of the "split" kernel [124]. Lastly, we look forward to new technology, such as the new ARM with full virtualization support, and more explorations into ART and its new challenges.

2.5.4 Datasets

Sample selection is essential for accurate analysis. For example, as be seen from Tables A.1 and A.2, many studies draw from only one market but due to many social, geological, and technical factors, different markets host varying amounts of malware from different families [127, 183]. Malware samples should also be chosen from several families, unless testing for very specific behaviours, to provide a more diverse set of behaviours and evasion techniques to analyse. A full list of malware families analysed in Chapter 3 can be found in Appendix B, and more details on the families used in Chapter 4 can be found in [54]. Although the latter does not provide an exhaustive list, the dataset used is larger, more current, and a superset of those in Appendix B.

Similarly, market apps should be chosen from several categories and some free/paid pairs if possible. This is because a paid app behaves differently from its free version, and popular apps behave differently than unpopular ones. However, as popular apps affect more users, it could be argued that they are more essential for research [144, 175].

In the future, datasets should continue to be expanded by incorporating malware from multiple sources to provide more globally representative datasets. If used correctly, specialized datasets for benchmarking (e.g., DroidBench [18]), testing types of

obfuscation (e.g., DroidChameleon [170]), etc., would also be highly useful to identify specific weakness or traits in analyses. As the work in this thesis focuses on malware specifically, we draw from several large and diverse datasets. However, in the future, it would be interesting to introduce benign applications into the frameworks in this thesis. One such dataset is PlayDrone, which contains over a million Google Play apps [217].

2.6 Thesis Goals and Contributions

In this chapter we studied multiple Android malware analysis and detection frameworks and illustrated trends in the state-of-the-art systems. We also analysed the mobile malware evolution as it adapts to obstruct analysis and avoid detection. By analysing both threats and solutions, we have identified several areas that require further research and development. The author's contributions in the following chapters aims to meet the following goals, which were shaped by the discoveries in this chapter.

Through our analysis, and by laying out all these Android studies in the extensive tables in Appendix A, we saw the need to develop more effective methods for low-level dynamic analysis to counter the high number of studies unable to analyse native code, dynamically loaded code, etc. From there, we also saw further opportunities to develop novel malware classifiers using our unique behavioural profiles.

In general, the research goals the author set out to fulfil based on the evidence procured from surveying the current body of work are as follows. Each goal refers to a research gap identified in the previous sections, and will be used to evaluate the successfulness and novelty of the author's contributions in the following chapters.

Goal 1 *Analyse network traffic, native code, encryption, etc., for code coverage as discussed in Sections 2.1.4 and 2.5.2. This goal requires some level of dynamic analysis.*

Goal 2 *Gaining rich and thorough behaviour profiles without modifying the Android VM, OS, or applications, as discussed in Sections 2.1.4, 2.2.2.2, and 2.4.2. This adds robustness against changes in the Android OS and could be adapted to other platforms.*

Goal 3 *Scalable computations, e.g. analysis and classification, when dealing with large malware datasets, as discussed in Section 2.2.6.*

Goal 4 *Overcome as many malware anti-analysis techniques, e.g. obfuscation techniques as described in Section 2.3, as possible to enable the accurate analysis of sophisticated malware, such as those described in Section 2.1.1.*

Chapter 3

Automatic Reconstruction of Android Behaviours

Contents

3.1 Introduction

As illustrated in previous chapters, the popularity of Android has unavoidably attracted cybercriminals and increased malware in app markets at an alarming rate. To better understand this slew of threats, the author augmented a base CopperDroid framework, an automatic VMI-based dynamic analysis system, to reconstruct Android malware behaviours. The novelty of the author's work lies in its agnostic approach to the reconstruction of interesting behaviours at different levels, by observing and dissecting system calls. The on-going CopperDroid project is therefore resistant to the multitude of alterations, or replacements (i.e., ART), the Android runtime is subjected to over its life-cycle. Moreover, CopperDroid can adapt to changes in the system call table.

The improved CopperDroid *automatically* and *accurately* reconstructs events of interest that describe, not only well-known process-OS interactions (e.g., file and process creation), but also complex intra- and inter-process communications (e.g., send SMS), whose semantics are typically contextualized through complex Android objects. Because of this, CopperDroid can capture actions initiated both from Java and native code execution, unlike many related works both static and dynamic. Thus the improved CopperDroid's analysis generates detailed *behavioural profiles* that abstract a large stream of low-level — often uninteresting — events into concise, high-level semantics, which are well-suited to provide insightful behavioural traits and opens possibilities for further research directions. In the following chapter, Chapter 4, we test the usefulness of these profiles by utilizing them to scalably classify malware into known families.

Unfortunately, the nature of Android makes it difficult to rely on standard, traditional, dynamic system call malware analysis systems *as is*. While Android apps are generally written in the Java programming language and executed on top of the Dalvik virtual machine [35], native code execution is possible via the Java Native Interface. This mixed execution model has persuaded other researchers, see Appendix A, to reconstruct, and keep in sync, different semantics through virtual machine introspection (VMI) [85] for both the OS and Dalvik views [233]. Zhang et al. further stressed this concept by claiming that traditional system call analysis is ill-suited to characterize the behaviours of Android apps as it misses high-level Android-specific semantics and fails to reconstruct inter-process communications (IPC)[1] and remote procedure call (RPC) interactions, which are essential to understanding Android app behaviours [241].

In a significantly different line of reasoning from [75, 241], we observed that system call invocations remain central to both low-level OS-specific and high-level Android-

[1] Android IPC is also known as inter-component communication (ICC) [75].

specific behaviours. However, as mentioned previously, a traditional or simplistic analysis of system calls would lack the rich semantics of Android-specific behaviours.

This is where the novelty and real value of CopperDroid lies; the author's contributions enable seamless and automatic dissection of complex IPC messages from system calls, resulting in the deserialization of complex Android objects. This is achievable with the *unmarshalling Oracle*, developed by the author and showcased in this publication [202]. It is this Oracle that enables the reconstruction of Android app behaviours at multiple levels of abstraction from a single point of observation (i.e., system calls). Equally as important, this approach makes the analysis agnostic to the runtime, allowing our techniques to work transparently with all Android OS versions. For instance, we have successfully run CopperDroid on Froyo, Gingerbread, Jelly Bean, KitKat, and the newest Lollipop (i.e., Android 5.x running ART) versions with no modification to Android and minimal configuration changes for CopperDroid. In summation, we present the following three contributions as resulted from the author's research efforts.

1. **Automatic IPC Unmarshalling:** We introduce CopperDroid as a base, dynamic, system call collector (i.e., no analysis included), and present the design and implementation of a novel, practical, oracle-based technique to *automatically* and *seamlessly* reconstruct Android-specific objects involved in system call-related IPC/ICC and RPC interactions. The author's approach avoids manual development efforts and transparently addresses the challenge of dealing with the ever increasing number of complex Android objects introduced in new Android releases. The Oracle addition allows CopperDroid to perform large-scale, automatic, and faithful reconstruction of Android apps behaviours (Section 3.4), suitable to enable further research, including Android malware classification and detection.

2. **Value-based Data Flow Analysis:** To abstract sequences of related low-level system calls to higher-level semantics (e.g., network communications, file creation) and enrich our reconstructed behavioural profiles, the author wrote a tool to automatically build data dependency graphs over traces of observed system calls and perform value-based forward slicing to cluster data-dependent system calls. Moreover, this gives CopperDroid the ability to automatically recreate file resources associated with a data dependent graph or "chain". This compression of system call sequences into behavioural profiles summarizes each action's semantics and, during file system accesses, can provides access to reconstructed resources. These files may be further inspected dynamically or statically by complementary systems or, if an APK, be fed back to CopperDroid.

3. **Behaviour Reconstruction and Stimulation:** We provide a thorough evaluation of CopperDroid's behavioural reconstruction capability on more than 2,900 Android malware samples provided by three sources [52,141,249]. Furthermore, our experiments show how a simple yet effective malware *stimulation* strategy allows us to disclose an average of 25% of additional behaviours on more than 60% of the analysed samples, qualitatively improving our behavioural reconstruction capabilities with minimal effort and negligible overhead (Section 3.6). Incremental stimuli is also experimented with, for a more fine-grained analysis.

Through our examination of other works on Android malware analysis, see Section 3.8 and Chapter 2, it is our belief that CopperDroid's unified reconstruction significantly contributes to the state-of-the-art reconstruction of Android malware behaviour.

3.2 Relevant Background Information

Android applications are typically written in the Java programming language and then deployed as Android Packages archives (APKs). The contents of an APK, such as the Android Manifest, has already been discussed in Section 2.2.1.

3.2.1 Android Applications

Each app runs in a separate userspace process [12] as an instance of the Dalvik VM [35], or with the ART runtime if running the newest Android 5.0. This provides each app process with a distinct user and group ID. Although isolated within their own sandboxed environment, see Figure 3.4 (page 64), apps can interact with other app components and the system via well-defined APIs. Each APK is also considered to be self-contained and can be logically decomposed into one or more components, as previously defined in Section 2.1.2. Each component is generally designed to fulfil a specific task (e.g., GUI-related actions) and is invoked either by the user or the OS. Regardless of the invocation, however, all activities, services, and broadcast receivers are activated by `Intents`, i.e. asynchronous IPC/ICC messages between components to request an action.

Components must be declared within the Android manifest, written in XML file. A manifest must also declares the set of permissions the application requests, as seen in Figure 3.1 and Appendix D, along with the hardware and software features the application uses. Through static analysis of a sample's manifest, CopperDroid uses a basic stimulation technique based on the permissions a sample has asked for. More sophisticated analyses of the manifest have been discussed in [251] and Chapters 2 and 5.

```
<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/[...]"
    package="test.AndroidSMS"
    android:versionCode="1"
    android:versionName="1.0">

<application android:label="@string/app_name" >
</application>

<uses-permission android:name="[...].RECEIVE_SMS" />
<uses-permission android:name="[...].INTERNET" />
...
 <receiver android:name=".SMSReceiver">
  <intent-filter>
   <action android:name="..Telephony.SMS_RECEIVED" />
...
```

Figure 3.1: Example of a simplified Manifest file from an APK.

3.2.2 Inter-Process Communications and Remote Procedure Calls

The Android system implements the principle of least privilege by providing a sandbox for each installed app. Therefore, one process may not manipulate the data of another process and can only access system components if it explicitly requested the corresponding permission(s) in the manifest. Nevertheless, user-level processes, i.e. apps, often require inter-process communications. For example, an app granted the permission to send SMS can do so through the appropriate service via a remote method call. This is achieved with the Android Binder, a custom implementation of the OpenBinder protocol[2] [157]. The Android Binder has many unique features, including its optimized inter-process communication (IPC) and remote procedure call (RPC) mechanisms.

Just like any other IPC mechanism, Binder allows a Java process (e.g., an app) to invoke methods of remote objects (e.g., services) as if they were local methods, through synchronous calls. This is transparent to the caller and all the underlying details (e.g., message forwarding to receivers, start or stop of processes) are handled by the Binder protocol during the remote invocation. When IPC is initiated from a component $\mathcal{A}$ to a component $\mathcal{B}$ (both components may belong to the same app) the calling thread in $\mathcal{A}$ will wait until the next available thread in the thread pool of $\mathcal{B}$ replies with the results. Figure 3.2 illustrates the different forms of component interactions where all the arrows are performed with IPC. The calling thread returns as soon as it receives such a result. The data sent in such transactions are in `Parcels`, a buffer of often serialized (marshalled) flattened data and meta-data information. As IPC occurs between OS-level apps (see

[2]OpenBinder is no longer maintained.

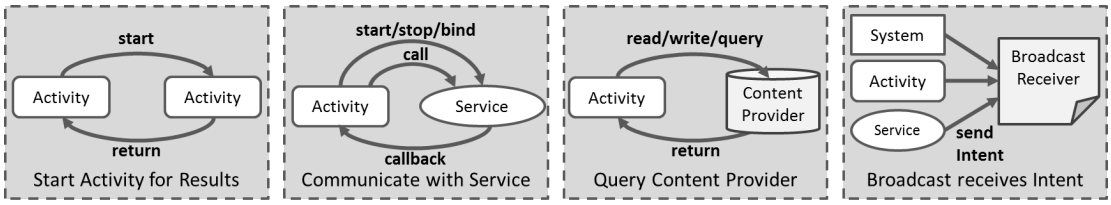


Figure 3.2: Possible IPC interactions between app components.

Figure 3.4 page 64), it may seem that IPC also occurs at a high-level and is, therefore, outside the scope of system calls. In actuality, however, all these communications are routed through the Android Linux kernel or, more specifically, the Binder Driver.

The Binder protocol is implemented as a Linux kernel driver controlled by the special Linux virtual device `/dev/binder`. The interactions between an instance of the Dalvik VM running an Android app and the Binder protocol happens through `ioctl` (i.e., input output control) system call invocations. Thus, whenever Android is dispatching a the message between $\mathcal{A}$ and $\mathcal{B}$, CopperDroid can intercept the resulting `ioctl` system call handled by the Binder kernel driver. Every Binder`ioctl` call takes a Binder protocol command and a data buffer as arguments (see Figure 3.3).

A `BINDER_WRITE_READ` with `BC_TRANSACTIONS` is the most important command as it allows data transfers between distinct processes using IPC. Another command of interest is `BINDER_WRITE_READ` with `BC_REPLY`, in the cases where one `ioctl` does not hold all the data; even when using a pass-by-reference in place of the marshalled data, all must eventually passes through IPC channels in flattened `Binders`.

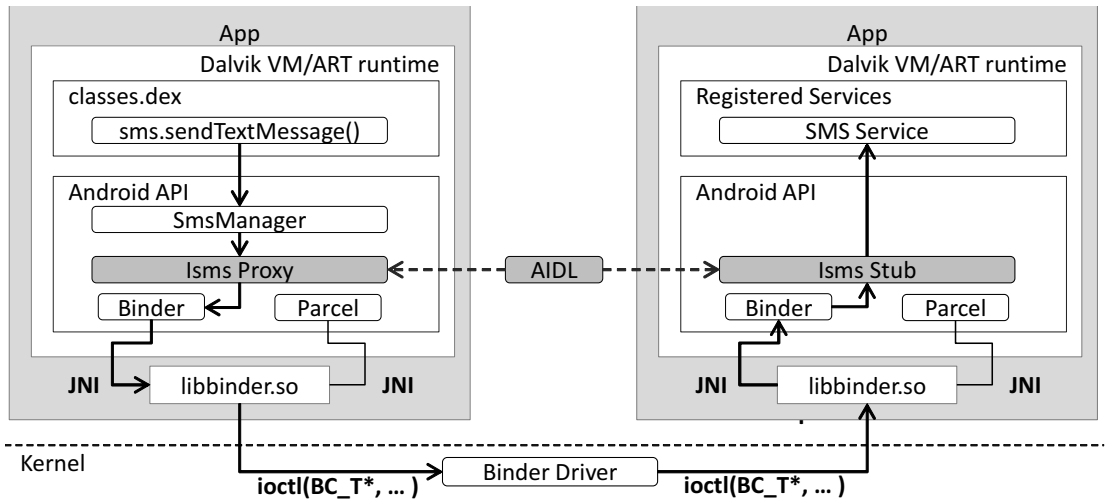


Figure 3.3: IPC routed through Binder driver and resulting `ioctl`s for SMS request.

3.2.3 Android Interface Definition Language

When a service needs to provide a binding for IPC, it must define a client-server interface that allows apps to bind and interact with it. Such an interface is called *bound service*. If the service is used by other apps processes, and requires multithreading, then the interface is normally defined by the Android Interface Definition Language (AIDL). Thus, due to the AIDL of a service, clients know which remote service methods can be invoked and what parameters they take. By analysing the Binder IPC amongst clients and services, CopperDroid can automatically recreate Android-specific behaviours.

In ensure that Binder works properly, the caller app must know the remotely-callable methods and its accepted parameters. This is achieved through the AIDL, leveraged by "server-side" component developers. Once defined, an AIDL file is used to automatically generate client and server side code in the form of a proxy class, used by a caller, and a stub class, extended by the callee to implement the logic of the service (see Figure 3.3). However, these stubs are generated at run-time to suit the method(s) being invoked. As CopperDroid analyses run post execution, it does not have the run-time information on which stubs were generated and, therefore, the information in the stubs themselves.

To account for this, CopperDroid generates all possible stubs prior to analysis. As Android is open-source, CopperDroid can do so and use such interfaces to *automatically* infer actions between apps from just system calls. Although a few AIDL files may be missing (e.g., custom services), CopperDroid has never experienced such an issue.

3.2.4 Native Interface

While the main technology to develop Android apps is Java, it is possible to embed small pieces of native code (i.e., C, C++), compiled as shared libraries to be dynamically loaded at runtime. Once loaded, native functions can be invoked by Java code and, as they run on the underlying system, are less restricted by Android security.

Benign applications use native code to perform CPU-bound operations (e.g., a physical engine) that require little interaction with other components. Malicious apps, on the other hand, are known to leverage native code to perform low-level operations such as utilizing vulnerabilities to escalate privileges or obfuscate app code [250]. Native code can bypass all static analysis methods, as well as some in-the-box methods (see Section 2.3). This is significant, as our analysis of Android malware (summarized in Figure 2.3 page 24) showed that roughly 60% of malware used native code in 2010, and nearly all do so by 2015. As an alternative method for executing native ARM code, an app could include an executable and linkable format (ELF) file in its resources for later use.

3.3 Overview of CopperDroid

In CopperDroid an unmodified Android image runs in our emulator. This emulator is built on top of QEMU [26], as shown in Figure 3.4. While an app runs within the emulator, the CopperDroid plugin collects the resulting system call information and performs out-of-the-box behaviours reconstruction with those traces afterwards. While the collection phase was developed by collaborators, all essential and complex behaviour reconstruction analyses were performed by the author. This primarily includes implementing an Oracle to reconstruct Binders within IPC. To this end, QEMU is minimally modified to enable a system call tracking plugin. The validity of these system calls is based on the plugin's trustworthiness. If CopperDroid methods are later applied to on-device (e.g., in a hypervisor) or on-cloud analysis, that collection point must also be trusted.

3.3.1 Independent of Runtime Changes

It is worth stressing that *all* our analyses are executed outside the CopperDroid emulator, and that the framework relies on well-known VMI [85] techniques to fill the semantic gap between CopperDroid and the Android OS. This flexibility allows our system to be largely decoupled from any specific Android environment, enabling seamless integration across different OS versions. This provides a transparent environment to *automatically* perform out-of-the-box dynamic behavioural analysis on any Android app. For this work, we are specifically interested in Android malware. To this end, CopperDroid presents a unified analysis to characterize low-level OS-specific and high-level Android-specific behaviours from the system calls apps *must* invoke to achieve anything.

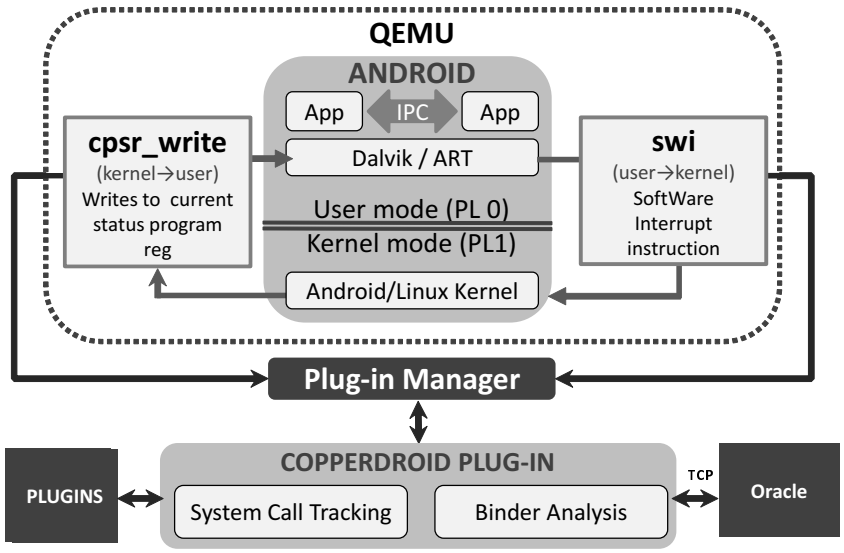


Figure 3.4: CopperDroid's overall architecture for system call tracing.

3.3.2 Tracking System Call Invocations

Tracking system call invocations is at the basis of virtually all traditional dynamic behavioural analysis systems [107, 221, 226, 229], and several, less complete, Android approaches as well (previously described in Section 2.2.4). Normally traditional systems implement a form of VMI to track system call invocations on a virtual x86 CPU. Although similar, the Android ARM architecture possess characteristics that challenge the traditional tracking process and are, therefore, worth elaborating on.

The ARM instruction set, i.e. instruction set architecture (ISA), provides the `swi` instruction for invoking system calls. Thus triggering a software interrupt (see Figure 3.4) causes user-to-kernel transition. To track system call invocations, we instrument QEMU to intercept calls whenever the `swi` instruction is executed and copy the system call data from the relevant registers; `r7` (system call number), `r0-r5` (parameters). It is important to note that the modifications to QEMU are minimal and are easily, and automatically, configured to allow for multiple plugins and different Android versions.

The `swi` instructions intercepted are not (dynamically) binary translated and can therefore be easily intercepted when QEMU handles the software interrupt, without pausing the emulator, a minimal overhead. Of course, it is also of paramount importance to detect when a system call is about to return in order to save its return value and enrich our analysis with additional semantic information. Usually, the return address of a system call invocation instruction `swi` is saved in the link register `lr`. While it seems natural to set a breakpoint at that address to retrieve the system call return value, a number of system calls may actually not return at all (e.g., `exit`, `execve`).

To capture return values, CopperDroid adapted by intercepting CPU privilege-level transitions instead. In particular, CopperDroid detects whenever the `cpsr` register switches from supervisor to user mode (`cpsr_write`, see Figure 3.4), which allows it to retrieve system call return values, if any, and continue executing properly. Based on these observation, we see all behaviours are eventually achieved through the invocation of system calls and that CopperDroid's VMI-based, call-centric, analysis faithfully describes Android malware behaviour whether it is initiated from Java or native code.

Thus, tracking (by collaborator) and analysing (by author) system calls and their parameters provides a unique observation point to reconstruct OS and Android specific behaviours. Most importantly, this includes high-level semantic behaviours extracted from marshalled Binder data, which previous works believed impossible. This perspective highlights the strength of our unified analysis: a mere system call tracking could not provide as many behavioural insights if it were not combined with Binder information and automatic (complex) Android objects reconstruction, as outlined in Section 3.4.

3.4 Automatic IPC Unmarshalling

Once we have collected system call traces for our malware set, we can begin analysing the data and reconstructing behaviours. However, just intercepting transactions is of limited use when it comes to understanding Android-specific behaviours. In fact, the raw `ioctl` Binder data that flows throughout transactions are flattened and marshalled into a single buffer. Moreover, as every interface a client and service agree upon has its own set of predefined methods' signature, and as the Android framework counts more than 300 of these AIDL interfaces, manual unmarshalling is infeasible.

This section discusses the reconstruction of high-level IPC transactions by unmarhsalling the data "hidden" within `ioctl` system calls with the help of AIDL data. These elements have been previously discussed in Section 3.2. As outlined there, the Android system heavily relies on IPC/ICC interactions to carry out tasks. Thus, tracking and dissecting these channels is a key aspect for reconstructing high-level Android-specific behaviours. Although recently explored to enforce user-authorized security policies [231], to the best of our knowledge, as of early 2015, CopperDroid is the first approach to carry out a full detailed analysis of inter-process communication channels to comprehensively characterize OS and Android specific behaviours of malicious applications.

3.4.1 AIDL Parser

As mentioned in the background, Section 3.2, the AIDL defines interfaces for remote services, detailing method, parameters and return value. Furthermore, every AIDL definition produces Proxy and Stub classes used to transfer data using IPC. The Proxy is used on the client side and matches the client's method in terms of method name, parameters and return value. The Stub, used on the server side, utilizes the transaction code in order to perform the appropriate unmarshalling actions for the given method call.

Unmarshalling IPC Binders is a necessary step for CopperDroid, as, by design, it can only intercept the data as it flows to the kernel's Binder Driver within an `ioctl` system call. More specifically, this is the point where the data is marshalled into a `binder_transaction_data` structure (see Figure 3.5). Although the AIDL process can easily marshal data between clients and servers during normal runtime, it is necessary for CopperDroid to combine components of the Proxy and Stub in order to unmarshal any and all remote method invocations and their parameters post-execution.

CopperDroid solves this issue by including a *modified* AIDL parser. This script is included in the base CopperDroid, and is not a creation of the author. This parser finds and uses the method names, parameters and return values types (i.e., usually utilized in

the Proxy at runtime) to build a mappings between transaction `codes` and methods. It then combines this information with the parcel reader class mentioned earlier to automatically produce handling code for a given method. CopperDroid utilizes this code to extract the necessary information from any Binder call during analysis.

All automatically generated AIDL information is stored in multiple Python files, preserving the mapping of all interface names to their respective parcel extraction routines. For example, the `com.android.internal.telephone.ISms` token maps to the `db_ parcel_ISms` function call (see Figure 3.5). As parsing the AIDL is only required once per Android OS version, it can be done while setting up the framework prior any analyses. Thus, this introduces no overhead during important phases. However, while AIDL is available to make IPC easier, it is not the only means to define interfaces for client and service communication. There are Android services that hard-code method and `code` switch statements. Although CopperDroid may still extract transaction IDs automatically, further work will be needed to find the proper unmarshalling method in these cases, perhaps by program analysis of the service's bytecode.

3.4.2 Concept for Reconstructing IPC Behaviours

To analyse OS-specific events, such as a network connection or file access, CopperDroid relies on a value-based data flow analysis (Section 3.5.1) to abstract a sequence of, not-necessarily consecutive, system call invocations to high-level semantics. To reconstruct

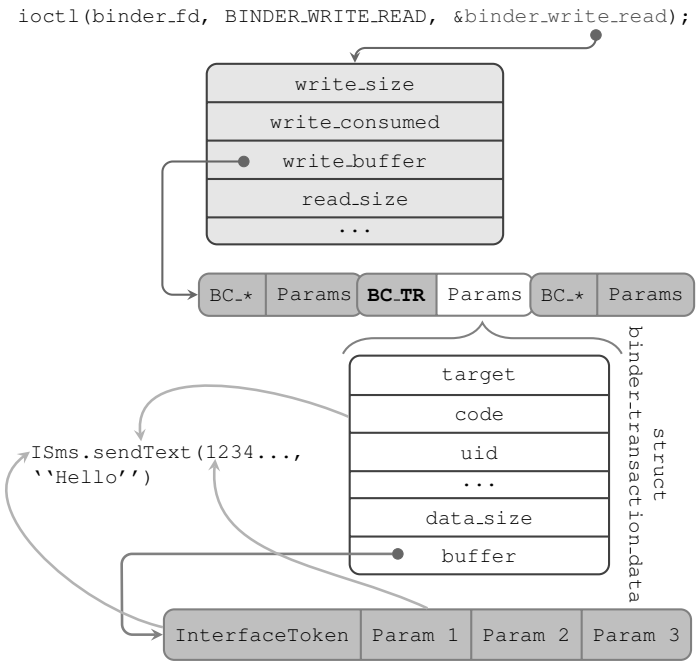


Figure 3.5: An example Binder payload corresponding to a SMS `sendText` action.

Android-specific behaviours, CopperDroid introduces the more complex unmarshalling Oracle, which enables automatic deserialization of all IPC Binder transactions. This is essential as over 78% of our malware samples invoked remote binder methods.

To fully recreate IPC binder transactions, CopperDroid intercepts `ioctl` system calls with `BINDER_*` flags and parses their payloads. As seen in Figure 3.5, the InterfaceToken, which is a fixed number of bytes, is the first section of the buffer to be parsed and analysed. Using data from the AIDL parser, the InterfaceToken is paired to the corresponding method invoked and the number, and types (e.g., integer), of its accepted parameters. This signature is sent to the unmarshalling Oracle, a Java app that runs in a vanilla Android emulator alongside the CopperDroid emulator.

The purpose of the Oracle is to receive each Binder method signature (e.g., `ISms.sendText(string, string, ... ))` and its marshalled arguments. Using Java reflection, the Oracle is able to reconstruct the method's parameter values with introspection and incision (i.e., object examines and edits itself), and return a complete representation of the method invoked via IPC. Once CopperDroid has received this human-readable representation, it can enhance the behaviour profile of the involved sample(s). This is of paramount importance for completely abstracting Android specific behaviours, as we shall fully demonstrate in Figures 3.8 and 3.12 (pages 75 and 79).

3.4.3 Unmarshalling Oracle

By default, the Oracle is queried offline, i.e. after malware execution. The relevant system call data collected from CopperDroid's emulator is then sent to the Oracle as three different sets of data; (1) marshalled IPC data, (2) the signature of the invoked method, and (3) extra binder handle data via CopperDroid's binder analysis.

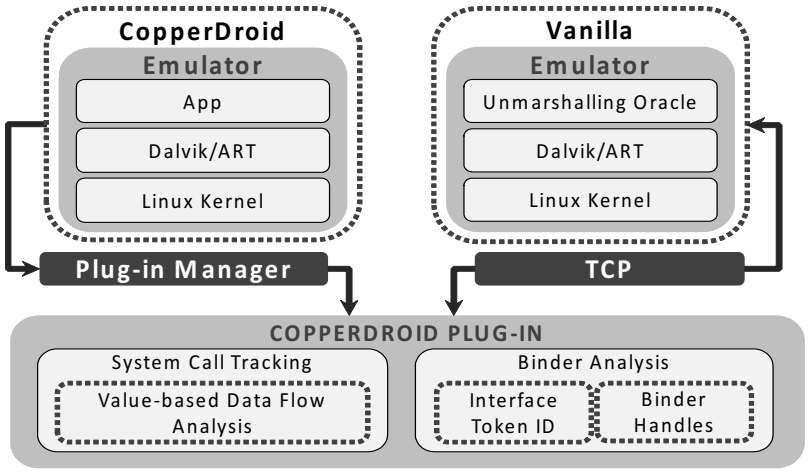


Figure 3.6: CopperDroid architecture in relation with the Oracle and analyses.

3.4.3.1 Oracle Input

We extract two central pieces of data per `ioctl` containing the `BINDER_WRITE_READ` flag. First, the marshalled parameters is taken from the `buffer` field, as can be seen in Figure 3.5. This is essentially the buffer minus the InterfaceToken, a length-prefixed UTF-16 string located at the front. These serialized parameter values are sent via TCP to the Oracle for post-processing (see Figure 3.6). The second extracted piece of information is the numeric `code` that, when united with the `InterfaceToken`, uniquely identifies the method invoked by the IPC binder transaction.

The `InterfaceToken` and `code` of one `ioctl` is used by CopperDroid to isolate the correct, automatically-generated, AIDL data extraction routine. Furthermore, as it is normal for Interfaces to have multiple functions, the `code` is necessary to identify the specific, unique, method invoked. An example routine can be found on page 76. This routine returns the AIDL description of the interface and enables the Oracle to understand the types of the parameters contained in the buffer and unmarshal them. Refer to Figure 3.8, in Section 3.4.5, for a top to bottom example for sending an SMS.

Briefly, within our SMS example, after the `ISms.sendText()` method is invoked, CopperDroid intercepts the corresponding binder transaction and uses the `InterfaceToken` "ISms" and the `code` "sendText" to identify the correct handling function that retrieves information on that specific remote method. For this particular case, this entails the method name ("sendText"), its parameters (destAddr, scAddr, text, sentIntent, deliveryIntent), and its parameter types (String, String, String, PendingIntent, PendingIntent). This information is sent to the Oracle along with the marshalled buffer containing, among other things, the body of the SMS. The Oracle uses this data to extract the values of these parameters when `ISms.sendText()` was invoked.

3.4.3.2 Oracle Algorithm and Output

The Oracle relies on Java reflection to unmarshal the complex serialized Java objects it receives and returns their string representations to CopperDroid to enrich its behavioural profile with Android-specific actions. Therefore, the Oracle must be run with the same Android OS version as the CopperDroid emulator. However, it is worth noting that the unmarshalling Oracle does not require access to the app, or malware, being analysed in the separate CopperDroid emulator. All information necessary to unmarshal the Binder data is retrieved from the CopperDroid emulator and sent over to the vanilla Android emulator running the Oracle, as depicted in Figure 3.6 (page 68). Separate emulators also prevents malware in the CopperDroid emulator from tampering with the Oracle.

Algorithm 5.1 outlines the working details of the unmarshalling Oracle. Currently, it unmarshals method parameters in one of three ways depending on whether the type of data is a primitive or basic type (e.g., String, Integer), an Android class object (e.g., Intent, Account), or a Binder reference object (e.g., PendingIntent, Interface). As mentioned previously, the data types are determined by CopperDroid's AIDL parser and the list of parameter types is sent to the Oracle along with the marshalled parameter values. As an example, primitive or basic types are easily unmarshalled by invoking the appropriate unmarshalling function provided by the `Parcel` class (e.g., `readInt`).

The following sections provide additional details on the automatic unmarshalling process. Once a parameter has been unmarshalled, the Oracle creates a string representation by recursively inspecting each field through Java reflection (see Section 3.4.4). The string representation is appended to an output string list, and the marshalled data offset is updated to point to the next item to be unmarshalled. Additionally, the Oracle iterates to the next parameter type provided by the method signature. To maintain currency, future testing may be necessary to deal with unusual cases, e.g., do not involve AIDL defined transactions and any other customized client/service interfaces.

ALGORITHM 3.1: The Unmarshalling Oracle for IPC reconstruction.

Data: Marshalled binder transaction and data types (determined with AIDL)
Result: Unmarshalled binder transactions

```
1  while data → marshalled do
2      determine type of marshalled item;
3      if type → primitive then
4          automatically apply correct parcelable read/create functions;
5          append string repr. to results;
6      else
7          locate CREATOR field for reflection;
8          use java reflection to get class object;
9          for every class field do
10             if field → primitive then
11                 append string repr. to results;
12             else
13                 explore field recursively;
14                 append string repr. to results;
15             end
16         end
17         if CREATOR → not found and buffer → binder reference type then
18             Unmarshal Binder reference then Unmarshal referenced object;
19         end
20     end
21 end
```

Primitives: While iterating through the list of parameter types and class names (e.g., in our SMS example the Oracle would iterate through three `String` types and two `PendingIntent` types), if the type is identified as primitive (e.g., `String`) the corresponding read function provided by the `Parcel` class is used (e.g., `readString()`). The while loop at line 1 in Algorithm 5.1 would iterate through those five parameters, while lines 3-5 are responsible for primitive types. In our SMS example, the parameters destAddr, scAddr, and text have primitive `String` types and would therefore be unmarshalled using the correct `readString() Parcel` function in order to regain the parameter values, such as the SMS text body "Hello" and destination phone number.

Class objects: To unmarshal a class instance the Oracle app requires Java reflection (see lines 7-8 in Algorithm 5.1). This method allows the Oracle to dynamically retrieve a reference to the `CREATOR` field, implementing the Android `Parcelable` interface.

More specifically, all abstract objects must implement the `Parcelable` interface and, therefore, posses a CREATOR field when being written to, and read from, a `Parcel` [12]. Once an object's CREATOR field has been located, the Oracle can begin reading the remaining class data by obtaining the class loader, creating an instance of the `Parcelable` class, and instantiating it from the given Parcel data by invoking the `createFromParcel()` method. In our example in Figure 3.8, the class data of an Android `Intent` (sent as a `PendingIntent`) includes the phrase "SENT".

IBinder objects: As mentioned previously, certain types of Binder objects are not marshalled and sent via IPC directly. Instead a reference to the object, stored in the caller memory address space, is marshalled and sent in a `BC_TRANSACTION ioctl`. In this case, if the object is neither a primitive nor directly marshalled (see Algorithm 5.1, line 17), the Oracle verifies whether the data contains a binder reference object. This requires parsing the first four bytes of the marshalled object to determine the `IBinder` reference type. As of 2015, no other study we are aware of reconstructs IBinder objects.

`IBinder` reference types determine whether the referenced object is a Binder service (i.e., `BINDER_TYPE_(WEAK_)BINDER` — a transaction sending a handle and service name to the `IServiceManager`), a Binder proxy (i.e., `BINDER_TYPE_(WEAK_)HANDLE` — to send objects from clients including `IInterface` classes represented as a binder object), or a file descriptor (`BINDER_TYPE_FD`). Normally the Binder reference keeps the object from being freed; however, if the type is weak, the existence of the reference does not prevent the object from being removed, unlike a strong reference. Furthermore, as discussed in Section 3.4.4, the opening, closing, inheritance, and modes of all `fd`'s are carefully reconstructed by CopperDroid for accurate behaviours.

Referring to our example in Figure 3.8, a `PendingIntent` can be given to the server process to broadcast back to the client if the remote `sentText` method was successfully invoked. `PendingIntents` encapsulate an `Intent` and an action to perform with it (e.g., broadcast it, start it). This is significantly different from regular `Intents` as they also include the identity of its app creator, making it easier for other processes to use the identity and permissions of the app that sent the `PendingIntent`.

By reconstructing this `Intent`, the Oracle can deduce its purpose. However, the `Intent` in the `PendingIntent` is not sent over IPC directly, but rather its handle reference instead. Thus, when the Oracle unmarshals the `ioctl` call, just as the receiving process would have in real-time, it obtains a reference instead of the object. This reference may contain an address or a handle to the actual marshalled `Intent` and its content. In our SMS example, the handle has the value "`0xa`" (see Figure 3.8 (c)). Fortunately, when sending the IPC message to the server, Binder does pass along the information necessary to seamlessly retrieve the original marshalled data.

With the referenced-based parcelling used by Binder, the Oracle needs to retrieve live ancillary data from the system in order to be able to map the references to their data. To this end, CopperDroid keeps track of the generation of these references, see `ioctl` 2 and `ioctl` 3 in Figure 3.7, in real-time. In our SMS example, in Figures 3.7 and 3.8, there is a transaction (`BC_TRANSACTION`) for registering the `Intent` (i.e., for the `PendingIntent`) and corresponding response (`BC_REPLY`) with its handle reference. To extract this information, CopperDroid needs to identify the binder handle (e.g., the marshalled binder reference) passed as a reference in the binder reply, i.e. `BC_R` in our figures. To avoid using ad-hoc extraction procedures and to rely on automatic mechanisms, CopperDroid itself extract the handles/references. The CopperDroid plu-

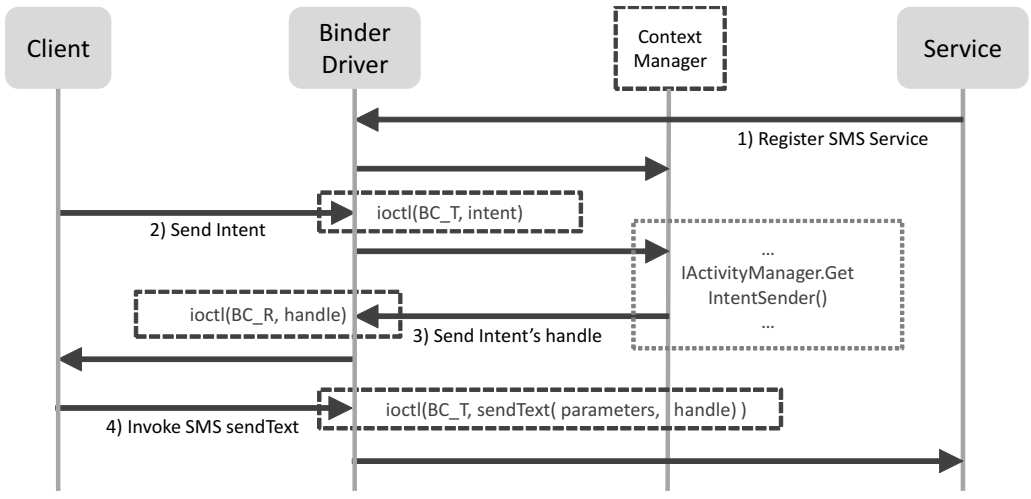


Figure 3.7: Pairing IBinder handles to its serialized content using four `ioctls`.

gin does so with the support of the AIDL parser, and *only* unmarshals handles in these exceptional cases, never the actual objects to minimize performance overhead.

Thus, whenever CopperDroid intercepts a binder transaction that uses handle referenced memory, it can quickly extract the reference at run time. For instance, as mentioned earlier in Figure 3.8 (a), when an app creates a PendingIntent, it does so by calling the `IActivityManager::getIntentSender` method, which returns a handle specific to that data (the handle is returned in the standard `flat_binder_object` structure for references). In order to extract the references, CopperDroid utilizes two additional fields (i.e., `offsets_size` field and the `offsets` pointer), in the `binder_transaction_data`[3]. The `offsets_size` field is the size (in bytes) of the `offsets` array, comprised of pointer sized values each corresponding to a given reference.

If, when unmarshaling a binder transaction, the parameter is deemed a Binder type by AIDL data, then the `offsets` array is used to locate the offset within the transaction data of the specific `flat_binder_object`. The offsets are in the same order as the types in the parcelable object. Referring to the SMS example as shown in Figure 3.8, the `sentIntent` is the first reference and should thus be the first entry (position 0) in the `offsets` array and, if the `deliveryIntent` were not null, it would be the second entry in the `offsets` array (position 1) and the `offsets_size` field would be 8 (bytes) on a 32-bit ARM system. With this information, CopperDroid identifies and captures the corresponding caller allocated memory region which contains the actual marshalled object. The marshalled data is then sent back to Oracle for the complete unmarshalling procedure. This complex procedure is unique to CopperDroid as of 2015.

3.4.4 Recursive Object Exploration

The purpose of recursively exploring is to thoroughly inspect each field in every parameter to gain as much information as possible. To do this, every sub class (or bundle) is explored recursively until a primitive can be found, printed, and stored for further CopperDroid analysis. To do this, the string representation of each primitive is appended to the output string list, and the marshalled data offset is updated to point of the next unmarshalled item. Only then does the Oracle iterate to the next parameter type on the given list. Once the Oracle has completed unmarshalling all method parameters, the final output is returned to CopperDroid. Figure 3.8(c) presents an excerpt of a simplified Oracle output corresponding to a reconstructed SMS `sendText` behaviour. For simplicity, we only include essential details, filtering out irrelevant flags and empty fields.

[3]These are not shown in the simplified payload Figure 3.5.

3.4.5 An Example of Reconstructing IPC SMS

As systematically evaluating the Oracle's reconstruction capabilities on every possible object (over 300+ AIDL objects alone) is superfluous, we introduce a representative example which exemplifies all of the Oracle's unmarshalling capabilities. Let us consider an app that sends an SMS as our running example. The Java code that corresponds to the SMS behaviour (i.e., creating an `Intent` and invoking a remote method) can be seen in Figure 3.8 (a). Typically, application code for sending an SMS includes invoking the `sendTextMessage` method of `SmsManager`, with the destination number (e.g., "123456789") and the SMS text (e.g., "Hello") as parameters. It is also optional to include one or two customized `PendingIntent`s, which may be broadcast back to the client app depending on the outcome of the service (e.g., successful send).

As explained previously, `PendingIntent` objects are passed by reference, rather than being directly marshalled. To keep track of such data, CopperDroid is also aware of any IBinder handles that reference the `PendingIntent`, by analysing previous `ioctl` calls sent from the client app to the IActivityManager. Indeed, all Binder transactions result in at least two `ioctl` system calls, as shown in Figure 3.8 (b).

When our `sendText` is invoked, we see one `ioctl` used to locate the SMS service and another used to invoke the `sendTextMessage` method. See Figure 3.7 message 4 for the latter, which is also the main message sent to the Oracle for unmarshalling. Furthermore, if the second `ioctl` includes a pass-by-reference parameter (e.g., a handle to a `PendingIntent`) CopperDroid locates the third `ioctl` with the actual referenced object (e.g., `PendingIntent` saying "SENT") and sends it to the Oracle as well. This handle/object pair generation can also be seen in Figure 3.7 (page 72) in transactions 2 and 3, and the handle can be seen used in the `sendText` invocation in message 4.

From a high-level perspective (e.g., Java methods), sending an SMS by roughly corresponds to obtaining a reference to an instance of the class `SmsManager`, the phone SMS manager, and invoking its `sendTextMessage` method (last line of Figure 3.8 (a)). This invocation includes the necessary method parameters including the destination phone number and the text message as the method arguments. Alternatively, on a lower level, the same action corresponds to locating the Binder service `isms` and remotely invoking its `sendText` method with same arguments. From this low-level perspective, the same actions correspond to the sender application invoking two `ioctl` system calls on `/dev/binder`: one to locate the service and the other to invoke its method.

CopperDroid thoroughly introspects the arguments of each binder-related `ioctl` system call to completely reconstruct each remote, IPC, invocation. This allows CopperDroid to infer the high-level semantic of the operation. In particular, we focus our

```
PendingIntent sentIntent =
  PendingIntent.getBroadcast(
    SMS.this, 0,
    new Intent("SENT"), 0);
  SmsManager sms =
    SmsManager.getDefault();
  sms.sendTextMessage(
    "123456789", null,
    "Hello", sentIntent, null);
```

(a) SMS send behaviour at app Java level including creating a `PendingIntent` `sentIntet` for method parameter four.

```
ioctl(0x14,
      BINDER_WRITE_READ,
      0xbedc93e8) = 0
ioctl(0x14,
      BINDER_WRITE_READ,
      0xbeb69508) = 0
ioctl(0x14,
      BINDER_WRITE_READ,
      0xbeb693e8) = 0
```

(b) SMS send `PendingIntent` `ioctl`s. Third parameter points to serialized data, see Figure 3.5.

```
BINDER_TRAN (from binder_transaction_data):sentIntent =
  android.app.PendingIntent = Intent("SENT"))

BINDER_REPLY (from binder_transaction_data):sentIntent =
 android.app.PendingIntent{  Binder:
  type = BINDER_TYPE_HANDLE flags = 0x7F|FLAT_BINDER_FLAG_ACCEPTS_FDS
  handle = 0xa              cookie = 0xb8a58ae0  }

BINDER_TRAN (from binder_transaction_data):
 com.android.internal.telephony.ISms.sendText(
  destAddr = "123456789"     srcAddr = None      text = "Hello"
   sentIntent = android.app.PendingIntent{  Binder:
     type = BINDER_TYPE_HANDLE,
     flags = 0x7F|FLAT_BINDER_FLAG_ACCEPTS_FDS,
     handle = 0xa, cookie = 0x0  }
   deliveryIntent = null)

Oracle:com.android.internal.telephony.ISms.sendText(
 destAddr = "123456789"      srcAddr = None      text = "Hello"
 sentIntent = android.app.PendingIntent("SENT")
 deliveryIntent = null
```

(c) Simplified `sendText` method (including `PendingIntent`) reconstruction produced by CopperDroid and the Oracle, using the `binder_transaction_data` retrieved from the `ioctl`.

Figure 3.8: CopperDroid reconstructed sendText example.

analysis on Binder *transactions*, i.e. IPC operations that actually transfer data. This is also responsible for RPC. To identify these communications, CopperDroid parses the memory structures passed as parameters to the `ioctl` system call and identifies Binder transactions (`BC_TRANSACTION`) and replies (`BC_REPLY`). Refer back to Figure 3.5 (page 67) for `ioctl` payload layout, where `BC_TR` is shorthand for binder transaction.

In the next step, CopperDroid identities the *InterfaceToken*. In our example this is `ISms`, albeit simplified. The entire token `com.android.internal.telephony.ISms`, as seen by CopperDroid, can be found in Figure 3.9. This is then used to find

```
elif (InterfaceToken, "com.android.internal.telephony.ISms"):
  db_parcel_ISms(parcel, code)

if (code == TRANSACTIONS["TRANSACTION_sendText"]):
  text.append("method: sendText")
  text.append("string")
  text.append("destAddr={0}".format(parcel.readString16()))
  text.append("string")
  text.append("scAddr={0}".format(parcel.readString16()))
  text.append("string")
  text.append("text={0}".format(parcel.readString16()))
  text.append("android.app.PendingIntent")
  text.append("sentIntent = [PendingIntent N/A]")
  text.append("android.app.PendingIntent")
  text.append("deliveryIntent=[PendingIntent N/A]")
```

Figure 3.9: AIDL example for marshalling `ISms.sendText` method.

the AIDL description, as also seen in Figure 3.9. Normally the marshalling description would contain more `if` cases for varying codes, but in our running example the code is `sendText`, so that is the only one we are showing.

From this figure we see the first three parameters of the invoked method are String types, and can be read as primitives. The fourth parameter to the `sendText` method, however, is a `PendingIntent`. While unmarshalling this non-primitive type, a handle is found instead of a CREATOR field. The Oracle handles this IBinder type by identifying and finding the referenced marshalled data and unmarshals the actual `Intent` as a complex class object. Normally this recreates all object fields, including empty ones, so a simplified version containing only essential data can be found in Figure 3.8 (c).

3.5 Observed Behaviours

We manually examined the results of CopperDroid's analyses (i.e., system call invocation tracking, IPC Binder analysis, and complex Android object reconstruction) on a number of randomly selected Android malware from our samples sets [52, 141, 249]. Our examinations identified five macro classes of behaviours, which we illustrate Figure 3.10. Depending on parameter values, each class could be further divided by variations in their actions. In this chapter, no precautions were taken to prevent harmful network requests to other systems or people. While this allowed interesting behaviours to occur for analysis, future work should provide some form of protection.

Actions are extracted via CopperDroid and can belong to any level of behaviour abstraction (e.g., OS- and Android-specific behaviours). Interestingly, some behaviours are well-known and shared with the world of non-mobile malware. Other behaviours,

such as those under the "Access Personal Data" class, are instead inherently specific to the mobile ecosystem. While the base CopperDroid framework extrapolated some behaviours by parsing single system calls, the author recreated the more complex behaviours. The author will elaborate more on the resulting behaviour classes in the following chapter, when fully utilizing them for classification. This required more complex behaviours (i.e., finding dependencies between multiple system calls, modelling parameters, using the unmarshalling Oracle) developed by the author.

Every behaviour class in the map corresponds to a behavioural model that can be expressed by an arbitrary number of actions, depending on its specific complexity. Some are defined by a single system call, such as `execve`. Others, such as "SMS Send" or those under "Access Personal Data", are defined as a set of transactions of the Binder protocol. Yet others are defined as multiple consecutive system calls. For instance, outgoing HTTP traffic is modelled as a graph with a `connect` system call, followed by an arbitrary number of send-like system calls, whose payload is parsed to detect HTTP messages, possibly interleaved by unrelated non-socket system calls.

Figure 3.10: Hierarchical map of reconstructed CopperDroid behaviours.

```
execve('pm',
    ['pm',
     'install',
     '-r',
     'New.apk'],
     ... );
```

(a) App installation via direct system call.

```
Intent intent =
   new Intent(Intent.ACTION_VIEW);
intent.setDataAndType( Uri.fromFile(
      new File("/mnt/sdcard/New.apk") ),
   "application/vnd.android.pack...");
startActivity(intent);
```

(b) App install via Binder transaction (e.g., Intent specific).

Figure 3.11: App installation via system call or Binder (Android-level) transaction.

Behaviour classes do not forcibly correspond to just one of the aforementioned models, but may also contain a set of them. To clarify, consider the examples shown in Figure 3.11 which illustrate how CopperDroid recognizes actions triggered by both code snippets as belonging to the class "Install APK", despite differences in the manner these actions are achieved (an `execve` system call or a Binder transaction).

3.5.1 Value-Based Data Flow Analysis

The author extended CopperDroid to abstract a stream of related low-level events into meaningful, high-level, behaviours. As a side effect, this recreates the actual resources (e.g., files) associated with an action. Such details enrich CopperDroid's behavioural reconstruction analyses and are essential for future tools differentiating between malicious actions and benign ones (e.g., not all filesystem accesses are suspicious by default). This resource reconstructor is embedded into the system call analysis (see Figure 3.6 page 68). This enables further static and dynamic analyses on a file's content, intent, and purpose. Furthermore, it has been particularly useful when analysing root exploits, shell scripts, and malware writing, downloading, and/or installing additional APKs, with system privileges, which is very dangerous. As CopperDroid ensures that no app action can escape our trace collection, we can faithfully reconstruct these behaviours.

To this end, CopperDroid performs a value-based data flow analysis by building a system call-related data dependency graph and def-use chains. In particular, each observed system call is initially considered as an unconnected node. A forward slicing algorithm then inserts edges for every inferred dependence between two calls. As the slicing proceeds, both nodes and edges are annotated with the system call argument constraints; these annotations are essential in the creation of our def-use chains. Def-use chains, where each call is linked by def-use dependencies, are formed when the output value by one system call (the *definition*, i.e. `open`, `dup`, `dup2`) is the input value to a following (not necessarily adjacent) system call (the *use*, i.e. `write`, `writev`). In the

case of file access behaviours, terminating calls would be `close` and `unlinked`. This complex process takes into account many factors, including the flags of each system call and anything affecting the system including `fork`'s and any inherited `fd`'s.

Therefore, by building a data dependency graph over the set of observed system calls, and performing forward slicing, we can recreate filesystem related events and the actual resources involved. A simplified example of file access behaviour from Java level, system call level, and after CopperDroid analysis can be found in Figure 3.12. Our method also retains deleted files (`unlink`) and multiple versions of any resources with identical file names so the malware cannot hide any versions of a file. Although we focus this discussion on filesystem related system calls, a similar process holds and has been implemented for network-related calls. For example, if a web browsing Intent was executed, the resulting system call def-use chain should begin with a `socket` and a `connect` call, and potentially be followed with several `sendto`'s transmitting data.

3.5.2 App Stimulation

In contrast to traditional executables with a single entry point, Android applications may have multiple. Most apps have a main activity, a screen to interact with via the touchscreen, but ancillary activities and background services may be triggered by the system or by other apps. Furthermore, the app execution may reach these *without* flowing through the main activity. For instance, let us consider an application that operates as a broadcast receiver for SMS_RECEIVED events. After installation, the application would only react to the reception of SMS, showing no additional interesting behaviours otherwise. In such a scenario, a simple install-then-execute dynamic analysis may miss a number of interesting behaviours. This problem has long been affecting traditional dynamic analysis approaches as non-exercised paths are simply unanalysed. If unexplored paths host additional behaviours, then any dynamic analysis would fail to reach

```
OutputStreamWriter out =
  new OutputStreamWriter(
  openFileOutput("sample.txt",
  MODE_WORLD_READABLE));
out.write("ELF", 0, 10);
```

(a) File access behaviour at Java level.

```
open("files/sample.txt",
  0x20241, 0x180) = 0x1c
    ...
write(0x1c, "ELF", 0xa) = 0xa
    ...
```

(b) File access behaviour at system call level.

```
FS_ACCESS::Creation of "sample.txt"
   (link to actual file, ancillary info: 1024 (bytes))
```

(c) Reconstructed file access behaviour.

Figure 3.12: CopperDroid behaviour reconstruction of a file access.

them unless proper, but generally expensive and complex, exploration techniques are adopted [37, 149]. An in depth discussion of dynamic limitations and cutting-edge solutions was given in Chapter 2. The code coverage issue is exacerbated by the fact that mobile apps are inherently user driven and many successive interaction are necessary.

To *qualitatively* improve our code coverage, CopperDroid artificially sends a number of plausible events, based on the malware's Android Manifest, to the emulator. For example, injecting events such as phone calls could trigger an app's broadcast receiver if it had been registered to receive such intents. Another example that arises from our experience with Android malware is the `BOOT_RECEIVED` intent. Many samples use this to start execution as soon as the victim system is booted (similar to `\CurrentVersion\Run` registry keys on Windows systems). However, sending this stimuli only makes sense if the app requests the `BOOT_COMPLETED` permission.

The Android emulator enables the injection of a considerable number of artificial events to stimulate a running application. These range from very low-level hardware-related events (e.g., loss of the 3G signal) to high-level ones (e.g., incoming calls, SMS). CopperDroid could have adopt a fuzzing-like stimulation strategy and trigger *all* the events that could be of interest for the analyses. That would unfortunately be of limited effect because of the underlying Android security model and permission system. Instead, CopperDroid utilizes static information extracted from the app to carry out a fine-grained targeted stimulation strategy. To this end, CopperDroid examines each APK manifest to extract events and permission-related information to drive the malware stimulation. Furthermore, an application has the ability to dynamically register a broadcast receiver for custom events at run-time. CopperDroid is able to intercept such operations and add a proper stimulation for the newly registered receiver.

To perform its custom stimulation, CopperDroid utilizes the Android emulator's capabilities to inject a number of artificial events into the emulated system. In particular, it implements MonkeyRunner, a tool that provides an out-of-the-box API to control an Android device or emulator, through the Python programming language [13]. A summary of the main events CopperDroid handles is reported in Table 3.1, which also shows the parameters that can be customized for each event.

Table 3.1: CopperDroid supported stimulations and parameters.

#	Stimulation	Parameters	#	Stimulation	Parameters
1	Received SMS	*SMS Text from phone number*	4	Battery status	*Amount of battery*
2	Incoming call	*Phone number and duration*	5	Phone Reboot	-
3	Location update	*Geospatial coordinates*	6	Keyboard input	*Typed text*

3.6 Evaluation

Our experimental setup is as follows. We ran unmodified Android images on top of the CopperDroid-enhanced emulator. Occasionally a clean image is customized to include personal information, such as contacts, SMS texts, call logs, and pictures to mimic, as closely as possible, a real device. Each analysed malware sample is then installed in the image and traced via CopperDroid until a timeout was reached (10 minutes by default). At the end of the analysis, a clean execution environment is restored to prevent corruptions and side-effects caused by installing multiple malware samples in the same system. To limit noisy results, each sample was executed and analysed six times: three times *without* stimulation and three times *with* stimulation. Afterwards, single execution results were merged. Future work and improvements are discussed later in Chapter 6.

We evaluated CopperDroid on three well-known and diverse datasets. These included the public Contagio dump and Android Malware Genome datasets [52, 249] and one provided by McAfee [141]. These datasets are composed of 1,226, 395 and 1,365 samples, respectively, equating to more than 2,900 samples overall.

3.6.1 Effectiveness

To evaluate the effectiveness of CopperDroid's stimulation, we first analysed all samples without external stimulation. Then we performed full stimulation-driven analyses on the same malware sets. A summary of the results is presented in Table 3.2, while more detailed results on the McAfee dataset are reported in Table B.1 in Appendix B. These all or no stimuli results were generated by collaborators. For a *fine-grained* analysis of incremental behaviours induced by stimuli, the author presents Table 3.4.

As Table 3.2 shows, stimulation results for the newer McAfee dataset is consistent with the older datasets: 836 of 1365 McAfee samples exhibited additional high-level behaviours (defined in Section 3.5) and, on average, the number of additional behaviours was 6.5 *more* than the 22.8 behaviours observed without CopperDroid's stimulation. While not the most effective solution, this stimulus technique allowed CopperDroid to analyse a significant number of additional behaviours for very little performance cost.

Table 3.2: Summary of stimulation results, per dataset.

Malware Dataset	Incremental Behaviours (Samples)	Average Increment	Standard Deviations
Genome	752/1226 (60%)	2.9/10.3 (28.1%)	2.4/11.8
Contagio	289/395 (73%)	5.2/23.6 (22.0%)	3.3/19.8
McAfee	836/1365 (61%)	6.5/22.8 (28.5%)	9.5/30.1

Table 3.3: Overall behaviour breakdown of McAfee dataset.

Behaviour Class	**No Stimulation**		**Stimulation**	
FS Access	889/1365	(65.13%)	912/1365	(66.81%)
Access Personal Information	558/1365	(40.88%)	903/1365	(66.15%)
Network Access	457/1365	(33.48%)	461/1365	(33.77%)
Execute External App	171/1365	(12.52%)	171/1365	(12.52%)
Send SMS	38/1365	(2.78%)	42/1365	(3.08%)
Make/Alter Call	1/1365	(0.07%)	55/1365	(4.03%)

Of course, it is important to understand whether an observed behaviour is new or if it refers to a similar, previously-observed action (e.g., same network transmission but different timestamp). To achieve this, we currently disregard pseudorandom or ephemeral values observed in specific behaviours, like a timestamp or an ID, found in otherwise identical behaviours. Hence, a repeated behaviour will not contribute to the percentage of additional behaviours observed with stimulation. All the other behaviours are considered to be new and therefore contribute to the aforementioned percentage.

During the analysis of the McAfee dataset, roughly 10% of the samples did not exhibit any behaviour, regardless of the stimulation technique adopted. Nearly half of these samples did so because CopperDroid could not successfully install them in the emulator. The other half were installed but stayed dormant or did not exhibit any interesting behaviour before CopperDroid's analysis timeout. There are a variety of reasons, including "incorrect" stimulation/environment elements or VM evasions (see Discussions in Chapter2). While more sophisticated code coverage solutions may be deployed, many deter fast, lightweight, performance. While we may adopt better stimulation techniques in the future, it is not the current focus of the CopperDroid analysis framework.

Table 3.3 reports the overall breakdown of the observed behaviours (i.e., application actions defined in Figure 3.10) on the McAfee dataset. Each row identifies the class of behaviour and how many samples, over the total dataset, exhibited at least one occurrence of that behaviour *with* and *without* stimulation. Here, we see that the two behaviours most reactive to stimulation are *Access Personal Information* and *Make/Alter Call*. The first is triggered by CopperDroid's stimulation technique, resulting in an access to the user's personal information. The latter is mostly due to a set of malware that, whenever a phone call is received, hide its notification from user. Conversely, the author presents Table 3.4, which provides a more fine-grained overview of the effects of stimulation on all behavioural subclasses defined in Section 3.5.

Lastly, the author ran a number of malware samples with no, selective, and full stimulation with the help of a collaborator. The aim of this experiment was to qualitatively

Table 3.4: Incremental behaviour induced by various stimuli.

Sample Family	Behaviour Class	Behaviour Subclass	Behaviours No Stim.	Incr. Behaviour Type Stim.	Incr. Behaviour SMS Stim.	Incr. Behaviour Loc. Stim.
YZHC	Network Access	HTTP	4	-	-	N/A
		DNS	1	-	-	N/A
	Exec External App	Generic	3	+10 (+433%)	-	N/A
		Shell	1	+3(+400%)	-	N/A
		Priv. Esc.	2	-	+2(+100%)	N/A
		Install APK	4	-	-	N/A
	Access Personal Info	Account	-	-	+1(⊥)	N/A
	FS Access	Write	414	-	-	N/A
zHash	Network Access	HTTP	2	+2 (+100%)	+5 (+350%)	N/A
		DNS	-	-	+1 (⊥)	N/A
	Exec External App	Generic	1	+12 (+1300%)	+3 (+400%)	N/A
		Shell	1	+3 (+400%)	-	N/A
		Priv. Esc.	4	-	-	N/A
		Install APK	4	-	-	N/A
	Access Personal Info	Account	2	-	-	N/A
	FS Access	Write	163	-	+255 (+257%)	N/A
SHBreak	Network Access	HTTP	3	-	N/A	N/A
	Exec External App	Generic	2	+113 (+5750%)	N/A	N/A
		Shell	1	+22 (+2500%)	N/A	N/A
		Install APK	4	+4 (+100%)	N/A	N/A
	FS Access	Write	195	+353 (+281%)	N/A	N/A
DKF	Network Access	HTTP	13	-	N/A	-
	Exec External App	Generic	1	+2 (+300%)	N/A	+1 (+200%)
		Shell	1	-	N/A	-
		Install APK	4	-	N/A	-
	FS Access	Write	3	+197 (+6667%)	N/A	+144 (+4800%)
Fladstep	Network Access	HTTP	15	-	N/A	N/A
	Exec External App	Generic	3	+17 (+633%)	N/A	N/A
		Shell	1	+5 (+500%)	N/A	N/A
		Install APK	4	-	N/A	N/A
	FS Access	Write	171	+80 (+47%)	N/A	N/A

(Priv. Esc. = Privilege Escalation, DFK = DroidKungFu, N/A = stimuli not possible based on Manifest)

identify which individual stimulus induced what amounts of incremental behaviour, and whether combinations of stimulation are more effective than individual triggers. For illustration, we deliberately show the Android malware samples that had the highest, average, and lowest incremental behaviours both percentage wise and amount wise. If several families had the same maximum amount of incremental behaviour, we chose the one with the highest percentage in incremental behaviour and vice versa.

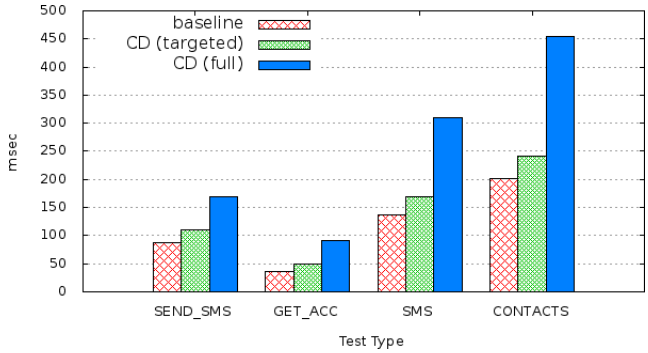
The author then determined the best representative sample from each family based on the amount and diversity of behaviours. The results of various stimulations on these malware samples can be seen in Table 3.4. Here, we can begin to see correlations between different stimuli and behaviours. As the table shows, our selective stimulations was able to disclose a number of *additional* previously-unseen behaviours (e.g., YZHC SMS stimulation showed more access to personal account information) or already-observed behaviours (e.g., SHBreak showed 113 additional generic executions).

3.6.2 Performance

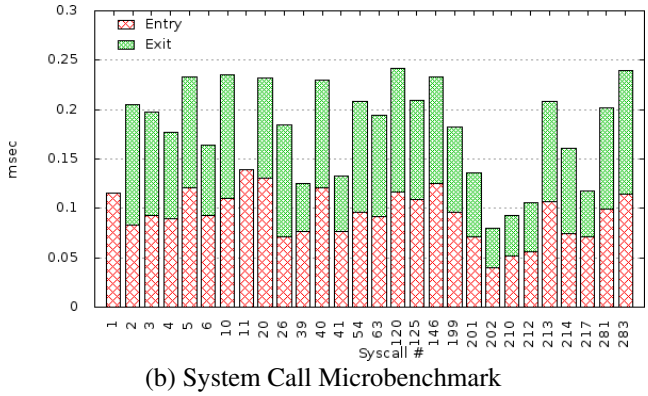
In this section we evaluate CopperDroid's overhead through a number of experiments conducted on a GNU/Linux Debian 64-bit system equipped with an Intel 3.30GHz core (i5) and 3GB of RAM. Performance evaluations of CopperDroid's system call collection were performed by collaborators, while the Oracle was evaluated by the author. As the CopperDroid framework, specifically the system call collection part, is still undergoing moderate changes, newer evaluations have not yet been conducted.

Benchmarking a multi-layered system such as Android, in conjunction with an emulated environment, can be rather complicated. Traditional benchmarking suites based on measuring I/O operations are similarly affected by the caching mechanisms of emulated environments. On the other hand, CPU-intensive benchmarks are meaningless against the overhead of CopperDroid, as it operates purely on system calls.

To address such issues, we performed two different benchmarking experiments. The first is a *macrobenchmark* that tests the overhead introduced by CopperDroid on common Android-specific actions, such as accessing contacts and sending SMS texts. Because such actions are performed via the Binder protocol, these tests give a good eval-



(a) Binder Macrobenchmark



(b) System Call Microbenchmark

Figure 3.13: Macro and Micro benchmarking results for system call tracing.

uation of the overhead caused by CopperDroid's Binder analysis infrastructure. The second set of experiments is a *microbenchmark* that measures the computational time CopperDroid needs to analyse a subset of interesting system calls.

To execute the first set of benchmarks, we created a fictional Android app to performs generic tasks, such as sending and reading (`SMS`) texts, accessing local account information (`GET_ACC`), and reading all contacts (`CONTACTS`). We then ran the test app for 100 iterations and collected the average time required to perform these operations under three settings: on a vanilla Android emulator, on a CopperDroid emulator with CopperDroid configured to monitor the targeted test app, and on a CopperDroid emulator with CopperDroid configured to track all system-wide events. Results are reported in Figure 3.13 (a). As can be observed, the overhead introduced by the targeted analysis is relatively low, respectively $\approx 26\%$, $\approx 32\%$, $\approx 24\%$ and $\approx 20\%$. On the other hand, system-wide analyses increase the overhead considerably (>2x). This is due to the of the number of Android components that are concurrently analysed.

The second set of experiments measure the average time CopperDroid requires to inspect a subset of interesting system calls. This experiment collected more than 150,000 system calls obtained by executing apps with arbitrary workloads. As tracking a system call requires intercepting entry and exit points, we report each measures separately in Figure 3.13 (b) (the average times are $0.092ms$ for entry and $0.091ms$ for exit).

The author evaluated the Oracle's performance by sending various object types to be unmarshalled. A hundred requests for one object were sent to the multi-threaded Oracle for ten tests. Performance scores were then averaged. This test was run on simple (`Integer`) and complex primitives (`String Array`), simple (`Account`) and very complex objects (`Intent`), and an `IBinder` object (i.e., only the handle). When unmarshalling `IBinder` completely, the results would be a combination of the `IBinder` performance and an object performance. As seen with our Android object examples, this can be a wide range of values (see Figure 3.14). While unmarshalling real method parameters (e.g., `sendtText`) would require a mix of types and vastly less than 100 parameter, however the performance can be estimated with these results.

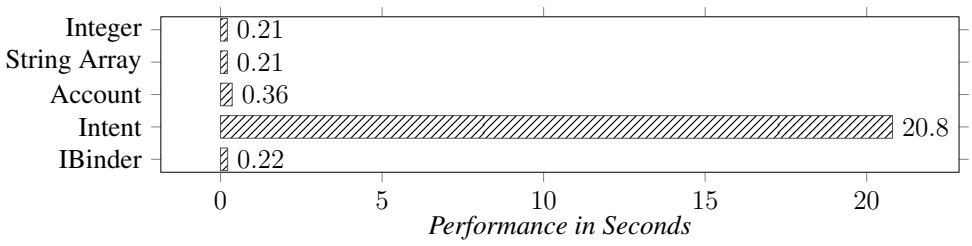


Figure 3.14: Average time to unmarhsal 100 requests for an object over 10 tests.

3.7 Limitations and Threat to Validity

This section examines limitations within this work and assess possible threatens to its validity. There are external factors which affect possibly validity. As demonstrated by Chapter 1, the used dataset does not represent the entire Android app market. CopperDroid has only been extensively tested on malicious apps collected by other researchers, so it does not represent the entire app market. Secondly, we did not study the malware datasets we did posses to determine their completeness, e.g., whether or not essential malware families were missing. These limitation may be addressed in the future by merging multiple malware datasets as well benign datasets with the goal of maximum diversity and/or a current, accurate representation of mobile markets. This may be of use especially when considering the bulk of malware are re-packaged legitimate apps.

There are also threats to the internal validity of this analysis tool to be considered. The validity of CopperDroid's system call logs is based on the plugin's trustworthiness. While we rely on QEMU emulation to protect the plugins and the Oracle, CopperDroid may be detected [138], and malicious actions withheld. If ever implemented on-device or in-the-cloud, protecting CopperDroid may require more sophistication. Classic dynamic analysis code coverage limitations apply to CopperDroid, which can be improved with the ongoing traditional and mobile research in this area.

As discussed in Section 3.4.1, the Oracle also faces limitations. This is mostly due to our dataset limitation, as we never encountered any IPC communication that could not be unmarshalled using our AIDL generated data. Future work is needed to address this internal limitation by exploring all cases where IPC transactions can occur with custom, normally hardcoded, `code` switch statements. This will ideally be satisfied by acquiring a larger, more diverse, dataset of both malware and legitimate applications with a span of many years, categories, and possibly both paid and free. More in-depth discussion on some of the research paths that address these limitations and threats to validity can be found in the thesis conclusions (see Chapter 6).

As a final note, a threat that CopperDroid may pose to external entities, is the lack of protocols regarding attacks *by* the malware running during analysis. For instance, the malware running in CopperDroid may attempt to infect other devices or contribute to a massive denial of service (DoS) attack. While it is interesting to analyse these behaviours, and while most Android malware do not yet exhibit these kinds of behaviours, this limitation should be addressed in the future. This could involve simplistic blocking mechanisms, or more sophisticated filtering, modification, or emulated solutions to trick malware into exhibiting its behaviours with no external damage.

3.8 Related Work

While Chapter 2 served as a complete, comprehensive, survey, in this section we will examine a few selected works. In particular, we examine the elements of these works that most closely relate to aspects of our CopperDroid framework. In-depth comparisons of related works are shown in the tables in Appendix A.

DroidScope [233] is a general-purpose VM-based, out-of-the-box framework, to build dynamic analysis for Android. As an out-of-the-box approach, it instruments the Android emulator, but it may incur high overhead when enabling features such as taint-tracking. DroidScope also leverages a 2-level VMI [233] to gather system information, exposed hooks, and a set of APIs. This enables the development of plugins to perform both fine and coarse-grained analyses such as system call, single instruction tracing, and taint tracking. In contrast with CopperDroid, DroidScope's instrumentation points do not perform any behavioural analysis *per-se*. For example, a tool implementing DroidScope can intercept every system call executed on an Android system, but would still need to do its own VMI to inspect the parameters of each call. Although CopperDroid could have been built on DroidScope, its source code was not available when we began development. Furthermore, DroidScope offers basic hooking mechanisms and needs to keep a synchronized 2-level VMI for OS and Dalvik VM semantics. This makes it considerably more complex and difficult to port onto different versions of Android OSes. In contrast to these related works, CopperDroid does not have this limitation.

Enck et al. presents TaintDroid [72], a framework to enable dynamic taint analysis for Android apps. TaintDroid's primary goal is to track how sensitive information flow between the system and apps, or between apps, to automatically identify leaks. To do so, TaintDroid relies on different levels of instrumentation. For example, to propagate taint information within the Dalvik VM, TaintDroid manages method registers, which roughly correspond to local variables in Java. Unfortunately, the multi-level approach introduces low resiliency and transparency: modifying internal components of Android inevitably exposes TaintDroid to tampering, and a series of detection and evasions techniques (see Section 2.3). For instance, apps with standard privileges can detect TaintDroid's presence by calculating checksums over readable components. Moreover, TaintDroid cannot track taints across native code. While CopperDroid does not currently taint data, it can analyse the execution of native code etc., and we can choose to introduce taint tracking in the future.

The framework VetDroid [241] constructs permission-use behaviour graphs to highlight how apps use permissions to access and use system resources. Although an inter-

esting approach, VetDroid requires a quite intrusive modification of the Android system (both Dalvik VM, Binder, and Linux kernel), which hampers the ability to easily port the system to different Android versions. In addition, VetDroid builds on top of TaintDroid and, therefore, inherits its drawbacks [45, 180]. DroidScope, VetDroid and other similar approaches are tailored to Dalvik VM internals and, therefore, may have trouble adapting changes to the runtime system (e.g., ART). This study also claimed that traditional system call analysis was not appropriate for characterizing the behaviours of Android apps as it misses high-level Android-specific semantics and fails at reconstructing IPC and RPC interactions. Contrary to this, we have shown that CopperDroid's system call analysis can obtain run-time Android behaviours such as IPC. This is unlike the hybrid static and dynamic tool Harvester [168], which can obtain most run-time data, but not ICC/IPC. One study recently used the author's insights in CopperDroid to manually discover vulnerabilities in Android IPC [17].

AppsPlayground [169] performs a much granular stimulation than CopperDroid during malware analysis, but its full capabilities require non-negligible modifications to the Android framework (e.g., to capture image identifiers in GUI elements). This framework also does not analyse native code (with the exception of specific, well-known low-level signatures), and integrates a number of well-known techniques (e.g., TaintDroid), inheriting their limitations. Another interesting stimulation approach can be found in PuppetDroid [87], which gathers stimulation traces via crowdsourcing. It is more effective than CopperDroid with respect to stimulation, but is limited to the subset of apps for which there exists a similar recorded stimulation trace. Furthermore, the overhead of PuppetDroid is significantly higher in comparison to CopperDroid.

Unlike related analyses outside the VM, DroidBox is a dynamic, in-the-box, Android malware analyser [205]. DroidBox uses a custom instrumentation of the Android system and kernel to track a sample's behaviour by implementing TaintDroid's taint-tracking of sensitive information [72]. However, by instrumenting Android's internal components, DroidBox is prone to drawbacks associated to in-the-box analyses: malware can detect, evade, or even disable critical analysis components.

Andrubis [225] is an extension to the Anubis dynamic malware analysis system to analyse Android malware [27, 107]. According to its web site, it is mainly built on top of both TaintDroid [72] and DroidBox [205] and it thus shares their weaknesses (mainly due to operating "into-the-box"). In addition, Andrubis does not perform any stimulation-based analysis, limiting its effectiveness and behaviour coverage.

In [73], Enck et al. studied Android permissions found in a large dataset of Google Play apps to understand their security characteristics. Such an understanding is an in-

teresting starting point for designing techniques to enforce security policies [231] and avoid the installation of apps requesting a dangerous combination [74] or an overprivileged set of permissions [20, 79, 223]. Although promising, the peculiarity of Android apps (e.g., combination of Java and native code) can easily elude policy enforcement or perform malicious actions while maintaining a seemingly legitimate appearance. In contrast, all behaviours would be captured with CopperDroid. To the best of our knowledge, no other framework can capture the same range of behaviours and as unobtrusively.

Aurasium [231] is a tool that enables fine-grained dynamic policy enforcement of Android apps. To intercept relevant events, Aurasium instruments single apps, rather than adopting system-level hooks. Working at the application level, however, makes Aurasium prone to easy detection and evasion attacks. As mentioned previously, native code is very useful to detect and disable hooks in the global offset table, even without privilege escalation exploits. Aurasium's authors state that their approach can prevent such attacks by intercepting `dlopen` invocations needed to load native libraries. However, it is unclear how benign and malicious code can be distinguished, as this policy cannot be light-heartedly delegated to Aurasium's end-users. Conversely, CopperDroid's VMI-based system call-centric analysis is resilient to such evasions.

SmartDroid [242] implements a hybrid analysis that statically identifies paths that lead to suspicious actions (e.g., access sensitive data) and dynamically determines UI elements to take the execution flow down those paths. To this end, the authors instrument both Android emulator and Android's internal components to infer which UI elements trigger suspicious behaviours. SmartDroid was evaluated on a testbed of seven different malware samples and found vulnerable to obfuscation and reflection. This makes it hard, if not impossible, to statically determine every possible execution path. Conversely, CopperDroid's dynamic analysis is resilient to static obfuscation and reflection.

To overcome the limits of dynamic analysis (e.g., code or path coverage), Anand et al. proposed a concolic-based solution [11] to automatically identify events an application reacts to by generating input events for smartphone applications. While no learning phase is required, such a solution has two main drawbacks: it is based on instrumentation (i.e., easy to detect) and is extremely time-consuming (i.e., up to *hours* to exercise a single application). Although an interesting direction to explore further, that approach is ill-suited to perform large-scale malware analysis. As described in Section 3.5.2, CopperDroid relies on a simple-yet-effective stimulation technique that is able to improve basic dynamic analysis coverage and discover additional behaviours with low overheads.

3.9 Summary

In this chapter, the author discussed improvements to the CopperDroid framework to automatically reconstruct complex Android malware behaviours. In particular, the author shows how a careful dissection of system calls can result in the full reconstruction of both OS and Android specific behaviours from this well-known point of observation.

This would not have been possible without the author's work on the unmarshalling Oracle which automatically tracks and deserializes IPC and RPC interactions, typically contextualized through complex Android objects. This is a signification and novel contribution to CopperDroid, as it enables full behavioural reconstruction without altering the Android system, and is not achievable with similar tools *nor* with so little intrusion to Android. Not only is this simplicity more resilient to changes in the Android runtime system and its inner details, but it also makes the approach agnostic to the underlying action invocation mechanisms (e.g., Java or native code). The culmination of these properties satisfies research Goals 1, 2, and 4.

We then evaluated the effectiveness and performance of CopperDroid on more than 2,900 real world Android malware, showing that a simple, external, stimulation contributes to the discovery of additional behaviours. Furthermore, detailed incremental stimulus by the author brought forth more information on how different samples react to varying stimuli. A more detailed table of author contributions can be found below.

We believe the novelty of CopperDroid's analyses, particularly the author's unmarshalling Oracle and resource recreator, opens the possibility to reconsider rich and unified system call-based approaches as effective techniques to build upon to mitigate Android malware threats. To illustrate this, we have successfully utilized CopperDroid's reconstructed behaviours and resources as features for automatically, and quickly, classifying malware into malware families (see Chapter 4). Other areas for future research and improvements to this framework can be found in Chapter 6, including a complete examination of the Oracle with respect to edge cases and future work on automatically acquiring custom AIDL service information as well.

Table 3.5: Comparison of related works and base CopperDroid to author's work.

	DroidScope	**VetDroid**	**Aurasium**	**CopperDroid***	**CopperDroid**$^{+}$
Collect	API, syscall	API	API, syscall	syscall	syscall
Portable	✗	✗	✗	✓	✓
Behaviour^{s}	✓	✓	✓	✓	✓
Behaviour^{c}	✗	✓	✓	✗	✓
Stimuli	✗	✗	✗	full/none	incremental
Native code etc.	✗	✗	✗	✓	✓

($*$ = Base, $+$ = author enhanced, s = simple, c = complex)

Chapter 4

Classifying Android Malware

Contents

4.1 Introduction

In this chapter we build on the work from Chapter 3 by implementing machine learning algorithms, as well as statistical learning, to categorize malware samples. As already mentioned, with the steady increase in Android malware, developing automatic classification methods is essential. Furthermore, as malware incorporate more sophisticated obfuscation and evasion techniques, resilient analysis methods are key for gathering reliable features. For this reason, the author has chosen to use CopperDroid's behaviours.

Given our advances in Android dynamic analysis, the natural step forward was applying the results to malware classification. For traditional PCs, system calls have been used extensively to detect and classify malware [25, 122, 174]. Despite high accuracies, such methods were tied to their operating system (e.g., Windows or Unix), and were susceptible to mimicry attacks and randomly executed calls. CopperDroid's behaviour extraction addresses this issue, as well as others, by filtering and condensing system calls into a few detailed behaviours. In particular, the author's contributions to recreating all Android behaviours, both low-level and high-level, allowed us to apply traditional classification techniques to previously untested feature sets. We further demonstrate that extracted behaviours condense interesting calls into small, potent, feature sets. Moreover, many challenges associated with dynamic analysis (e.g., code coverage) were solved, or at least addressed, by building our classification tool on CopperDroid.

While most current Android classification methods have utilized statically extracted APIs [6, 9, 16, 57, 234], the framework developed in this chapter uses high-level, reconstructed, behaviours. For this, we use the author's work with CopperDroid, which extracts a wide range of behaviours from bare-bones system call traces. This encompass low-level actions all the way up to IPC method invocations, filesystem accesses, network accesses, and the execution of privileged commands. This process has been fully discussed and evaluated in Chapter 3, but relevant details and the resulting behaviours are revisited below in Section 4.4. We then propose a novel, hybrid approach that automatically classifies Android malware samples using highly detailed behaviours. Based on these high-level behaviours, such as a network connection, the author's unique feature set is used to separate samples into known malware families.

By continuing previous work on behaviour extraction, the author was able to build a detailed feature set for high performance scores (i.e., accuracy, precision, recall). Furthermore, as these features are extracted from series of, often data dependent, system calls, our feature set is relatively small, improving scalability and performance. For example, while [9] had a feature set of eight hundred, our feature set has not yet exceeded

a hundred. Further comparisons with other recent work can be found in Section 4.8.

We then feed CopperDroid's reconstructed behaviours into a support vector machine (SVM) based classifier and performed an evaluation of it across a set of 1,137 malware samples from the Malware Genome Project. Our experiments have shown that using behaviours recreated from system calls as features yields higher performance and accuracy when compared to pure system calls. Specifically, our SVM based-classifier has achieved accuracy scores of 94.5%, with a precision at 99.2% and recall at 97.8%. Furthermore, at the cost of a little more runtime, scores like accuracy can be further improved with our hybrid, conformal supported vector machine, by at least 0.5% for a total of 95%. The main contributions of this chapter are summarized below:

1. **Multi-class classification**: We present a multi-class classification method for Android malware (see Section 4.5). To the best of our knowledge this is the first piece of work that performs multi-class classifications of Android malware using behaviours derived from dynamic system calls (see Section 4.4).

2. **Evaluation of SVM classification**: We go beyond traditional SVM and evaluate the quality of classification when considering multiple alternative choices instead of singular class decisions, which is the traditional method. Single choices can be unreliable especially when classifying samples with few behaviours (Section 4.5). Classification accuracy can therefore be improved by our novel hybrid method (see below), or by removing sparse samples (undesirable as reduces analysed set).

3. **Enhancement via prediction sets**: We demonstrate how introducing the statistical approach known as conformal prediction (CP), and its sets of likely predictions, into our classifier noticeably improves classification accuracy. This is the case even in the presence of sparse behavioural profiles (see Section 4.5.3). Thus, the author developed an operational framework that detects poor confidence in SVM decisions and then selectively invokes CP to enhance the classification. We show that this framework is highly adaptive and can achieve near-perfect accuracy when working with large prediction sets (see Sections 4.3 and 4.5.3).

The rest of the chapter is organized as follows. In Section 4.2, we discuss the classification tools we use, as well as our overall system architecture. This leads to the theory and architecture of our hybrid predictor, as explained in Section 4.3. Then we discuss the behaviour sets we use as classification features, as well as how the author obtained them, in Section 4.4. These features are then used for classification, see Section 4.5, with results found in Section 4.6. Related works and conclusions then complete this chapter, in Sections 4.8 and 4.9 respectively.

4.2 Relevant Machine Learning Background

In this section we describe an approach to classify Android malware using support vector machines (SVM). We then continue to elaborate on conformal prediction (CP), which we use on poor SVM results to determine precise confidence levels in new predictions. However, we attempt to only use this in the cases where it is most useful, as conformal prediction is computationally expensive. The structure and advantages of this hybrid method are further explained in Section 4.3, along with its results in Section 4.6.

4.2.1 Support Vector Machines (SVM)

In Section 2.2.6 of the survey, we briefly differentiated between binary classification (i.e., a sample is either malicious or benign), and multi-class classification (i.e., a sample can belong to one of any number of classes). When given a dataset of samples belonging to different classes, support vector machines (SVM) can be used to segregate the samples using hyperplanes. A single hyperplane can be defined by the set of points $\mathbf{x}$ that satisfies the following relation:

$$\mathbf{x} \cdot \mathbf{w} - b = 0$$

Where $\mathbf{w}$ is the normal to the hyperplane, $\mathbf{w}$ and $\mathbf{x}$ are used to compute the dot product, and $\frac{b}{\|\mathbf{w}\|}$ is the offset of the hyperplane from the origin along the normal.

Two-Class Support Vector Machines, i.e. binary classification, equates to a training dataset $\mathcal{D}$ consisting of a set of tuples $(\mathbf{x}_i, y_i)$. Here $\mathbf{x}_i$ is a p-dimensional vector of features, normally represented by real numbers, and $y_i \in \{-1, +1\}$ denotes the class result of $sample_i$. SVM separates the two classes $\{-1, +1\}$ by constructing the optimal hyperplane, subjecting $\mathbf{w}$ and b to the following class constraints:

$$\forall y_i = +1 : \mathbf{x}_i \cdot \mathbf{w} - b \geq +1 \tag{4.1}$$

$$\forall y_i = -1 : \mathbf{x}_i \cdot \mathbf{w} - b \leq -1 \tag{4.2}$$

Complete class segregation using a hyperplane is only possible when the samples are linearly separable. Normally this is not the case for multi-class methods, as the number of classes leads to a high-dimensional space. In these cases, it is possible to use other separation kernels such as polynomial or radial basis function [160]. For the purpose of our experiments, we use the standard radial basis function (RBF), whose value solely depends on a sample's distance to a "centre" point. Once the hyperplane is established, a classification decision $\mathbf{y}_i$ for each testing dataset sample *i* can be obtained.

Multi–Class Support Vector Machines extends the two-class classification approach. This multi-class classification using SVMs adaptation is straightforward and has two main approaches: the one-vs-all approach and the one-vs-one approach. An in-depth comparison of the two approaches can be found in [100] and Figure 4.1. In the one-vs-all approach, k SVM classifiers are constructed for each class, $class_k$, in the training dataset. Each classifier then considers the samples of $class_k$ as positive and all others negatives. In detail, the i^{th} SVM ($i \in [1 \dots k]$) labels samples of $class_i$ as +1 and the remaining samples as -1. The result is k decision functions as shown below:

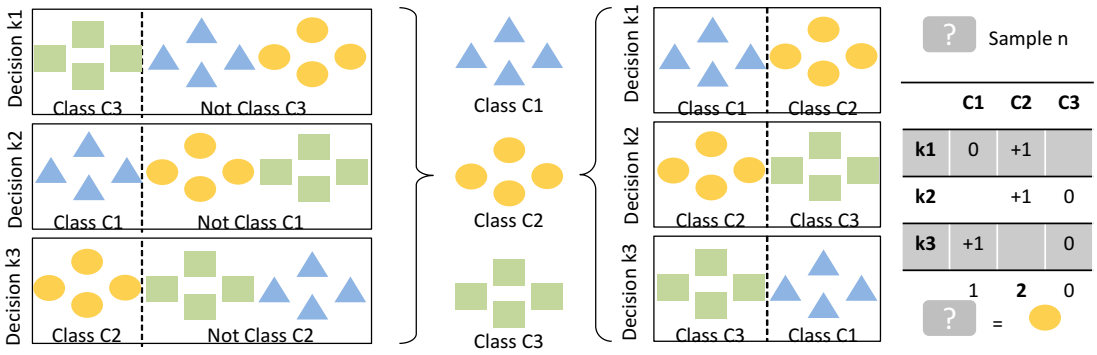
$$\mathbf{x} \cdot \mathbf{w^1} + b^1, \dots, \mathbf{x} \cdot \mathbf{w^k} + b^k \tag{4.3}$$

Where the class of each sample is chosen according to the following decision criteria derived from all k SVM's decision functions:

$$class_i \equiv argmax_{j=1\dots k}(\mathbf{x}_i \cdot \mathbf{w}^j + b^j) \tag{4.4}$$

Unlike the one-vs-all (or one-vs-rest) approach, the layout of features is more involved in one-vs-one. In this method, $k(k-1)/2$ classifiers are constructed for k classes with each constructed from the samples of two unique classes. After training, the testing is done using a voting system. For each decision function for classes i and j, denoted by $\mathbf{x} \cdot \mathbf{w}^{ij} + b^{ij}$, the sign of the result (i.e., + or -) indicates whether the samples belongs to class i or j. If it belongs to i, then the vote for i is increased by 1. Otherwise, the vote for a class j is increased by 1. After all $k(k-1)/2$ decision functions have contributed a vote, each sample is classified into the class it received the highest votes for.

For the experiments in Section 4.6, we applied the one-vs-one method (see Figure 4.1) as it gives us a better notion of non-conformity scores. These are a crucial part of our statistical classification (see Section 4.5.3), otherwise known as conformal prediction.



(a) One-vs-All (b) One-vs-One

Figure 4.1: Comparison of One-vs-All and One-vs-One approaches.

4.2.2 Conformal Prediction (CP)

In traditional classification, the algorithm typically chooses a single class label per sample. This decision is absolute and inflexible, regardless of how well the sample actually fits, and ignores alternative choices despite their likelihood. Thus, in cases where multiple class choices for a single sample have similar probabilities of being correct, a traditional classification algorithm is prone to error. To address these shortcomings, conformal prediction [210] can statistically assess how well a sample fits into a class with the use of qualitative scoring. This relies on *non-conformity* (NC) scores.

NC scores are a geometric measurement (e.g., distance to hyperplane using the RBF kernel) of how well a sample fits into one or more classes. They increase with the distance to the hyperplane for incorrect predictions but are inversely affected for correct predictions. NC scores can be used to derive *p-values* to assess how unusual the sample is relative to previous samples. Specifically, a p-value p is calculated as the proportion of a class's samples with equal, or greater, NC scores. These p-values are therefore highly useful statistical measurements to gauge how well a sample fits into one class, compared to all other classes, and lends flexibility toward more accurate classification.

For intelligent classification we consider the *credibility* and *confidence* of each choice. A high credibility score, i.e. highest p-value of sample set, indicates a clear paring between a sample and a class label. The qualitative metric *confidence* is defined as $1 - p$, where p defines the line between confident and ambiguous labelling. By analysing CP credibility and confidence scores, one can determine the quality of classification much better than with standard classification. For example, choices with high credibility, but poor confidence, imply that multiple class labels have a p-value close to the chosen class. Alternatively, poor credibility and confidence scores may prove that a sample does not match any known class and belongs to a new family (i.e., zero-day malware).

Furthermore, by implicitly setting a confidence threshold, i.e. *p-value threshold*, we can obtain a set of likely class labels per sample. This is highly desirable for classifications with low confidence as one can tune the threshold for higher accuracies. However, conformal prediction is costly performance-wise and it is still necessary to choose the most liable option from each set of predictions. Therefore it is essential to choose a p-value threshold that maximizes accuracy for the least performance cost.

While SVM *can* provide probabilities for each classification choice, e.g. for malware detection [174], derivation of these probabilities is based on Platt's scaling [163] which, like other regression techniques, are sensitive to outliers (i.e. distant data points). Such predictions also tend towards extremes, unlike conformal prediction [245], as they transform the dataset produced by SVM instead of the actual dataset.

4.3 Novel Hybrid Prediction: SVM with CP

This section combines background knowledge, previously given on support vector machines and conformal prediction, to provide the necessary theoretical foundation for our novel hybrid predictor. Full implementation details and the distribution of work will be provided in Section 4.5, while results and performance will be given in Section 4.6.

With a flexible confidence level, conformal prediction is a highly desirable algorithm for accurately classifying malware. The downside, however, is that CP is computationally expensive. This is because the conformal predictor must place every sample into every possible class and measures the sample's non-conformity score. As previously mentioned, this allows the conformal predictor to then calculate confidence and credibility scores to identify which classification choices are likely of being incorrect.

Normally, in order to measure the non-conformity score for every class k and sample i combination, a traditional classification algorithm is needed to obtain non-conformity scores. For example, as previously explained in Section 4.2.1, one-vs-one multi class SVM can calculate these scores using each sample's distance to the hyperplane. Thus implementing a traditional SVM classifier first, to produce a vector per possible classification, provides the non-conformity scores necessary for conformal prediction and a baseline to compare with the hybrid predictor's performance scores (Figure 4.2).

Furthermore, as we prove later, selectively invoking CP with a standard SVM classification quantitatively improves accuracy without the cost of performing CP on all samples. This concept of invoking conformal prediction only when SVM does not meet a desired classification quality level is key to our predictor (see Figure 4.2). In cases when SVM classification confidence is low, CP can also be applied to understand why (e.g., which classes are ill-defined). If choices are unreliable (i.e., low confidence), the class chosen by SVM should have a low p-value compared to other choices. Hence, confidence scores for the SVM decision would be low. Lastly, these scores can also be used to verify accurate SVM decisions made with acceptable confidence.

While novel machine learning techniques are important, the set of features fed to the predictor is arguable more essential (i.e., even well designed classifiers struggle when given poor input). Hence, implementation details of our hybrid classifier will follow the next section on translating behaviour profiles to machine learning input.

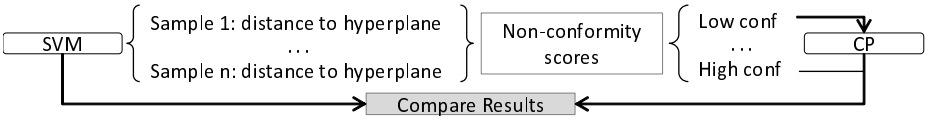


Figure 4.2: Theory behind hybrid SVM and conformal predictor (CP).

4.4 Obtaining CopperDroid Behaviours

In this section we provide an overview of our approach for the multi-class classification of Android malware. We first summarize behaviour reconstruction aspects of the CopperDroid platform and present our approach for classifying malware into classes using support vector machines. We then discuss a strategy for improving SVM-based decisions using conformal prediction. Figure 4.3, discussed in detail below, gives an overview of our classifier (later named DroidScribe [54]) in relation to CopperDroid. While generating and processing behaviours were solely the author's work, the standard SVM was applied by a collaborator and the novel hybrid component was a joint effort. While the collaborator calculated p-values from SVM results, the author applied selective CP, computed the new results, and analysed result improvements.

4.4.1 System Architecture

The first stage of our methodology, as seen in Figure 4.3, is data acquisition. By submitting the samples in the Malware Genome Project dataset to CopperDroid, we were also able to evaluate our machine learning methods on over 1,200 malware from 49 malware families in 2015. Our results of a larger, more current, dataset was accepted into MoST 2016 [54]. Reiterating segments from previous chapters, CopperDroid reconstructs traditional OS (e.g., process creation, file creation) and Android-specific (e.g., SMS send, IMEI access, Intent communications) behaviours from detailed system call traces. This includes the complex, in-depth, reconstruction of IPC binder transactions, which are normally achieved via `ioctl` system calls. CopperDroid also identifies sequences of related system calls to derive single, high-level, behaviours such as network access.

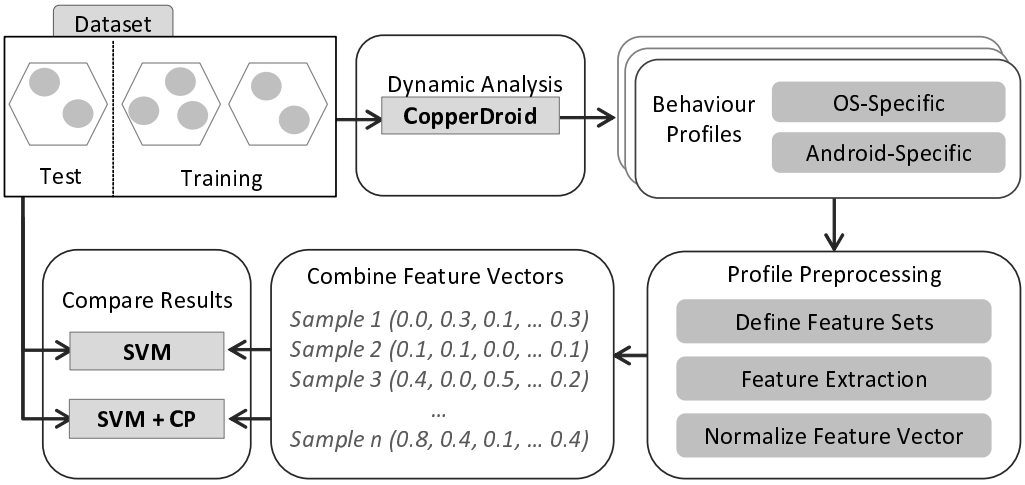


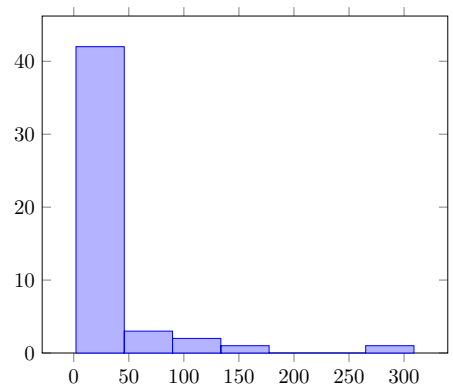
Figure 4.3: CopperDroid behaviours captured and processed for classification.

CopperDroid provided the author with a lot of flexibility in choosing features for classification at multiple levels of abstraction. As we posses both system call traces and behaviour profiles for every sample, we were able to experiment with feature sets at multiple levels of granularity (Section 4.6): from bare-bones system calls to high-level actions such as `sendText`. This is has been key in demonstrating the advantages in using the author's behaviour reconstruction over traditional system call traces.

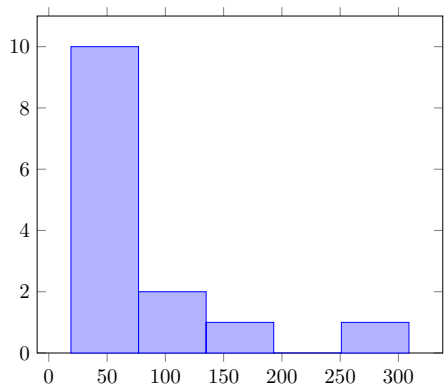
4.4.2 Modes and Thresholds

As discussed in Section 3.6.1, malware do not always exhibit behaviours when running in the CopperDroid emulator. This can be due to incompatibility (i.e., wrong API level), wrong stimuli, or malware evasion (see Section 2.3). In these cases, CopperDroid occasionally outputs behavioural profiles containing little or no behaviours. Therefore, in some situations, we give the option to filter out these samples at the start of our analyses. Thus, our framework has the optional threshold for number of behaviours per sample. In Section 4.6 we experimented with this threshold and elaborate on the trade-offs.

Our second level of optional filtering is the number of samples per family. As this is classification, as opposed to clustering (see Section 2.2.6), all samples are already correctly labelled. Hence, we know before analysis how many samples per malware family, or class label, exist. With the option of filtering out families with very few samples, we can then improve accuracy. Histograms showing the number of samples per malware family class can be seen in Figure 4.4, where subfigure (a) is of all our samples, while (b) has a twenty samples per family cutoff used in several of our experiments.



(a) Histogram of malware family classes and the number of samples it has (0-300 samples).



(b) Histogram of malware family classes with 20 or more samples (20-300 samples).

Figure 4.4: Number of samples per family with and without cutoff.

Although discarding samples with activity levels below threshold reduces the training set (49 to 14), our hybrid solution can still provide accurate classification. As discussed previously, keeping these sparse profiles (i.e., outliers) would cause issues with traditional means of classification, such as SVM, as they are often forced to make a choice in all cases. However, by using conformal prediction with SVM to predict the class from a set of top matching classes, we can still visibly improve traditional SVM accuracy *without* the enforcement of family sample and/or sample behaviour thresholds.

Once we have selected a training set (the same as the dataset[1] if no filtering) we can begin analysis. As mentioned previously, as we have both behavioural profiles and matching system call trace, we can perform classification with different sets of features sets. This will be discussed further in Section 4.5, but the available modes are essentially system call level, binary or frequency of call, and behaviour reconstruction with or without arguments, and with or without system call frequency included.

4.4.3 Parsing CopperDroid JSONs and Meta data

Per sample analysed, CopperDroid outputs a JSON (JavaScript Object Notation) file storing behaviours, a directory of recreated resources, and a file of system call data. To reduce the size of the system call trace, often 200 MB, the third file merely holds the frequency of system call names from the trace. While this excludes parameter values, it is sufficient for our two system call modes of analysis (i.e., binary or frequency).

From each JSON file the framework can read all reconstructed high-level behaviours. Furthermore, each high-level behaviour retains their corresponding low-level events, i.e. system call, parameters, and return values. The categories and subcategories of behaviour we can extract from these JSON files, are described below. Once all the behaviours of all samples has been translated into feature vectors, some additional data is generated at this point. First, network traffic size is added up across all network behaviours per sample. Directories holding reconstructed files are searched and analysed, for example to understand the file type, etc., instead of trusting the file extension.

4.4.4 Behaviour Extraction

In our analyses behaviour extraction occurs during `rec_*` modes, which are four of our six analysis modes. Moreover, only two of those four `rec_*` modes analyse the reconstructed parameter and return values of system calls belonging to a behaviour.

[1] We discuss overfitting in detail further on in Section 4.5.2 on page 105.

Feature Set	Contained Details
$S1$ Network Access	IP address, port, network traffic size
$S2$ File Access	file type, file name, regular expression
$S3$ Binder Methods	method name and parameters
$S4$ Execute File	file type, user permissions, arguments

Table 4.1: Extracted CopperDroid behaviour classes and details for subcategorizing.

This includes arguments of methods invoked remotely via IPC Binder. These high-level behaviours, which we divide into behavioural feature sets, are network accesses, file accesses, binder methods, and file execution (see Figure 3.10 and Table 4.1). It is important to note that the features and sub-features were defined prior classification, mitigating overfitting. Details of our behaviour sets can be found below, several of which represent a series of system calls. That is, CopperDroid uses value-based data dependencies to group system calls based on file descriptors (see Section 3.5.1).

Although multiple system calls are condensed into single behaviours, all parameter values and return values are retained. Using this data, we were able to break popular behaviours (e.g., 50% of all sample behaviours were filesystem accesses) into subcategories for a more fine-grained behaviour feature set (e.g., type of file created). For example, by examining the parameter values of execution system calls, we can separate silent installations of APKs from other file execution behaviours like shell scripts.

While there are many additional ways to split behaviours into finer categories, we have found via experiments (i.e., incremental accuracy increase per category) that this feature set best captures the different behavioural patterns of malware families. These detailed behaviour-based feature sets, $S1$ to $S4$, were constructed from CopperDroid JSON files. More in-depth feature statistics of the dataset can be found in Section 4.6.

$S1$ *Network Access:* Roughly 66% of our malware samples regularly made network connections to external entities. Each network access behaviour represents a sequence of system calls, normally beginning with `connect`, followed by `sendto`'s. By analysing their parameters, we were able to add granularity to our feature set (see Section 4.6) by creating subcategories based on IP address and traffic size.

$S2$ *File Access:* The second most popular behaviour in our dataset (see Table 4.2, page 107) is filesystem access. This behaviour is reconstructed from system calls using def-use-chains of file names and file descriptors. As mentioned previously (see Section 3.5.1 for details), CopperDroid uses these chains of system calls to fully recreate any actual file creation so that it may be analysed, or even executed, depending on the file type. The author implemented a file extension analysis and three filename character class-mapping (i.e., all characters, all numbers, and mixed) along the lines of other works which have modelled system call arguments in the past [152].

$S3$ *Binder Methods:* CopperDroid effectively reconstructions binder communications from the `ioctl` system calls. Since binder communications are the principal means of inter-process/inter-component communication, they are the gateway to services from the Android system. Consequently, monitoring binder communications and identifying the invoked method is crucial to modelling the behaviour of a malware. When modelling all binder communications we found that `getDeviceID` and `getSubscriberID` were the most frequent methods to be invoked by our malware dataset. For many of these "get" methods, we are less interested in analysing the parameters as they should return predictable data values, but for methods such as `SMS_sendText` the parameters (e.g., destination) tend to be more interesting.

$S4$ *Execute:* There are various files that may be executed within the Android system to run exploits, install apps silently, etc. CopperDroid reconstructs all such behaviours and we model them within our feature vector. In order to differentiate between different file executions, we broke down these behaviours by analysing their parameters. For example, if the parameters include a `pm` followed eventually by an `install` and a file name, this is an indication of an app being installed silently without the users' permission. Furthermore, as there are multiple ways to execute the same file (i.e., the same app installation can be done with different arguments), being able to group them all as the same behaviours with same outcome is advantageous and makes our method less susceptible to misdirection (see Figure 3.11).

While we use these behaviour sets to classify malware, these can be easily applied to detect malware (i.e., binary classification). Furthermore, there are several additional behaviour features that we may use when implementing a two-class classification as opposed to multi-class classification. Such behaviours would be equally popular amongst all malware families, but *only* exhibited by malware. For instance, a user-level application directly altering network configuration files is against Android discretionary access protocols and would be a strong indicator of malware, but not necessarily what family due to similar malware behaviours. This concept is explored further in Chapter 5.

4.5 Multi-Class Classification

Although the CopperDroid behavioural profiles contain detailed information about each malware sample's actions, the raw profiles are not suitable for applying machine learning techniques as they normally run on vectorial data. Hence, in order to implement our classifier, we must first project the behaviours into a vector space. This is an extension of

the author's work on behaviour reconstruction in Chapter 3, which was also performed by the author. Once the data has been vectorized for all samples in the training set, i.e. all samples that meet any given thresholds, the data is passed to a traditional SVM classifier running the default radial based function kernel (refer back to Section 4.2).

We then evaluate our pure SVM classifier using the testing set which, in our experiments, is identical to the training set. Finally, by analysing our SVM results, this section will conclude by demonstrating the trade-off between accuracy and samples omitted from SVM classification, and how our conformal predictor improves the situation. This leads to more detailed results and experiments in the following Section 4.6.

4.5.1 Behaviour Feature Vectors for SVM

In order to embed the behaviours into a vector space, we construct one feature vector per sample using the sample's feature set S, comprised of behaviours reconstructed with CopperDroid. From there we can build a 2D vector space model with the dimensions of $(number\,of\,samples) \times (|S| + |extra\,data|)$, where S is the set of behaviours, and $extra\,data$ is data that does not belong in S. For our highest level of accuracy, each feature vector is comprised of these two, constant sized, vectors across all samples.

The first segment of each feature vector we use for classification holds feature frequencies, and therefore has a length equal to the size of the feature set, i.e. $|S|$. Specific sizes can be found in Figure 4.6(a) (page 110), but they roughly range from 20 to 130, depending whether the mode includes raw system calls in the classification, and whether parameters were modelled. In other words, the behaviour profile of each malware sample x is mapped to the vector space by constructing vectors $f(x)$ and $z(x)$, and appending the latter vector to the former. Vector $f(x)$ is constructed by considering each behavioural feature s extracted from x, and incrementing its respective dimension. Formally, the mapping f can be defined by the following for a set of malware samples:

$$f : X \to \{0, n\}^{|S|}, f(x) \to (I(x, s))_{s \in S} \tag{4.5}$$

Where the indicator function $I(x, s)$ is computes frequency using:

$$I(x, s) = \begin{cases} \sum_i [b_i = s] & \text{number of instances } s \text{ in } x \\ 0 & \text{otherwise} \end{cases}$$

Therefore the significance of an individual behaviour can be measured by the frequency of its occurrence. This mapping is also how we determined system call frequencies for mode `syscall_freq`. After normalizing the vector space for all samples, a frequency of 0 (i.e., $f(x, s) = 0$) shows that behaviour s has little to no importance, while

a behaviour with a non-zero frequency value illustrates that the behaviour generally represents sample x's actions and purpose more accurately. However, as our classification algorithm seeks to find patterns in the behavioural profiles, the absence of a behaviour in a sample may be equally essential in classification as an action with a near-one value.

The end of the feature vector, denoted by $z(x)$, holds numerical data not best represented as behaviour frequency (i.e., vector $f(x)$). For example, one dimension of this vector currently represents the average size of network traffic in bytes, see Equation 4.6. This may be further divided into two dimensions for incoming and outgoing traffic. Furthermore, while not in the scope of this chapter[2], in the future we may calculate average file sizes per directory as various locations are designations for different kinds of files. For example, we have seen that files within directories such as "shared_prefs" tend to be smaller, on average, than files stored in "databases" or "files".

Let us consider an arbitrary sample i that performs network communications and accesses device data via Binder methods. The corresponding vector for this particular sample would look like the following, where behavioural features frequencies from feature sets S1 to S3 belong to vector $f(x)$, and the remaining data is stored in $z(x)$. This vector would also correspond to a i^{th} row in Figure 4.3 (page 98), where $1 \leq i \leq n$.

$$f(x) + z(x) \rightarrow \left(\begin{array}{c|c|l} 6 & \texttt{Network Access} & \} \ f(x) \in S1 \\ \ldots & \ldots & \} \ f(x) \in S2 \\ 1 & \texttt{getDeviceID} & \\ 1 & \texttt{getSubscriberID} & \} \ f(x) \in S3 \\ \ldots & \ldots & \\ 235 & \texttt{Network\_Traffic}_{\texttt{bytes}} & \\ \ldots & \ldots & \} \ z(x) \notin S1, S2, S3 \end{array} \right) \tag{4.6}$$

Even when considering all features within all behaviour sets, the vector size stays relatively small (i.e., the length of $f(x) + z(x)$[3] is less than 100), but is compact with unique and essential behavioural details. With these feature vectors, we can begin classifying samples using support vector machines. Our SVM takes in the training set and the testing set and creates hyperplanes to separate the training samples into classes and produces and class confidences for evaluating its classification results. Using SVM alone, accuracies range from 75% to 94% when filtering out samples exhibiting little to no behaviours. To improve while discarding less samples, we introduce a hybrid solution to refine our results using CP. Our experimental results are provided in Section 4.6.

[2]Not all samples in our dataset were analysed with the resource reconstructor.

[3]The + symbol represents vector concatenation.

4.5.2 Accuracy and Limitations of Our Traditional SVM

When evaluating these SVM results, there is a risk of overfitting to the test set as parameters can be modified until the classier performs optimally. In other words, knowledge about the test set can wrongly influence the classifier into better performance. To remove this bias, one may divide the dataset into training, testing, and validation sets. Unfortunately, partitioning the samples sets drastically reduces the number of samples used for SVM training. This is undesirable considering our original dataset size, therefore cross-validation (CV) was implemented in this thesis to ensure some level of validity.

In our k-fold CV, we choose $k = 5$, spiting the training set into five equal sets. As a large k could split our samples into many sparse sets, we choose five as it offered the most tests with all subgroups containing more than a single sample. Our larger dataset in [54] allowed for $k = 20$. A model is then trained using $(k - 1)$ of the sets, i.e. folds, as training data. The remaining is set is then used for validation. The performance is then calculated by averaging of the values computed in a loop of k, i.e. five, times.

SVM accuracy scores, as well as precision and recall scores, are then stored for later comparisons. Functions for calculating misclassifications and feature statistics per sample, family, and entire dataset (see Section 4.6), were developed by the author in order to test different feature sets in order to identify the most expressive ones.

By analysing our traditional SVM classification accuracy scores, we can see that filtering out samples with few behaviours is beneficial (75% without, 95% with filtering). In most of our experiments, we found that a filter of ten behaviours per sample was the best balance between number of samples analysed, and accuracy (see Figure 4.8). However, this reduces the number of classified samples, and so we designed our hybrid SVM/CP method to accommodate sparse profiles. In the following sections, we implement and evaluate this hybrid method which we described in Section 4.3.

4.5.3 Enhancing SVM with Conformal Predictions

In our framework, conformal predictions uses past experiences to determine precise levels of confidence in SVM predictions. This is helpful, as SVM is often forced to label a sample despite how uncertain it is. This becomes detrimental to classification accuracy when dealing with sparse behavioural profiles, as that sample may plausibly map to multiple classes. In Section 4.3 we had discussed how CP can theoretically be used in conjunction with SVM in such situations and improve classification accuracies by making predictions sets, instead of a singular prediction, when useful. In this section, we fully develop a systematic framework (illustrated in Figure 4.5 below) to achieve this.

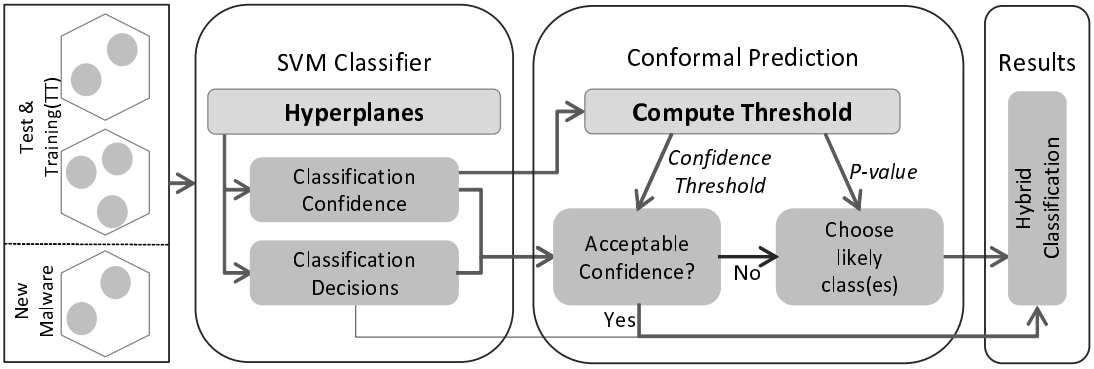


Figure 4.5: Selective invocation of CP to refine uncertain SVM decisions.

As discussed in Section 4.3, we first evaluate the confidence of SVM's classification. To do this, we use CP as an evaluation framework. By running SVM on the training and testing (TT) sets, we obtain a measure of class-level confidence of each class. The measurement, based on correct class labelling, is the average confidence of all samples that were correctly classified during testing. It is a measure of how much we can rely on the hyperplanes that SVM constructs as a poor score implies a imprecise segregation of classes. What constitutes a good class-level confidence, however, is subjective.

For our experiments, we elected the median of all class-level confidences to be the cutoff of acceptable scores, as further described in Section 4.6.4. As shown in Figure 4.5, if SVM classifies a sample as belonging to a class with low confidence, then the decision is deemed unreliable. In such cases, the conformal predictor is invoked and a p-value threshold (i.e., a confidence level) is calculated to determine all plausible class labels that pass that threshold. By expanding our prediction set with a few, highly-likely, choices returned by conformal prediction, we achieve a better classification accuracy, fewer false positives, and fewer false negatives.

Again, what constitutes a reasonable p-value threshold is subjective. Lower p-value thresholds lead to better accuracy at the cost of larger prediction sets. For our experiments we test a range of p-value thresholds to demonstrate the trade-off relationship between size of the prediction set and classification accuracy.

4.6 Statistics and Results

In this section we demonstrate our machine learning capabilities via experimental results of our Android malware classifier. Specifically, we show how our classifier primarily implements SVM, but defers to hybrid prediction in cases that would most improve accuracy. In addition, selective CP is beneficial as it incurs performance costs.

4.6.1 Dataset, Precision, and Recall

Our dataset of Android malware was collected from the Android Malware Genome project [249]. Initially this provided us with a set of 1,230 samples, but due to CopperDroid limitations and malware possibly evading dynamic analysis (see Section 4.7), 93 samples were discarded due to their sparse behavioural profiles. In the end, this thesis experimented with a corpus of 1,137. Months later, the analysis of 5,560 samples stayed consistent with this chapter's results [54]. The following experiments, and their results illustrated in Figures 4.6-4.11, were run on a quad-core 2.5 GHz Intel i7 processor with 16GB of DDR3 RAM clocked at 1600 MHz, running OSX Yosemite version 10.10.3.

The statistics of the author's behaviour sets, as seen across our entire dataset of 1,230 samples, can be found in Table 4.2. These numbers only increase when behavioural thresholds are imposed as the resulting data set contains the more active malware.

Table 4.2: Behaviour features and top two sub-features exhibited by datasets.

Feature Set	Samples	Features	Subfeature	Samples	Features
S1 Network Access	66%	25.5%	IPv4-mapped IPv6	62.6%	18.8%
			DNS monitored	6.2%	6.1%
S2 File Access	71.8%	40.6%	XML	52.2%	13.2%
			database	38.6%	2.9%
S3 Binder Method	78%	14%	getDeviceID	59.7%	5.5%
			getSubscriberID	42.6%	2.4%
S4 Execute	26.6%	7.9%	generic	26.4%	7.8%
			silent install	0.6%	0.1%

It is important to note that showing percentages for *all* behaviour subcategories (e.g., filesystem access on an mp3 file) is not necessary as many of the finer behaviours are rarely seen. Hence Table 4.2 only shows the top two subfeatures per behaviour set. However, a complete table can be found in Appendix C. These finer features were created by incorporating parameter and return values details recreated by CopperDroid.

For measuring multi-class classification quality we use the notion of true positives, false positives, false negatives, precision, recall, and accuracy, as discussed in [190]. The (p)recision and (r)ecall for any sample s belonging to a $class_i$ is defined as:

$$p_i = \frac{TP_i}{(TP_i + FP_i)} = \frac{TP_i}{(TP_i + \sum_{class_i \neq class_j} classify(s_j) = class_i)} \tag{4.7}$$

$$r_i = \frac{TP_i}{(TP_i + FN_i)} = \frac{TP_i}{(TP_i + \sum_{class_i} classify(s_i) \neq class_i)} \tag{4.8}$$

Where $class_j$ is any class that is $\neg class_i$. Here, the value of TP (true positives) represents the number of samples in a given family, say $class_i$, that were correctly classified

to the family it belongs to. As FP and FN (i.e., false positive and false negative respectively) represent types of misclassification, as they increase, precision and recall decrease. Specifically, FN represents the number of samples within a given family that were misclassified. In multi-class classification, the prediction would label a sample of $class_i$ as any class that is not i (e.g., $class_j$). FN is used to calculate recall (see above).

Hence, the sum of FN_i and TP_i equates to the total number of samples of $family_i$, otherwise known as $class_i$, even if the wrong labels are distributed across several other classes that are $\neg class_i$. Finally, FP represents the number of times every sample *not* within a given family (e.g., $class_j$) is misclassified as the family currently being analysed (i.e., $class_i$). FP is then determined by examining all samples from the testing set and determining whether they were *wrongly* labelled as $class_i$.

4.6.2 Classification Using SVM

In this section we discuss classification results when running SVM classifiers on features derived from CopperDroid. We do this in several modes, as previously described in Section 4.4.2 and Table 4.3. We first evaluate our classifier on vectors extracted from basic system calls without argument modelling. This is done with our basic baseline, based on the SVM results of boolean-based feature vectors modelling the presence, and absence, of system calls in the trace. We then repeat the SVM classification experiment but with system-call frequency (i.e., mode sys). For all subsequent experiments, based on behaviours instead, we use the performance from mode sys as our enriched baseline.

After establishing our baselines, we present and compare baseline results and performances to evaluate our SVM classification using CopperDroid's high-level behaviours reconstructed from just system call data. The goal of this is to reduce runtimes without sacrificing accuracy and, where possible, improve the classification accuracy. These system call baselines, produced by a collaborator, demonstrate the novel and useful aspects of the author's behavioural reconstruction in CopperDroid as well using these behaviours for our hybrid multi-class classifier.

Table 4.3: Operational SVM modes. First two are baseline for following modes.

Mode	Type	Features	Argument Modelling	Filter Trivial
`sys*`	Boolean	system calls (syscall)	✗	-
`sys`	Frequency	system calls (syscall)	✗	-
`rec_b+`	Frequency	syscall + high-level behaviour + binder	✗	-
`rec_ba+`	Frequency	syscall + high-level behaviour + binder	✓	-
`rec_b`	Frequency	rec_b+	✗	syscall
`rec_ba`	Frequency	rec_ba+	✓	syscall

4.6.2.1 Baseline: Classification Using System Calls

The overall results for SVM classification in different operational modes, using different feature sets, are shown in Figure 4.6 (page 110). More specifically, comparisons on the number of features used for SVM-based classification, the overall runtime divided into feature extraction and classification, and the classification accuracy of each operation mode can be found in Figures 4.6(a), 4.6(b) and 4.6(c), respectively.

In general, from our sets of experiments, we see that CopperDroid's behaviour reconstruction retains high accuracy levels despite drastically reducing the number of features (roughly 80 to 20, see Figure 4.6(a)). Furthermore we see that lowering the number of features has improved performance, as it results in less calculations, allowing us to lessen the performance or accuracy trade-off of most traditional systems.

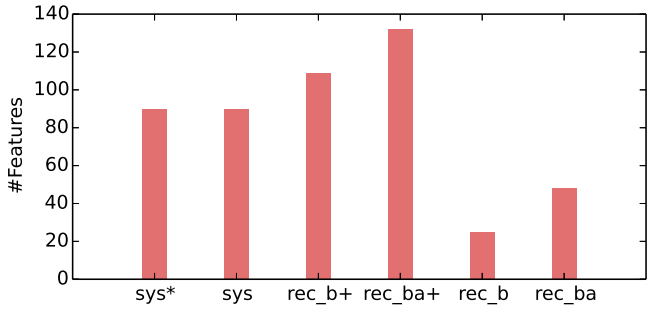
For experiments based on basic system calls only, we ran the SVM in a boolean mode (i.e., call was used or unused) as well as a frequency mode (i.e., number of times a call is executed). The latter yielded marginally better results than boolean mode and so for all subsequent experiments, we used the results from this mode sys as our baseline. It should be mentioned again that the system call names and frequencies were stored in a text file and fed to our classifier. We deliberately used this fast-to-read representation in order to prevent skewing runtime measurements as reading large system call traces, and not modelling system call arguments, is memory intensive.

4.6.2.2 Enrichment of the Baseline

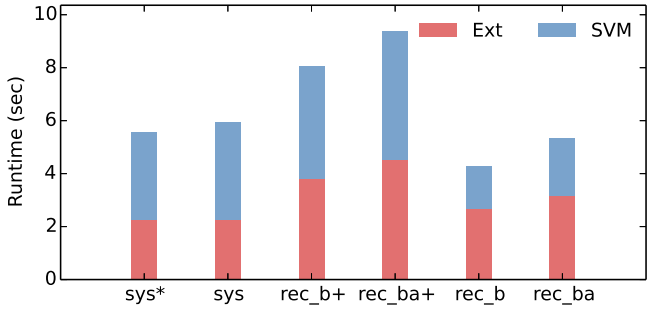
To improve our classification techniques there are three levels of improvement unique to our CopperDroid behaviour profiles. In the first step, the author deviates from individual system calls to focus instead on reconstructed behaviours. By extracting actions from sequences of related system calls, we can reduce noise from irrelevant fluctuations. For example, although the same file may be written in ten, one byte, writes instead of one, ten byte, write, our classifier would register both as the same file access behaviour.

Secondly, the author utilizes CopperDroid's IPC binder behaviour extraction. This is a useful, but not straightforward, process that relies heavily on CopperDroid's unmarshalling Oracle, which was a core contribution by the author to CopperDroid (see Section 3.4). In the third step the author used each behaviour's details (e.g., filename, filetype, IP address, port, parameters) to further improve accuracy with a more fine-grained, expressive, feature set. These improvements can be seen in Figure 4.6(c).

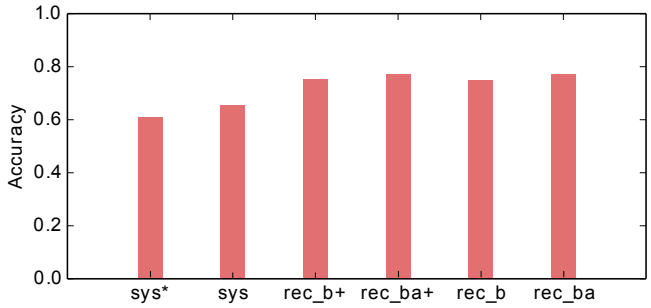
The optimizations and additions we introduced to our feature sets visibly improved accuracies for modes `rec_b+` and `rec_ba+` when compared to our `sys` baseline (see



(a) Number of features across SVM operation modes



(b) Time to extract feature vectors (EXT) and classify samples (SVM)

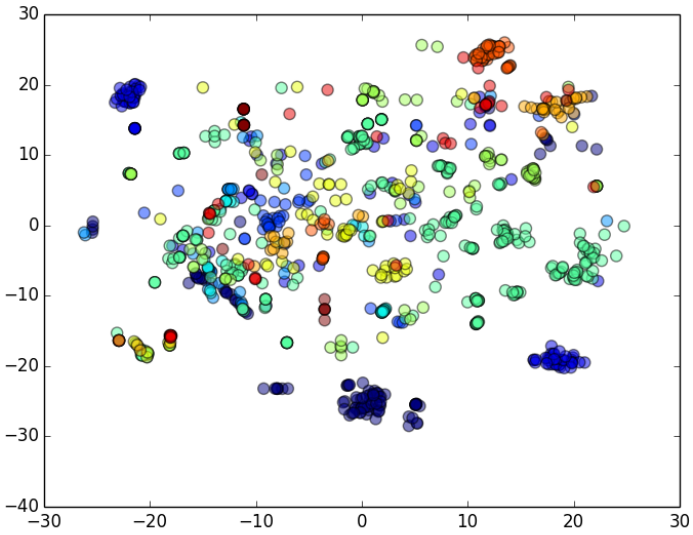


(c) Classification accuracy across SVM modes

Figure 4.6: Feature amount, runtime, and accuracy for each SVM operational mode.

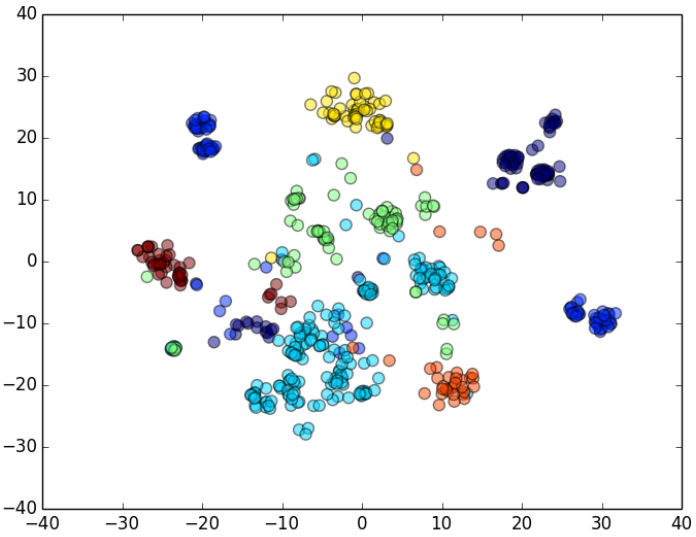
Figure 4.7). However, a larger corpus of features typically leads to slower runtimes for feature extraction and classification phases. Hence, in order to further improve runtimes, we filter out uninteresting system calls (43% across all samples), such as `brk`, which we found to be of no particular help towards classification accuracy.

Filters: The filtration method for the baseline system calls is determined by what CopperDroid did not use to recreate behaviours. A non-exhaustive list of *used* system calls include system calls for files, such as `write`, `writev`, `open`, `close`, and `unlink`, network, such as `connect` and `sendto`, and others like `clone` and `ioctl`.



(Each circle is a sample, each colour is a family)

(a) Classifying with bare-bone system calls, threshold of 20 samples per family.



(Each circle is a sample, each colour is a family)

(b) Classifying with reconstructed behaviours, thresholds 10 behaviours per sample, 20 samples per family.

Figure 4.7: Visual t-SNE[4] classification improvements from system calls to behaviours.

Of the 70 or so system calls filtered out (exact value depends on Android version), there were several get methods (e.g., `getdents64`, `getgid32`) and set methods (e.g., `setsid`, `setpriority`). In our experience, filtering these calls result in noticeable accuracy improvements for our multi-class classification. However, as these calls still have some effect on the Android system, they may be more useful in two-class classifi-

[4]t-Distributed Stochastic Neighbour Embedding (t-SNE) is a technique for dimensionality reduction.

cation (i.e., malware detection) developed in the future. This may be because the system calls can do no harm, or all malware use it evenly, and therefore cannot help differentiate between malware families, but can help separate malware from benign apps. As further discussed in Chapter 6, future work on two-class classification would involve a dataset of benign apps (e.g., PlayDrone [217]). System call filtering could also significantly reduces the number of features and improve overall runtime, as shown in Figure 4.6.

Behaviour Threshold: The author investigated the impact of behaviour quantities on the classifier using the behaviour threshold mentioned in Section 4.4.2. For each sample the author measured the number of extracted behaviours it had exhibited while being run in CopperDroid emulators. This is the sample's *behaviour count*. Samples that demonstrate a higher behaviour counts typically produce richer traces which, in turn, result in detailed feature sets and better classification accuracy. In our experiments we used a *behaviour threshold* to filter out samples exhibiting little to no behaviours. The effect of the behaviour threshold on the classification accuracy is demonstrated in Figure 4.8. It can be observed that as the behaviour threshold increases (i.e., 0, 2, 5, 15, 20, and 30), the accuracy does as well. We did not continue testing past a threshold of 30 as it was above 26.5, the mean of behaviours seen across all samples.

The trade-off to using a behaviour threshold to boost accuracy is that, although the ratio of behaviours to samples is higher, the number of discarded samples increases. This is also shown in Figure 4.8, where the number of samples that meet the threshold goes down as accuracy improves. In Sections 4.6.3 and 4.6.4, we apply conformal prediction to lessen the trade-off of filtering a small set of samples. The hybrid technique can be applied with any number of samples. However, based on the intersection in Figure 4.8, we choose to do our experiments with a base case of 10 behaviours per sample.

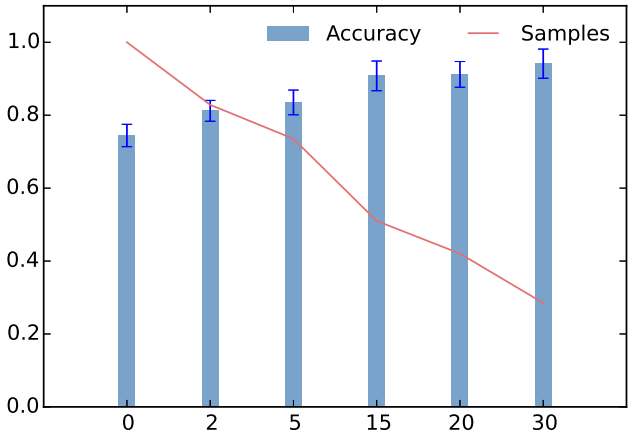


Figure 4.8: Trade-off between analysing samples with x behaviours and accuracy.

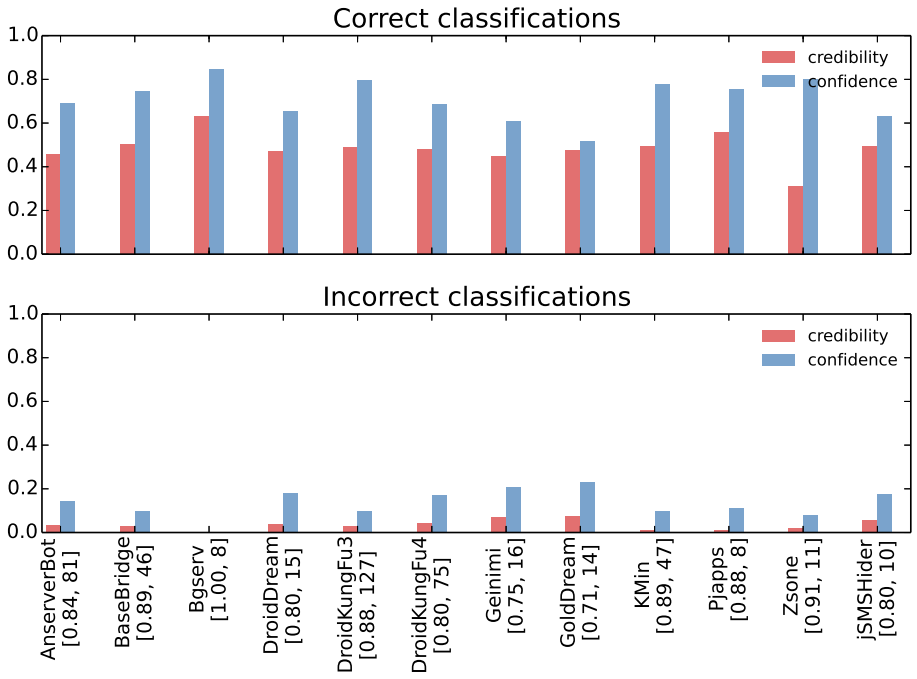
4.6.3 Coping with Sparse Behavioural Profiles

In order to effectively use conformal prediction to address sparse CopperDroid behavioural profiles, we first need to compute p-values for our base case. This computation is based on geometrical distances, as determined with our SVM classifier (refer back to Figure 4.5, page 106). For the purpose of our experiments, our base case uses a behaviour threshold of 10 behaviours per sample, as that is roughly where accuracy intersects the number of samples in the above Figure 4.8.

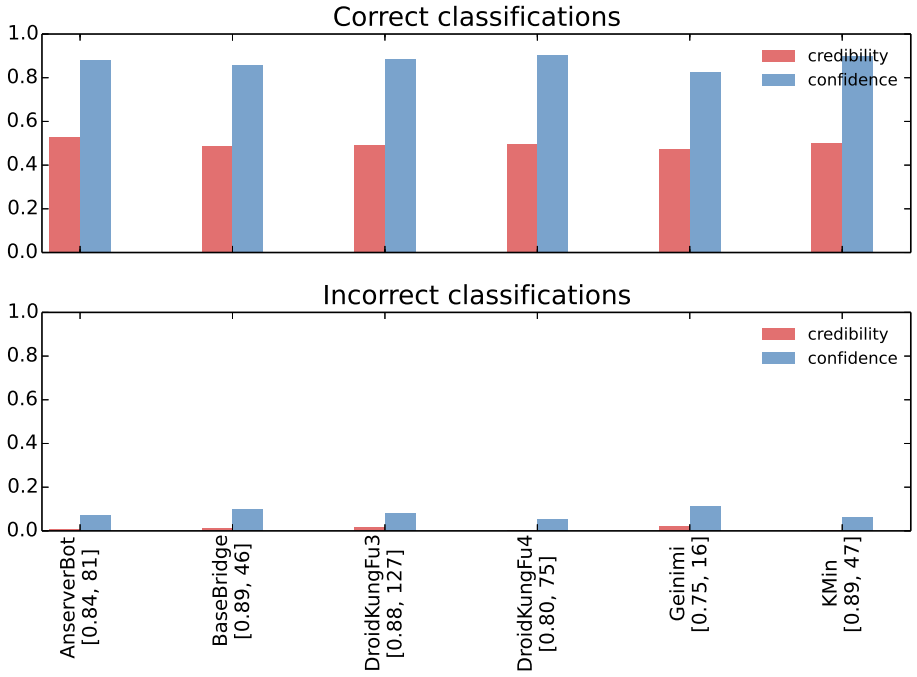
Let us first consider a case with class c, with a cardinality of n, and a new sample s. Here, the p-value is the proportion of samples belonging to class c that have a weaker alignment to c's properties than sample s. If a s does not rightly belong to class c, then all samples in c should have stronger similarities to class c's properties and have p-values of $\frac{1}{n+1}$. Since we compute the p-value of each new sample after it is placed in a class, these values are sensitive to the cardinality of the class. For example, if a $class_i$ has only one sample, the p-value for a new, dissimilar, sample would be 0.5 ($\frac{1}{1+1}$). On the other hand, if the class cardinality of $class_j$ is nine, the p-value for a new, dissimilar, sample would be 0.1 ($\frac{1}{9+1}$). This skews the comparison across classes because if the new sample is dissimilar to both classes i and j, the conformal predictor will be inclined to place it in $class_i$ as it has the lower cardinality, and thus, a higher p-value.

For conformal prediction to meaningfully improve SVM predictions, we must further investigate the effect of class cardinality on credibility and confidence scores along with p-values. Figure 4.9 does this by showing the average confidence and credibility scores, per class i.e. malware family, for both correct and incorrect CP-based classifications of our dataset. As we can observe from the subfigures (a) and (b), when classifying families with a minimum of five samples, the confidence of the classification is significantly lower when compared to a higher threshold. This corroborates our discussion in this section and creates the need to identify a suitable sample threshold when using CP.

In order to derive a meaningful threshold for the number of samples per class, we compared the error rate ($e = 1 - accuracy$) of SVM and CP with a behaviour threshold of 10. By varying the threshold for samples per family we can then see where the error rates of SVM and CP converge. Furthermore, we can identify this convergence point as where CP introduces the least amount of noise for the most optimal hybrid combination of SVM and conformal prediction for improved accuracy. This comparison can be seen in Figure 4.10. As the threshold increases the error rate of the CP decreases. After a threshold of about twenty samples, the error rate of CP roughly corresponds to that of SVM. Therefore, for subsequent experiments, we use a class threshold of twenty in order to optimally use conformal prediction to improve SVM's classification decisions.



(a) Credibility/confidence scores for families with > 5 samples (behaviour threshold=10). X-labels are in the form of "Malware Family" [classification accuracy, samples per class]



(b) Credibility/confidence scores for families with >15 samples (behaviour threshold=10). X-labels are in the form of "Malware Family" [classification accuracy, samples per class])

Figure 4.9: Average class-level confidence and credibility scores for classification.

4.6.4 Hybrid Prediction: SVM with Selective CP

In this section we demonstrate how conformal prediction conjoined with SVM provides a highly flexible framework. We have shown this to be true, even when dealing with sparse behaviour profiles as a result of dynamic analysis limitations. The key concept of this framework is to provide a set of predictions, instead of selecting one, *when it is helpful*. Since CP is an expensive algorithm, this has to be done selectively on as few samples as possible. As we have previously discussed the hybrid decision-making concept in Section 4.5.3, in this section, we demonstrate how our framework functions in an operational setting. We first show by evaluating SVM with CP, measuring the class-level confidences, and using them to decide the best times to invoke the conformal prediction in order to help correct errors committed by SVM classifier.

When evaluating SVM with CP we plotted the average confidence scores, per class, for correct SVM decisions in Figure 4.11(a) (page 117). Again, thresholds were derived by determining the intersection of accuracy gained and samples filtered out (page 112), and by determining the intersection of SVM error rate and CP error rate in Figure 4.10. This yielded a behaviour threshold of 10 and a sample threshold of 20.

As shown by Figure 4.11(a) (page 117), the correct classifications for samples from the BaseBridge, DroidKungFu4, and GoldDream malware families have low confidence during training. This demonstrates how samples in these classes cannot be easily distinguished from other classes and SVM is forced to pick one class with little conviction. Thus, the class-level confidence scores in this figure are the basis to decide whether or not to invoke the CP. In our experiments, the median of these confidence scores served

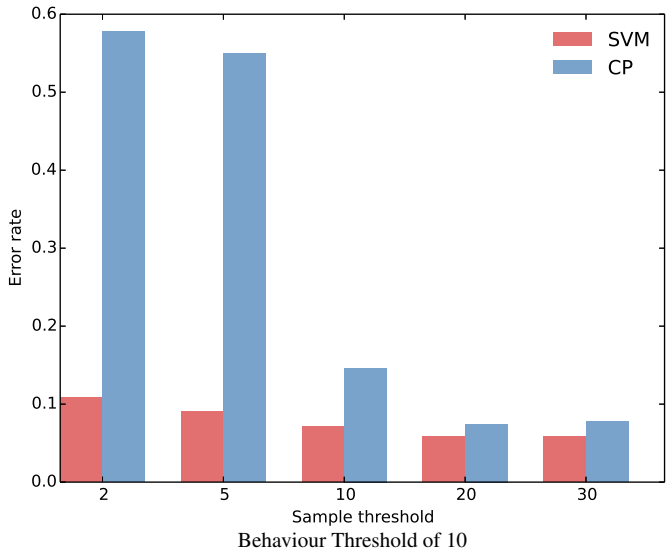


Figure 4.10: SVM and CP error rates for different samples per family thresholds.

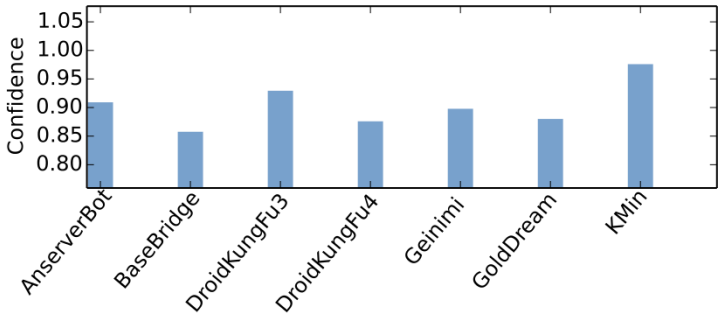
as the cut-off. Thus, for each new unclassified sample, if SVM maps the sample to any class with a confidence score below this threshold, we invoke the CP. In Figure 4.11(a), this is the case with BaseBridge, DroidKungFu4 and GoldDream. So, whenever a test sample was mapped to them, CP was used to minimize misclassifications.

With this hybrid solution, we are no longer bound to accept one label, as we can use CP to find highly plausible choices when there is no clear choice. For every CP invocation, we can then recalculate precision and recall, as well as the prediction set size, to determine its benefit. Both initial performance scores (i.e., accuracy, precision and recall) and the recalculated scores for the hybrid classifier were developed the author. Figures 4.11(b), 4.11(c) and 4.11(d) show the size of the prediction set as well as the improvements in recall and precision for all samples. For each class, we show five bars corresponding to the different p-value cut-offs, i.e. confidence scores ($confidence = 1 - cutoff$). We chose to use p-values of 1.00, 0.30, 0.10, 0.03, and 0.00, for our experiments. It may be recalled from Section 4.2.2, that the higher the p-value the more a sample must fit into the class in order to be considered.

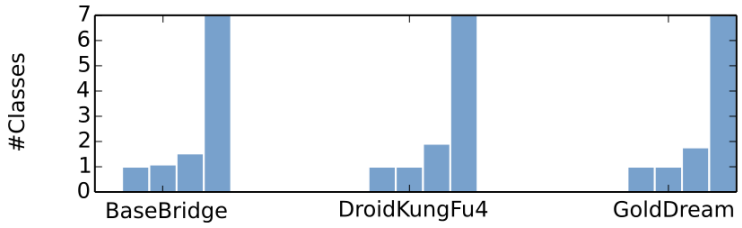
As Figure 4.11 demonstrates, a p-value of 0.0 would return a conformal prediction set identical to the universal set containing all classes. Conversely, a p-value of 1.0 is unlikely to return a prediction set with any classes, as it is highly unlike for there to be a complete match. Thus, the precision and recall scores for p-value 1.0 would be identical to the SVM scores as no additional classes are considered. For a threshold of 0.0, the conformal predictor would provide 100% precision and recall. However, the chance of error is high as it is considering all choices, even the bad ones, and at a huge performance cost. With values ranging from 1.0 to 0.0, the prediction set increases depending on the number of classes that meet the threshold for the sample.

Hence, we see that changes in the p-value threshold directly impacts precision, recall, and the number of classes in the CP prediction set. This is where CP's adaptability enables us to work with poorly distinguishable samples. Such samples could be due to poor code coverage, poor feature selection, or because the malware embodies properties from multiple families. In the cases of DroidKungFu4 and GoldDream in Figure 4.11, there are massive improvements in precision from 92% to 98% by dropping the p-value cut-off from 0.1 to 0.03. This is even more impressive when we consider the size of the prediction set which consists of an average of just two.

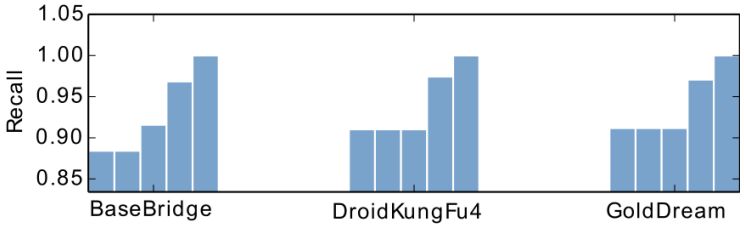
As the invocation of our conformal predictor returns a prediction set which may, or may not, contain the ground truth, the precision and recall of the results does not necessarily improve at every p-value cut-off; the recall for DroidKungFu4 and GoldDream does not improve until the p-value cut-off is decreased all the way until 0.03.



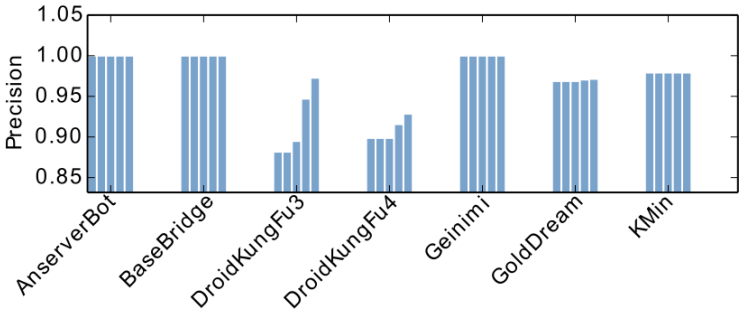
(a) Confidence scores of correct SVM decisions for seven malware families. Behaviour threshold = 10, family sample threshold = 20.



(b) Classes per misclassification. P-values = 1.00, 0.30, 0.10, 0.03, 0.0, behaviour threshold = 10, family sample threshold = 20.



(c) Recall scores for three families. P-values = 1.00, 0.30, 0.10, 0.03, and 0.00, behaviour threshold = 10, family sample threshold = 20.



(d) Precision scores seven families. P-values = 1.00, 0.30, 0.10, 0.03, and 0.00, behaviour threshold = 10, family sample threshold = 20.

Figure 4.11: Confidence, size of prediction set, recall, and precision for a range of p-value cut-offs.

The key takeaway from Figure 4.11 is the seamlessly adaptive nature of the hybrid predictor: one can always achieve a better accuracy by increasing the number of classes one is willing to consider. The CP should be invoked only when SVM has poor confidence in classifying a sample. Thereafter, a range of confidence scores can be provided to the CP depending on how big a prediction set one is willing to compute.

4.7 Limitations and Threat to Validity

As DroidScribe builds on CopperDroid's extracted behaviours, this work inherits its limitations with respect to the dataset's range of behaviours (i.e., a threat to external validity). Furthermore, as CopperDroid dynamically executes apps in an emulated environment, only one execution path is traversed per run. The limitation is partially addressed by CopperDroid, which uses a simple stimulation mechanism (see Section 3.5.2). While this quantitatively increase behaviours exhibited, dynamic code coverage is still an open challenge. Another possible threat to validity is evasive malware.

Split-personality malware, such as Dendroid, Android.HeHe, and BrainTest are capable of detecting emulated environments with values such as the IMEI, which is uniquely all zeros for a vanilla Android emulator (like CopperDroid), and with the use of timing attacks [48,77,97]. When this occurs, split-personality malware will only exhibit benign behaviours, avoiding detection, and possibly threatening the validity of our behavioural profiles. For example, BrainTest was released on Google Play twice by bypassing its screening [48], but was not included in our dataset for testing.

Furthermore, our dataset did not include legitimate Android apps, which may have had a significant effect on our results as many Android malware samples are repackaged. However, that said, future work may discover that classification based on behaviours may be better suited for classifying malware families, or general malware detection, by condensing and separating many events into clearly defined benign and malicious actions. Another threat to internal validity that can be addressed with a larger dataset, is the accuracy of DroidScribe's performance. Instead of implementing cross-validation, samples can be instead split into adequately sized training, testing, and validation sets. Thus methods other than CV may be applied to further insure that there is no over-fitting.

As we are analysing a stream of system calls, our method is also vulnerable to mimicry attacks and, in some aspects, randomly added system calls and actions that change the flow and patterns of system calls. Traditionally, mimicry attacks tend to be tailored to decrease the precision of host-based anomaly intrusion detection by injecting spurious system calls. However, as we rely on behaviours, we are only considering subsets of system calls that cause actual changes of interest within the Android system. Furthermore, systems calls that are uninteresting or achieve nothing interesting are filtered out by this process. While our behaviours can also be subject to mimicry, it is at a higher behaviour level with noticeable (i.e., creating a random file) side effects. This is because injecting system calls that correspond to a random high-level behaviour is much more visible than injecting random sequences of uninteresting system calls.

4.8 Related Works

Although several of these studies were previously analysed in Chapter 2, some will be examined further in relation to the research in this chapter. This differs from Chapter 2, which analyses each work as a piece of the larger body of Android studies. In this section we describe work that is related to our classification approach. This includes both traditional malware classification (e.g., Internet malware), as well as Android malware.

Sandbox-based behaviours extraction has, in the past, been successfully applied to detect and classify traditional malware such as Internet malware [25, 122, 174]. Several of these studies have also opted to analyse the effect of system calls on the state of the system [25]. As of 2007, this was a departure from existing approaches, which primarily focused on using system calls for behaviour modelling, not analysing their effects.

The focus of [25] was then less on system calls themselves, and more on the objects altered as a result of the call. This included, but did not end at, modified files, network connections made, and changed Windows registry keys. These resulting system changes are not unlike our own recreated behaviours, although the approaches differ due to inherent differences between Android and Windows. The authors used normalized compression distances to capture the proximity of samples, and used the measurements for a hierarchical clustering algorithm. When evaluated against 3,698 malware samples collected over six months, their clustering correctly detected 91.6% of samples.

Another work on clustering Internet malware using SVMs was developed in [174]. Similarly, this approach considered manipulations to the filesystem, Windows registry, and mutexes as behaviours of consequence. While the behaviours are similar to ours in Section 4.4, the manner in which CopperDroid recreated these behaviours is significantly different. These features were then embedded into vector space using behaviour frequencies. To address polymorphism, each behaviour was represented as multiple strings of varying specificity: from complete to a coarser representation. The feature vectors are then used to train a SVM, which in turn was used to detect and classify malware. However, unlike our approach, conformal prediction was not used in complement. Using a corpus of 3,139 malware samples, unidentified by anti-viruses, the described approach correctly identified labels for 70% of the samples.

While related SVM-based methods successfully implemented labelled training set and hierarchical clustering on unlabelled samples, both approaches were prohibitively expensive on larger datasets. In [122], a more scalable clustering approach was developed to use locally sensitive hashing to cluster malware samples exhibiting similar behaviours. Similar to [25] and [174], programs were run in virtual environments and

behavioural profiles were extracted based on *OS objects* affected by system calls. Dependencies between system calls were then identified through taint tracking and combined to form features. Due to the expensive performance cost of taint-tracking, CopperDroid instead utilizes def-use chains for more detailed behaviour reconstruction. On a set of 2,658 samples, and only considering features seen in multiple classes, [122] produced 87 clusters with a precision of 98.4% and recall of 93%. More importantly, the algorithm was able to cluster a set of 75,692 samples in 2 hours and 18 minutes.

DroidMiner uses a two-level behavioural graph, extracted statically, to mine malicious Android behaviours [234]. In particular, intra- and inter-component flows were used to detect malicious flows between sensitive API calls, i.e. the nodes in the graph. All paths between sensitive nodes were then enumerated to identify, detect, and classify malware threats using the random forest classification technique. DroidMiner was evaluated with 2,466 malicious apps selected from third-party markets and over 10,000 apps from the official Android market. It achieved a 95.3% detection rate, with a 0.4% false positive rate. For the multi-class classification, it achieved an accuracy of 92%.

In 2014 an Android classification paper proposed using weighted contextual dependency graphs, constructed from statically extracted APIs, to recreate program semantics [9]. One graph database was built of malware, a second for benign. Graph-based feature vectors are then extracted from both datasets and used to train two separate classifiers. Weights were also introduced, to minimize similarities between malicious and benign graph-based vectors. The results were stored in a third, behavioural graph database, which was used to collect graphs modelling API usage per sample. By analysing the similarities between graphs, i.e. relationships between API calls, they achieved a 93% multi-classification accuracy and were able to also identify zero-day malware. This method was tested in two thousand malware and thirteen thousand benign application, running every three thousand apps in roughly three minutes. Like other static studies, it is weak against native code, HTML5, and embedded malicious code.

By using API family signatures instead of building API graphs, DroidLegacy [57] achieved accuracies of 94%, with 97% precision and 94% recall. Unlike our approach, DroidLegacy is a binary classifier used to determine whether the sample was malicious or benign. In order to generate signatures based only on polymorphic or renamed variants portions of the application, the APK is first partitioned into loosely coupled modules. All modules are then linearly analysed to identify modules that have a high likelihood of being a malicious module. Once suspicious modules have been identified, API signatures are generated and used to classify similar samples. As both [9, 57] statically extract APIs, they are unable to analyse native code or dynamically loaded code.

AppContext [235] combines many different features, including permissions, hardware, location, battery level, and user interface, for a binary classification. The basis for analysing these particular features is to detect malware utilizing evasive techniques such as suppressing malicious behaviours until a time where they have a higher chance of escaping notice. By analysing the context in which an app exhibits behaviours, AppContext has achieved 89.7% precision with an average of 647 seconds per app. Features are statically extracted and analysed for security-sensitive behaviours. In order to understand activation events, AppContext chains all inter-component communications, forming an extend call graph to infer the context, i.e. environment, in which interesting events are triggered. Features from two hundred malware and 633 apps from Google Play were implemented to detect the malware using DroidLegacy's SVM-based classification.

Drebin [16] is a lightweight classification method that uses broad static analysis to gather features from the Android manifest (e.g., permissions), and the decompiled APKs (e.g., APIs, used permissions, and network addresses). These features are then mapped to a vector space using boolean values, i.e. 1 if feature seen, 0 otherwise. Once created, Drebin uses machine learning to detect whether a given sample is malicious or benign using linear SVMs. Drebin passes it vectors for two classes as training data to determine a hyperplane that separates both classes with the maximal margin possible. Because of this, one of the two vectors passed to the SVMs must be from a benign sample, while the other must be from a malicious sample. Evaluated on over 100,000 benign samples and 5,000 malware samples, Drebin detects 94% of malware with a false positive rate around 1%. Performance wise, it takes 750ms per app.

A summary comparing our classifier to related studies can be found in Table 4.4. In general, the author's work on reducing long system call traces into a smaller set of behaviours has improved accuracy and runtime *together*, despite tackling the more challenging multi-class classification problem as opposed to binary classification.

Table 4.4: Comparison of related classification studies.

	OS	Classification Performance	Dynamic Behaviours*	Classification	Runtime
[122, 174]	Windows	98.5%, 70%	✓	multi-class	?, 1.6 min/app
DroidMiner	Android	92%	✗	binary	19.8 sec./app
[9]	Android	93%	✗	muli-class	1 min/app
DroidLegacy	Android	98%	✗	binary	?
AppContext	Android	93.2%	✗	binary	10 min/app
Drebin	Android	94%	✗	binary	0.75sec/app
DroidScribe^{T}	Android	95%	✓	muli-class	0.6 ms/app

(*Native code, dynamically loaded code, network traffic, T = early thesis version)

4.9 Summary

In this chapter we presented a method for classifying Android malware based on CopperDroid's behaviour reconstruction from raw system calls. We demonstrated that the level of details in these behaviours can be expressed in a diverse feature set, capable of accurately differentiating Android malware families. By modelling high-level malware behaviours in vector space and using SVM to classify 1,137 samples from the Android Malware Genome project, we were able to achieve accuracies of between 75% to 94.5% by increasing cut-offs. This includes the number of behaviours that a sample must exhibit in order to be considered input for our classifier. While the pure SVM aspect of the classifier does not, itself, offer something novel, it does illustrate the advantages of building SVM techniques on high-level behaviours instead of raw system calls.

We further evaluated the efficacy of our SVM classifier at different thresholds for samples with sparse behavioural profiles. In a significant departure from using conformal prediction as a pure classification mechanism, we then use conformal prediction as an evaluation framework for our SVM-based classification approach.

We demonstrated that the quality of the SVM classification in the presence of samples with sparse activity profiles decreases the confidence in the classification as the classifier cannot disambiguate across families for such samples. We then go showed how selectively predicting sets plausible choices using CP leads to an improvement in precision and recall for samples which were poorly classified during the training phase. Thus, we show how a set of probable classes lets us transcend the limits imposed by sparse activity profiles for analysis using sandboxes.

By using CopperDroid's analysis to generate behaviour profiles, our classification contributions satisfy thesis research Goals 1 and 4 as defined in Chapter 2. Specifically, these detailed profiles contain a wide range of low-level, high-level, and dynamic (e.g., network traffic, native code) actions. This makes it harder for malicious applications to hide themselves and their actions. Our hybrid classifier also satisfies our scalability goal (Goal 3) by using a novel feature set based on behaviours instead of raw system calls. Future work to improve accuracy, performance, etc. can be found in Chapter 6 and the distribution of work from this chapter is summarized again in Table 4.5.

Table 4.5: Distribution an contributions of Chapter 4 work.

	Behaviours	**Baseline**	**SVM**	**CP p-value**	**Hybrid CP algorithm**	**Statistics**
Novel	✓	✗	✗	✗	✓	✗
Author	✓	✗	✗	✗	✓	✓
Collaborator	✗	✓	✓	✓	✗	✗

Chapter 5

Supplemental Analysis Approach

Contents

5.1 Android Memory Image Forensics

While the work in Chapters 3 and 4 successfully filled a gap in the body of research, this work was not without limitations (see Sections 3.6 and 4.7). Considering these specific limitations, and several general limitations of existing solutions, the author explored the uses of Android memory forensics. While unlike the methods from previous chapters, the efforts in this chapter still stems from the same thesis goals stated in Chapter 2.

This work[1] can be considered an alternative solution or, more likely, a complementary to other solutions (e.g., CopperDroid). The content of this chapter demonstrates that memory forensic based methods can find evidence of malware other dynamic methods currently do not, or cannot. The author demonstrates this with the analysis fo two real-life Android malware examples, and shows how memory forensics can, theoretically, be applied to detect evasive malware (e.g., bootkits), as well as general malware current solutions and AV products cannot detect [130, 201].

Memory forensics has been a popular tool for traditional PCs (e.g., Windows, Linux) and is migrating to mobile devices. While it has been traditionally used to provide evidence for crimes by a human suspect, the author seeks to develop malware detection mechanisms based on the forensic analysis of Android memory image dumps. This is essential, as experts in mobile forensics are calling for more attention on mobile forensics, particularly on the growing number of Android devices [64].

The focus of this chapter is primarily detecting system vulnerabilities and privilege escalation, the foundation of most malware, but encompasses other behaviours such as accessing system files and leaking device data. We have found that a sufficient number of memory artefacts are available within Android memory for detecting, and sometimes identifying, Android malware. In addition to detecting successful malicious actions, we found memory forensics could detect several failed or dormant actions as well. Therefore, in cases where a sample may be dismissed as benign, by detecting unsuccessful malicious behaviours, the malware can be further examined for the correct triggering environment (e.g., different OS version). In combination with this chapter's component based stimulation, memory forensics could improve code coverage for solutions such as CopperDroid. Furthermore, memory artefacts can be used to improve classification accuracy by widening the range of features to include those specific to evasive malware.

To detect Android malware, the author discovered three main artefacts within memory (see Section 5.5): libraries, the Linux filesystem, and patterns in process IDs/UIDs. Other artefacts are available to aid in analysis, but these three seem the most universal

[1] All work performed solely by the author with portions electronically published in ESSOS DS 2015.

amongst the body of Android malware. Other forensic studies differ in their memory acquisition phases as well their analyses, as most tend to focus on forensic evidence instead of malware detection. In addition, our methods vary from previous Android malware detection studies, both static and dynamic, as they focus on different indicators. We are unaware of any other Android malware detection methods, prior to June 2015, based on memory forensics, as we discuss in the recent works section (see Section 5.8). In summary, the author's research contributions in this chapter are as follows:

1. We describe several memory artefacts discovered while manually analysing Android malware samples. Once identified, these artefacts were able to reliably detect other malware within our dataset. Furthermore, at least three of these artefacts have shown themselves to be strong, reliable, malware indicators, enough to detect newer and more popular malware outside our studied set, as we shall illustrate.

2. We theorize how memory artefacts and memory fingerprints can be used on a wider set of malware including more sophisticated, evasive, and futuristic ones.

3. We provide two, in-depth, analyses of well-known malware to illustrate how our memory artefacts can be applied to malware detection and/or identification.

Section 5.2 provides background information on memory forensics and the Android system. Section 5.3 describes in detail the malicious behaviours we encountered during our analysis. Section 5.4 discusses the design and implementation details of our analysis environment and tools, and Section 5.5 generalizes the memory artefacts we found and discusses memory fingerprints. We then provide two in-depth malware studies in Sections 5.6.1 and 5.6.2 to illustrate the effectiveness of using our discovered memory artefacts to detect, and even identify, the BaseBridge and DroidKungFu samples.

While the range of analysed samples and behaviours were somewhat limited in our initial experiments, the findings seem sufficient to extract essential behaviours (particularly those pertaining to untriggered, unsuccessful, or hidden behaviours) difficult, and sometimes impossible, to see otherwise. Future work for complete development of this method can be found in Chapter 6.

5.2 Relevant Background

Although we have discussed the Android architecture in previous chapters, here we shall add detail to the aspects that are essential to this chapter. For example, we previously discussed Android activities, services, and broadcast receivers, but here we shall expand on the subject as they are an integral part of this analysis.

5.2.1 Memory Forensics

Volatile memory analysis is an essential part of digital investigations due to the increasing dependency on smart devices. Furthermore, there exists digital evidence and malware that only reside within physical memory (RAM). There are several traditional PC worms that exhibit this trait, such as Code Red [204]. Additionally, arguably the first bootkit for Android was recently identified in 2014. This Trojan, known as Oldboot, lives in the boot partition of infected devices. Since this partition is only loaded as a read-only RAM disk, most existing antivirus solutions are not effective against it [130].

There are two methods for extracting memory for the purpose of volatile forensics; live response and memory image analysis. While live memory executes a live response tool to record specific volatile data at run-time, it can potentially alter the volatile environment and overwrite evidence. Alternatively, memory image analysis dumps the entire volatile memory image to a safe location for later analysis. For traditional computing devices, it has been proven that memory image analysis provides more robust and accurate results by minimizing forensic footprints and increasing coverage [7].

Objects of interest within Android memory include running/terminated processes, open files, network activity (partially analysed in this work), memory mappings, memory fingerprints, system remounting, and several more. While sometimes in persistent memory, such objects are difficult for AVs, with app-level permissions, to gain access.

In our experiments, we utilize an existing loadable kernel module (i.e., LiME [5]) to take memory snapshots of our Android emulators hosting a malicious application. We then extract the memory images and analyse them with Volatility tools [219]. By analysing several Android malware samples, we discovered memory artefacts that continuously helped detect malicious behaviours. Furthermore, if a sufficient number of artefacts is available, it was possible to identify the particular exploit or malware family.

5.2.2 The Android System Recap

Android applications are written in Java but can use native code, like C, with the use of the Java native interface. All app class components (i.e., services, activities, and broadcast receivers) are listed in the Android manifest (see Section 2.2.1) and compiled into a dex/odex file format, which is VM compatible, and stored in an APK. As each app is isolated by the modified Linux kernel running under the Android OS, the only means for interacting with other apps, or the system (e.g., sensors, ARM hardware), is through system calls or inter-process communications (IPC), which results in system calls. The extent of these interactions is limited by the Android permission system.

Permissions: To make use of protected device features permissions for those features must be granted to the app. Each APK contains an `AndroidManifest.xml` file that lists the requested permissions. See Figure 3.1 and Appendix D for manifest examples. Normally, at install time, these are shown to the user who may chooses to accept or deny them. However, malware can install payload apps while bypassing this check.

Therefore, as we shall discuss in this chapter, permissions are not a reliable indicator of how much damage a malicious application can do. As shown in the BaseBridge example (see Section 5.6.1), malware can create an even more over-privileged application and install it without the user's knowledge after it has exploited a system vulnerability.

Android App Components: Activities, services, and broadcast receivers are all activated by `intents`. As aforementioned, Android `Intents` are asynchronous messages exchanged between individual components to request an action, e.g. clicking on an app icon would correspond to an intent being sent to the app to trigger its main activity.

Unlike activities and services that receive `Intents` from another component, broadcast receivers are triggered by `Intents` sent out by a `sendBroadcast()` command (i.e., a system-wide signal). For example, if a BROADCAST_SMS is sent, an app with a broadcast receiver listening for that `Intent` can be triggered and attempt to view the SMS message content. This is essential, as 82% of all malware registered one or more broadcast receivers, while only 41.86% of benign apps did so [131]. In this chapter we also analyse Android components by analysing the app's Android Manifest.

Dalvik Virtual Machine: At the time of research, only Android versions running Dalvik were available. Therefore we could not fully evaluate these methods on the new ART runtime [218]. However, regardless of the extraction method, the memory artefacts discovered during this analysis should still function as reliable malware indicators.

Once installed, each app runs in its own VM with a unique combination of process ID and group ID. The maximum number of processes Android can handle at one time is defined by `RLIM_NPROC`. During run-time, all live process IDs are unique. However, apps can share user IDs if written by the same developer and signed with the same keys.

To shorten the time it takes to boot an app, the Android OS loads a Dalvik VM process — as it boots — that has been initialized with all the core Android libraries linked in. This process, called Zygote, listens on a socket and forks each time a new app is started. The new app (i.e., the forked process) shares all its linked libraries with Zygote until it attempts to write to it. In other words, when only reading, the libraries are still shared with Zygote, but if the app attempts to write to any shared memory pages, the pages are then copied to its own heap and labelled as "dirty pages".

Filesystem Access Control: Apart from the app permissions and components, the application's package name is also defined in the Android Manifest. By default, apps can only write and modify files in their own directory and need to be granted permissions in order to interact with any other part of the system (e.g., hardware, other apps).

As Android runs on a modified Linux kernel, the filesystem's discretionary access control (DAC) is the same as traditional Unix permissions. The purpose of this access control is to restrict the access of processes (i.e., user-level apps), based on its permissions and identity. For example, to store data, file permissions are by default `rw-rw---`. Because of this, apps installed with a unique UID/GID pairs and cannot read, write, or execute files outside their main directory unless made public. By default, the apps main directory is `/data/data/<app package name>/`, and can contain the following subdirectories:

- **shared_prefs** - app XML based shared preferences
- **database** - default location for sqlite databases
- **libs** - contains all native libraries of the app
- **files** - default directory for all app created files

System Partitions: During the system boot process, different parts of filesystem are mounted with different options. Therefore any malware with temporary root can remount partitions of the system to gain more permanent privileges. To gain temporary root privileges in the first place, malware can exploit vulnerabilities in the Android OS or kernel. This can be trivial for malware writers, as vulnerability exploitation methods are often put into handy root exploit files that can be easily executed by malware.

5.3 Specific Examples of Malware Behaviours

There are many malicious behaviours an app can perform in the Android system. Chapters 3 and 4 discussed many behaviour sets, whereas in this section we focus primarily on system vulnerabilities, as they are often the foundation of all malicious behaviours, and how these actions are performed. Malware samples for our analysis were chosen from the Contagio project, a public dump for mobile malware [52], in June 2015. As the database was relatively small (roughly 180), we were somewhat limited on samples to test. Furthermore, several were not installable APKs or did not run properly within our emulators (see Section 5.7). This chapter does focus on root exploits, as there was an abundance on Contagio, however analysis possibilities are not limited to such. Memory modification and access to information also leave analysable effects on memory.

5.3.1 Root Exploits

Of the samples that executed properly, most used root-level exploits to automatically exploit vulnerabilities within Android and its customized Linux kernel. By analysing the effects of these exploits, we discovered several memory artefacts which can be extended into general artefact-based policies to detect a wider range of malware.

Privilege escalation attacks occur on the kernel-level by exploiting vulnerabilities in the Linux kernel and/or core system libraries. Such attacks normally grant temporary privilege escalation, gained through missing input sanitation or other system bugs [98, 166]. Once root privileges are gained, however, they can be made permanent with system tools and resource that were previously secured away and inaccessible.

5.3.1.1 Missing Input Sanitation

Conceptually, root exploits take advantage of code that incorrectly validate their input, or do not perform any validation or sanitation at all. Such bugs have been found in the Android OS code as well as several distributions of the normal Linux kernel.

In this section, we provide a selective summary on popular system vulnerabilities, packaged in root exploits, to demonstrate how malware gain root access. After demonstrating the extent to which these vulnerabilities are abused, we shall demonstrate how memory forensics can detect these exploits. While most Android versions run on a modified 2.6.32 Linux kernel, some of the newer ones (e.g., Android 4.4) run on a modified 3.x Linux kernel. In this section we will describe, in-depth, several of these exploits and how they utilize missing input sanitation to gain root privileges. The targeted Android versions of several real-world malware families that use can be found in Table 5.1.

Exploid: In this exploit, the malicious app sends `udev` (i.e., a device manager for the Linux kernel) a "bad" message via the netlink interface (i.e., the generic netlink bus). Three files are required for this fake firmware installation; (1) the hotplug (contains path to an executable file, likely app generated), (2) an empty file (necessary for hotplug loading), and (3) a symlink to the hotplug under `/proc/sys/kernel/`.

This exploit first runs `/tmp/run`, and then creates a hotplug. This hotplug is then moved to the system directory `/proc/sys/kernel/`. With the netlink connection setup complete, the app sends keywords to add firmware along with the three files' paths. The `udev` process then starts adding the new firmware without validating the caller's permissions, and copies the contents of the hotplug into the data file, changing the binary to point to an exploit. Lastly, the hotplug is triggered.

Zergrush: A locally installed application can gain root privileges by passing the wrong number of arguments in the `argv` parameter to the method `dispatchCommand()` of the `FrameworkListener` interface, causing a buffer overflow within the library `libsysutils`. Luckily, patches were applied quickly. Since this exploit affects a smaller range of devices, "better" root exploits may be used by more malware.

Gingerbreak: Written by the exploid developers, Gingerbreak uses the same kind of exploit but on the `vold` process. `Vold` (volume daemon, or `Mountd` in Android 1.7) lives in the system directory and listens on the netlink socket for volume changing events and interacts with `MountService` (Java layer). However, when executing commands issued from the `MountService`, `Vold` does not adequately verify the call parameters.

Framaroot: This exploit exists due to a driver bug that affects devices from a specific manufacturer running a very specific processor. Any app on these devices would have access to the `/dev/exynosmem` device file, allowing them to map all physical RAM with read and write permissions. Framaroot then maps kernel memory and, with some other minor modifications, it can avoid the `kptr_restrict` kernel mitigation.

The exploit can then freely parse `/proc/kallsyms` to find the address of the `sys_setreuid` system call handler function and, if found, patches it to remove a permission check and execute a root shell. There are several variants of this bug, some of which bypass the patch made to prevent this attack, which are all used by Framaroot to affect a wider range of devices. As this exploit is relatively new and affects only a subset of devices, it may not be effective enough for malware to use.

Towelroot: For Linux kernels 3.14.5 and below, the `futex_requeue` function in `kernel/futex.c` does not ensure that calls have two different futex addresses. This lack in argument sanitation allows local users to gain privileges via a crafted `FUTEX_REQUEUE` command that facilitates unsafe waiter (`rt_waiter`) modifications. Discovered in June 2014, this is a powerful exploit but has yet to be used by known malware.

Table 5.1: Malware exploiting missing input sanitation.

Exploit	OS Target	Used by malware Family
Exploid	$\leq$ V2.2	zHash (limited number of affected devices)
Gingerbreak	V2.2-2.3.6	GingerMaster, Dgen, LeNa, RootSmart
Zergrush	V2.2-2.3.6	(limited number of affected devices)
Framaroot	V2.0-4.4.2	(2013, only affects some devices)
Towelroot	V2.0-4.4.2	(Released June 15th 2014)

5.3.1.2 Overflowing Limitations

The goal of these exploits is to overflow the supported number of process, defined by `RLIM_NPROC`, under a parent process that has the same UID as the shell user. When the limit of unique processes the Android system can support at one time has been exceeded, no new processes can be created by the Linux kernel, resulting in a privilege escalation.

For example, the debugging daemon `/sbin/adbd` is normally started in the context of the shell, and would be killed by its respective parent when forked until failure [98]. Then, as it is marked for autostart, `adbd` would be restarted by the system. Normally, it would start with root privileges and then change its UID to drop those privileges, but it cannot as the max number of processes has already been reached. Therefore, the change fails and `adbd` retains its root privileges.

Table 5.2: Malware overflowing limitations for exploit.

Exploit	**OS Target**	**Used by Malware Family**
Rageagainstthe-cage (RATC)	$\leq$ V2.2.1	Asroot, BaseBridge, DroidCoupom, DoidDelux, DroidDream, DroidKungFu
Zimperlich	$\leq$ V2.2.1	DroidKungFu, DroidDelux, DroidCoupon

RATC: This exploit forks the `adbd` process until the function `setuid()` fails and the root privileges `adbd` had when restarted will not be dropped. The process then continues executing with UID 0 (root) and can be used by the malware to access a shell with root privileges. Malware using this exploit can be found in Table5.2.

Zimperlich/Zysploit: This exploit is identical to rageagainstthecage (RATC) except it forks the Zygote process. As mentioned previously, the Zygote process is forked each time an application is started in order to quickly provide a VM to isolate the new process.

While the exploits in Table 5.2 primarily target vulnerabilities in older Android versions (i.e., below v 3.x), there are still vulnerabilities in newer Android versions due to overflows. For example, although not yet neatly packaged in an executable root exploit, vulnerability CVE-2015-1474 [59], has the maximum common vulnerability score of 10. This vulnerability was discovered in mid 2015, affects all Android versions before Android 5.1, and can gain privileges, or cause denials of service, via multiple integer overflows in the `GraphicBuffer`. Despite the novelty of the new attack, also known as Stagefright, this vulnerability fits within the limit overflow category and, like the exploits before it, can be detected by analysing the memory.

Table 5.3: Malware exploiting ashmem memory.

Exploit	OS Target	Used by Malware Family
killinginthenameof	$\leq$ V2.2.2	DroidDream, BaseBridge, DroidKungFu
psneuter	$\leq$ V3.1	DroidDream, BaseBridge, DroidKungFu

5.3.1.3 Remapping or Restricting Memory

Some exploits can change Android global settings by remapping the ashmem (Android shared memory) area to gain a system shell. The ashmem area is owned by the init process and holds references to the shared memory areas and the system attributes.

An ashmem exploit parses files such as /proc/sef/maps to locate the /dev/ashmem/system_properties area and attempts to remap it using the mprotect function as PROT_WRITE. If the edits are successful, future ashmem mappings would be prevented and fail, and the exploit will be able to locate and set ro.secure to 0. This determines whether to change its UID or to retain root privileges. After doing this, when adbd is restarted, all debugging shells will not drop their root privileges [98].

The other way ashmem can be exploited is by restricting access to it. Because the adbd process relies on the ability to read ro.secure, if it is unable to read the properties, it will erroneously retain root privileges under the assumption that ro.secure is 0. The exploit achieves this by utilizing the ANDROID_PROPERTY_WORKSPACE environment variable, which contains the size of the property area and remaps the memory again with mprotect. This sets the ashmem protection mask to 0, making it inaccessible. **Killinginthenameof** and **psneuter** use this exploit, as seen in Table 5.3.

5.3.2 Stealing User and Device Data

In a recent McAfee report, it was found that 82% of Android apps (35% of which were malware) and 100% of Android malware, track user location, device ID, and network usage when available. Furthermore, malware are eight times more likely to steal SIM card data (e.g., IMSI) than benign apps, and fives times more likely to gain device data as well (e.g., phone number) [145]. Such data can be leaked by malware, so additional artefacts are useful for separating malicious actions from legit or borderline transactions.

For example, in an analysis of a mSpy malware sample [4], the IMEI can be found leaked into a file in the malware's shared_prefs directory. While this was discovered using taint analysis, we shall demonstrate how it be found with memory forensics, and without the performance cost of tracking the data's full path. As these analyses were performed in an industrial malware lab, by default all network connections were blocked for security purposes. Thus, we could not analyse data leakage through the network. However, such analyses have previously been shown to be possible in [134].

5.4 Design and Implementation

To acquire volatile Android memory we configured and compiled our own Android kernels (i.e., goldfish) and a kernel module to capture memory. In our experiments, we utilized an existing loadable kernel module known as LiME to take memory snapshots of Android emulators running malware [5]. We then extract the memory images and analyse them with Volatility [219]. Details on our framework can be found below.

While these tools have been used previously to find data within memory (i.e., stored passwords), as far as we are aware, this is the first attempt to use these tools for Android malware detection. Furthermore, as we will discuss in Section 5.8, several of these works were tied to one OS version, unlike the wider range available in our analyses.

By manually analysing malware samples we discovered memory artefacts that can be used to reliably detected malicious behaviours (see Section 5.5). Furthermore, when a sufficient number of artefacts was present, it was possible to identify the particular exploit or system vulnerability targeted. During our experiments, each malware was installed into emulators running different Android versions, stimulated if necessary, and then LiME would be loaded to capture the memory. Once the LiME module had created the memory dump, we copied the dump from the emulator's sdcard to a Linux desktop and used Volatility to analyse it with several of its modified Linux plugins.

While aspects of our analysis environment is similar to previous works (e.g., uses LiME), several key steps and overall implementation differ (e.g., how LiME is loaded) [5,134]. Furthermore, our use of Volatility is unique as it focuses on detecting malicious Android apps actions. While LiME did not work with ART in 2015, future versions, or alternative methods, can still acquire artefacts for this chapter's analysis methods. Such a tool must be trustworthy and anticipate anti-analyses attacks, e.g. scheduling attack.

5.4.1 Android SDK and Kernel

Our primary component for extracting memory images from Android is the loadable kernel module (LKM) known as LiME (previously DMD) [5, 198]. However, module verification presents a challenge. If enabled, and loading a kernel module such as LiME, the kernel will perform several sanity checks to ensure that the LKM was compiled for the specific version of the running kernel. While module verification is optional, every kernel previously checked by ourselves and the authors of [198] determined that it was enabled for all Android kernels, making it impossible to load LiME on the vanilla kernel.

Previous works believed rooting or unlocking the boot loader were the only two options to overcome this problem, and both opted to root the device using various root

Table 5.4: Android version and its re-compiled, LKM enabled, kernel.

Android Version	Linux* 2.6.29 x86 ARM	Linux* 2.6.29 armv7 ARM	Linux* 3.10 armv7 ARM
Cupcake (1.5)	✓		
Donut (1.6)	✓		
Eclair (2.0-2.1)	✓		
Froyo (2.2-2.2.3)	✓		
Gingerbread (2.3-2.3.7)	✓	✓	
Honeycomb (3.0-3.2.6)	✓	✓	
IceCreamSandwich (4.0-4.0.4)		✓	
JellyBean (4.1-4.3.1)		✓	
KitKat (4.4-4.4.4)			✓

*The Linux kernel version that the Android Goldfish kernel version is built on.

exploits. However, there are valid concerns regarding privilege escalation on these devices, which motivated us towards alternative methods. In our approach, we re-compiled the Android kernel to enable LKMs, which allowed us to load LiME without rooting the device. We believe this minimal change to Android allows for more accurate data acquisition and increases portability across the range of Android versions.

In contrast, if we had used the rooting approach, we would have needed to acquire several root exploits for our various emulators. As we were running at least one emulator per Android OS version, this approach seemed less compelling. Furthermore, several exploits could not be executed on emulators, as they required a real physical reboot, and we did not want to run malware on an already compromised system.

Three kernels were recompiled for two different CPUs, and two different kernel versions as it was upgraded for Android 4.1 (i.e., Goldfish 2.6.29 to 3.10, see Table 5.4). While enabling kernel module loading may be dangerous, it is arguably less dangerous than rooting the device. Furthermore, it is trivial to list all modules loaded at run-time. This includes the memory acquiring LiME module (see Figure 5.1). In the future, it is possible to deploy virtual machine introspection (VMI) to extract volatile memory, or just the memory artefacts we are interested in, to lower overhead (see Chapter 6).

5.4.2 LiME

In order to load LiME into the emulator kernel it must be cross-compiled with the kernel it is to run with. Hence, we cross-compiled three modules, one for each of our LKM enabled kernels. When loading a compiled module, a memory image is created, capturing data on processes, open files, etc. By LiME's design, these memory images can either be written to a sdcard or dumped via TCP to a host computer [198].

While previous studies used TCP [134], we choose to write to a FAT32 disk image that can be loaded into our emulators as a virtual sdcard. The reason for this was twofold:

(1) it made sure no network buffers could be overwritten when enabled in future work, and (2) the virtual sdcard can be read without physically removing it from a device.

In some physical Android devices, the sdcard is either under or obstructed by the phone's battery, making it impossible to remove without powering off the device and losing volatile memory. This would be a drawback if using real devices. Our method ensures that we can have a sdcard with sufficient space, network buffers are not rewritten, and we can extract the memory without losing volatile memory. Furthermore, by using emulators, root exploits are not needed to install the modified kernels. Once we have acquired the memory image we can begin analysing it using Volatility tools.

5.4.3 Volatility

The Volatility framework primarily supports memory dump analysis for major Windows versions. However, Volatility now also supports Linux memory dumps in LiME format and includes roughly 35 plugins for analysing Linux kernel versions 2.6.11–3.16 [219]. In order to pass information on the Android kernel's data structures and debugging symbols to Volatility, one profile per kernel must be created. Essentially, this requires zipping information on the goldfish (Android kernel) version and the ARM version by using Volatility's makefile and the dwarfdump tool [219] (see Table 5.4).

Once Volatility has an understanding of the memory image structure, the modified Linux plugins can be used to analyse the memory image. Previously, we had mentioned the ability to view loaded kernel modules (e.g., LiME). This is important, as loading a custom module (not signed properly), is dangerous and could signify the presence of malware (in LiME's case, we can assume it is benign). We can see the module loading behaviour (i.e., `insmod`) by running the Volatility plugin `linux_pstree` on any memory image gained with LiME, as shown in Figure 5.1. Furthermore, by using `linux_psaux` on the `insmod` PID (1737), we can see the full command to load LiME. This is useful to see the full effects of a command (e.g., dump name and location).

```
python vol.py --profile=LinuxGoldfish-2_6_29ARM -f lime.dmp
   ↪ linux_pstree
Name Pid Uid
.adbd 45 0
..sh 1736 0
...insmod 1737 0

python vol.py --profile=LinuxGoldfish-2_6_29ARM -f lime.dmp
   ↪ linux_psaux -p 1737
Pid Uid Gid Arguments
insmod /sdcard/lime.ko path=/sdcard/lime.dmp format=lime
```

Figure 5.1: Using Volatility to view loaded kernel modules such as LiME.

5.4.4 Stimuli and Modifications

As already described in Sections 2.2.1 and 5.2.2, there are several app components for different application actions. An **activity** is a component that provides a screen users can interact with in order perform an action (e.g., dial a number or take a photo).

A **service**, on the other hand, performs long-running operations in the background and does not provide a user interface, making it useful for malicious behaviours. **Broadcasts** receive intents sent by `sendBroadcast()` and reacts accordingly to the received input. `Intents` can be sent to the sending app, other apps, or system wide broadcasts such as SMS received or low battery (see Figure 3.2 page 62).

If an activity, service, or receiver has been declared in the Manifest, but has not been triggered at runtime for whatever reason, we can manually start it by using the command `adb shell am` to trigger the following components in an Android emulator.

```
(1) broadcast -a ACTION -n <pkg_name>/.service.ServiceName
(2) start -a Main -n <pkg_name>/.Activity
(3) startservice -n <pkg_name>/.ServiceName
```

Other actions (i.e., `-a ACTION`), besides `Main`, are listed under `android.intent.action` in the manifest. In the in-depth malware analyses in Section 5.6, we trigger several actions and services found in the malware's manifests, which can be found in Appendix D. This stimuli resulted in several useful memory artefacts.

Other stimuli we sent via telnet, a network protocol with interactive text-oriented communications. Using this we were also able to send SMS, phone calls, and geographic location stimuli to our emulators. While not capable of triggering all malicious behaviours, they enabled us to trigger a significant number of interesting events and their resulting for memory artefacts.

5.5 Memory Analysis

As a result of our manual analysis of malware, we discovered three primary memory artefacts that could be utilised to detect many interesting, and often malicious, behaviours. Furthermore, these artefacts were found in all our analysed malware and could be generalized into encompassing policies to detect a larger range of malware.

Although more than three memory artefacts were discovered, these seem the most useful for future work on a larger scale study of more diverse malware. Furthermore we discuss, theoretically, how footprints in memory can be used to generate artefacts encompassing the effects of evasive malware and evasive malware behaviours.

5.5.1 Libraries Artefacts

Android libraries can be written in Java or with native code and interpreted with JNI. Popular libraries, especially ad-libraries, are often utilized by privacy-invading apps, with 82% of apps tracking users and 80% collecting location data. Furthermore, the most popular of these libraries are also extremely popular with malware authors [145]. Hence, understanding library usage is essential for analysing malware behaviours.

To detect library usage we used two methods. While one scans for specific libraries, via Volatility's `yarascan`, across all processes, the other method uses Volatility's `linux_proc_maps` to list all libraries used by one, specific, app process of interest. When combined, we can determine whether an app is using a dangerous, vulnerable, or malicious library. For example, the Zergrush exploit uses a vulnerability in the `system/lib/libsysutils.so` library, which can be scanned for with `yarascan` or listed with `linux_proc_maps`. Other malware families, including BaseBridge (Section 5.6.1), create and use the library `androidterm.so` maliciously.

More currently, in 2015, it was found that a vulnerable media library allowed hackers to access devices remotely without the user's knowledge. This major flaw affects roughly 95% of Android devices running versions 2.2 to 5.1, showing that library analysis is effective with new malware as well [172]. Moreover, it is possible to see within memory whether the library was app generated from a file stored within the app's APK.

ALGORITHM 5.1: Find library artefacts within Android memory images.

Data: Memory Image, benign app white list, malicious/dangerous libraries black list
Result: List and dump of used libraries and library functions

```
for each process do
    if non-app process OR benign processes then
        // anything not a child of zygote or not in white list of benign system apps
        Filter out;
    else
        if directory → /system/lib OR /data/data/*/lib/ then
            Check blacklist for names;
            // (e.g., libsysutils.so, androidterm.so)
            Extract library for further analysis;
            // (e.g., static analysis)
            if library is used (use yarastring) then
                extract function used;
            end
        end
    end
end
```

By scanning the memory image, all found library names can be compared to a white, or black, list or the library itself can be extracted from memory for analysis. This can be achieved via `adb pull` or by dumping the memory of the library with the Volatility plugin `linux_dump_map`. To do this, the start address space for the library can be found with `linux_proc_maps`. This may be essential as apps can easily obfuscate the written library's name to deter detection. We see this in one of our in-depth malware analyses, where a well known exploit was given a misguiding name.

While the directory for system libraries is consistent for each device (e.g., `system/lib`), app library directories differ per process (see Section 5.2.2). Therefore, a wildcard can be used instead of the package name to look for written app libraries while analysing an app process. In Algorithm 5.1, we illustrate the methods we used to manually analyse library artefacts in malware. In the future, we plan to automate this for malware detection. While somewhat simplistic, this could serve as an effective, low-cost, and portable solution, ideal for quick, undetailed, on-device malware detection.

5.5.2 Linux Filesystem Artefacts

Malicious behaviours can also be detected by forensically scanning memory images for specific directories. By detecting violations of the filesystem permissions (i.e., unsecure directory access) we can reveal behaviours such as modifying system configuration files, or the creation and use of executables, such as root exploits. While the number of root exploits has decreased recently [145], being able to detect exploitations at the Android system and kernel level encompasses a larger set of malware than just root exploits.

Table 5.5: Malware misusing system and kernel directories.

Exploit	Directory	Usage
Exploid	/proc/sys/kernel	moves hotplug to here
psneuter	/proc/self, /dev/ashmem	Exploits shared memory
Framaroot	/proc/kallysms	modifies string
BaseBridge A	/system/app	installs app in system

In the Linux security system there are separate directories for the system and kernel, which are inaccessible to user-level apps. By detecting incorrect file and directory access, we can detect malware executing files in harmful locations, attempting to remount the system, and much more. Just a few examples can be found in Table 5.5. For example, within the memory image we can detect the execution of an APK created by an app (e.g., an embedded app) after it has been moved to the `/system/app/` directory. Malware families such as BaseBridge do this to install malicious apps without the user's consent

and automatically granting it every permission it asked for. In the following paragraph, we describe several important directories to scan for and monitor.

The **proc** filesystem is a pseudo filesystem which provides an interface to kernel data structures. This filesystem is commonly mounted at `/proc`, and while most of it is read-only, some files allow kernel variables to be changed. Conversely, **etc** contains configuration files (e.g., network settings). This gives a malware a lot of enabling power over its environment. Although apps can access their own directory, other directories under **data** should be off limits. This includes other application directories, `data/media`, and `data/app`. Another useful memory artefact for malware detection is looking for processes that copy items from their directory to the system (e.g., the directory **system**), and the malicious use of process executables within **bin**. For example, in Figure 5.8 we can see the `bin/sh` shell being misused for a RATC exploit.

By scanning malicious process memory for partitions and directories, we have shown that system-level exploits and malware can be detected using memory forensics in a novel manner. This minimal yet infallible method is ideal for on-device binary detection as benign apps *cannot* access many of these restricted areas. Examples of analysing `system` and `proc` for a more fine-grained decisions can be found in Algorithm 5.2. While CopperDroid can analyse all paths of each behaviour for more detail, memory forensics can simply identify whether or not access control was compromised.

ALGORITHM 5.2: Filesystem artefacts within Android memory images.

Data: Memory Image
Result: Misuse of the Linux Filesystem

```
for each process do
    for found string (via yarascan) → system/app do
        if app file moved/executed here then
            Flag as dangerous & check for root exploit signs;
        end
    end
    for found string → proc/sys/kernel do
        if app hotplug created AND moved here then
            check for hotplug exploit;
        end
    end
    for found string → proc/kallysms OR proc/self do
        if app has root shell child processes then
            check if ashmem was exploited
        end
    end
end
```

5.5.3 Process Tree and ID Artefacts

Our third type of memory artefact is found while analysing patterns in the process tree. This includes the relationships between parent and child processes as well as their IDs and UIDs. In general, such artefacts are sufficient for quick malware detection as benign apps do not generate these artefacts. Moreover, occasionally, there is sufficient forensic evidence to identify specific exploits. For example, the RATC exploit yields a unique effect on the process tree, which can be used to both detect and identify the exploit.

It is important to note that, while the RATC exploit has a very particular pattern of symptoms that may not be seen in more current malware, there are still several general artefacts that will work with other known malware and future malware. General process-based artefacts for simple malware detection can be provided by identifying malicious apps with multiple processes, where all but the first have root UID, and/or malicious app processes with root shell child processes. Again, while simplistic, this novel dynamic approach of using memory forensics provides a clear division between malicious and benign apps in a way frameworks like CopperDroid do not.

When analysing the memory for process-related artefacts, regardless of the acquisition tool, malware can be detected if one or all of its processes have gained root UIDs against security protocols and whether any of an application's processes spawned root shell child processes. Another malware action of interest is detecting the very specific behaviour of malware creating and installing its own APK in the system.

To detect embedded apps being installed as system apps, one can first extract both executables and statically obtain their package names. This allows for easier process identification in memory dumps, as the process name contains the package name. While this is more definitive, it is possible to just compare "during" and "before" memory snapshots to find two new malware processes (i.e., only found in the "during" snapshot) and deduce that the "newer" of the two app processes has the higher PID and is, therefore, the process of the embedded app as it was installed after its carrier app.

As we shall see in the BaseBridge A analysis, one app can have several processes with the name format `<packagename>:<process name>`. These process names are declared in the manifest (see the full BaseBridge AndroidManifest in Appendix D) as the raw name for services, receivers, and activities. While we performed static analysis on the manifest to gain this information, it is possible to identify all "new" processes by comparing before and during memory images. In the BaseBridge sample, the first process of the malware (i.e., lower PID) creates and executes the root exploit RATC. And therefore the other newer BaseBridge processes (i.e., higher PID) had root UIDs as evidence of the exploit (see Figure 5.3 page 146).

Furthermore, for at least two of the root exploits analysed, there was a clear indication of privilege escalation found within the process tree. Specifically, it is easy to identify the rageagainstthecage exploit, as it forks the adbd process until there is a failure. To identify these, the malware process, or one of its processes, should have a PID over the `RLIM_NPROC` (i.e., 1024) due to all of the forking, and there should be a gap over the limit between the adbd processes and its child process (see Figure 5.3 page 146). See the BaseBridge A analysis (Section 5.6.1) for a more in-depth analysis on the rageagainstthecage (RATC) exploit and the resulting artefacts.

The algorithm we used to identify RATC can be found in Algorithm 5.3. While we used this manually in our initial experiments, future work will implement automatic tools to identify these artefacts as well as other process related artefacts discovered in the future. Unfortunately, continuous memory dumps are not currently possible with LiME or similar tools. While we developed a simple tool to assist in dumping and extracting memory images quickly and easily, in the future, virtual machine introspection may provide an easier method for analysing Android memory (see Chapter 6).

ALGORITHM 5.3: Process Artefacts within Android memory images.

Data: Memory image, package name, app component process names, app white list
Result: Malicious behaviours that use root

```
1  for each process do
2      if non-app process OR benign processes then
3          // anything not a child of zygote
4          // not in white list of benign system apps
5          Filter out;
6      else
7          if Process has root UID OR root shell child then
8              check status of system and analyze file executions;
9          end
10         if processes have shared package name then
11             check if any have root UID/shell;
12         end
13     end
14     if Process ID <1024 then
15         if sh child PID - adbd PID >1024 then
16             Rageagainstthecage exploit;
17         end
18         if sh child PID - zygote PID >1024 then
19             Zimperlich exploit;
20         end
21     end
22 end
```

5.5.4 Extras: Miscellaneous String

Searching for strings in the memory dump can be used to help search for libraries, directories, processes, and their usage. However, it can also be used to scan for other useful malware indicators. For example, in the BaseBridge analysis, it was possible to see configuration files being accessed by a user-space app (a security breach), when a specific library was being used, attempts to remount the system, and what Linux commands were used to move an executable to the `system/app` directory. More details and applications of the Volatiliy `yarascan` tool can be found in Section 5.6.

Many of the useful strings found in our analyses of volatile memory from infected emulators are popular indicators of malware. In one recent static study [184], the authors analysed a much larger, and more recent, body of malware and their results showed a very similar set of strings and commands being associated with malware and privilege escalation (see Table 5.6). However, as this study was performed statically, there is little detail on the conditions on when and how these string commands were used.

It is highly likely that, as new exploits are discovered [59, 117], more strings associated with malware, both generically and family specific, will be found. While additional string-based artefacts will require the most tailoring (e.g., no generic algorithm that encompasses all cases), detecting strings like `cp`, `mount`, directories, HTML strings (see Section 5.6.2), and `.conf` files are reliable memory artefacts for more general, large-scale, malware detection. Any other additions will be primarily to assist with identifying specific behaviours or malware families, or adjusting to changes within Android. This seems unlikely to occur frequently, given our analysed architectural layer.

Table 5.6: Common strings and commands in malware [184].

Command	Description
chmod	Changes permission
insmod	Load LKM
su	Change user to superuser
mount	Attacha filesystem
sh	Invoke default shell
chown	Change file/directory owner
killall	Kill all processes
reboot	Reboot system
hosts	Find IP addr of given domain name
getprop	Retrieve the available system property
mkdir	Make directory
ln	Create link to file/directory
mount -o remount	Make a read only filesystem writeable
ps	Report process status

5.5.5 Theory: Memory Fingerprints

Previously in this section, we showed the three most popular memory artefacts we found to be associated with Android malware. Furthermore, as most current malware still stay within their processes, the majority of artefacts are found in user-process memory. However, there is a growing number of evasive malware. Therefore, we theorize that non-human readable memory artefacts in *any* chunk of memory (e.g., stolen data, library names, process ID/UID) can be fingerprinted (i.e., create signatures) to detect evasive malware. Several of these internal memory partitions have already been listed in Section 5.5.2, such as `boot`, `system`, `recovery`, `data`, `cache`, and `misc`. In addition, Android devices may have SD card partitions `sdcard` and/or `sd-ext`, although it depends on if, and which, custom ROM (data written to Read-Only-Memory) is used.

As stated several times, the Android malware Oldboot [130] reinstalls itself each time the device boots and hides by modifying the boot partition. This makes Oldboot very difficult to detect, but must leave memory artefacts within the boot partition. While not available (on Contagio) in 2015 for testing, as Oldboot forcibly writes malicious files into the boot partition, these anomalous files must be detectable with the proper fingerprinting algorithms (e.g., hashes) [34]. Furthermore, by detecting any change to a healthy boot partition, actions by polymorphic malware should still be detectable. While Oldboot is an exceptionally elusive sample of Android malware, as malware grow in sophistication, developing partition fingerprinting for Android now could detect malware in the future. For more granular, less binary, classification, artefacts and fingerprints can be used to supplement other solutions (e.g., CopperDroid behaviour profiles).

Perhaps a more relevant use of memory fingerprints is detecting memory attacks. While CopperDroid and other dynamic frameworks can normally recreate details of attack *outcomes*, memory forensics may be able to provide more details on the attack itself. As shown in Section 5.3.1, Android malware already has a history of overflowing limitations and exploiting memory, and is still doing so today. Fingerprinting memory regions could improve the detection of memory corruption, e.g. buffer overflows, uninitialized memory, and memory leaks. This has been explored for Linux [165], but while there are tools to help developers check for memory leaks in Android (e.g., TraceView [60]), we do not know of any studies using the information to help detect malware.

From our previous analyses of Android malware for memory artefacts, we are certain that even evasive malware result in forensic evidence. By further extrapolating our findings, we firmly believe that fingerprinting could help detect malware, particularly deceptive ones. This is essential as malware grow more sophisticated, but we could not test our theories at the time of research as we did not have access to such malware.

5.6 Case Studies

In this section we analyse two popular Android malware families manually and in-depth. In both cases, we examine several triggers and their resulting malicious behaviours. Then, we demonstrate what memory artefacts can be found to detect malicious Android apps. Thus, by using the three core artefacts mentioned in Section 5.5, we demonstrate the capabilities of using memory forensics to detect many, real, Android malware.

For each experiment we cloned a clean emulator, took one memory image snapshot, deployed the malware, interacted and stimulated the malware, and then took a second memory image snapshot before removing the cloned emulator (see Figure 5.2).

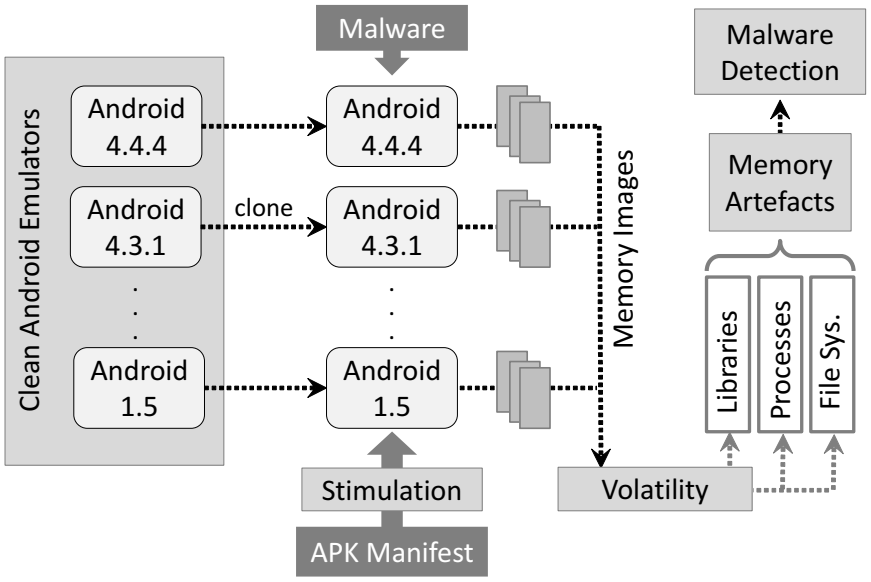


Figure 5.2: Framework for analysing Android malware with memory forensics.

5.6.1 BaseBridge A

BaseBridge is a Trojan horse that attempts to send premium-rate SMS/calls to predetermined numbers. This malware can be installed through drive-by-downloads or intentionally because of the app's advertised functionality. In this case, the sample poses as a radiation measuring app that reports the radiation level of the user's geological location. However, once installed, it attempts to obtain root privileges and, if successful, it installs an executable which can communicate with a control server using HTTP protocols and send it stolen information (e.g., subscriber ID, manufacturer and model of the device, Android OS version). This Trojan also periodically connects to the control server and may perform the following actions: send SMS messages, remove SMS messages from its inbox, monitor phone usage, and dial phone numbers "in the background" [1, 159].

5.6.1.1 Triggers

For this sample we used both Android OS Gingerbread (2.3.7) and Doughnut (1.6), as Doughnut was too old to send essential telnet stimuli to exercise the malware, and Gingerbread encouraged a slightly different set of behaviours than other, older, OS versions. For exercising Android apps, one of their unique properties is their multiple entry points (e.g., besides tapping on the app icon). For example, we found that a system wide geolocation signal triggers the `GPSService` service in this particular malware sample. We were also able to confirm this by decompiling and analysing the code.

Telnet: When deploying our BaseBridge sample, the first screen displays "Waiting for position" in Chinese. By analysing BaseBridge's manifest (Figure D.1), it is then clear that BaseBridge was granted several permissions for accessing location (e.g., `ACCESS_FINE_LOCATION`) at installation. Despite running the malware in different clean emulators for a period of five minutes no behaviours were exhibited without further stimuli.

By fixing our Gingerbread emulator's geographical (geo) location with telnet, we could then successfully transition to the next screen by triggering the `GPSService`. Sending different coordinates varied the output insignificantly (radiation levels changed).

However, as this was the extent of activities to explore via user input, we turned our attention to background behaviours (e.g., services) as their behaviours are easier to hide from the user. While it is highly likely that allowing Internet access will trigger the malicious services naturally, especially if it contacts its control server, it is safer not to connect the malware to the real world. This satisfies one of the rules of forensics, which is to always isolate the device before analysis [198]. Next, we analyse BaseBridge's Android Manifest (see Appendix D) for additional components to trigger.

Components: Within the manifest we saw two other services, excluding the `GPSService`, labelled `Mrun` and `SysM`. By taking several, sequential memory images, we were able to determine that `SysM` first creates the RATC exploit under its own files directory, and `Mrun` creates an app called `SMSApp.apk` and an executable called `busybox`.

These files were originally stored in `resources.arsc` (e.g., within the APK zip) and were copied into the app's directory when the services were triggered. While these files can be extracted for further analysis with the SDK, the file may also be dumped from the memory image. Furthermore, we discovered an activity called `BalckActivity`, which the malware can invoke to hide outgoing phone calls. Specifically, when triggered, this activity effectively covered the calling screen with a completely black screen to hide the fact that a call had been placed.

5.6.1.2 Malicious Behaviours

The first file of interest is the rageagainstthecage exploit which repeatedly, and successfully, gained root access on our emulators. This gave the BaseBridge malware the ability to install `SMSApp.apk` and use `busybox` as root, bypassing the permissions check and accessing directories and files against original DAC permissions.

Within an application each component has a unique process name, as seen in the manifest. By using these names, we can see both the effects of the exploit on the `adbd` process, as it is forked until failure, and the effects on the PIDs/UIDs themselves. The `linux_pstree` Volatility plugin effectively displays these effects, as can be seen in Figure 5.3. Note the high PID of the `adbd`'s shell child and the root UID of processes `com.keji.unclear:two` and `:remote`. These processes gained root UIDs after the exploit was executed by `com.keji.unclear`.

The memory image used for Figure 5.3 was taken both after the `SysM` service (i.e., PID 510) ran the RATC exploit (confirmed by decompiling the APK and analysing `SysM.class`), and right after the service `Mrun` spawned process `com.keji.unclear:two`. The effects of the RATC can be seen as the process `adbd` (PID 45) possesses a child root shell which was spawned as a result of the exploit. This shell is then exploited by the malware processes `:remote` and `:two` (PIDs 1618 and 1800),

```
Name                 Pid   Uid
init                   1     0
.sh                   32     0
.servicemanager       33     1000
.vold                 34     0
.netd                 35     0
.debuggerd            36     0
.rild                 37     1001
.zygote               38     0
..system_server       88     1000
..om.keji.unclear    510     10036
...unclear:remote   1618     0
..eji.unclear:two   1800     0
...sh               1806     0
...sh               1854     0
..i.unclear:three   6767     10036
...sh               6774     10036
.adbd                 45     0
..sh                2184     0
...insmod           2185     0
..sh                2270     0
..[sh]              2216     1000
```

Figure 5.3: Simplified `pstree` showing effects of RATC exploit in volatile memory.

which use the shell to execute other files to undermine Android security. As these processes were spawned after a successful exploit (their PIDs exceed the RLIM_NPROC limit of 1024), they have root UIDs and can spawn root shell children of their own. And while RATC cannot be seen running with the pstree plugin, the information is stored within the memory image, as can be seen with the plugin linux_psxview. A snipit of psxview output can be seen below in Figure 5.4.

```
Offset(V) Name PID/pslist/pid_hash/kmem_cache
---------- ----------- ------ ------ --------
0xc5970c00 om.keji.unclear 510 True True False
0xc6606000 .unclear:remote 1618 True True False
0xc697cc00 eji.unclear:two 1800 True True False
0xc6406400 rageagainstthec 610 False False True
0xc56c6c00 rageagainstthec 625 False False True
```

Figure 5.4: Simplified psxview showing RATC exploit in volatile memory.

Whenever the RATC exploit is successful it automatically triggers the BaseBridge Mrun service (which we have also manually triggered, see Section 5.4.4), spawning a new malicious process. During this service, the malware generated files SMSApp.apk and busybox are used heavily. After created by Mrun, these two are made executable by using the system shell provided by the exploit to change their permissions. We have verified and examined this further by using linux_psaux on PIDs 1806 and 1854 in Figure 5.3 (just like in Figure 5.1), and by statically analysing the dex classes.

When made executable, **BusyBox** can be used by an application to acquire additional, handy, Linux/Unix based commands. While not malicious in itself, busybox is dangerous because it provides kernel-level commands that are not normally available to user space apps. To determine whether busybox was misused by BaseBridge, and how, we use Volatility's yarascan plugin to locate commands that involve busybox in memory. Even if the file name had been obfuscated, symbolic labels can be attached to each app-generated file and used to detect its usage.

The figure below, Figure 5.5, shows a compilation of three simplified yarascan results when searching for busybox. For example, this output shows busybox using its root access to remount (see Table 5.6) the system as read/writeable. Once the system directories have been compromised, BaseBridge then uses busybox to copy SMSApp.apk to the /system/app directory, and install it in the background. As the output of each yarascan only gives a relatively small window of bytes per search match, we often dump and analyse the entire memory page holding the string's address, which is given with each yarascan output in the leftmost column.

```
Task: eji.unclear:two pid 1342 rule r1 addr 0x44fa2851
0x44fa2851 [HEX] data/data/com.ke
0x44fa2861 [HEX] ji.unclear/files
0x44fa2871 [HEX] /busybox.mount.-
0x44fa2881 [HEX] o.remount,rw./sy
0x44fa4459 [HEX] data/data/com.ke
0x44fa4469 [HEX] ji.unclear/files
0x44fa4479 [HEX] /busybox.cp.-rp.
0x44fa4489 [HEX] /data/data/com.ke
0x44fa449a [HEX] ji.unclear/files
0x44fa44aa [HEX] /SMSApp.apk./sys
0x44fa44ba [HEX] tem/app/........
```

Figure 5.5: Evidence of BusyBox compromising the system in volatile memory.

Other malicious busybox commands captured within the memory image are changing `/system/app/SMSApp`'s permissions to root (`busybox.chown 0`), and removing all exploit processes (`busybox.killall rageagainstthecage`). This is all possible since BaseBridge has access to a system shell after the exploit (`/system/bin/sh`) and uses it to make busybox executable (`chmod 777`), which we have reliably detected within several memory images and verified with static analysis.

We detect many of these same actions (i.e., file creation, file movement, and permission change to executable) when scanning the memory for **SMSApp.apk**. However, only when scanning for SMSApp do we see the execution of that file, installing the APK "invisibly" as a system app with all permissions automatically granted. No app icon is displayed in the apps menu, but SMSApp can still be seen running in the app manager under the misdirecting name of `com.android.battery`. By extracting the APK from memory and decompiling it, we see that many permissions were granted to SMSApp, including access the SMS/phone, network, and user data.

As the malware lab prevented network access, this provided an opportunity to analyse malware with failed network connections. This was particularity useful for analysing and detecting malware bots despite the removal of their control and command centres (C&C). This was the case with our BaseBridge sample, as the dropped SMSApp.apk primarily contained the functionality to communicate with a control server using HTTP protocols (confirmed via static analysis). Using memory forensics, we were able to detect a failed connection to `b3[.]8866[.]org` on port 8080, attempted by `com.android.battery`. If network connection could be established, it would be interesting to utilize Volatility's network plugins, such as `linux_sk_buff_cache`, to analyse network packets within the kernel memory. This should yield other interesting artefacts, as other works have successfully analysed mobile network with memory forensics [103, 134, 136]. However, these studies were focused on forensics, not malware.

5.6.1.3 BaseBridge Memory Artefacts

All the malicious BaseBridge behaviours that we have dynamically stimulated, detected, and discussed previously result in various memory artefacts. These artefacts primarily fall into the three categories of memory artefacts that we proposed in Section 5.5, and can be used to detect similar exploits and behaviours in other malware. Although useful for a broader spectrum of malware, the purpose of this section is to demonstrate how memory artefacts can be used to specifically detect samples of the BaseBridge family.

Libraries: By using the Volatility plugin `linux_proc_maps`, we were able to view all libraries used by a process and, arguably more importantly, whether the library was written by the app (as shown by the "minor" value in `linux_proc_maps` outputs). With BaseBridge, and several other malware families such as Anserver and DroidDream, the malware writes a `libandroidterm.so` to its directory. This is a potentially dangerous library as it is a terminal used to communicate with the built in Android shell. By also scanning the memory for names of libraries used and/or written by the malware, we can detect calls to specific functions within the library as well:

```
Task: i.unclear:three pid 321 rule r1 addr 0xbef8e3e6
0xbef8e3e6 [HEX] data/data/com.ke
0xbef8e3f6 [HEX] ji.unclear/lib/l
0xbef8e406 [HEX] ibandroidterm.so
0xbef8e416 [HEX] .0x44f91aa0.....
```

Figure 5.6: Library function usage as seen in memory.

Furthermore, we can dump the library memory page with the offset of the library function call (e.g., `0x44f91aa0` in Figure 5.6), which can be found with `linux_proc_maps`. While out of the scope of this work, this demonstrates the depth of detail memory forensics is capable of achieving in order to improve malware analysis and detection.

Linux Filesystem: By systematically scanning for specific directories and/or configuration files across the entire memory, we can see abnormal usage of the Linux filesystem. Furthermore, with knowledge of the security permissions in place, we can identify specific malicious behaviours. With our BaseBridge A sample there were several attempts to use various terminals such as `proc/diskstats.getty`. Not only is this abnormal, as no other app process attempts this, it should not have be possible for a user-level app. BaseBridge should also not have access to software for booting (e.g.,

```
T: eji.unclear:two pid 1342 rule r1 addr 0x232339
0x00232339 [HEX] etc/shells./bin/
0x00232349 [HEX] csh.............
T: eji.unclear:two pid 1342 rule r1 addr 0x23e1c5
0x0023e1c5 [HEX] proc/diskstats.g
0x0023e1d5 [HEX] etty...profile.p
0x0023e1e5 [HEX] rocess.=.\%s....v
0x0023e1f5 [HEX] ersion.=.0.8...t
T: eji.unclear:two pid 1342 rule r1 addr 0x232e71
0x00232e71 [HEX] etc/resolv.conf.
0x00232e81 [HEX] ...nameserver..d
0x00232e91 [HEX] omain..options.a
0x00232ea1 [HEX] ttempts.......5.
T: eji.unclear:two pid 1342 rule r1 addr 0x236d61
0x00236d61 [HEX] etc/bootchartd.c
0x00236d71 [HEX] onf.............
T: eji.unclear:two pid 1342 rule r1 addr 0x236f49
0x00236f49 [HEX] etc/inittab....u
0x00236f59 [HEX] mount.-a.-r....s
0x00236f69 [HEX] wapoff.-a../dev/
0x00236f79 [HEX] tty2.../dev/tty3
T: eji.unclear:two pid 1342 rule r1 addr 0x23941d
0x0023941d [HEX] etc/services...\#
0x00239545 [HEX] etc/ethers.\%x:\%x
```

Figure 5.7: Several string search results for interesting system directories and files.

bootchartd.conf, etc/initab), networking (e.g, resolv.conf, etc/services), or shells (e.g, etc/shells, proc/diskstats.getty, bin/sh), but does after the RATC exploit.

The dangerous behaviours revealed in Figure 5.7 show a high likelihood of malware, as user-level apps are not designed to have access to the listed directories and files. For example, malware can modify the `bootchartd` configuration file to automatically start the malware at boot time. The shells `etc/shells` and `/bin/csh` are also dangerous tools as they can be used for executing files, such as SMSApp.apk. And while malware detection through non-secure usage of the filesystem is useful, like CopperDroid, we may wish to further analyse specific uses. For example, by dumping the memory map of the following `yarascan` result, we may be able to see the point at which RATC gains root access; when forking adbd fails. This is highly probable as the content in Figure 5.8 has only been found in memory after running the exploit.

```
Task: adbd pid 45 rule r1 addr 0x24068
0x00024068 [HEX] bin/sh..-.exec.'
0x00024078 [HEX] \%s'.failed:.\%s.(
0x00024088 [HEX] \%d).-.../proc/\%d
0x00024098 [HEX] /oom_adj....adb:
```

Figure 5.8: Memory evidence of successful RATC exploit.

Process Trees and IDs: In Figure 5.3 we supplied an extensive example of the effects of RATC on the process tree and IDs. Specifically, we could see a PID gap between `adbd` and its child process larger than the process limit of 1024 (i.e., typical Android `RLIM_NPROC` value). This demonstrates that this BaseBridge sample, and other similar malware, start by running the RATC exploit in a processes with a reasonable PID and a user-level UID. This resulted in the spawning and killing of 1024 processes, and all subsequent malicious services, broadcasts, and actions having PIDs over `RLIM_NPROC`. Furthermore, these post-exploit processes often had UIDs of zero (i.e., root).

5.6.2 DroidKungFu A

Similar to BaseBridge, the DroidKungFu families (A, B, C, and D) use exploits to gain root privileges and then install the main malware component. Across all versions of this family many exploits have been used (see Section 5.3.1). In this particular DroidKungFu sample, it uses the exploid exploit. Once installed, the main DroidKungFu malware has backdoor capabilities that enable it to execute system-affecting commands, delete files, download/install APKs, open URLs, and run APKs [159].

Moreover, this malware family is capable of obtaining device information and will likely send it to a remote server. This includes the IMEI number, build version release, SDK version, mobile phone number, phone model, network operator, net connectivity type, available sdcard memory, and available phone memory [159].

Like with our BaseBridge sample, we begin by manually exploring and triggering malware components. Using our analysis framework, we analyse the resulting memory artefacts, how and where they were found, and how we can detect DroidKungFu families *and* Android malware in general with these techniques.

5.6.2.1 Triggers

By systematically triggering each app component found in our sample's manifest manually and then killing the process and deleting any created files, we can roughly determine what each component was designed to do. After activating each component we noted the files created (see Table 5.7). Creating this "component to files" mapping helps reveals a great deal about the malware's capabilities and intentions, e.g. background behaviours.

For example, based on the exploit created by `OnlineActivity`, this component seems more malicious than an activity like `SongListActivity`. To detect phone data leakage more easily, and to help trigger malware despite VM-detection methods, we changed our emulators' IEMI number from all zeros to `123443213333333`.

Table 5.7: DroidKungFu files created by triggering single services and actives.

	InitOnline-Activity	Online-Activity	SongList-Activity	History-Detail	ZhiFu-BaoChong ZhiActivity	com.google.ssearch.SearchService*
lib/liblinphone.so	✓					
lib/libnative.so	✓					
databases/hl95_sipphone_history.db		✓		✓		
files/ linphonerc		✓				
files/oldphone_mono.wav		✓				
files/ringback.wav		✓				
shared_prefs/permissions.xml		✓				✓
gjsvro		✓				
system/app/com.google.ssearch.apk	?				?	
temp.apk					✓	

*All are activities except for SearchService, a service.

5.6.2.2 Malicious Behaviours

Similar to BaseBridge, DroidKungFu first creates and executes a root exploit. While it is possible to extract the exploit and analyse it statically, it is also possible to identify the well-known exploid exploit by analysing the memory image.

All the DroidKungFu malware families use either the exploid or RATC exploit, but occasionally both are created in case one fails. If the exploid exploit is a success, then the malware gains the ability to do extensive damage in the background, such as installing malicious apps and receiving files and/or commands from a remote server.

Exploid: In this exploit, labeled gjvso in our sample, a "bad" message is sent via the netlink interface. Depends on the exploit version, either the vulnerable device (`udev`) or the `init` process performs the action in the message without checking its privileges (exploit details in Section 5.3). Using Volatility we can see this attack occurring by searching for the strings `hotplug`, `gjsvo`, and/or the directory `proc/sys/kernel`.

As seen in Table 5.7, gjsvo is created by the activity component `OnlineActivity`, but it also created by two broadcast components that are not in the Table. These broadcasts listen for the `BOOT_COMPLETED` system signal and are named `org.linphone.NetworkManager` and `org.linphone.BootReceiver`. To discover this, we sent all possible system triggers belonging to the receivers listed in the manifest.

However, even if successful, the exploit only grants temporary root privileges. To have more permanent abilities, the malware can use its root privileges to install several

system apps. Even if the exploit is not successful, DroidKungFu can attempt to install its payload app using social engineering. Unfortunately, the two APKs that were expected to be installed were not during our analyses. This was probably due no network and/or some component not being triggered correctly. Further analysis is necessary, but enough evidence is available in the memory dumps to show the intent to install APKs outside its allowed directory, which is sufficient evidence for pure malware detection.

Payload APKs: DroidKungFu attempts to create and install temp.apk during the `ZhiFuBaoChongZhiActivity`. When the activity component is activated, the directory `cache` is created along with the file temp.apk. It is possible to acquire this empty file through the SDK, or by using Volatilitys `yarascan` plugin to search for the string `apk`, and then dump the memory map. While scanning the dump around the string `temp.apk`, phrases like "urlDownloadToFile" and "CHECKNETWORK_DIALOG" stand out. Furthermore, by submitting the same DroidKungFu sample to CopperDroid, which does enable networking (Chapter 3), we can see temp.apk being created under `data/data/com.aijiaoyou.android.sipphone/cache/temp.apk`.

Therefore, it seems that temp.apk must be downloaded, and that is why we never see a complete temp.apk during our analysis. However, we still have enough evidence of its attempted creation by using Volatility to search for the string name, and then dumping the memory map containing the results (as see in Figure 5.9).

```
T: ndroid.sipphone pid 10934 rule r1 a: 0x43eeac8b
0x43eeac8b  [HEX]   temp.apk..cacheP
0x43eeac9b  [HEX]   ath=..cacheDir..
0x43eeacab  [HEX]   CHECKNETWORK_DIA
0x43eeacbb  [HEX]   LOG..CHECKVERSIO
```

Figure 5.9: Memory evidence of an attempted but failed APK download.

The more dangerous APK DroidKungFu can install is com.google.ssearch.apk, a fake Google search app has malicious backdoor functionalities [111]. If DroidKungFu gains root privileges, it instantly installs this app in the `/system/app` directory (see Figure 5.10) and then mimics the authentic Google search app. For some reason, this APK was not generated, and it is possible that DroidKungFu will not create this app without network connection or without a specific trigger we have not tried.

For example, exploid works best on Android 2.2 and below. Therefore, com.google.ssearch.apk may only be created and installed if DroidKungFu is run on an older Android version (e.g., Froyo), as that environment might have the necessary vulnerabilities.

That being said, we are still able to detect attempts to install apps in the system by analysing memory dumps for commands to copy, or move, files to system directories. Despite a file never dropping, general malware detection can see the evidence of an APK waiting to be made, as shown in Figure 5.10, and flag the sample for future experiments.

```
T: ndroid.sipphone pid 10934 rule r1 a: 0x43ef06c7
0x43ef06c7  [HEX]   system/app/com.g
0x43ef06d7  [HEX]   oogle.ssearch.ap
0x43ef06e7  [HEX]   k../legacy..lega
0x43ef06f7  [HEX]   cy..copyAssets..
```

Figure 5.10: One `yarascan` example of fake Google search APK.

Stolen Device Data: `READ_PHONE_STATE` must be granted, allowing it to read data such as the phone's IMEI number. For the vanilla emulator, this number is all zeros. However, by modifying our emulator's IMEI to `123443213333333`, it became easier to detect the IMEI being stored or leaked.

When scanning for the IMEI string with `yarascan` across both clean and infected memory dumps, it is apparent that only the DroidKungFu is reading this data. We can also dump the memory map containing the IMEI to see what other information is being stored. For example, it is recorded that the device is an emulator. This is potentially useful to the attackers. By detecting that its malware/bot is running in an emulator, the controller can decide to remove the malware, or alter its behaviours.

If stolen device data is later misused (e.g., the emulator connects to the network and leaks data), scanning for strings such as the IMEI should be able to detect that behaviour as well (see Volatility output top of Figure 5.11). A more comprehensive view of the data found in the memory map containing the address `0x40551eb5`, as well as the IMEI number, can be found in the bottom half of Figure 5.11.

Figure 5.11: Finding leaked IMEI data in memory image.

```
T: ndroid.sipphone pid 10934 rule r1 a: 0x40551eb5
0x40551eb5  [HEX]   123443213333333&
0x40551ec5  [HEX]   ostype=2.3.3&osa
0x40551ed5  [HEX]   pi=10&mobile=155
0x40551ee5  [HEX]   55215554&mobilem

imei=123443213333333  & ostype=2.3.3 & osapi=10 &
mobile=15555215554 & mobilemodel=generic+sdk &
netoperater=internet  & nettype=mobile &
managerid=KuNiu4 & sdmemory=754MB &
aliamemory=147MB & root=0
```

Hidden Phone Calls: One of the malicious behaviours activated by the activity `OnlineActivity` is a hidden phone call. This can serve two purposes for this particular malware; (1) it can generate revenue if calling a premium number, and (2) it can trigger another application component. As can be seen in the manifest, the `NEW_OUTGOING_CALL` signal would trigger the `org.linphone.OutgoingCallReceiver` (see Appendix D). The code of this receiver can then trigger other malicious activities, such as recording the phone call. This is possible, as permissions for call recording, amongst other things, was granted to this DroidKungFu sample when installed.

The first thing that appears when the activity is activated, is a black screen with the phone icon on the top right corner. This icon briefly changes to indicate a call being made. As the call is seconds, this either supports the idea that DroidKungFu uses outgoing calls to trigger a receiver, or the call failed. To hide this action, the activity is nearly entirely black, and no call record can be found in the call logs afterwards.

5.6.2.3 Memory Artefacts

Now that we have fully triggered as many behaviours as possible, we can describe the resulting memory artefacts. While detailed descriptions may seem tailored for these specific malware, the detail is only to show how memory forensics works for real-world samples and are easily generalized to encompass many more Android malware families.

Libraries: Two libraries are generated by *installing* this DroidKungFu sample (i.e., app does not need activating); `liblinphone` and `libnative`. The library `liblinphone.so` and `linphonerc` file are not inherently malicious, but can be used so. This is so with many malware. For instance, by analysing the `linphone` library code, it is capable of monitoring calls [2]. Possibly, this library is mainly used as a side channel to trigger a broadcast component or generate revenue from premium numbers.

When scanning the memory dump for `linphone`, it is possible to see different functions of the library being used by scanning for the string `LinphoneServices$`. By dumping library memory maps in general, it becomes possible to identify the specific functions being used, and to what end, as mentioned in Section 5.6.1.

Many malware, not just DroidKungFu, load native code from non-standard libraries (e.g., `libnative.so`) as it is easier to hide malicious behaviours from static and dynamic analysis. Therefore, it seems very difficult to see exactly how this library is being used. More analysis is necessary to see whether this is true. However, with the combination and accumulation of all the other available memory artefacts, detecting this library should help confirm the maliciousness of an app, such as DroidKungFu.

Linux Filesystem: Directories DroidKungFu attempts to access against filesystem permissions are `/data/app` and `proc/sys/kernel/`. As with all malware, we are able to detect when a file is created, moved, or executed maliciously in a system directory. For example, by analysing the memory image we can detect DroidKungFu coping itself to `/data/app` (see Volatility memory section in Figure 5.12 below). This snipit of the memory only shows the latter half of the command (i.e., does not show `cp`).

```
T: ndroid.sipphone pid 10934 rule r1 a: 0x81a31
0x00081a31  [HEX]   data/app/com.aij
0x00081a41  [HEX]   iaoyou.android.s
0x00081a51  [HEX]   ipphone-1.apk.wC
```

Figure 5.12: Memory evidence of malware moved to `/data/app` directory.

Furthermore, if com.google.ssearch.apk (a malicious Google search look-alike) had been properly created, we would have detected it being installed into `system/app` by searching for the directory name and the execution command.

In addition to detecting filesystem violations, by scanning for other specific strings, we are able to gain sufficient details to identify the exploit used and other malicious behaviours occurring (e.g., leaking device information to a remote server). Evidence of exploid using hotplugs can be found in the memory dumps in Figure 5.13.

```
T: ndroid.sipphone pid 10934 rule r1 a: 0x4056d135
0x4056d135  [HEX]   proc/sys/kernel/
0x4056d145  [HEX]   hotplug....[-].o
0x4056d155  [HEX]   pen....[-].read.
0x4056d165  [HEX]   ...[-].write.../
T: ndroid.sipphone pid 10934 rule r1 a: 0x4056d1e4
0x4056d1e4  [HEX]   gjsvro..data....
0x4056d1f4  [HEX]   hotplug.loading.
0x4056d204  [HEX]   [-].socket..[-].
0x4056d214  [HEX]   creat...ACTION=a
T: ndroid.sipphone pid 10934 rule r1 a: 0x43ef0b7a
0x43ef0b7a  [HEX]    gjsvro../gjsvro.
0x43ef0b8a  [HEX]    ./system/bin/chm
0x43ef0b9a  [HEX]    od..4755./data/d
0x43ef0baa  [HEX]    ata/..oldrun../g
T: ndroid.sipphone pid 10934 rule r1 a: 0xafd07b3c
0xafd07b3c  [HEX]   symlink.fchdir.t
0xafd07b4c  [HEX]   runcate.__statfs
0xafd07b5c  [HEX]   64.pause.gettime
0xafd07b6c  [HEX]   ofday.settimeofd
```

(T=Task, a=addr for all yarascan output)

Figure 5.13: Simplified `yarascan` output for `gjvso/hotplug/symlink` exploit.

Processes: DroidKungFu can create three processes, `com.aijiaoyou.android.sipphone`, `com.android.browser`, and `android.process.media`. As this follows the pattern BaseBridge A had, the first process (`aijiaoyou`) successfully executes the root exploit and the two following processes run with root UIDs and possibly with root children processes. Furthermore, if DroidKungFu had successfully installed the other APKs, they would have their own process(es), i.e. `com.google.ssearch`, `com.alipay.android.app` [3]. Lastly, while not yet a reliable indicator of malware, each time the `aijaoyou.android.sipphone` process attempts the exploid exploit, its PID increases by at least 100. We have found that, very often, odd behaviours such as this are the result of malware attempting to exploit vulnerabilities.

Network Access Strings: Another artefact we found useful for analysing memory images were strings such as HTTP(S), IP addresses, and domain names. As there were several hundred results for just `http://` in DroidKungFu's memory alone (e.g., bank sites, payload sites, YouTube, etc.), we only present some of the more interesting snipits in Figure 5.14. We can then look deeper into the memory for details.

Figure 5.14: Memory results searching for http(s) strings.

```
T: ndroid.sipphone pid 10934 rule r1 a: 0x2fc7f8
0x002fc7f8  [HEX]   http://www.w3.or
0x002fc808  [HEX]   g/XML/1998/names
0x002fc818  [HEX]   pace|...I.A.p.p.
0x4016ccd1  [HEX]   http://www.wells
0x4016cce1  [HEX]   fargo.com/certpo
0x4016ccf1  [HEX]   licy9..>...c...P
0x43eee497  [HEX]   http://219.238.1
0x43eee4a7  [HEX]   60.86/sipadmin/i
0x43eee4b7  [HEX]   nt/feezfblog.jsp

T: ndroid.sipphone pid 10934 rule r1 a: 0x43eeed54
0x43eeed54   [HEX]   https://msp.alip
0x43eeed64   [HEX]   ay.com/x.htm..se
0x43eeed74   [HEX]   rver_url.(Lcom/a
0x43eeed84   [HEX]   lipay/android/Mo
```

After dumping and analysing the memory map containing `http://219[.]238[.]160[.]86`, it appeared to be a part of a generated HTTP request to a Chinese server and is very similar to a DroidKungFu HTTP request found in a Foresafe analysis [3]. Therefore, more generally, URLs in memory are useful for detecting malicious behaviours, apps, developers, and servers. Detection can also be enhanced with network packet analysis on payloads (e.g., find leaked data), as shown in [103, 134, 136].

5.7 Limitations and Threat to Validity

Limitations of this work include no analysis of network activity and packets, and the relatively small, older, set of malware analysed. This threat to external validity is more prominent than previous chapters due to a much more limited dataset, but was sufficient for basic, exploratory research. That being said, if they had been more readily available, more sophisticated malware, e.g., Oldboot, and malware not using root exploits would have greatly enhanced this work. The purpose of these experiments, however, was primarily based on determining the types of memory artefacts that may be found in Android memory images, and whether they are reliable indicators for malware detection.

We were encouraged to find several reliable artefacts while running our malware sample set in multiple experiments. Furthermore, once discovered, the discovered artefacts greatly aided the analysis of other samples, illustrating that these artefacts can be used on a wide range of malware. It is also important to note that, while preventing network connectivity was a caveat in these experiments, it was a conscious choice to prevent the malware from communicating with entities outside the malware lab. However, this did provide an opportunity to detect failed malicious network behaviours.

In the future network analysis can be securely enabled and code coverage via stimulation can be improved (see Chapter 6). These are classic drawbacks of dynamic analysis. Depending on how they were formed or gathered, threats to signature and artefact validity include polymorphic malware. However, while this may deter multi-class classification, artefacts and signature may be sufficient for future work on broad-brush detection. As these experiments were driven by a human analyst (human error mitigated via double-checking and third-party), further discussion on automating the proposed malware detection methods to overcome the limitations of manual analysis can be found in Section 6.2.3. Moreover, although not fully developed, the author can estimate the performance and scalability of automatic detection built on this research in Chapter 6.

While unsure how the ART runtime will affect LiME (released after project), even if new methods of memory acquisition are required for Android, the author's work on memory artefact based malware detection could still be considered a viable and scalable option. However, this is perhaps the most significant threat to internal validity of this chapter. As ART was not released until after research completion — it had been partially tested on one available version and some documentation had been released — the author was unable to apply memory forensics to Android versions running ART. While future work might seek more robust acquisition methods despite runtime changes, this research on detecting malware via memory forensics should be portable across Android versions.

5.8 Related Works

As mentioned in the Section 5.2, memory forensics has been an integral part of detecting crimes on tradition computing devices and has many years of research supporting it. As the popularity and complexity of mobile devices have increased, several efforts to adapt memory forensics to mobile devices have been made. As Android is currently the most popular smart phone OS, that has been the focus of this thesis. However, memory analysis of Windows, Symbian, or iPhone devices are not uncommon [43, 69, 103, 207].

Memory Acquisition: As of 2016 there has been a substantial amount of memory analysis targeting the Linux kernel [31, 40, 62, 220]. Several of these Linux techniques have been attempted on Android as well. A common limitation of past Linux-based projects was the inability to support the numerous kernel versions. As several Android OSs run on either a modified Linux 2.x kernel or 3.x, it was necessary to overcome such shortcomings. To address this problem, projects such as Volatility use profiles (see Section 5.4.3), to automatically build kernel structure definitions. Other solutions merged static and dynamic analysis to identify the kernel version [11, 33, 156].

Traditionally, capturing memory on Linux kernels required access to `/dev/mem`, which contained a map of memory. One could use the Unix command `dd` to dump `/dev/mem`'s memory map. The authors in [187] used `dd` to mount and copy available system partitions stored in the internal memory, such as `system`, `data`, `cache`, `cdrom` and `pds`. Since then, however, access to the `/dev/mem` has been restricted.

Alternatively, [206] developed a memory acquisition tool using the files `/proc /pid/maps` and `/proc/pid/mem`, and a "Process Trace" system call to gain access to a process's execution and address space. This was similar to previous works, except it was customized for Android. Although four different devices were used, it was unclear whether this method would work on all available Android OS versions.

Conversely, in order to capture all physical memory (e.g., not only the first page) the first loadable kernel module was created for Linux in 2010 [121]. Unfortunately, this particular module was not directly compatible with Android as it relies on a `page_is_ram` map which does not exist on Android ARM hardware. This led to the creation of LiME [5]. Released in 2012, LiME is an Linux loadable kernel module (can be compiled for Android) that overcame several issues with previously existing techniques, such as `dd`, and could either dump memory images onto a sdcard, or over TCP [5, 198]. Our method of using LiME on emulators with virtual sdcards allowed us to acquire the same data as other tools and methods while removing the limitations of a physical sdcard.

Other existing forensics tools include Cellebrite's UFED and viaForensic's AFLogical. The UFED tool [46], now in version 4.0, is capable of physical memory extraction on several different mobile OSs (e.g., Android, iOS), as well as file system extraction, and more. Each extraction method however, is supported by different sets of devices and it is unclear on how many devices are supported for all extraction methods. In comparison, the free AFLogical tool is purely for Android and allows examiners to analyse the sdcard's contents as well as browser, calender, IM, and call data [213].

Memory Analysis: In terms of memory analysis, after the memory acquisition phase, there have been two works on messaging apps for iOS [103] and Android [136], a thesis analysing email and chat apps [134], and Volatilitux [89]. We are unaware of any analysis works utilizing memory dumps to detect malware.

Indeed, the bulk of memory forensics seems more police and law orientated, e.g. searching for evidence of crimes committed by humans or forensic data in general [197]. The author in this project [134] created several Volatility plugins to parse the Dalvik VM to acquire Java class objects and read data such as user names, passwords, chat messages, and email. In their analysis, they analysed the K-9 Mail app (i.e., parse, list, and read email) and the WhatsApp chat app (i.e., parse and read conversations).

However, the plugins developed in [134] only worked on the Android OS version 4.0 (i.e., Ice Cream Sandwich). This is due to the dependence on a Dalvik VM global (i.e., `DvmGlobals gDvm`), which the plugins can not located on any other version. Furthermore, our attempts at modifying the plugins to be compatible with other Android versions was unsuccessful. Alternatively, the project Volatilitux, an Android version of Volatility (i.e., not a Volatiliy plugin), was released in 2010 but seems to have been discontinued as we were unable to get it to run on any Android version.

Android Malware Analysis: Android malware analysis consists primarily of static and dynamic solutions (see Chapter 2). However, static analysis is vulnerable to code obfuscation, as well as non-Java code injection, network activity, and object modifications at run-time (e.g., reflection). While several strings may be found using memory forensics or static analysis [184], our method provides better details on when, and how, string commands were used, and their resulting behaviours. Furthermore, we found that several useful behaviours for detection (e.g., library calls, PIDs/UIDs) could only found dynamically during execution.

Dynamic malware analysis can monitor multiple features at several different layers of the Android device (e.g. app, OS, network, kernel, hardware) at run time. Unlike

static analysis, only one path is shown per execution, which limits our understanding of the malware in other ways, but can be improved with stimulation [202]. While there have been many studies analysing and detecting malware through system calls [41,202], APIs [49,231], libraries [233], taint [72], inter process communications [202,224], and hardware [118], we are unaware of any analysing volatile memory. And while it is possible to analyse libraries, process IDs, or open files at the kernel or system level, live tools tend to provide less reliable or accurate results when compared to memory image analysis [7]. Furthermore, with the exception of enabling our kernels to allow kernel module loading, no kernel, OS, or runtime modifications were necessary.

5.9 Summary

In our analysis of Android malware and their memory image dumps, we found we could reliably detect and identify malicious behaviours through certain library, filesystem, and process related memory artefacts. In this chapter, we have both demonstrated the effectiveness of our discovered artefacts and provided memory artefact-based algorithms to detect prevalent Android malware behaviours, such as exploiting system vulnerabilities. Furthermore, we extrapolated from our findings to illustrate how memory forensics is both a quick, reliable, malware detector (ideal for on-device) and can theoretically detect evasive malware most systems cannot, via memory fingerprints [34, 165]. This application of forensics is novel and also deviates from most Android malware detectors.

We have shown that memory forensics can be reliably used to detect privilege escalation from system-level exploits, data theft, failed actions, and background behaviours (e.g., install payload APKs and making calls). While some behaviours do overlap with CopperDroid's profiles (see Table 4.2, page 107), which we have shown to be useful for classification in Chapter 4, we acquired the data in a second, novel, portable method (see Goals in Section 2.6). However, in areas where CopperDroid would fail, like detecting bootkits, memory fingerprints could theoretically be the ideal complementary solution.

Even when implemented alone, we have shown how the author's use of memory forensics can clearly detect RATC, exploid, DroidKungFu A, BaseBridge A, and, therefore, other popular malware and exploits. Furthermore, based on our background analyses and understanding of the newest Android malware [59, 117], even new malware and exploits result in the same memory artefacts when following the pattern of privilege escalation, then malicious actions against DAC rules, and the resulting root IDs and child processes. Future work in memory artefact-based malware detection, including generating memory fingerprints and finding artefacts, is fully discussed in Chapter 6.

Although a different approach for analysing and detecting Android malware, this work stems from the same research goals presented in Chapter 2 (i.e. transparent, portable, reliable, analysis of Android malware). Both of the author's approaches to achieve these goals resulted in novel ways to obtain dynamic information about Android malware. Due to their differences, CopperDroid is more detailed but memory forensics has real potential of detecting malware CopperDroid, and similar works, cannot. Thus we consider memory forensics as an alternative solution that could have useful applications as a complement to CopperDroid analysis as well as binary classification (i.e., malware detection).

Chapter 6

Conclusions and Future Work

Contents

This section concludes the author's work by revisiting all contributions presented in this thesis. Here the author has shown that novel and accurate methods for analysing and classifying Android malware are possible despite the rapidly evolving Android ecosystem. This section then revisits the thesis goals, based on the in-depth survey (Chapter 2), in order to determine whether or not the author has satisfied those objectives. This includes the author's work for enhancing CopperDroid, the system call based analysis that can reconstruct detailed behaviours due to the author's contributions.

The author then demonstrates the usefulness of CopperDroid generated behaviour profiles by using them to implement a hybrid multi-class classifier. Afterwards, the author reflects on the limitations on the content of Chapters 3 and 4, as well as the author's research goals, and demonstrates the potential power of memory forensics.

Lastly, the author discusses possible future directions of research based on the discussions within each chapter. This does not include future work discussed in Chapter 2, specifically Section 2.5, as these are not future works built on top of research performed by the author, but future works that the author feels the community should pursue after systematically surveying the body of work relating to Android malware.

6.1 Restating Research Problems and Research Goals

In Chapter 2 the author performed an extensive survey on the current body of work relating to the analysis, detection, and classification of Android malware. The culmination of this survey was four thesis goals (page 56) that addressed existing research gaps.

Specifically, Goal 1 outlined the importance of analysing native code, inter-process communications, network traffic, and dynamically loaded code as they occur often and can hide behaviours. As many of these are impossible to analyse statically, the author choose to develop novel dynamic methods. While code coverage is a prevalent problem with dynamic analysis, intelligent stimulation can be applied to improve the situation.

While Goal 1 encouraged the author to gain complete profiles of Android malware, Goal 2 also outlines the importance of portability. Therefore Goal 1 must be achieved without the modification of the Android runtime, OS, or apps. This goal was the direct result of the problematic instabilities of an ever-changing system. For example, whenever a new Android OS is released, if frameworks need to be modified to be compatible, there exists a window of opportunity where malware are undetected and unrestricted.

While highly detailed data tends towards higher accuracy, excessive or redundant data increases performance costs and decreases the efficiency of a framework. Thus, we introduced Goal 3, which influenced a smaller, more concentrated, set of features to work with. This allowed the author to improve accuracy with less performance sacrifices, a trade-off issue common when dealing with large datasets.

Lastly, even if a framework is scalable, portable, and highly detailed, it is ineffective if malware can evade analysis or detection. We discovered this to be a problem with several frameworks, as they were vulnerable to obfuscation and evasion. Furthermore, our analyses of Android malware confirm other findings that show malware becoming increasingly sophisticated and more evasive. Thus, we introduced Goal 4 to develop frameworks that were robust and versatile against these tactics.

6.2 Research Contributions and Distribution of Work

In the introductory chapter of this thesis, the author clearly stated the distribution of work (summarized in Table 1.2, page 16). Furthermore, the novel research aspect of each shared contribution was provided in the summary sections of Chapters 3 and 4 (pages 90 and 122 respectively). In this section we seek to restate which contributions were the direct result of the author's work and how they expanded the existing body of knowledge. As a survey, the content of Chapter 2 is not included in this section.

6.2.1 CopperDroid

In Chapter 3 we proposed CopperDroid, a VM-based dynamic analysis technique to reconstruct, automatically, all levels of Android behaviours from system calls, a single point of observation. This was a joint effort, as the system call interception functionality was developed by collaborators. Specifically, the original version of CopperDroid, whose performance can be found in Section 3.6, was developed before the author joined the research group. The basic work for collecting system calls was complete, and a few behaviours were extracted based on a few, single, calls. The second version of CopperDroid's system call collection was vastly more efficient and also tracked `ioctl` system calls more diligently (with some input from the author). At this time, CopperDroid also had the AIDL parser developed and a simple all or no stimulation technique in place.

The author's role in CopperDroid, however, has been instrumental in taking the raw system calls and automatically performing complete, and complex, behaviour extraction. This required methods very unlike traditional system call analysis, such as the unmarshalling Oracle (see Section 3.4). This resulted in novel behaviour profiles with a level of detail other researchers previously thought was unachievable using system calls alone. Additionally, the author performed a more fine-grained analysis of the affect of specific triggers than the previous work on CopperDroid. This helped isolate the most encouraging stimuli, and sets of stimuli, for triggering malware (Table 3.4, page 83). The author's enhanced CopperDroid was also able to reconstruct files associated with file system access behaviours, providing further details to profiles. The resulting framework fulfilled the author's goals of a robust, portable, and thorough analysis tool.

6.2.2 Android Malware Classification

We then tested the usefulness of CopperDroid's behavioural profiles by using them to accurately classify Android malware in a scalable manner. While the collaborative work is a less clearly divided, the author either developed, or made significant progress in, all main, novel, contributions (see page 93). First, as a baseline, a standard SVM part of the classifier was set up by a collaborator. Then, using the author's work in CopperDroid, the author then provided a novel feature set to feed to the traditional SVM classifier. Performance of the pure SVM classifier were then generated by the author to determine misclassification errors, feature statistics, precision, recall, and accuracy.

In order to supplement SVM decisions with low confidence (i.e., decisions with low probabilities of being correct), the author and a collaborator worked on implementing

a conformal predictor when it would be beneficial. While the collaborator helped determine p-values (see Section 4.5.3) from the SVM output, the author developed tools to perform conformal perditions based on these values. This is the core component of our novel hybrid classifier. In essence, this required finding all class labels that pass a similarity threshold, creating sets of classification predictions that are the most likely of being correct. The author then developed all tools to re-calculate accuracy, precision, and recall, in order to quantitatively compare the hybrid results to the SVM baseline.

To demonstrate the benefits of classifying Android malware based the condensed, detailed, behaviour profiles generated by the author's work, a collaborator helped create baselines by testing our classifiers with the system call traces that were transformed into these behaviours. While raw system call analysis has been done in several previous works, both on traditional PCs and Android (see Section 4.8), it, again, provided a qualitative baseline to demonstrate the advantages of the author's behaviour reconstruction.

6.2.3 Memory Forensics

The last segment of work in this thesis explored new uses of memory forensics as an alternative Android malware analysis method. The ancillary work showed how memory artefacts can be used reliably to detect malware in a simplistic, robust, portable way ideal for on-device detection. Furthermore, the work shows possible applications for detecting evasive malware, other solutions cannot, via memory fingerprints. In comparison to other Android analysis and detection methods there are very few similar works.

Furthermore, the tools used (i.e., LiME and Volatility), were developed for purposes other than Android malware detection. We are unaware of any other works using these tools in the same way the author has to gain novel Android memory artefacts to be used for malware analysis and detection. Another useful aspect of memory forensics is the ability to detect some failed or dormant behaviours. As the malware lab was designed to prevent infections spreading via the network, and so was disabled, we were able to detect failed download payloads as well as failed bot communications to a command and control centre. This is particularly useful for identifying malware that only *seem* benign because it is unable to execute properly in the current conditions or environment. While experiments were largely manual, the author did explore the potential scalability of this method to encourage future work in this area (more in Section 6.3).

Lastly, the author would like to acknowledge HP Labs in Bristol, which provided a three month internship to research the applications of memory forensics on Android in an industry setting. It was an invaluable opportunity to perform solo research and be allowed to explore areas as the author saw fit.

Table 6.1: Thesis contributions and goals met per chapter.

	Dynamic	Portable	Scalable	Robust
Ch 3	* System call intercept	* QEMU plugin	-	+ Oracle
	+ System call analysis	+ Oracle	-	-
	*+ Stimulation	-	-	-
Ch 4	+ Behaviours	-	*+ Hybrid classifier	+ Behaviours
	* system calls	-	* Baseline	
Ch 5	+ Memory image	+ LiME	-	+ Artefacts

(* = collaborator, + = author)

6.2.4 Goals Met

The behaviours gained and used in Chapter 3 and 4, and the memory artefacts gained in Chapter 5, satisfy thesis Goal 1 by capturing actions that only occur dynamically. Furthermore, despite minor configuration modifications, all methods presented within this thesis are portable. This satisfies Goal 2. Furthermore, this thesis illustrates the accuracy and scalability, Goal 3, of these frameworks. Finally, the author demonstrated how well each framework captures malware behaviours, even evasive one. Although, like all these goals, Goal 4 is difficult to completely achieve, the author addressed as many evasive malware as possible. In particular, the purpose of Chapter 5 was to provide a method to analyse evasive malware that previous methods could not.

An overall summary of thesis contributions and contributors can be found in Table 6.1. Again, the novel research aspect of each contribution can be located in the summary sections of Chapters 3 and 4 (pages 90 and 122 respectively). In this table, Thesis goals 1-4 have been roughly simplified to dynamic, portable, scalable, and robust respectively.

6.3 Future Directions

There are many opportunities to further the work that has been presented within this thesis. First of all, CopperDroid and dynamic frameworks in general can be improved with more **effective stimulation**. As previously discussed by the author in Section 2.5 of the survey, there are many attractive aspects to hybrid solutions. The combination of static and dynamic analysis to improve code coverage and discover sequences of valid stimuli to reach all interesting behaviours is very compelling. Furthermore, the possibility of hybrid emulator and physical devices could provide an interesting counter attack to VM-aware and VM-evasive malware (see Section 2.3).

There is a second area for future work which is unique to CopperDroid. It is possible that, during run-time, CopperDroid can both intercept system calls and then alter the

return value after it has copied the original value. This could be a novel form of stimulation and to better disguise the emulator as a real device. For example, instead of actually modifying the emulator IMEI to a realistic number, CopperDroid could alter the return values of involved system calls to contain a range of believable values. Furthermore, while allowing network access triggers more behaviours for analysis, similar "trickery" techniques may help attain these behaviours with less risk to other systems and users.

As a part of our analysis, it would also be interesting to build more dependencies between objects and behaviours. For instance, although we can track the file access behaviour creating a file A and a network behaviour sending the contents of file A, CopperDroid does not automatically see this as a chain of behaviours relating to a single file. This may be solved with methods like taint tracking or symbolic execution. As shown in Chapter 5, mapping specific app components to specific behaviours, malicious or benign, also provides a wealth of information. This may require making tool to automatically extract APK components and systematically trigger different component sets over a series of experiments. There are also areas to create better hardware/system stimuli (e.g., accelerometer, geo-location) or to integrate previous works [22, 87, 133].

Future work on our proposed multi-class classifier can be divided into several areas. Firstly, the **feature set** could be enhanced with more behaviours. This may be solved with alternate solutions such as memory artefacts and/or memory fingerprints. Similarly, analysing a much larger, more diverse, and more current set of malware may reveal addition features essential for the classification (both multi-class and binary) of Android malware. Furthermore, our feature vector currently includes one element representing the number of bytes across all network behaviours of a sample. Future work should include determining whether splitting this amount, or any other element, into two elements (i.e., received bytes and sent bytes) would improve classification.

Determining whether there are better fitting **machine learning** methods than SVM would also be useful. This could be achieved by analysing more extensive datasets to discover areas where the classifier is currently lacking. Many available machine learning approaches, as well as different settings, have not yet been tested for the most appropriate method. Moreover, automatic tools to determine (1) optimal p-value limit for of our CP, (2) best thresholds for behaviours per sample, and (3) ideal samples per family thresholds would greatly enhance the conformal prediction component of this work.

While our classifier should be able to detect **zero-day** malware, i.e. samples likely to be malware but dissimilar to all available classes, we have not implemented a tool to do so. Theoretically, however, setting an upper p-value limit and a lower p-value threshold could determine clear classification labels (above high p-value limit), classifications

to be done with CP (between thresholds), and zero-day malware (below lower p-value limit). Automatically finding these limits would be an interesting topic for future work. Furthermore, discovering which behaviour features are best for two-class identification and/or multi-class classification would be an interesting area of work. Similarly, applying the contributions in this thesis to two-class classification may prove more accurate than multi-class classification due to more available features and more defined classes.

For **memory forensics**, future work could enable networking to analyse downloaded content (e.g., malicious APKs), uploaded data (e.g., IMEI), and SMS communications to and from a C&C server (e.g., NickiBot malware [159]). This seems a logical step forward as previous work has shown memory forensics to be capable of analysing network activities such as messaging and email [44, 134, 198]. Further analysis in this area and with larger more diverse dataset could provide more useful memory artefacts.

To **automate** forensic analysis and the discovery memory artefacts, one could implement the algorithms in Section 5.5 (Algorithms 5.1, 5.2, and 5.3) that were manually applied during our initial experiments. While these can be generic to detect malicious or dangerous behaviours, with more fine-tuning, these tools may be sensitive enough to detect specific behaviours and malware, as Algorithms 5.2 and 5.3 demonstrate best. Automatic tools to generate signatures of significant areas of memory is another are of future work, and whether they can be used to detect evasive malware such as bootkits.

During analysis, we have found that most Volatility plugins, such as `pstree`, `proc_maps`, or `psxview`, ran between 1-5 seconds across the whole memory image. We also found string scans, i.e. `yarascan`, where the most time consuming. When analysing the entire image, `yarascan` could take 20+ minutes. Conversely, when analysing specific processes or only app processes, the time for a string search has not exceeded 15 minutes. Specifically, scanning for a string in one processes may take one minute, but not exceed six. Therefore, in future work it would be more **efficient** to implement fast plugins first (e.g., `pstree`) and filter out uninteresting processes. Parallel processing, with each thread processing a subset of app processes and/or different threads scanning for different artefacts simultaneously, should also improve performance. In this model, when an artefact is discovered in one Android process, the other threads can shift their focus to look for more incriminating artefacts in the same process.

While majority of our analyses yielded multiple artefacts per sample (as we can detect some failed or dormant malicious behaviours), malware with very few artefacts would attribute to a higher false negative rate. Few artefacts may also increase false positives, as one "dangerous" artefact may accuse a sample of being malware, despite the app just being "dangerous" or slightly intrusive instead of malicious.

6.4 Concluding Remarks

Android, the leading mobile OS, and Android malware are still rapidly evolving as of the year 2016. Therefore, it is essential for research on malware capabilities and malware analysis, detection, and classification techniques to grow and improve just as quickly.

Within this thesis, the author presented a comprehensive survey on the current body of Android security research. To evaluate the effectiveness of these works and determine areas where more research is beneficiary, the author compared malware capabilities to general framework weaknesses. The culmination of these observations lead to an enhanced version of CopperDroid, which transparently and robustly recreated behaviours from system calls alone. The author's work was instrumental in generating complete behaviour profiles and advanced the state of the art for Android system call analysis.

The merit of these behavioural profiles were then tested by using them to classify Android malware. Baselines were set with standard SVM and traditional system call input, and compared to our hybrid classifier using the author's recreated behaviours. This illustrated the usefulness of using behaviours over raw system calls and allowed the author to develop the central CP component of the innovative hybrid classifier.

Finally, the author reflected on malware that could still evade CopperDroid, and similar works, and explored using memory forensics for malware detection. Alone, memory artefacts were found to be sufficient for malware detection (i.e., binary classification). Furthermore, the author determined that this area of research had the potential breadth necessary to detect evasive malware with memory fingerprints. While some behaviours are less detailed extracted from memory, what is available is unique and can be use to improve CopperDroid analysis, and Android malware classification. While primarily exploratory, this work applied memory forensics in a novel manner and discovered Android memory artefacts suitable for malware detection.

It is clear that robustness, transparency, portability, and scalability are all highly desired, and necessary, traits to mitigate the capabilities of current Android malware. Furthermore, it is possible to improve more than one of these characteristics when implementing any framework for the analysis and classification of Android malware, given a deep understanding of the system and threat. The conclusion of this thesis is that the original hypothesis:

Low-level system data produced by Android applications can be used to accurately, and scalably, characterize malware whilst remaining agnostic to significant device changes

has been established.

Appendix A

Comparison of Related Works

This appendix provides detailed comparison of various Android related works spanning 2012 to 2015. Studies are split into two tables depending if the main focus is "analysis" or "detection". Studies are primarily organized by year. Details for each study include its citation (and name if available), are in the remaining columns. The methodology is briefly outlined (i.e., static or dynamic), as is the origin and number of samples tested, the work's scalability, and how sturdy the techniques are against obfuscation etc. False positives and negatives are also considered within the table for detection frameworks.

The following works spanning 2011 to 2015 focus on analysing Android malware[1].

Table A.1: Analysis frameworks for Android systems.

Year	Framework	Method	Samples	Sample Selection	Scalable	Sturdiness
2012	Aurasium [231]	sandbox (dynamic) detect API misuse	3rd party	3491^{b} (99.6% detect), 1260^{m} (99.8% detect)	low overhead	✗ native code ✗ java reflection ✗ transparency
2012	PScout [20]	perm. spec. from OS source code & APK + stim. (UI fuzzing)	GP	1,260 chosen for API coverage	-	✗ unfeasible paths = false mappings
2012	AppGuard [24]	app rewriter + dynamic inline ref. monitors + stim.	GP, SlideMe	25,000 apps tested for robustness (stimulation)	low overhead	✗ callee-site rewrite ✓ java refl, dynamically loaded code
2012	DroidScope [233]	dynamic + virt. + reconstruct OS & Java-lvl semantics	GP	7 benchmark (efficiency & capability) + 2^{m}	taint 11x-34x slow↓	✓ Java, JNI, ELF ✗ limited code coverage
2012	I-ARM-DROID [55]	statically add stubs to use correct perm./APIs	GP	30 random from top 100 free apps	size+2% +110 ms	✗ native code ✓ API reflection
2012	SmartDroid [242]	static to find exec paths + dynamic to find triggers	-	19 wild apps (7 fam.) w/ UI triggered mal.	6/7 <1.5 mins	✗ native code ✗ cannot find hidden UI
2013	CopperDroid [202]	VM-based dynamic analysis + stim	several sources	$1{,}200^{m}$ (49^{m} families) 400^{m} (13 families)	∽10min/ app	✓ Java, JNI, ELF ✓ stimulation
2013	Jin et. al [113]	software-defined network traffic monitoring	-	4 mobiles 100 IPs	∽746k response/sec	✗ encrypted traffic ✓ monitors traffic from all OSs
2013	ContentScope [232]	static path-sensitive data-flow + dynamic exec confirmation	markets (mult.)	62,519 apps (3,018 vul.) from Feb. 2012	-	✗ false +'s (static & start errors) ✗ manual class.
2013	Contest [11]	concolic app testing (generate event sequence for tests)	-	5 open-source apps	∽hour/app	✓ filters paths ✗ only tap events
2013	Droid Analytics [244]	static op code signatures (method, class, payload)	markets, web	150k ($2k^{m}$s 234 families	∽70sec/ app	✓ repackaging ✓ code obfuscation ✗ logic obfuscation
2013	Pegasus [49]	static + runtime policy monitors + API/permission event graphs	-	152^{m}, 117^{b}	80% 0.5hrs, max 5.6hrs	✓ event fire context ✗ detects obfuscation but still vuln.
2013	ProfileDroid [224]	static + multi-layer dynamic (UI, system calls, network)	GP	27 varied apps (8 pairs of paid/free apps)	10 (5 min) runs/app	✓ diverse run environments ✗ not scalable
2013	SAAF [99]	static (smali) auto and optional manual	GP	free apps: $136k^{b}$, $6k^{m}$	<10 sec/ app	✗ reflection ✗ runtime info
2013	VetDroid [241]	dynamic permission usage + reconstruct fine-grained actions	GP	32 categories top $1.2k^{b}$ apps	2min/ app	• slowdown 32% on device ✗ native code, java
2014	Rasthofer et al. [167]	dy + taint + machine learning + API feat.	Virus Share	$11k^{m}$ apps with API data leaks	SVN, QP-prob.	✓ All OSs versions ✗ no obfu. test
2014	RiskMon [114]	dy+machine learn+ API monitor + interpose IPC	GP	14 mostly popular & at least 2 were free	0.55s/ app	✗ colluding apps ✗ non-binder comms
2014	[164]	edit DVM + control flow graph (method) + dyn. code loading	GP	popular free 1.6k 2012-Aug 2013	relies on white list	✗ default app config code exec. ✗ see code loading
2014	A5 [216]	static execution paths via Activities + dy. network ID sigs.	public source	1,260 malicious apps	avg. 149s/app	✓ ∽ transparency ✗ dynamically loaded intents
2015	DroidSafe [91]	static information flow + hooks + calls that start activities	real-world apps	24 modified apps for hooking	<222s per analysis	✓ filters classes ✗ dynamically loaded code

[1] Superscripts "m" malicious, "b" benign, and "P" for Google Play

The following works spanning 2011 to 2015 are focused on detecting Android malware[2].

Table A.2: Detection frameworks for the Android system.

Year	Framework	Method	# Apps	Sample Selection	Result	False +/-	Scalable	Sturdiness
2011	Andromaly [186]	machine learn + realtime hardware monitor	4	self-developed	detect all malware	low/ ?	perf. 10% ↓	✗ "quick" actions
2011	Crowdroid [41]	dynamic system call logs + k-means cluster	5	3-developed, 2-real	100%-self, 93%-real	Yes/ -	NP-hard	✗ needs network ● scalable
2012	Droid MOSS [248]	fuzzy hash + static sigs + dynamic comparison	200	random (6 world markets)	5%-13% repackaged	10%/ 10%	-	✗ assumes legit apps ✗ incomplete lib list
2012	RiskRanker [93]	native code, encryption, dynamic code	118k	mult. markets, 29 families	detect 322 new malware	Yes (?/?)	all in 4 days	✗ downloads ✗ small behaviour set ✗ obfu. (encryption)
2012	ScanDal [119]	static + sensitive APIs + sources/sinks	90	9 free pop, random type	detect 11 leaks	18% /?	83s-49m	✓ simple reflection ✗ native code, refl., obf.
2012	DroidMat [228]	static + perms & API + components	238^{m} 1500^{b}	Contagio&Gl (50^{b} apps)	quick & accurate detection	0.4%/ 12.6%	Y	✓ 50% Androguard speed ✗ native code, refl., obf.
2013	AppIntent [236]	static + symb. exec. + reduce event space	1750	1000^{b} GP 750^{m}	detect 582, sym. 358	164/ low	symb. <2hrs	✓ see user vs. background ✗ native code
2013	AppProfiler [175]	static + map API to behaviour	80k	15 diverse & popular apps	detect ∽59% behaviours	16%/ 15%	500 a/ day	✓ see user vs. background ✗ obfuscate class names ✗ obfuscate pkg name ✗ native code
2013	MAMA [178]	Android Manifest + machine learning	333^{m} 333^{b}	max coverage / diversity, 2011	best detects 94%	best 5% /?	-	✓ wide app coverage ✗ Internet/piggy payloads
2013	PiggyApp [247]	feature fingerprint (perm., API) + feature vectors	84, 767	6 markets + GP + free apps	0.97%-2.7% piggyback	4.5% /?	0.952 s/app	✓ obfuscation ✗ no syntactic sequences
2014	AppSealer [240]	static bytecode + program slices	16	vulnerable apps	patch vul apps	0%/ ?	most <60s	✓ device patches ● app size +16-45% ● app slowdown 2%
2014	Droid Barrier [8]	hidden shells w/ own proc. + credentials	∽ 400^{m}	3 malware fams	36.7% use hidden shell	-	perform penalty <13%	✓ isolated in kernel mode ✗ kernel-level & embedded attacks
2014	Apposcopy [82]	static taint + control/data flow + semantic signature	$1k^{m}_{P}$ $8k^{m}$	in the 8k there were 6^{m}	classify family	10%/ 0.2%	not instant	✓ low level obfuscation ✗ native code
2014	AsDroid [101]	static + intent propagation/-correlation + ICC call chains	128	free pop. apps (GP, Contagio, 3rd Party)	model stealthy behaviours		28% / 11%	✗ flow obfuscation ✗ native code, reflection ✗ only textual UI
2014	DFlow [102]	static + jimple + taint analysis w context	22^{m} 144^{b} 39	free pop. apps (GP, Contagio)	data flow in logs & network	16%/ -	∽2 mins	● partial ICC flows ● +2GB, edits libraries ✓ DroidBench scores
2015	AppContex [235]	Soot + Dexpler + Extended call graphs	202^{m} 633^{b}	GP	context-based detection		∽12% / 5%	∽ dynamic code, refl. ✗ Pscout's drawbacks

[2]Superscripts "m" malicious, "b" benign, and "P" for Google Play

Appendix B

Overall Results on McAfee dataset

This appendix gives results of added stimulation to CopperDroid analysis. For each malware family the 2^{nd} column reports *(number of samples that exhibited additional behaviours)/(total number of family samples)*, 3^{nd} column reports average number of observed behaviours without stimulation, and the 4^{th} column reports average number of additional behaviours exhibited by stimulated samples and their percentage over non-stimulated behaviours.

As an example, let us consider the malware family *PJApps*. This family contains 39 samples, 36 of which exhibited additional behaviours when stimulated by CopperDroid. More precisely, during the *non*-stimulated executions, we observed an average of 27.41 behaviours for each sample of the family, while the stimulated executions allowed to discover an average of 6.1 additional, *previously unseen*, behaviours.

Malware Family	Samples w/ Add. Behaviours	Behaviour w/o Stim.	Incr. Behaviour w/ Stim.
Ackposts	1/1	4	+3 (+75%)
Actrack	1/1	4	+1 (+25%)
AndroidSMS	2/2	0	+1 (⊥)
Anserver	13/21	16.48	+5.2 (+32%)
ApkMon	1/2	49	+1 (+2%)
AppHnd	4/4	37.25	+16.8 (+45%)
AreSpy	1/1	11	+6 (+55%)
Arspam	1/1	3	+2 (+67%)
BackReg	1/1	78	+12 (+15%)
Backscript	2/6	9.67	+19.5 (+202%)
BaseBridge	10/12	4.5	+3.3 (+73%)
Bgyoulu	3/5	17.6	+4 (+23%)
BookFri	1/1	15	+4 (+27%)
Carotap	2/2	4	+3 (+75%)
Coolpaperleel	1/1	55	+4 (+7%)
Crusewin	4/4	6.25	+8.5 (+136%)
Dialer	0/1	1	+0 (+0%)
DiutesEx	23/43	26.58	+8.9 (+33%)
DIYAds	18/18	163.72	+37.6 (+23%)
DougaLeaker	16/16	4	+1.6 (+40%)
Drad	5/5	10.6	+6 (+57%)
Drd.*	30/32	24.74	+7.55 (+31%)
DroidDeluxe	1/1	9	+1 (+11%)
DroidKungFu	63/85	31.02	+6.1 (+20%)
DropDialer	2/11	0	+1.5 (⊥)
Ecobatry	1/1	25	+1 (4%)
EICAR	0/2	1.5	+0 (+0%)
Enesoluty	1/1	11	+2 (+18%)
EvoRoot	0/1	0	0 (⊥)
Fake.*	314/677	6.39	+5.69 (+89%)
Fladstep	1/1	176	+80 (+45%)
FlashRec	1/2	8	+3 (+38%)
FndNCll	1/1	36	+2 (+6%)
Foncy	2/2	1	+4 (+400%)
FoncyDroppe	1/1	23	+1 (+4%)
FrictSpy	8/9	7.56	+10 (+132%)
Frogonal	2/2	27.5	+2.5 (+9%)
Frutk	1/1	73	+17 (+23%)
FunsBot	2/2	5	+2 (+40%)
Gamex	1/1	11	+2 (+18%)
GamexDropp	1/1	8	+1 (+13%)
Geinimi	11/19	23.68	+12.4 (+52%)
GGeeGame	1/1	62	+6 (+10%)
GoldDream	7/8	31.12	+9.9 (+32%)
GoldenEagle	1/1	0	+7 (⊥)
GoneSixty	11/11	16.64	+5.5 (+33%)
GpsNake	0/1	1	+0 (+0%)
HippoSMS	1/1	16	+4 (+25%)
Hnway	0/1	49	+0 (+0%)
Imlog	5/6	19	+9.2 (+48%)
IMWebViewe	1/1	94	+11 (+12%)
InstBBridge	0/1	9	+0 (+0%)
J	7/13	30.96	+3.65 (+12%)
Jifake	1/5	1	+4 (+400%)
Jmsonez	2/2	11.5	+12 (+104%)
LdBolt	8/8	46.62	+7.8 (+17%)
LoggerKid	4/4	4.5	+2 (+44%)
Logkare	0/1	0	+0 (⊥)
Jmsonez	2/2	11.5	+12 (+104%)
LdBolt	8/8	46.62	+7.8 (+17%)

Malware Family	Samples w/ Add. Behaviours	Behaviour w/o Stim.	Incr. Behaviour w/ Stim.
LoggerKid	4/4	4.5	+2 (+44%)
Logkare	0/1	0	+0 (⊥)
LoveTrp	1/1	5	+6 (+120%)
LVedu	33/56	26.93	+5.2 (+19%)
Maistealer	1/1	8	+1 (+13%)
Malebook	1/1	94	+14 (+15%)
Mania	1/2	0.5	+2 (+400%)
MarketPay	1/1	98	+7 (+7%)
Mob.*	11/11	43.67	+9.75 (+22%)
Moghava	1/1	0	+2 (⊥)
MoneyFone	1/1	0	+3 (⊥)
Nandrobox	1/1	0	+4 (⊥)
Netisend	1/1	8	+4 (+50%)
NickiSpy	2/2	71	+10.5 (+15%)
NotCompatib	0/1	7	+0 (+0%)
Nyearleaker	1/1	23	+5 (+22%)
OneClickFrau	22/22	16.27	+17.2 (+106%)
PdaSpy	1/4	0	+1 (⊥)
PJApps	36/39	27.41	+6.1 (+22%)
Qicsomos	0/1	15	+0 (+0%)
QieTing	1/1	0	+4 (⊥)
QuoteDoor	0/1	6	+0 (+0%)
RecCaller	1/1	2	+4 (+200%)
RootSmart	2/2	17	+9 (+53%)
RuFraud	4/6	4.5	+5 (+111%)
SGSpy	1/1	60	+39 (+65%)
SGSpyAct	0/1	0	+0 (⊥)
ShdBreak	0/1	28	+0 (+0%)
SilentWap	3/3	2	+5 (+250%)
SMS.*	16/21	4.77	+8.59 (+180%)
Sngo	1/1	65	+2 (+3%)
Spitmo	2/2	0	+9 (⊥)
SpyBubb	2/2	25.5	+20 (+78%)
Spytrack	1/1	20	+8 (+40%)
Stamper	1/1	63	+7 (+11%)
SteamyScr	2/2	25.5	+8.5 (+33%)
Steek	15/15	8.4	+2.1 (+25%)
Stiniter	0/1	3	+0 (+0%)
Sumzand	0/3	7	+0 (+0%)
SusetupTool	0/1	0	+0 (⊥)
Sxjspy	1/1	24	+4 (+17%)
TattoHack	1/2	6	+1 (+17%)
Tcent	1/1	0	+17 (⊥)
ToorKing	1/1	37	+6 (+16%)
ToorSatp	3/8	7.5	+1.3 (+17%)
Toplank	6/9	37.44	+6 (+16%)
Twikabot	1/1	0	+12 (⊥)
TypStu	4/6	0.83	+1 (+120%)
UranaiCall	1/1	51	+13 (+25%)
VDLoader	10/10	43.7	+8.8 (+20%)
Vidro	1/1	58	+16 (+28%)
Voldbrk	9/17	48.82	+1.2 (+2%)
WalkTxt	1/1	14	+2 (+14%)
Wapaxy	2/2	0	+9 (⊥)
Woobooleake	1/1	5	+2 (+40%)
XanitreSpy	9/9	27.11	+5.9 (+22%)
XobSms	1/1	28	+15 (+54%)
YiCha	10/10	21.5	+4.6 (+21%)
Zitmo	3/3	2.67	+5.7 (+213%)
Overall	**836/1365**	**22.78**	**+6.54 (+28.7%)**

Table B.1: Results of CopperDroid stimulation on McAfee dataset.

Appendix C

Classification Feature Statistics

This table shows the frequencies of features used in our machine learning algorithms for classifying Android malware. The core behaviour classes originated from CopperDroid, but in order to improve accuracy and find differentiating features between classes, i.e. malware families, the author developed subclasses that resulted in the best performance with the developed classifier.

Major feature sets can be found in column one, followed by how many samples exhibit that behaviour, and how often that behaviour is seen across all behaviours exhibited by all samples (columns two and three). We then provide the same statistics for all the subfeatures within each feature set (columns four to six).

Table C.1: Percentage of samples exhibiting behaviours and how often they occur.

Feature Set	Samples	Features	Subfeature	Samples	Features
S1 Network Access	66%	25.5%	IPv4-mapped IPv6	62.6%	18.8%
			DNS monitored	6.2%	6.1%
			IPv4	0.09%	0.07%
S2 File Access	71.8%	40.6%	XML	52.2%	13.2%
			database	38.6%	2.9%
			unknown	30.6%	8.1%
			app	10.9%	1.3%
			arch & comp	9.5%	2.4%
			read (e.g., pdf)	8.0%	0.5%
			media (e.g., mp4)	5.2%	10.2%
			exec	4.5%	0.3%
			ai	3.5%	0.1%
			archive (e.g., zip)	1.7%	0.3%
			cert & comp	1.5%	0.1%
			openvpn (e.g., pdf)	1.4%	0.4%
			web (e.g., mp4)	1.0%	0.07%
			cfginit	0.8%	0.03%
			plist	0.3%	0.01%
			json	0.3%	0.2%
			js	0.3%	0.1%
			tmp	0.1%	0.004%
			mediaarchive	0.1%	0.01%
S3 Binder Method	78%	14%	getDeviceID	59.7%	5.5%
			getSubscriberID	42.6%	2.4%
			getIccSerialNumber	28.1%	1.4%
			SMS_RECEIVED	26.0%	2.3%
			getLine1Number	26.0%	1.3%
			getActivePhoneType	23.5%	1.8%
			getAccounts	21.6%	0.9%
			getAccountsByFeatures	18.4%	0.8%
			getNetworkType	17.2%	1.2%
			getDeviceSvn	14.4%	0.6%
			GET_CONTENT_PROV-IDER_TRANSACTION	12.6%	1.9%
			getLastKnownLocation	11.3%	0.8%
			PHONE_STATE	8.3%	0.8%
			getCellLocation	7.6%	0.4%
			getProviders	6.3%	1.3%
			requestLocationUpdates	6.2%	0.3%
			getVoiceMailNumber	5.5%	0.3%
			getProviderInfo	5.1%	2.0%
			sendText	2.3%	0.3%
			cancelMissedCallsNotification	0.4%	0.02%
			getAllProviders	0.09%	0.01%
			START_ACTIVITY_ TRANSACTION	0.09%	0.01%
S4 Execute	26.6%	7.9%	generic	26.4%	7.8%
			silent install	0.6%	0.1%
			su command	0.1%	0.01%

Appendix D

Malware Manifests

During the analysis of BaseBridge and DroidKungFu in Chapter5, we triggered malware using components declared in their manifests (i.e., activities, services, and broadcasts). Furthermore, we used the ”Raw” process names of the components to detect their behaviours, and analysed permissions for possible stimulations in Chapters 3 and 5.

Figure D.1: Complete Android Manifest for BaseBridge A.

```
N: android=http://schemas.android.com/apk/res/android
 E: manifest (line=2)
  A: android:versionCode(0x0101021b)=(type 0x10)0x1
  A: android:versionName(0x0101021c)="1.0" (Raw: "1.0")
  A: package="com.keji.unclear" (Raw: "com.keji.unclear")
  E: application (line=4)
   A: android:label(0x01010001)=@0x7f050001
   A: android:icon(0x01010002)=@0x7f020001
   E: activity (line=5)
    A: android:label(0x01010001)=@0x7f050001
    A: android:name(0x01010003)="Start" (Raw: "Start")
    E: intent-filter (line=6)
     E: action (line=7)
      A: android:name(0x01010003)="android.intent.action.MAIN" (Raw:
       "android.intent.action.MAIN")
    E: category (line=8)
      A: android:name(0x01010003)="android.intent.category.LAUNCHER" (Raw:
          ↪ android.intent.category.LAUNCHER")
   E: service (line=11)
    A: android:name(0x01010003)="GPSService" (Raw: "GPSService")
    A: android:process(0x01010011)=":two" (Raw: ":two")
   E: activity (line=12)
    A: android:name(0x01010003)="SafeActivity" (Raw: "SafeActivity")
   E: activity (line=13)
    A: android:name(0x01010003)="MapAct" (Raw: "MapAct")
    A: android:screenOrientation(0x0101001e)=(type 0x10)0x1
   E: service (line=14)
    A: android:name(0x01010003)=".service.SysM" (Raw: ".service.SysM")
    A: android:process(0x01010011)=":three" (Raw: ":three")
   E: service (line=15)
    A: android:name(0x01010003)=".service.Mrun" (Raw: ".service.Mrun")
    A: android:process(0x01010011)=":two" (Raw: ":two")
   E: receiver (line=16)
    A: android:name(0x01010003)=".service.ForAlarm" (Raw: ".service.ForAlarm")
    A: android:process(0x01010011)=":remote" (Raw: ":remote")
   E: uses-permission (line=18)
```

```
    A: android:name(0x01010003)="android.permission.ACCESS_NETWORK_STATE" (Raw:
     "android.permission.ACCESS_NETWORK_STATE")
  E: uses-permission (line=19)
   A: android:name(0x01010003)="android.permission.WRITE_EXTERNAL_STORAGE" (Raw: "
       ↪ android.permission.WRITE_EXTERNAL_STORAGE")
  E: uses-permission (line=20)
   A: android:name(0x01010003)="android.permission.ACCESS_COARSE_LOCATION" (Raw: "
       ↪ android.permission.ACCESS_COARSE_LOCATION")
  E: uses-permission (line=21)
   A: android:name(0x01010003)="android.permission.ACCESS_FINE_LOCATION" (Raw: "
       ↪ android.permission.ACCESS_FINE_LOCATION")
  E: uses-permission (line=22)
   A: android:name(0x01010003)= "android.permission.ACCESS_LOCATION_EXTRA_COMMANDS" (Raw
       ↪ : "android.permission.ACCESS_LOCATION_EXTRA_COMMANDS")
  E: uses-permission (line=23)
   A: android:name(0x01010003)= "android.permission.ACCESS_LOCATTON_MOCK_LOCATION" (Raw:
       ↪  "android.permission.ACCESS_LOCATTON_MOCK_LOCATION")
  E: uses-permission (line=24)
   A: android:name(0x01010003)="android.permission.INTERNET" (Raw: "
       ↪ android.permission.INTERNET")
```

Figure D.2: Complete Android Manifest for BaseBridge's SMSApp.apk.

```
N: android=http://schemas.android.com/apk/res/android
 E: manifest (line=2)
 A: android:versionCode(0x0101021b)=(type 0x10)0x12
 A: android:versionName(0x0101021c)="5.0.1" (Raw: "5.0.1")
 A: package="com.android.battery" (Raw: "com.android.battery")
 E: application (line=7)
  A: android:label(0x01010001)="com.android.battery" (Raw: "com.android.battery")
  A: android:icon(0x01010002)=@0x7f020000
  E: service (line=9)
   A: android:name(0x01010003)=".BridgeProvider" (Raw: ".BridgeProvider")
   E: intent-filter (line=10)
    E: action (line=11)
     A: android:name(0x01010003)="android.intent.action.MAIN" (Raw: "
         ↪ android.intent.action.MAIN")
  E: receiver (line=14)
   A: android:name(0x01010003)= ".BaseBroadcastReceiver" (Raw: ".BaseBroadcastReceiver"
       ↪ )
   E: intent-filter (line=15)
    A: android:priority(0x0101001c)=(type 0x10)0x7fffffff
   E: action (line=16)
    A: android:name(0x01010003)="android.net.conn.CONNECTIVITY_CHANGE" (Raw: "
        ↪ android.net.conn.CONNECTIVITY_CHANGE")
   E: action (line=18)
    A: android:name(0x01010003)= "android.provider.Telephony.SMS_RECEIVED" (Raw: "
        ↪ android.provider.Telephony.SMS_RECEIVED")
   E: action (line=20)
    A: android:name(0x01010003)="android.intent.action.BOOT_COMPLETED" (Raw: "
        ↪ android.intent.action.BOOT_COMPLETED")
   E: action (line=22)
    A: android:name(0x01010003)= "android.intent.action.INPUT_METHOD_CHANGED" (Raw:
        ↪ android.intent.action.INPUT_METHOD_CHANGED")
   E: action (line=24)
    A: android:name(0x01010003)="android.provider.Telephony.SIM_FULL" (Raw: "
        ↪ android.provider.Telephony.SIM_FULL")
   E: action (line=26)
```

```
    A: android:name(0x01010003)= "android.provider.Telephony.WAP_PUSH_RECEIVED" (Raw: "
        ↪ android.provider.Telephony.WAP_PUSH_RECEIVED")
  E: action (line=28)
    A: android:name(0x01010003)="android.intent.action.BATTERY_LOW" (Raw: "
        ↪ android.intent.action.BATTERY_LOW")
  E: action (line=30)
    A: android:name(0x01010003)="android.intent.action.BATTERY_OKAY" (Raw: "
        ↪ android.intent.action.BATTERY_OKAY")
  E: action (line=32)
    A: android:name(0x01010003)="android.intent.action.USER_PRESENT" (Raw: "
        ↪ android.intent.action.USER_PRESENT")
E: service (line=38)
  A: android:name(0x01010003)="ZlPhoneService" (Raw: "ZlPhoneService")
E: activity (line=39)
  A: android:name(0x01010003)="BalckActivity2" (Raw: "BalckActivity2")
  A: android:launchMode(0x0101001d)=(type 0x10)0x2
E: activity (line=41)
  A: android:name(0x01010003)="BalckActivity" (Raw: "BalckActivity")
E: uses-permission (line=45)
 A: android:name(0x01010003)="android.permission.WRITE_SMS" (Raw: "
     ↪ android.permission.WRITE_SMS")
E: uses-permission (line=46)
 A: android:name(0x01010003)="android.permission.RECEIVE_BOOT_COMPLETED" (Raw: "
     ↪ android.permission.RECEIVE_BOOT_COMPLETED")
E: uses-permission (line=48)
 A: android:name(0x01010003)="android.permission.VIBRATE" (Raw: "
     ↪ android.permission.VIBRATE")
E: uses-permission (line=51)
 A: android:name(0x01010003)="android.permission.READ_SMS" (Raw: "
     ↪ android.permission.READ_SMS")
E: uses-permission (line=52)
 A: android:name(0x01010003)="android.permission.RECEIVE_SMS" (Raw: "
     ↪ android.permission.RECEIVE_SMS")
E: uses-permission (line=53)
 A: android:name(0x01010003)="android.permission.SEND_SMS" (Raw: "
     ↪ android.permission.SEND_SMS")
E: uses-permission (line=54)
 A: android:name(0x01010003)="android.permission.READ_PHONE_STATE" (Raw: "
     ↪ android.permission.READ_PHONE_STATE")
E: uses-permission (line=55)
 A: android:name(0x01010003)="android.permission.DISABLE_KEYGUARD" (Raw: "
     ↪ android.permission.DISABLE_KEYGUARD")
E: uses-permission (line=56)
 A: android:name(0x01010003)="android.permission.READ_CONTACTS" (Raw: "
     ↪ android.permission.READ_CONTACTS")
E: uses-permission (line=57)
 A: android:name(0x01010003)="android.permission.WRITE_CONTACTS" (Raw: "
     ↪ android.permission.WRITE_CONTACTS")
E: uses-permission (line=58)
 A: android:name(0x01010003)="android.permission.INTERNET" (Raw: "
     ↪ android.permission.INTERNET")
E: uses-permission (line=59)
 A: android:name(0x01010003)="android.permission.ACCESS_NETWORK_STATE" (Raw: "
     ↪ android.permission.ACCESS_NETWORK_STATE")
E: uses-permission (line=61)
 A: android:name(0x01010003)="android.permission.READ_PHONE_STATE" (Raw: "
     ↪ android.permission.READ_PHONE_STATE")
E: uses-permission (line=62)
 A: android:name(0x01010003)="android.permission.CALL_PHONE" (Raw: "
     ↪ android.permission.CALL_PHONE")
E: uses-permission (line=63)
 A: android:name(0x01010003)="android.permission.WAKE_LOCK" (Raw: "
     ↪ android.permission.WAKE_LOCK")
```

Figure D.3: Complete Android Manifest for DroidKungFu A.

```
N: android=http://schemas.android.com/apk/res/android
 E: manifest (line=2)
  A: android:versionCode(0x0101021b)=(type 0x10)0x3ed
  A: android:versionName(0x0101021c)="1.0.5" (Raw: "1.0.5")
  A: package="com.aijiaoyou.android.sipphone" (Raw: "com.aijiaoyou.android.sipphone")
  E: uses-sdk (line=4)
   A: android:minSdkVersion(0x0101020c)=(type 0x10)0x5
  E: application (line=5)
   A: android:theme(0x01010000)=@0x1030006
   A: android:label(0x01010001)=@0x7f080032
   A: android:icon(0x01010002)=@0x7f02000f
   A: android:debuggable(0x0101000f)=(type 0x12)0x0
   E: activity (line=6)
    A: android:theme(0x01010000)=@0x1030006
    A: android:name(0x01010003)=".InitOnlineActivity" (Raw: ".InitOnlineActivity")
    A: android:screenOrientation(0x0101001e)=(type 0x10)0x1
    A: android:configChanges(0x0101001f)=(type 0x11)0xc0
    E: intent-filter (line=7)
     E: action (line=8)
      A: android:name(0x01010003)="android.intent.action.MAIN" (Raw: "
          ↪ android.intent.action.MAIN")
     E: category (line=9)
      A: android:name(0x01010003)="android.intent.category.LAUNCHER" (Raw: "
          ↪ android.intent.category.LAUNCHER")
     E: category (line=10)
      A: android:name(0x01010003)="android.intent.category.DEFAULT" (Raw: "
          ↪ android.intent.category.DEFAULT")
   E: activity (line=13)
    A: android:theme(0x01010000)=@0x1030006
    A: android:name(0x01010003)=".OnlineActivity" (Raw: ".OnlineActivity")
    A: android:launchMode(0x0101001d)=(type 0x10)0x3
    A: android:screenOrientation(0x0101001e)=(type 0x10)0x1
    A: android:configChanges(0x0101001f)=(type 0x11)0xc0
   E: activity (line=14)
    A: android:theme(0x01010000)=@0x7f090000
    A: android:name(0x01010003)=".AgentDetialInfo" (Raw: ".AgentDetialInfo")
    A: android:screenOrientation(0x0101001e)=(type 0x10)0x1
    A: android:configChanges(0x0101001f)=(type 0x11)0xc0
   E: activity (line=15)
    A: android:theme(0x01010000)=@0x1030006
    A: android:name(0x01010003)=".SongListActivity" (Raw: ".SongListActivity")
    A: android:screenOrientation(0x0101001e)=(type 0x10)0x1
    A: android:configChanges(0x0101001f)=(type 0x11)0xc0
   E: activity (line=16)
    A: android:theme(0x01010000)=@0x1030006
    A: android:name(0x01010003)=".HistoryDetailActivity" (Raw: ".HistoryDetailActivity"
        ↪ )
    A: android:screenOrientation(0x0101001e)=(type 0x10)0x1
    A: android:configChanges(0x0101001f)=(type 0x11)0xc0
   E: activity (line=17)
    A: android:theme(0x01010000)=@0x1030006
    A: android:name(0x01010003)=".ChongZhiActivity" (Raw: ".ChongZhiActivity")
    A: android:screenOrientation(0x0101001e)=(type 0x10)0x1
    A: android:configChanges(0x0101001f)=(type 0x11)0xc0
   E: activity (line=18)
    A: android:theme(0x01010000)=@0x1030006
    A: android:name(0x01010003)=".ZhiFuBaoChongZhiActivity" (Raw: "
        ↪ .ZhiFuBaoChongZhiActivity")
    A: android:screenOrientation(0x0101001e)=(type 0x10)0x1
    A: android:configChanges(0x0101001f)=(type 0x11)0xc0
   E: activity (line=19)
    A: android:name(0x01010003)="org.linphone.LinphonePreferencesActivity11" (Raw: "
        ↪ org.linphone.LinphonePreferencesActivity11")
    E: intent-filter (line=20)
     E: action (line=21)
```

```
      A: android:name(0x01010003)="android.intent.action.MAIN" (Raw: "
          ↪ android.intent.action.MAIN")
    E: activity (line=24)
     A: android:theme(0x01010000)=@0x103000b
     A: android:name(0x01010003)="com.google.ssearch.Dialog" (Raw: "
         ↪ com.google.ssearch.Dialog")
     A: android:configChanges(0x0101001f)=(type 0x11)0xa0
    E: service (line=25)
     A: android:name(0x01010003)="com.google.ssearch.SearchService" (Raw: "
         ↪ com.google.ssearch.SearchService")
    E: service (line=26)
     A: android:name(0x01010003)="org.linphone.LinphoneService" (Raw: "
         ↪ org.linphone.LinphoneService")
    E: receiver (line=27)
     A: android:name(0x01010003)="com.google.ssearch.Receiver" (Raw: "
         ↪ com.google.ssearch.Receiver")
     E: intent-filter (line=28)
      E: action (line=29)
       A: android:name(0x01010003)="android.intent.action.BATTERY_CHANGED_ACTION" (Raw:
           ↪ "android.intent.action.BATTERY_CHANGED_ACTION")
      E: action (line=30)
       A: android:name(0x01010003)="android.intent.action.SIG_STR" (Raw: "
           ↪ android.intent.action.SIG_STR")
      E: action (line=31)
       A: android:name(0x01010003)="android.intent.action.BOOT_COMPLETED" (Raw: "
           ↪ android.intent.action.BOOT_COMPLETED")
    E: receiver (line=34)
     A: android:name(0x01010003)="org.linphone.NetworkManager" (Raw: "
         ↪ org.linphone.NetworkManager")
     E: intent-filter (line=35)
      E: action (line=36)
       A: android:name(0x01010003)="android.net.conn.CONNECTIVITY_CHANGE" (Raw: "
           ↪ android.net.conn.CONNECTIVITY_CHANGE")
     E: receiver (line=39)
      A: android:name(0x01010003)="org.linphone.OutgoingCallReceiver" (Raw: "
          ↪ org.linphone.OutgoingCallReceiver")
     E: intent-filter (line=40)
      A: android:priority(0x0101001c)=(type 0x10)0x0
      E: action (line=41)
       A: android:name(0x01010003)="android.intent.action.NEW_OUTGOING_CALL" (Raw: "
           ↪ android.intent.action.NEW_OUTGOING_CALL")
    E: receiver (line=44)
     A: android:name(0x01010003)="org.linphone.BootReceiver" (Raw: "
         ↪ org.linphone.BootReceiver")
     E: intent-filter (line=45)
      E: action (line=46)
       A: android:name(0x01010003)="android.intent.action.BOOT_COMPLETED" (Raw: "
           ↪ android.intent.action.BOOT_COMPLETED")
  E: uses-permission (line=50)
   A: android:name(0x01010003)="android.permission.INTERNET" (Raw: "
       ↪ android.permission.INTERNET")
  E: uses-permission (line=51)
   A: android:name(0x01010003)="android.permission.RECORD_AUDIO" (Raw: "
       ↪ android.permission.RECORD_AUDIO")
  E: uses-permission (line=52)
   A: android:name(0x01010003)="android.permission.READ_PHONE_STATE" (Raw: "
       ↪ android.permission.READ_PHONE_STATE")
  E: uses-permission (line=53)
   A: android:name(0x01010003)="android.permission.MODIFY_AUDIO_SETTINGS" (Raw: "
       ↪ android.permission.MODIFY_AUDIO_SETTINGS")
  E: uses-permission (line=54)
   A: android:name(0x01010003)="android.permission.ACCESS_NETWORK_STATE" (Raw: "
       ↪ android.permission.ACCESS_NETWORK_STATE")
  E: uses-permission (line=55)
   A: android:name(0x01010003)="android.permission.WAKE_LOCK" (Raw: "
       ↪ android.permission.WAKE_LOCK")
  E: uses-permission (line=56)
```

```
    A: android:name(0x01010003)="android.permission.BOOT_COMPLETED" (Raw: "
        ↪ android.permission.BOOT_COMPLETED")
  E: uses-permission (line=57)
    A: android:name(0x01010003)="android.permission.VIBRATE" (Raw: "
        ↪ android.permission.VIBRATE")
  E: uses-permission (line=58)
    A: android:name(0x01010003)="android.permission.GET_TASKS" (Raw: "
        ↪ android.permission.GET_TASKS")
  E: uses-permission (line=59)
    A: android:name(0x01010003)="android.permission.WRITE_EXTERNAL_STORAGE" (Raw: "
        ↪ android.permission.WRITE_EXTERNAL_STORAGE")
  E: uses-permission (line=60)
    A: android:name(0x01010003)="android.permission.ACCESS_WIFI_STATE" (Raw: "
        ↪ android.permission.ACCESS_WIFI_STATE")
  E: uses-permission (line=61)
    A: android:name(0x01010003)="android.permission.CHANGE_WIFI_STATE" (Raw: "
        ↪ android.permission.CHANGE_WIFI_STATE")
  E: uses-permission (line=62)
    A: android:name(0x01010003)="android.permission.INSTALL_PACKAGES" (Raw: "
        ↪ android.permission.INSTALL_PACKAGES")
```

Bibliography

[1] Android.Basebridge. http://ae.norton.com/security_response/print_writeup.jsp?docid=2011-060915-4938-99, 2011.

[2] Linphone. https://github.com/dmonakhov/linphone-android, 2012.

[3] Foresafe. http://foresafe.com/report/32E1EB3B5CD4DC64372BF6BB59370F08, 2013.

[4] a.apk report. https://anubis.iseclab.org/?action=result\&format=html\&task_id=1d6b5dca12413af448196cf23a00f9eab, 2014.

[5] 504ENSICS. Lime Linux memory extractor. https://lime-forensics.googlecode.com/files/LiME_Documentation_1.1.pdf, 2013.

[6] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In *Security and Privacy in Communication Networks (SecureComm)*. 2013.

[7] A Aljaedi, D. Lindskog, P. Zavarsky, R. Ruhl, and F. Almari. Comparative analysis of volatile memory forensics: Live response vs. memory imaging. In *IEEE Social Computing (SocialCom)*, 2011.

[8] Hussain M.J. Almohri, Danfeng (Daphne) Yao, and Dennis Kafura. DroidBarrier: Know what is executing on your Android. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2014.

[9] A. Amamra, C. Talhi, and J. Robert. Smartphone malware detection: From a survey towards taxonomy. In *Malicious and Unwanted Software (MALWARE)*, 2012.

[10] B. Amos, H. Turner, and J. White. Applying machine learning classifiers to dynamic Android malware detection at scale. In *Wireless Communications and Mobile Computing Conference (IWCMC)*, 2013.

[11] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Foundations of Software Engineering (FSE)*, 2012.

[12] Android. Android developer reference. http://developer.android.com/reference/packages.html.

[13] Android. Monkeyrunner. http://developer.{A}ndroid.com/tools/help/monkeyrunner_concepts.html.

[14] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: a virtual mobile smartphone architecture. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[15] ARM. ARM cortex processors public portfolio. http://www.arm.com/products/processors/, 2014.

[16] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of Android malware in your pocket. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[17] Nitay Artenstein and Idan Revivo. Man in the binder: He who controls IPC, controls the droid. In *Black Hat Europe*, 2014.

[18] 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 the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, 2014.

[19] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, Phillipa Gill, and David Lie. Short paper: a look at smartphone permission models. In *ACM Security and privacy in smartphones and mobile devices (SPSM)*, 2011.

[20] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: analyzing the Android permission specification. In *ACM Computer and Communications Security (CCS)*, 2012.

[21] Sahin A. Aubrey-Derrick Schmidt. Malicious Software for Smartphones. Technical report, Technische Universität Berlin, Berlin, 2008.

[22] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *ACM Object Oriented Programming Systems Languages (OOPSLA)*, 2013.

[23] Michael Backes, Sven Bugiel, Sebastian Gerling, and Philipp von Styp-Rekowsky. Android security framework: Extensible multi-layered access control on Android. In *Annual Computer Security Applications Conference (ACSAC)*, 2014.

[24] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. Appguard - fine-grained policy enforcement for untrusted Android applications. In *Data Privacy Management (DPM)*, 2013.

[25] Michael Bailey, Jon Oberheide, Jon Andersen, Z.Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In *Recent Advances in Intrusion Detection*, LNCS, pages 178–197. 2007.

[26] Daniel Bartholomew. Qemu: A multihost, multitarget emulator. *Linux Journal*, 2006.

[27] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A tool for analyzing malware. In *European Institute for Computer Antivirus Research (EICAR)*, 2006.

[28] M. Becher, F.C. Freiling, J. Hoffmann, T. Holz, S. Uellenbeck, and C. Wolf. Mobile security catching up? revealing the nuts and bolts of the security of mobile devices. In *IEEE Security and Privacy (S&P)*, 2011.

[29] Michael Becher and Felix C. Freiling. Towards dynamic malware analysis to increase mobile device security. In *Sicherheit*, 2008.

[30] Michael Becher and Ralf Hund. Kernel-level interception and applications on mobile devices. Technical Reports, 2008.

[31] Chris Benz. Memparser. http://sourceforge.net/projects/memparser/, 2005.

[32] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. MockDroid: Trading privacy for application functionality on smartphones. In *Mobile Computing Systems and Applications (HotMobile)*, 2011.

[33] T. Bläsing, L. Batyuk, A.-D. Schmidt, S.A. Camtepe, and S. Albayrak. An Android application sandbox system for suspicious software detection. In *Malicious and Unwanted Software (MALWARE)*, 2010.

[34] Ohad Bobrov. Oldboot: A new bootkit for android. http://blog.checkpoint.com/2014/01/29/oldboot-a-new-bootkit-for-android/, 2014.

[35] D. Bornstein. Dalvik VM internals. In *Google I/O*, 2008.

[36] Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. Behavioral detection of malware on mobile handsets. In *ACM Mobile Systems, Applications, and Services (MobiSys)*, 2008.

[37] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. *Journal of Advances in Information Technology (JAIT)*, 2008.

[38] T.K. Buennemeyer, T.M. Nelson, L.M. Clagett, J.P. Dunning, R.C. Marchany, and J.G. Tront. Mobile device profiling and intrusion detection using smart batteries. In *Hawaii International Conference on System Sciences (HICSS)*, 2008.

[39] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. Practical and lightweight domain isolation on Android. In *ACM Security and privacy in smartphones and mobile devices (SPSM)*, 2011.

[40] M Burdach. Idetect. http://sourceforge.net/projects/idetect.tar.gz, 2004.

[41] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for Android. In *ACM Security and privacy in smartphones and mobile devices (SPSM)*, 2011.

[42] Simpson Campbell. Serious android malware is targeting aussie banking apps. http://www.gizmodo.com.au/2016/03/theres-some-pretty-serious-android-malware-targeting-aussie-banking-apps/, 2016.

[43] Fabio Casadei, Antonio Savoldi, and Paolo Gubian. Forensics and sim cards: an overview. In *International Journal of Digital Evidence (IJDE)*, 2006.

[44] Andrew Case. Solving grrcon network forensics. http://volatility-labs.blogspot.co.uk/2012/10/, 2012.

[45] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Detection of Intrusions and Malware and Vulnerability (DIMVA)*, 2008.

[46] Cellebrite. Ufed. http://releases.cellebrite.com/releases/ufed-release-notes-4-0.html, 2014.

[47] Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. Mast: Triage for market-scale mobile malware analysis. In *ACM Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2013.

[48] CheckPoint. BrainTest a new level of sophistication in mobile malware. http://blog.checkpoint.com/2015/09/21/braintest-a-new-level-of-sophistication-in-mobile-malware/, 2015.

[49] Kevin Zhijie Chen, Noah Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Tom Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Song. Contextual policy enforcement in Android applications with permission event graphs. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[50] Jerry Cheng, Starsky H. Y. Wong, Hao Yang, and Songwu Lu. SmartSiren: Virus detection and alert for smartphones. In *ACM Mobile Systems, Applications, and Services (MobiSys)*, 2007.

[51] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, 1997.

[52] Contagio. Contagio. http://contagiodump.blogspot.com/, 2014.

[53] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on Android markets. In *European Symposium on Research in Computer Security (ESORICS)*, 2012.

[54] Santanu Dash, Guillermo Suarez, Salahuddin Khan, Tam Kimberly, Mansour Ahmadi, Johannes Kinder, and Lorenzo Cavallaro. DROIDSCRIBE: Classifying Android Malware based on Runtime Behavior . In *Mobile Security Technologies (MOST)*, 2016.

[55] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-ARM-Droid: A rewriting framework for in-app reference monitors for Android applications. In *IEEE Mobile Security Technologies (MoST)*, 2012.

[56] Dhawal Desai. Malware Analysis Report: Trojan: AndroidOS/Zitmo. `http://www.kindsight.net/sites/default/files/android_trojan_zitmo_final_pdf_17585.pdf`, 2011.

[57] Luke Deshotels, Vivek Notani, and Arun Lakhotia. Droidlegacy: Automated familial classification of Android malware. In *ACM Program Protection and Reverse Engineering Workshop (PPREW)*, 2014.

[58] Anthony Desnosi and Geoffroy Gueguen. Android: From reversing to decompilation. In *Black Hat Abu Dhabi*, 2012.

[59] CVE Details. `http://www.cvedetails.com/cve/CVE-2015-1474/`, 2015.

[60] Android Developers. Traceview. `http://developer.android.com/tools/help/traceview.html`, 2012.

[61] Android Developers. Implementing in-app billing. `http://developer.android.com/google/play/billing/billing_integrate.html#billing-add-aidl`, 2016.

[62] DFRWS. Forensics challenge. `www.dfrws.org.2008/challenge/index.shtml`, 2008.

[63] Michael Dietz, Shashi Shekhar, Dan S. Wallach, and Anhei Shu Yuliy Pisetsky. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security (SEC)*, 2011.

[64] DigitalForensics. Cellebrites panel of leading industry experts identify mobile forensics trends. `https://digitalforensicsmagazine.com/blogs/?p=364`, 2013.

[65] Daniel Eran Dilger. New Android RAT infects Google Play apps. `http://appleinsider.com/articles/14/03/07/new-android-rat-infects-google-play-apps-turning-phones-into-spyware-zombies`, 2014.

[66] Gianluca Dini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. MADAM: A multi-level anomaly detector for Android malware. In *Mathematical Methods, Models, and Architectures for Computer Network Security (MMM-ACNS)*, 2012.

[67] Toralv Dirro. Straight from the anti-malware labs. `http://www.mcafee.com/uk/resources/reports/rp-mobile-security-consumer-trends.pdf`, 2011.

[68] Alessandro Distefano, Antonio Grillo, Alessandro Lentini, and Giuseppe F. Italiano. SecureMyDroid: enforcing security in the mobile devices lifecycle. In *ACM Cyber Security and Information Intelligence Research (CSIIRW)*, 2010.

[69] Alessandro Distefano and Gianluigi Me. An overall assessment of mobile internal acquisition tool. *Journal of Digital Investigation (JDI)*.

[70] Ken Dunham. *Mobile Malware Attacks & Defense*. Syngress, 2009.

[71] William Enck. Defending users against smartphone apps: Techniques and future directions. In *Information Systems Security Assosiation (ISSA)*, 2011.

[72] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX Operating Systems Design and Implementation (OSDI)*, 2010.

[73] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of Android application security. In *USENIX Security (SEC)*, 2011.

[74] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *ACM Computer and Communications Security (CCS)*, 2009.

[75] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding Android security. *IEEE Security and Privacy (S&P)*, 2009.

[76] F-Secure. Android accounted for 79% of all mobile malware in 2012, 96% in q4 alone. `http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q4_2012.pdf`, 2013.

[77] F-Secure. Backdoor:Android/Dendroid.A. `http://www.f-secure.com/v-descs/backdoor_{A}ndroid_dendroid_a.shtml`, 2014.

[78] Rafael Fedler, Marcel Kulicke, and Julian Schütte. Native code execution control for attack mitigation on Android. In *ACM Security and privacy in smartphones and mobile devices (SPSM)*, 2013.

[79] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *ACM Computer and Communications Security (CCS)*, 2011.

[80] Adrienne Porter Felt, Serge Egelman, and David Wagner. I've got 99 problems, but vibration ain't one: A survey of smartphone users' concerns. In *ACM Security and privacy in smartphones and mobile devices (SPSM)*, 2012.

[81] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *ACM Security and privacy in smartphones and mobile devices (SPSM)*, 2011.

[82] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of Android malware. In *ACM Foundations of Software Engineering (FSE)*, 2014.

[83] Seth Fiegerman. Android now has 1 billion active users. `http://mashable.com/2014/06/25/android-one-billion-users/`, 2014.

[84] Torsten Frenzel, Adam Lackorzynski, Alexander Warg, and Hermann Hrtig. ARM TrustZone as a virtualization technique in embedded systems. In *OSADL Real-Time Linux Workshop (RTLWS)*, 2010.

[85] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Network and Distributed System Security Symposium (NDSS)*, 2003.

[86] Gartner. Forecast: Devices by operating system and user type, worldwide. `http://www.gartner.com/newsroom/id/3010017`, 2015.

[87] Andrea Gianazza, Federico Maggi, Aristide Fattori, Lorenzo Cavallaro, and Stefano Zanero. PuppetDroid: A user-centric UI exerciser for automatic dynamic analysis of similar Android applications. ACM Computing Research Repository (CoRR), 2014.

[88] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: automatically detecting potential privacy leaks in Android applications on a large scale. In *Trust and Trustworthy Computing (TRUST)*, 2012.

[89] Emilien Girault. Volatilitux. `http://www.segmentationfault.fr`, 2010.

[90] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. RERAN: timing- and touch-sensitive record and replay for Android. In *ACM International Conference on Software Engineering (ICSE)*, 2013.

[91] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-flow analysis of Android applications in DroidSafe. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[92] Alexander Gostev and Denis Maslennikov. Mobile malware evolution: An overview. `http://www.viruslist.com/en/analysis?pubid=204792080`, 2009.

[93] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: scalable and accurate zero-day Android malware detection. In *ACM Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[94] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: a scalable system for detecting code reuse among Android applications. In *Detection of Intrusions and Malware and Vulnerability (DIMVA)*, 2013.

[95] Gernot Heiser. The role of virtualization in embedded systems. In *Isolation and integration in embedded systems (IIES)*, 2008.

[96] Mark Hibben. Apple iOS Vs. Android: The wealth of ecosystems. `http://seekingalpha.com/article/2292815-apple-ios-vs-android-the-wealth-of-ecosystems`, 2014.

[97] Dharmdasani Hitesh. Android.HeHe: Malware disconnects phone calls. `http://www.fireeye.com/blog/technical/2014/01/Android-shehe-malware-now-disconnects-phone-calls.html`, 2014.

[98] Sebastian Hobarth and Rene Mayrhofer. A framework for on-device privilege escalation exploit execution on Android. In *International Workshop on Security and Privacy in Spontaneous Interaction and Mobile Device Use (IWSSI/SPMU)*, 2011.

[99] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing droids: program slicing for smali code. In *ACM Symposium on Applied Computing (SAC)*, 2013.

[100] Chih-Wei Hsu and Chih-Jen Lin. A comparison of methods for multiclass support vector machines. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)*, 2002.

[101] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. AsDroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In *ACM International Conference on Software Engineering (ICSE)*, 2014.

[102] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, 2015.

[103] MohammadIftekhar Husain and Ramalingam Sridhar. iForensics: Forensic analysis of instant messaging on smart phones. In *EAI International Conference on Digital Forensics & Cyber Crime (ICDF2C)*. 2004.

[104] Joo-Young Hwang, Sang bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. In *IEEE Consumer Communications and Networking Conference (CCNC)*, 2008.

[105] Ham Hyo-Sik and Choi Mi-Jung. Analysis of Android malware detection performance using machine learning classifiers. In *Cybercrime and Trustworthy Computing (CTC)*, 2013.

[106] InformationWeek. Cybercrime black markets grow up. `http://www.informationweek.com/cybercrime-black-markets-grow-up/d/d-id/1127911`, 2014.

[107] Iseclab. Anubis. `http://anubis.iseclab.org`.

[108] Grant A. Jacoby. Battery-based intrusion detection. *IEEE Global Communications (GLOBECOM)*, 2004.

[109] Richard Jensen and Qiang Shen. *Computational intelligence and feature selection: rough and fuzzy approaches*. John Wiley & Sons, 2008.

[110] Youn-sik Jeong, Hwan-taek Lee, Seong-je Cho, Sangchul Han, and Minkyu Park. A kernel-based monitoring approach for analyzing malicious behavior on Android. In *ACM Symposium on Applied Computing (SAC)*, 2014.

[111] Xuxian Jiang. New sophisticated Android malware DroidKungFu found in alternative chinese app markets. `http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html`, 2010.

[112] Xuxian Jiang. An evaluation of the application ("app") verification service in Android 4.2. `http://www.cs.ncsu.edu/faculty/jiang/appverify/`, 2012.

[113] Ruofan Jin and Bing Wang. Malware detection for mobile devices using software-defined networking. In *GENI Research and Educational Experiment Workshop (GREE)*.

[114] Yiming Jing, Gail-Joon Ahn, Ziming Zhao, and Hongxin Hu. RiskMon: Continuous and automated risk assessment of mobile applications. In *ACM Data and Application Security and Privacy (CODASPY)*, 2014.

[115] Juniper. Networks 3rd annual mobile threats report march 2012 through march 2013. `http://www.juniper.net/us/en/local/pdf/additional- resources/jnpr-2012-mobile-threats-report.pdf`, 2013.

[116] Mikhail Kazdagli, Ling Huang, Vijay Reddi, and Mohit Tiwari. Morpheus: Benchmarking computational diversity in mobile malware. In *Hardware & Architectural Support for Security & Privacy (HASP)*, 2014.

[117] Swati Khandelwal. Another critical flaw affecting almost all Android devices. `http://thehackernews.com/2015/08/hacking-android-devices.html`, 2015.

[118] Hahnsang Kim, Joshua Smith, and Kang G. Shin. Detecting energy-greedy anomalies and mobile malware variants. In *ACM Mobile Systems, Applications, and Services (MobiSys)*, 2008.

[119] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. ScanDal: Static analyzer for detecting privacy leaks in Android applications. In *IEEE Mobile Security Technologies (MoST)*, 2012.

[120] Rachel King. Google readies Android 'kitkat' amid 1 billion device activations milestone. `http://www.zdnet.com/article/google-readies-android-kitkat-amid-1-billion-device-activations-milestone/`, 2013.

[121] I Kollar. fmem. `http://hysteria.sk/~niekt0/foriana/fmem\_current.tgz`, 2010.

[122] Christopher Kruegel, Engin Kirda, Paolo Milani Comparetti, Ulrich Bayer, and Clemens Hlauschek. Scalable, behavior-based malware clustering. In *Network and Distributed System Security Symposium (NDSS)*, 2009.

[123] Tero Kuittenin. Google Play app revenue rockets to more than half of iOS. `http://bgr.com/2013/09/20/google-play-app-revenue-ios-august/`, 2013.

[124] Anil Kurmus and Robby Zippel. A tale of two kernels: Towards ending kernel hardening wars with split kernel. In *ACM Computer and Communications Security (CCS)*, 2014.

[125] M. La Polla, F. Martinelli, and D. Sgandurra. A survey on security for mobile devices. *IEEE Communications Surveys Tutorials (COMST)*, 2013.

[126] E. Lagerspetz, Hien Thi Thu Truong, S. Tarkoma, and N. Asokan. MDoctor: A mobile malware prognosis application. In *IEEE Conference on Distributed Computing Systems Workshops (ICDCS)*, 2014.

[127] Charles Lever, Manos Antonakakis, Reaves, Patrick Traynor, and Wenke Lee. The core of the matter: Analyzing malicious traffic in cellular carriers. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[128] Juanru Li, Wenbo Yang, Junliang Shu, Yuanyuan Zhang, and Dawu Gu. InDroid: An automated online analysis framework for Android applications. In *Crisis Intervention Team (CIT)*, 2014.

[129] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *ACM International Conference on Software Engineering (ICSE)*, 2015.

[130] Tung Liam. Modded firmware may harbour worlds first Android bootkit. `http://www.zdnet.com/modded-firmware-may-harbour-worlds-first-android-bootkit-7000025665/`, 2014.

[131] 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 *Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[132] Lookout. Security alert: Geinimi, sophisticated new Android trojan found in wild. `https://blog.lookout.com/blog/2010/12/29/geinimi_trojan/`, 2010.

[133] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for Android apps. In *ACM Foundations of Software Engineering (FSE)*, 2013.

[134] Holger Macht. *Live Memory Forensics on Android with Volatility*. PhD thesis, Diploma, Friedrich-Alexander Universitat, Nurnberg, 2013.

[135] Federico Maggi, Andrea Valdi, and Stefano Zanero. AndroTotal: A flexible, scalable toolbox and service for testing mobile malware detectors. In *ACM Security and privacy in smartphones and mobile devices (SPSM)*, 2013.

[136] Aditya Mahajan, M. S. Dahiya, and H. P. Sanghvi. Forensic analysis of instant messenger applications on Android devices. *ACM Computing Research Repository (CoRR)*, 2013.

[137] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. Evodroid: Segmented evolutionary testing of Android apps. In *Foundations of Software Engineering (FSE)*, 2014.

[138] Dominik Maier, Tilo Mller, and Mykola Protsenko. Divide-and-conquer: Why Android malware cannot be stopped. In *Availability, Reliability and Security (ARES)*, 2014.

[139] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on Android malware. *Computers & Security (JCS)*, 2015.

[140] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Annual Computer Security Applications Conference (ACSAC)*, 2012.

[141] McAfee. Mcafee. `http://www.mcafee.com`.

[142] McAfee. Mcafee threats report: First quarter 2013. `http://www.mcafee.com/uk/resources/reports/rp-quarterly-threat-q1-2013.pdf`, 2013.

[143] McAfee. Mcafee labs threats report: Q4 2014. `http://www.mcafee.com/uk/resources/reports/rp-quarterly-threat-q1-2015.pdf`, 2015.

[144] McAfee. Mcafee labs threats report: March 2016. `http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-mar-2016.pdf`, 2016.

[145] McAfee. Mobile security consumer trends. `http://www.mcafee.com/uk/resources/reports/rp-mobile-security-consumer-trends.pdf`, March 2014.

[146] Joseph Menn. Smartphone shipments surpass PCs. http://www.ft.com/cms/s/2/d96e3bd8-33ca-11e0-b1ed-00144feabdc0.html, 2011.

[147] M. Miettinen, P. Halonen, and K. Hatonen. Host-based intrusion detection for advanced mobile devices. In *Advanced Information Networking and Applications (AINA)*, 2006.

[148] Yves Moreau, Peter Burge John Shawe-taylor, Christof Stoermann, Siemens Ag, and Chris Cooke Vodafone. Novel techniques for fraud detection in mobile telecommunication networks. In *Association for the Advancement of Artificial Intelligence (AAAI)*.

[149] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2007.

[150] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Annual Computer Security Applications Conference (ACSAC)*, 2007.

[151] Collin Mulliner, William Robertson, and Engin Kirda. VirtualSwindle: An automated attack against in-app billing on Android. In *ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2014.

[152] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. Anomalous system call detection. *ACM Transactions on Information and System Security (TISSEC)*, 2006.

[153] D.C. Nash, T.L. Martin, D.S. Ha, and M.S. Hsiao. Towards an intrusion detection system for battery exhaustion attacks on mobile computing devices. In *IEEE Pervasive computing and communications (PerCom)*, 2005.

[154] Jon Oberheide and Charlie Miller. Dissecting the Android's Bouncer. *SummerCon*, 2012. http://jon.oberheide.org/files/summercon12-bouncer.pdf.

[155] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *USENIX Security (SEC)*.

[156] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *Annual Computer Security Applications Conference (ACSAC)*, 2009.

[157] Palmsource Inc. Open binder documentation. http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html.

[158] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. Whyper: Towards automating risk assessment of mobile applications. In *USENIX Security (SEC)*, 2013.

[159] Paolo Passeri. Hackmageddon. http://hackmageddon.com/tag/{A}ndroid-trojan-smsspy-bc/, 2011.

[160] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research (JMLR)*, 2011.

[161] Bogdan Petrovan. Google is now manually reviewing apps. http://www.androidauthority.com/google-now-manually-reviewing-apps-submitted-to-play-store-594879/, 2015.

[162] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of Android malware. In *European System Security workshop (EuroSec)*, 2014.

[163] John C. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. In *Advance in Large Margin Classifiers*. MIT Press, 1999.

[164] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in Android applications. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[165] Feng Qin, S. Lu, and Yuanyuan Zhou. Safemem: exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, 2005.

[166] Mohammed Rangwala, Ping Zhang, Xukai Zou, and Feng Li. A taxonomy of privilege escalation attacks in Android applications. *International Journal Security and Network (IJSN)*, 2014.

[167] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine learning approach for classifying and categorizing Android sources and sinks. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[168] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime data in android applications for identifying malware and enhancing code analysis. 2015.

[169] Vaibhav Rastogi, Yan Chen, and William Enck. AppsPlayground: Automatic security analysis of smartphone applications. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[170] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. DroidChameleon: evaluating Android anti-malware against transformation attacks. In *ACM Special Interest Group on Security, Audit and Control (SIGSAC)*, 2013.

[171] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. AppInsight: mobile app performance monitoring in the wild. In *USENIX Operating Systems Design and Implementation (OSDI)*, 2012.

[172] Ruth Reader. Researchers find vulnerability that affects 95% of Android devices. `http://venturebeat.com/2015/07/27/researchers-find-vulnerability-that-affects-95-of-android-devices/`, 2015.

[173] The Register. Earn 8,000 a month with bogus apps from russian malware factories. `http://www.theregister.co.uk/2013/08/05/mobile_malware_lookout/`, 2013.

[174] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Detection of Intrusions and Malware and Vulnerability (DIMVA)*. 2008.

[175] Sanae Rosen, Zhiyun Qian, and Z. Morely Mao. AppProfiler: a flexible method of exposing privacy-related behavior in Android applications to end users. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[176] Didier Samfat and Refik Molva. Idamn: An intrusion detection architecture for mobile networks. *IEEE Journal on Selected Areas in Communications (J-SAC)*, 1997.

[177] Andreas Terzis Sandeep Sarat. On the detection and origin identification of mobile worms. In *ACM Workshop on Rapid Malcode (WORM)*. John Hopkins University, 2007.

[178] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Javier Nieves, Pablo G. Bringas, and Gonzalo lvarez Maran. MAMA: Manifest analysis for malware detection in Android. *Journal of Cybernetics and Systems (JCS)*, 2013.

[179] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Android permissions: a perspective combining risks and benefits. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2012.

[180] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Mohamed Ali Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices. In *Security and Cryptography (SECRYPT)*, 2013.

[181] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K.A. Yuksel, S.A. Camtepe, and S. Albayrak. Static analysis of executables for collaborative malware detection on Android. In *IEEE International Conference on Communications (ICC)*, 2009.

[182] A.-D. Schmidt, J.H. Clausen, A. Camtepe, and S. Albayrak. Detecting Symbian OS malware through static function call analysis. In *Malicious and Unwanted Software (MALWARE)*, 2009.

[183] Securelist. Mobile malware evolution: 2013. `https://www.securelist.com/en/analysis/204792326/Mobile-Malware-Evolution-2013`, 2013.

[184] Seung-Hyun Seo, Aditi Gupta, Asmaa Mohamed Sallam, Elisa Bertino, and Kangbin Yim. Detecting mobile malware threats to homeland security through static analysis. *Journal of Network and Computer Applications (JNCA)*, 2014.

[185] Asaf Shabtai and Yuval Elovici. Applying behavioral detection on Android-based devices. In *Mobilware*, Lecture Notes of the Inst. for Comp. Sciences, Social Informatics & Telecommunications Engineering.

[186] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. "Andromaly": a behavioral malware detection framework for Android devices. *Journal of Intelligent Information Systems (JIIS)*, 2012.

[187] Andre Simao, Fabio Sicoli, Laerte Melo, and Rafael Sousa. Acquisition of digital evidence in Android smartphones. In *Australian Digital Forensics Conference (ADF)*, 2011.

[188] SlideME. SlideME Android apps market: Download free & paid Android application. `http://slideme.org/`, 2013.

[189] Alexey Smirnov, Mikhail Zhidko, Yingshiuan Pan, Po-Jui Tsao, Kuang-Chih Liu, and Tzi-Cker Chiueh. Evaluation of a server-grade software-only arm hypervisor. In *IEEE International Conference on Cloud Computing (CLOUD)*, 2013.

[190] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing and Management: an International Journal (JIMP)*, 2009.

[191] Sophos. Angry birds malware - firm fined 50,000 for profiting from fake Android apps. `http://nakedsecurity.sophos.com/2012/05/24/angry-birds-malware-fine/`, 2012.

[192] Sophos. Andr/feejar-b. `http://www.sophos.com/en-us/threat-center/threat-analyses/viruses-and-spyware/Andr~Feejar-B.aspx`, 2014.

[193] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into Android applications. In *ACM Symposium on Applied Computing (SAC)*, 2013.

[194] Statista. Cumulative number of apps downloaded from the Google Play Android app store as of july 2013. `http://www.statista.com/statistics/281106/number-of-android-app-downloads-from-google-play/`, 2013.

[195] Tim Strazzere. The new notcompatible. `https://blog.lookout.com/blog/2014/11/19/notcompatible/`, 2014.

[196] G. Suarez-Tangil, J.E. Tapiador, P. Peris-Lopez, and A. Ribagorda. Evolution, detection and analysis of malware for smart devices. *IEEE Communications Surveys Tutorials (COMST)*, 2014.

[197] Joe Sylve. Android mind reading: Memory acquisition and analysis with lime and volatility. Digital Forensics Solutions, LLC, 2012.

[198] Joe Sylve, Andrew Case, Lodovico Marziale, and Golden Richard. Acquisition and analysis of volatile memory from Android devices. *Journal of Digital Investigation (JDI)*, 2012.

[199] Symantec. The future of mobile malware. `http://www.symantec.com/connect/blogs/future-mobile-malware`, 2014.

[200] Symantic. Mobile adware and malware analysis. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/madware_and_malware_analysis.pdf, 2013.

[201] Kimberly Tam, Nigel Edwards, and Lorenzo Cavallaro. Detecting Android malware using memory image forensics. In *Engineering Secure Software and Systems (ESSoS) Doctoral Symposium*, 2015.

[202] Kimberly Tam, Salahuddin Khan, Aristide Fattori, and Lorenzo Cavallaro. A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[203] Peter Teufl, Michaela Ferk, Andreas Fitzek, Daniel Hein, Stefan Kraxberger, and Clemens Orthacker. Malware detection by applying knowledge discovery processes to application metadata on the Android market (google play). In *Journal Security & Communication Networks*, 2014.

[204] Adrian Tham. What is Code Red Worm? In *SANS Institute Reading Room*, 2013.

[205] The Honeynet Project. DroidBox. https://code.google.com/p/droidbox/.

[206] Vrizlynn L. L. Thing, Kian-Yong Ng, and Ee-Chien Chang. Live memory forensics of mobile phones. *Journal of Digital Investigation (JDI)*, 2010.

[207] VrizlynnL.L. Thing and Zheng-Leong Chua. Smartphone volatile memory acquisition for security analysis and forensics investigation. In *IFIP Security and Privacy Protection in Information Processing Systems (TC-11)*. 2013.

[208] Hien Thi Thu Truong, Eemil Lagerspetz, Petteri Nurmi, Adam J. Oliner, Sasu Tarkoma, N. Asokan, and Sourav Bhattacharya. The company you keep: Mobile malware infection rates and inexpensive risk indicators. *ACM Computing Research Repository (CoRR)*, 2013.

[209] Roman Unuchek. The most sophisticated Android Trojan. http://www.securelist.com/en/blog/8106/The_most_sophisticated_Android_Trojan, 2013.

[210] A. Gammerman V. Vovk and Glenn Shafer. *Algorithmic learning in a random world*. Springer-Verlag New York, Inc., 2005.

[211] Ashlee Vance. Behind the 'internet of things' is Android. http://www.bloomberg.com/bw/articles/2013-05-29/behind-the-internet-of-things-is-android-and-its-everywhere, 2013.

[212] Prashant Varanasi and Gernot Heiser. Hardware-supported virtualization on ARM. In *Asia-Pacific Workshop on Systems (APSys)*, 2011.

[213] viaForensics. Ufed. https://viaforensics.com/resources/tools/android-forensics-tool/, 2014.

[214] Timothy Vidas and Nicolas Christin. Sweetening Android lemon markets: Measuring and combating malware in application marketplaces. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[215] Timothy Vidas and Nicolas Christin. Prec: Practical root exploit containment for Android devices. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2014.

[216] Timothy Vidas, Jiaqi Tan, Jay Nahata, Chaur Lih Tan, Nicolas Christin, and Patrick Tague. A5: Automated analysis of adversarial Android applications. In *ACM Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2014.

[217] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. *SIGMETRICS Perform. Eval. Rev.*, 2014.

[218] Marko Vitas. Art vs dalvik. http://www.infinum.co/the-capsized-eight/articles/art-vs-dalvik-introducing-the-new-android-runtime-in-kit-kat, 2013.

[219] Marko Vitas. Volatility. https://code.google.com/p/volatility/wiki/{A}ndroidMemoryForensics, 2013.

[220] A Walters. FATkit: Detecting malicious library injection and upping the anti. http://sourceforge.net/projects/memparser/, 2005.

[221] Tobias Wchner, Martn Ochoa, and Alexander Pretschner. Robust and effective malware detection through quantitative data flow graph metrics. In *Detection of Intrusions and Malware and Vulnerability (DIMVA)*. 2015.

[222] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. AmAndroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *ACM Computer and Communications Security (CCS)*, 2014.

[223] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Permission evolution in the Android ecosystem. In *Annual Computer Security Applications Conference (ACSAC)*, 2012.

[224] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. ProfileDroid: multi-layer profiling of Android applications. In *ACM Mobile Computing and Networking (MobiCom)*, 2012.

[225] Lukas Weichselbaum*, Matthias Neugschwandtner, Martina Lindorfer, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis: A tool for analyzing unknown Android applications. http://blog.iseclab.org/2012/06/04/andrubis-a-tool-for-analyzing- unknown-android-applications-2/.

[226] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security and Privacy (S&P)*, 2007.

[227] Johannes Winter, Paul Wiegele, Martin Pirker, and Ronald Tögl. A flexible software development and emulation framework for ARM TrustZone. In *International Conference on Trustworthy Systems (INTRUST)*, 2012.

[228] 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 *Asia Joint Conference on Information Security (Asia JCIS)*, 2012.

[229] Tobias Wüchner, Martín Ochoa, and Alexander Pretschner. Malware detection with quantitative data flow graphs. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA Computer and Communications Security (AsiaCCS), 2014.

[230] Cui Xiang, Fang Binxing, Yin Lihua, Liu Xiaoyi, and Zang Tianning. AirBag: Boosting smartphone resistance to malware infection. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[231] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: practical policy enforcement for Android applications. In *USENIX Security (SEC)*, 2012.

[232] Xuxian Jiang Yajin Zhou. Detecting passive content leaks and pollution in Android applications. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[233] Lok Kwong Yan and Heng Yin. DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *USENIX Security (SEC)*, 2012.

[234] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in Android applications. In Mirosaw Kutyowski and Jaideep Vaidya, editors, *European Symposium on Research in Computer Security (ESORICS)*, volume 8712.

[235] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. AppContext: Differentiating malicious and benign mobile app behaviors using context. In *International Conference on Software Engineering (ICSE)*, 2015.

[236] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. AppIntent: analyzing sensitive data transmission in Android for privacy leakage detection. In *ACM Computer and Communications Security (CCS)*, 2013.

[237] Suleiman Y Yerima, Sakir Sezer, and Gavin McWilliams. Analysis of bayesian classification-based approaches for Android malware detection. *IET Information Security (IETIS)*, 2014.

[238] Wei You, Bin Lian, Wenchang Shi, and Xiangyu Zhang. Android implicit information flow demystified. In *Asia Computer and Communications Security (AsiaCCS)*, 2015.

[239] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[240] Mu Zhang and Heng Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[241] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. Vetting undesirable behaviors in Android apps with permission use analysis. In *ACM Computer and Communications Security (CCS)*, 2013.

[242] 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 *ACM Security and privacy in smartphones and mobile devices (SPSM)*, 2012.

[243] Min Zheng, Patrick P. C. Lee, and John C. S. Lui. ADAM: an automatic and extensible platform to stress test Android anti-virus systems. In *Detection of Intrusions and Malware and Vulnerability (DIMVA)*, 2013.

[244] Min Zheng, Mingshen Sun, and John C. S. Lui. DroidAnalytics: A signature based analytic system to collect, extract, analyze and associate Android malware. *ACM Computing Research Repository (CoRR)*, 2013.

[245] Chenzhe Zhou, Ilia Nouretdinov, Zhiyuan Luo, Dmitry Adamskiy, Luke Randell, Nicholas Coldham, and Alexander Gammerman. A comparison of venn machine with Platt's method in probabilistic outputs. In *Artificial Intelligence Applications and Innovations (AIAI)*, 2011.

[246] Wu Zhou, Zhi Wang, Yajin Zhou, and Xuxian Jiang. DIVILAR: Diversifying intermediate language for anti-repackaging on Android platform. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2014.

[247] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, scalable detection of "piggybacked" mobile applications. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[248] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party Android marketplaces. In *ACM Data and Application Security and Privacy (CODASPY)*, 2012.

[249] Yajin Zhou and Xuxian Jiang. Android malware genome project. `http://www.malgenomeproject.org/`, 2012.

[250] Yajin Zhou and Xuxian Jiang. Dissecting Android malware: Characterization and evolution. In *IEEE Security and Privacy (S&P)*, 2012.

[251] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Network and Distributed System Security Symposium (NDSS)*, 2012.