The Evolution of Android Malware and Android Analysis Techniques

KIMBERLY TAM, Information Security Group, Royal Holloway, University of London
ALI FEIZOLLAH, NOR BADRUL ANUAR, and ROSLI SALLEH, Department of Computer System and Technology, University of Malaya
LORENZO CAVALLARO, Information Security Group, Royal Holloway, University of London

With the integration of mobile devices into daily life, smartphones are privy to increasing amounts of sensitive information. Sophisticated mobile malware, particularly Android malware, acquire or utilize such data without user consent. It is therefore essential to devise effective techniques to analyze and detect these threats. This article presents a comprehensive survey on leading Android malware analysis and detection techniques, and their effectiveness against evolving malware. This article categorizes systems by methodology and date to evaluate progression and weaknesses. This article also discusses evaluations of industry solutions, malware statistics, and malware evasion techniques and concludes by supporting future research paths.

CCS Concepts: • **Security and privacy → Operating systems security**; **Mobile platform security**

Additional Key Words and Phrases: Android, malware, static analysis, dynamic analysis, detection, classification

ACM Reference Format:
Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. 2017. The evolution of android malware and android analysis techniques. ACM Comput. Surv. 49, 4, Article 76 (January 2017), 41 pages.
DOI: http://dx.doi.org/10.1145/3017427

1. INTRODUCTION

Smartphones, tablets, and other mobile platforms have quickly become ubiquitous due to their highly personal and powerful attributes. As the current dominating personal computing device, with mobile shipments surpassing PCs in 2010 [Menn 2011], smartphones have spurred an increase of sophisticated mobile malware. Over six million mobile malware samples have been accumulated by McAfee as of Q4 2014, up 14% over Q3, and roughly 98% of them target primarily Android devices [McAfee 2015]. Given Android's all-pervasive nature and the threats against this particular mobile platform, there is a pressing need for effective analysis techniques to support the

Lorenzo Cavallaro would like to acknowledge that this research has been partially supported by the UK EPSRC grant EP/L022710/1. The second author would like to acknowledge that this work was supported in part by the Ministry of Science, Technology and Innovation, under Grant eScienceFund 01-01-03-SF0914.
Authors' addresses: K. Tam's present affiliation is Hewlett Packard Labs in Bristol, UK; K. Tam, 35202 Severn Dr. Newark, CA 94560, USA; email: kim.tam4@gmail.com; L. Cavallaro, Room 231, McCrea Building, Information Security Group Royal Holloway, University of London Egham Hill, Egham, Surrey TW20 0EX, United Kingdom; email: Lorenzo.Cavallaro@rhul.ac.uk; A. Feizollah, N. Badrul Anuar, and R. Salleh, Department Of Computer System & Technology, Faculty Of Computer Science & Information Technology, Jalan Universiti, 50603 Kuala Lumpur, Wilayah Persekutuan Kuala Lumpur, Malaysia; emails: ali.feizollah@siswa.um.edu.my, askbard@gmail.com, rosli_salleh@um.edu.my.

DOI: http://dx.doi.org/10.1145/3017427

development of reliable detection and classification tools. In an attempt to evaluate the progress of research within this specific area of work, this article provides the following contributions.

(1) This work first presents background information on mobile devices and their characteristics. This leads to a detailed description of the Android operating system, as well as notable Android malware and general mobile malware traits (see Section 2). Unlike previous mobile malware surveys, this article primarily focuses on the malware traits that hinder accurate studies and presents them in conjunction with a comprehensive snapshot of today's Android research techniques.
(2) This work presents a comprehensive study on an extensive and diverse set of Android malware analysis frameworks, including methods (e.g., static, dynamic, hybrid), year, and outcome. Similar studies are then reviewed to identify evolving state-of-the-art techniques in an attempt to identify their strengths, weaknesses, performance, and uses. For example, this article discusses how robust some techniques are to major changes within Android, such as replacing the Dalvik runtime. Studies were primarily selected from well-established and top-ranked research venues. However, this work does include, wherever appropriate, a number of additional studies in an attempt to demonstrate the entire breadth of this research area (see Sections 3 and 4).
(3) Section 5 addresses several Android malware tactics used to obstruct or evade analysis. This article classifies and describes transformation attacks and examines several advanced malware evasion techniques, such as encryption, native exploits, and Virtual machine (VM)-awareness. With that knowledge, this article performs a comparison of malware strengths to common analysis weaknesses, creating a more comprehensive view than surveys focused on individual aspects. We then confirm trends in evasive malware, found in similar studies, with our own experiments.
(4) This work further supports several directions of future research and highlights issues that may not be apparent when looking at individual studies, including malware trends and plausible research paths. While some have recently been receiving more attention, others have yet to be explored sufficiently. Section 6 gives an overview of the state-of-the-art and future research discussion.

Unlike previous works, this article is not a general study on mobile attack vectors or defense [Becher et al. 2011; Enck 2011; Suarez et al. 2014; Faruki et al. 2015] but instead focuses on Android-related analysis techniques systematically and in detail. As can be seen in Table I, this differs from a number of previous works. In similar surveys (e.g, on Android malware families, evolution, characteristics), although analysis techniques are often mentioned, the information is scattered throughout the article to support other material. Furthermore, when combined, those pieces often formed an incomplete picture of all available methods. This study fills that gap by presenting a method-focused view. Furthermore, unlike similar surveys, for example, Vidas et al. [2011], this article primarily concentrates on the malware aspects that hinder or deter analysis, detection, and classification, allowing us to explore the symbiotic relationship between malware and defense. These findings on how the newest malware and analysis techniques influence each other sets this survey apart from those focused on purely on malware threats or Android defense. However, while it is not the main focus, this article does discuss aspects of malware like market infections.

By narrowing the scope, this article provides in-depth studies on both sides of the arms race with respect to Android malware. A more general study on Android ecosystem weaknesses, for example, the level of app developer skills, and protection schemes can be found in Sufatrio et al. [2015]. This is unlike the focused details on analysis-related techniques, and anti-analysis methods, for Android malware in this article. The last

Table I. Comparison of Recent Surveys and Which Topics Have the Most/Least Coverage (✗= Little to No Content)

Survey	Background	Threat	Dyanmic Analysis	Static Analysis	Malware Tactics
This article (2016)	traditional + Android	discussion + small study	comprehensive coverage	comprehensive coverage	obfuscation + evasiveness
[Xu et al. 2016]	Android ecosys-tem	privilege escalation	dataflow and taint analysis	dataflow + mentions studies	obfuscation, moderate detail
[Faruki et al. 2015]	Android	malware types & actions	moderate coverage	moderate coverage	obfuscation
[Sufatrio et al. 2015]	Android	complete taxonomy	mention studies, × detail	mention studies, × detail	×
[Polla et al. 2013]	mobile tech.	attacks + evolution	mention studies, × detail	×	×
[Suarez et al. 2014]	smartphones	malware attacks	moderate coverage	mention studies, × detail	obfuscation, but × detail
[Zhou and Jiang 2012b]	Android malware	characterize malware	×	×	obfuscation mentioned
[Vidas et al. 2011]	Android	attack vectors	×	×	×
[Enck 2011]	traditional + Android	malware attacks	mainly dynamic taint analysis	details a few methods	×
[Felt et al. 2011]	Android perm.	permission abuse	×	details of used (one) method	challenges for used method
[Becher et al. 2011]	smartphon	mobile network	dynamic taint analysis	function call analysis	×

section containing discussions and future research possibilities also differs from the most recent, and most relevant, articles. This may be useful to a wide range of readers.

2. BACKGROUND

Prior to discussing current approaches to analyze Android malware, this article begins with this background section on the evolution of mobile malware. This concludes with a more in-depth section on the Android operating system (OS), which is the focus of this article.

2.1. Evolution of Mobile 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 was considerably low. However, as time progressed from the 1980s, the drive to create malware became less recreational and more profit driven as hackers actively sought sensitive, personal, and enterprise information. Malware development is now more lucrative and being aided by malware developing tools. In 2013 a report showed that attackers can earn up to 12,000 USD per month via mobile malware [Register 2013]. This, in part, resulted in PC malware samples exceeding millions [Dirro 2011], well before smartphones had even taken off; as of 2009, fewer than 1,000 mobile malware samples were known [Dirro 2011].

Since 2009, however, the rise of mobile malware has been explosive, with new technologies providing new access points for profitable exploitations [McAfee 2013, 2014]. Moreover, an increase in black markets (i.e., markets to sell stolen information, system vulnerabilities, malware source code, malware developing tools) has provided more

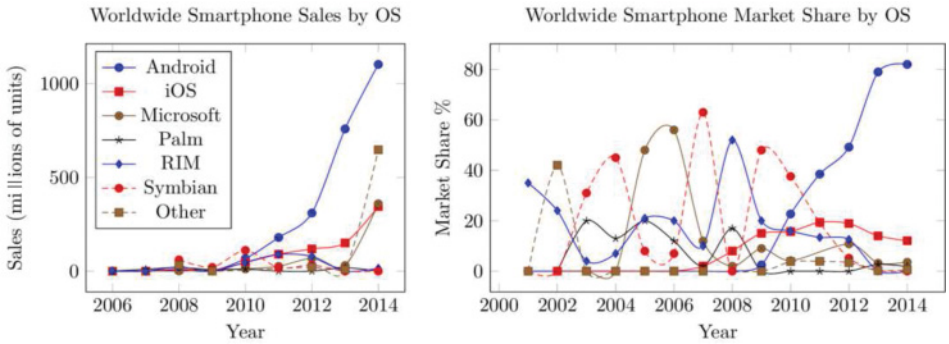


Fig. 1. Comparison of worldwide smartphone sales by operating systems (OSs).

incentive for profit-driven malware [InformationWeek 2014]. Although researchers may borrow and adapt traditional PC analysis 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 power), which limits on-device analysis. For further study on the similarities between traditional and mobile malware construction (e.g., in terms of features, methods, threats) refer to Felt et al. [2011], Branco et al. [2012], Bayer et al. [2009], and Rudd et al. [2016]. For more Android malware capabilities and vulnerability exploit details, see Drake et al. [2014].

2.1.1. Android Popularity and Malware. Based on a report from F-Secure, Android contributed to 79% of all mobile malware in 2012, compared to 66.7% in 2011 and 11.25% in 2010 [F-Secure 2013]. 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, in February 2014, Symantec stated that an average of 272 new malware and five new malware families targeting Android were discovered every month [Symantec 2014]. One of the prime contributing factors to this immense malware growth is Android's popularity (Figure 1), its open-source operating system [Teufl et al. 2014], 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 [2013]). 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. Looking towards 2016 and beyond, it is possible that Google will be adopting manual approaches for vetting applications in an attempt to lower malware existence further on the Google Play [Petrovan 2015].

Currently, the popularity of Android devices makes it a desirable target. However, its popularity is relatively recent, as illustrated in Figure 1. Its popularity began roughly in 2010, as shown by the statistics provided by Canalys (2001–2004) and Gartner [2015]. Interestingly, this figure also depicts a sizable dip in Symbian market shares during 2005, which may be the result of the first mobile worm, Cabir, discovered in 2004 and designed for Symbian [Gostev and Maslennikov 2009]. Figure 1 also illustrates why certain studies spanning 2000–2008 focus entirely on Symbian and Windows mobile malware threats; they were the most popular operating systems (OSs) during that period [Dunham 2009; Aubrey-Derrick and Sahin 2008].

As general smartphone sales rose dramatically in 2010, several alternatives rose to compete with Symbian. Studies such as La Polla et al. [2013] and Felt et al. [2011] reflected this shift by including emerging OSs such as Android and iOS, and by 2012 Android began to clearly dominate. Studies then began to focus purely on Android as Android malware skyrocketed [Symantec 2013; Zhou and Jiang 2012b]. 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 [Unuchek 2013]. In general, nearly half of all mobile malware as of 2014 are Trojans and are being tailored to target specific demographics. Together, Russia, India, and Vietnam account for over 50% of all unique users attacked in the world [Securelist 2013], while U.S. infections, as determined with 3 months of Domain Name System (DNS) traffic, is less than 0.0009% [Lever et al. 2013]. However, this method indirectly measured domain-name resolution traces. At the end of 2014, McAfee also analyzed regional infections rates of devices running their security products. They found the infection rates in Africa and Asia were roughly 10%, while Europe and both Americas had rates between 6% and 8%. Further discussions on varying infection rates due to different geological and virtual markets factors can be found in Section 6.

2.1.2. Traits of Android Malware. As mobiles are constantly crossing physical and network domains, they are exposed to more infection venues than traditional PCs. For example, by making full use of their host's physical movements, mobile worms are capable of propagating across network domains more easily [Sandeep Sarat 2007]. Additionally, with over one million available apps and near instantaneous installation, mobile devices are subjected to a high turnover of potentially malicious software [Kuittenin 2013]. 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 hard to automatically traverse all possible execution paths (see Section 5). Mobile devices are also accessible, and vulnerable, through multiple (sometimes simultaneous) "connections" to the outside world, such as email, WiFi, General Packet Radio Service (GPRS), High-Speed Circuit-Switched Data (HSCSD), 3rd Generation (3G), Long-Term Evolution (LTE), Bluetooth, SMS, Multimedia Messaging Service (MMS), and web browsers. They also utilize a complex plethora of technologies such as camera, compass, and accelerometers, which may also be vulnerable, for example, via drivers [Zaddach et al. 2014].

As an exploit 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 [Xiang et al. 2014] are still necessary. This is evident as background SMS are considered a high risk event by users, as shown in a study ranking smartphone user concerns [Felt et al. 2012] and since malware still exhibit this behavior [McAfee 2014]. As an example, it was estimated that over one 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 [Sophos 2012]. This is just one example of how, since 2010, the number of profit-driven malware has reportedly surpassed the number of non-profit-driven malware, and the gap continues to grow steadily [Techcrunch 2013].

Often, once malware is installed (e.g., social-engineering, drive-by-download), they use *privilege escalation* attacks to exploit Android OS or kernel vulnerabilities. When successful, the malware gains root access of the device [Zhou and Jiang 2012b]. Primarily, these attacks provide the malware with access to the lower, higher-privileged, architectural layers (see Figure 2). Once compromised, besides premium calls or SMS, malware often leak data pertaining to the device, owner, or both [McAfee 2014]. Similarly, malware known as *spyware* spy or monitor a target by exploiting mobile devices. Spying malware are also often *bots*, as they are controlled remotely via a command and control server. However, any malware can be one of many bots as long as there is network of compromised or malicious devices. These bots can also be used for *denial*

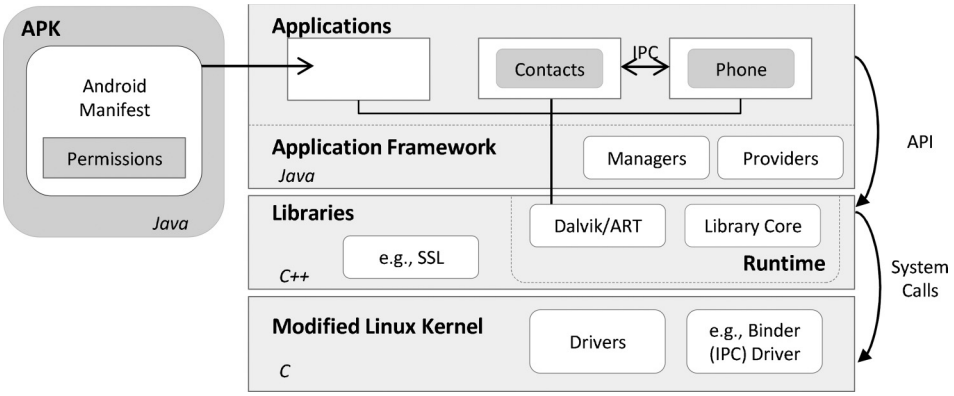


Fig. 2. Overview of the Android Operating System (OS) Architecture.

of service (DoS) attacks by rerouting traffic to specific address(es). Similarly, malware can deny services of other apps (including anti-virus apps) by overusing resources (e.g., battery, bandwidth) and tampering with necessary files or processes. See the Vidas et al. [2011] survey for focus on Android malware security threats and a few common defense mitigation techniques.

2.1.3. Notable Android Malware. There have been many malware families discovered from 2011 to 2015, but there have been only a few pivotal samples worth mentioning at this point. 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 has been discovered between 2014 and 2015, showing that mobile malware is, in some ways, catching up to traditional malware. The Android malware NotCompatible.C infected over four million devices to send spam emails, buy event tickets in bulk, and crack WordPress accounts [Strazzere 2014]. Furthermore, this malware is persistent and self-protecting via redundant actions and encryption, making static analysis very difficult. Conversely, malware such as Dendroid and Android.hehe are more difficult to analyze dynamically, as they are aware of emulated surroundings (details in Section 4.3) and have consistently evaded Google Play's vetting processes. The last notable Android malware mentioned here is the first Android bootkit, which can evade anti-virus (AV) products as it only exists in the boot partition, which is read-only memory. In the future, memory analysis may also be necessary to analyze malware, such as Oldboot, as they can only be found in volatile memory [Liam 2014].

2.2. Android Overview

2.2.1. Android Architecture. 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, this was changed to an ahead-of-time compiler, ahead of time compilation (ART), as opposed to the Dalvik just-in-time compiler. As discussed in Section 3 and 6, this change has negatively affected several current state-of-the-art analysis frameworks. The Android hardware consists of a baseband ARM processor (future tablets may use the Intel x86 Atom), a separate application processor, and devices such as GPS and Bluetooth. Figure 2 gives an overview of the described Android architecture.

In order to access the system, all Java-written apps must be granted permissions by the Android Permission System during installation (more in Section 3). Several studies

evaluating the effectiveness of Android permissions can be found in Felt et al. [2011], Au et al. [2011], Wei et al. [2012a], and Au et al. [2012]. Once installed, that is, permissions granted and enforced by the kernel, apps can interact with each other and the system through well-defined application program interface (API) calls. Unfortunately, this also applies to anti-virus apps, preventing these products from easily introspecting other apps. Because of this, most anti-virus solutions are signature-based and may be more viable implemented in markets instead of on-device (e.g., Chakradeo et al. [2013] and Zhou et al. [2012]).

The Android apps themselves are comprised of a number of components: activities, broadcast receivers, services, and content providers. 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 one component requests another component to do and is composed of asynchronous messages exchanged to perform this task. While broadcast receivers and services tend to run in the background, activities are the most visible component to the user and are often what handles user interactions like button clicking.

2.2.2. Comparison to Other Mobile Operating Systems. This section summarizes core differences between the Android OS and other mobile OSs (see their popularity in Figure 1). In particular, this article outline differences in architecture and how applications are handled and separated from the rest of the system. This helps determine their vulnerability to malware infections and malware exploits.

iOS: Released in 2007, iOS (previously iPhone OS) runs on XNU, a hybrid kernel. Apps run on top of the OS, which is comprised of four abstractions layers: Cocoa Touch, Media, Core Services, and Core OS. Users interact with the touch layer, triggering apps that then interact with the media and core layers for fundamental system services. All layers use low-level features supplied by the core layer, including the security framework. Unlike other systems, iOS does not possess a sophisticated permissions system and instead relies on the Apple store to screen apps [Apple 2015].

Windows OS: Developed by Microsoft and released in April 2000, Windows OS is based on the Windows CE hybrid kernel known as NT. Both custom and Windows applications are run on top of the OS in user mode. Apps in this less-privileged layer can be shut down without harming lower layers and are granted capabilities like Android permissions but with fewer options [Au et al. 2011]. The highest privileged mode is the kernel mode. Earlier OS versions had a loophole that allowed threads to be put in and out of kernel mode, giving attackers access to kernel-level resources.

Palm: Palm OS was released in 1996 but was discontinued after being succeeded by WebOS in 2009. WebOS runs on a monolithic (Linux) kernel and runs all apps in a User Interface (UI) System manager. Only the read permission is granted to third-party apps, but certified apps can have access to more sensitive APIs [Kingpin 2001]. These APIs are delivered as Mojo, a JavaScript framework that lies between the applications and the core OS, supporting common application-level functions, access to built-in applications, native services, and to protect the core OS from malicious applications.

BlackBerry: Created by Research in Motion, BlackBerry OS was released in 1999, with a Java virtual machine kernel type. Apps are organized into sections of the app infrastructure layer but separated from the OS system services. These partitions include native, web, and Android apps with their own respective infrastructure partitions in the layer below. These layers and their application context provide security to the OS and individual apps. Users define one set of permissions that is assigned to all apps on the device; permissions are not customizable per app [BlackBerry 2013].

Symbian: The Symbian OS, released in 1997, runs on an EKA2 kernel that enabled a real-time, priority, multithreaded OS. The kernel does as little as possible, outsourcing

the details to extensions, services, and drivers layered on top of the nano-kernel to maximize device stability. The topmost layer is the user interface, which interacts with both the application services layer below and the generic (not base) OS services layer. In Maemo, Symbian's successor, there is no permission system and no isolation between applications [Dunham 2009].

3. TAXONOMY OF MOBILE MALWARE ANALYSIS

The risks introduced by mobile malware motivate the development of robust and accurate analysis methods. One way to counter or detect malware is with the use of AV products. Unfortunately, as mentioned previously, on-device AV applications face difficulties as they are just as limited as normal applications. Hence cloud- and signature-based detection is more popular.

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. First, this method is ineffective for detecting unknown threats, that is, 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. Second, malware can easily bypass signature-based identification by changing small pieces of its software without affecting the semantics [Rastogi et al. 2013]. Section 3 provides further details on obfuscation techniques, including those that break signature-based detection. As a result of these downfalls, exemplified by the Google App Verification system released in 2012 [Jiang 2012], more effort has been dedicated to implementing semantic signatures, signatures based on functions or methods [Crussell et al. 2012; Zhou et al. 2012]. Alternatively, a wider set of available app features may be analyzed statically or dynamically to detect, or classify, malicious applications. In the remainder of this section, we examine such methods, their applications, and feature choice.

Although not discussed thoroughly within this article, it is natural that research on newer mobile environments builds on decades of traditional static and dynamic malware research. For example, although decompiling and virtualization are traditional methods, the particulars of code packaging and VM architectures 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, as discussed below.

3.1. 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. All static methods, however, are vulnerable to obfuscations (e.g., encryption) that remove, or limit, access to the code [Moser et al. 2007]. Similarly, the injection of non-Java code, network activity, and the modification of objects at runtime (e.g., reflection) are outside the scope of static analysis as they are only visible during execution. As later shown in Section 5.5, these do occur frequently in Android malware. Android app source code is also rarely available, so many frameworks analyze the app bytecode inside the app package (APK) instead. APK contents are described as follows, including changes introduced with the new ART runtime:

- **META-INF** folder holds manifest file, app RSA, list resources, and all resource SHA-1 digests;
- The **assets** directory holds files apps can retrieve with the AssetManager;
- **AndroidManifest.xml** is an additional Android manifest file describing package name, permissions, version, referenced library files for the app, and app components, that is, activities, services, content providers, and broadcast receivers;
- The **classes.dex** file contains all Android classes compiled into dex file format for the Dalvik VM. For ART, Dalvik bytecode is stored in an .odex file (pre-processed 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 of all ARM based processors);
- The folder **res** holds resources not compiled into resources.arsc;
- **resources.arsc** is a file containing precompiled resources.

Two essential APK components for Android static analysis and detection are (1) the `Android- Manifest.xml`, which describes permissions, package name, version, referenced libraries, and app components (e.g., activities), and (2) `classes.dex`, which contains all Android classes compiled into a Dalvik compatible, dex file format. We are unaware of any studies analyzing odex files.

3.1.1. 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 [Wu et al. 2012]. For example, before accessing the camera, the Android system checks if the requesting app has the CAMERA permission [Felt et al. 2011]. These requested permissions must be declared within the *AndroidManifest.xml*. As the manifest is easy to obtain statically, many frameworks, such as PScout [Au et al. 2012], Whyper [Pandita et al. 2013], and [Felt et al. 2011; Wei et al. 2012a], 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 behaviors [Felt et al. 2011]. 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 [Au et al. 2012; Pandita et al. 2013]. Furthermore, Wei et al. [2012a] 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 our experiments in Section 5.5, that malware requests 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.

3.1.2. Intents. Within Android, intents are abstract objects containing information on an operation to be performed for an app component. Based on the intent, the appropriate action (e.g., taking a photo, dialing a number) 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. In DroidMat [Wu et al. 2012], intents, permissions, component deployment, and APIs were extracted from the Manifest and analyzed with several machine-learning algorithms, such as k-means, k-nearest neighbors, and naive Bayes, to develop malware

Table II. Decompiled DEX Formats and Uses Based on How They Have Been Used by Existing Tools

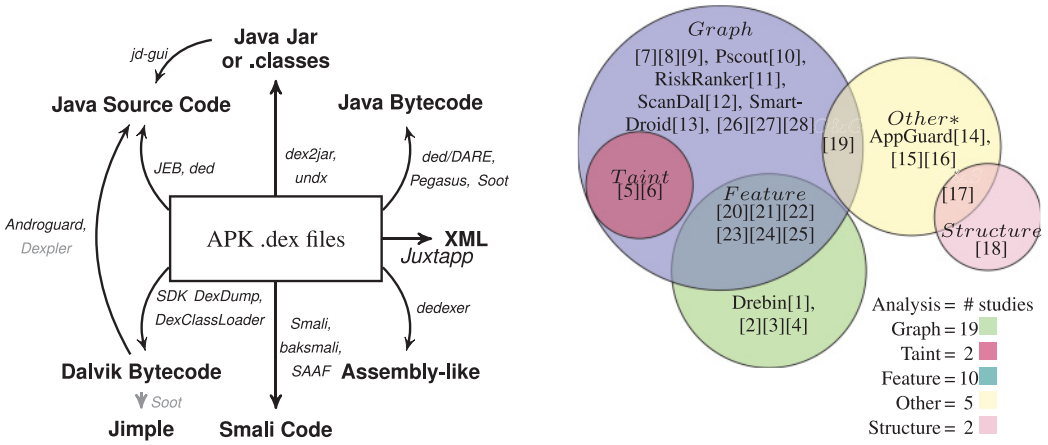
Format	Example Tool	Performance	Coverage
Dalvik Bytecode	dexdump [Kim et al. 2012]	false+ (15-18%) unknown (∼75%)	× dynamic code × instruction change ∼ reflection × JNI
Java Bytecode	Pegasus [Chen et al. 2013]	false+ (12.5%)	∼ reflection ∼ intents
Source Code	ded [Desnosi and Gueguen 2012]	accuracy (94)%	× dynamic code × instruction change ∼ number recovery
Smali	SAAF [Hoffmann et al. 2013]	accuracy (99.9%)	× obfuscation × runtime
Assembly	dedexer [Felt et al. 2011]	false+ (4%)	∼ reflection ✓ intents
Jar	dex2jar [Gibler et al. 2012]	false+ (35%)	× ad libs, JNI, intents, Java data structures
Jimple	FlowDroid [Arzt et al. 2014]	93% recall, 86% precision	×reflection

detection systems that were evaluated to be better than previous systems. Similarly, DREBIN [Arp et al. 2014] collected intents, permissions, app components, APIs, and network addresses from malicious APKs but instead used support vector machines. The results of the experiment showed that DREBIN detected 94% of the malware with a low false-positive rate.

3.1.3. Hardware Components. Another part of the Android Manifest that has been used for static analysis is the listed hardware components. DREBIN [Arp et al. 2014] utilized these components listed in the Manifest in its analysis. This can be effective as apps must request all the hardware (e.g., microphone, 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 calculator app to require 3G and GPS access. Dynamic analysis can be used to analyze hardware usage, but these normally analyze API calls, or system calls, as it is easier than analyzing the hardware directly.

3.1.4. Dex Files. The dex or classes.dex files can be found in the Android APK. They are difficult for humans to read 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 II for a brief comparison of disassembled formats. Both PScout [Au et al. 2012] and AppSealer [Zhang and Yin 2013] use Soot directly on the dex, see Figure 3(a), to acquire Java bytecode, while [Enck et al. 2011] uses ded/DARE, and Pegasus created its own "translation tool" [Chen et al. 2013]. Alternatively, Felt et al. [2011] decompiles dex into an assembly-like code with dedexer, while others choose to study Dalvik bytecode [Kim et al. 2012; Grace et al. 2012; Zhang and Yin 2013], smali [Hoffmann et al. 2013; Zheng et al. 2012, 2013a; Zhou et al. 2014], or the source code [Crussell et al. 2012; Desnosi and Gueguen 2012]. 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, features (e.g., classes, APIs, methods), structure sequences, and program dependency graphs can be extracted and analyzed. Dex files have also been decompiled and analyzed to track the flow of intents in interprocess communications (IPC), also known as inter-component communications (ICC) [Yang et al. 2015; Li et al. 2015], and to aid smart stimulation [Mahmood et al. 2014].

Different types of static analysis, such as feature, graph, or structure-based (details in Section 4), may also be combined for a richer, more robust analysis. For example, as seen in Figure 3(b), the framework [Zhou et al. 2013] combines structural and feature analysis by decoupling modules and analyzing extracted semantic feature vectors to detect destructive payloads. Also shown in Figure 3(b), Hoffmann et al. [2013] extracts



(a) APK decompiling/disassembly (b) Venn Diagram of static analysis methods (see Section 4)

Fig. 3. Approaches to Android static analysis. (Jimple = *simplified* Java source code, Other* = source code [Au et al. 2011], bytecode [Davis et al. 2012; Backes et al. 2013], Manifest [Wei et al. 2012a], and module decoupling [Zhou et al. 2013]).

#	Paper	10	[Au et al. 2012]	20	[Felt et al. 2011]
1	[Arp et al. 2014]	11	[Grace et al. 2012]	21	[Hoffmann et al. 2013]
2	[Wu et al. 2012]	12	[Kim et al. 2012]	22	[Zheng et al. 2013a]
3	[Hein and Myo 2016]	13	[Zheng et al. 2012]	23	[Amamra et al. 2012]
4	[Lagerspetz et al. 2014]	14	[Backes et al. 2013]	24	[Huang et al. 2014]
5	[Feng et al. 2014]	15	[Davis et al. 2012]	25	[Yang et al. 2015]
6	[Azim and Neamtiu 2013]	16	[Wei et al. 2012a]	26	[Mahmood et al. 2014]
7	[Crussell et al. 2012]	17	[Zhou et al. 2013]	27	[Arzt et al. 2014]
8	[Chen et al. 2013]	18	[Zhou et al. 2012]	28	[Li et al. 2015]
9	[Yajin Zhou 2013]	19	[Au et al. 2011]		

both feature and dependency graphs, via smali program slices, to find method parameter values. Conversely, Automated system for evaluating the Detection of Android Malware (ADAM) [Zheng et al. 2013a] tested if anti-malware products could detect apps repackaged by altering dependency graphs and obfuscated features. While obfuscation methods for mobile malware (e.g., native code, encryption) existed before Android in Symbian malware [Schmidt et al. 2009b], and despite well-established static methods, obfuscation is still an open issue as of 2014 [Sophos 2014]. This is further discussed in Section 5.

3.2. Dynamic Analysis

In contrast to static analysis, dynamic analysis executes a program and observes the results. Applied simplistically, it provides limited code coverage, as only one path is shown per execution, but can be improved with stimulation. As Android apps are highly interactive, many behaviors need to be triggered via the interface, received intents, or with smart, automatic event injectors [Azim and Neamtiu 2013; Machiry et al. 2013; Mahmood et al. 2014]. 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, then malware can detect and tamper with the analysis. This is

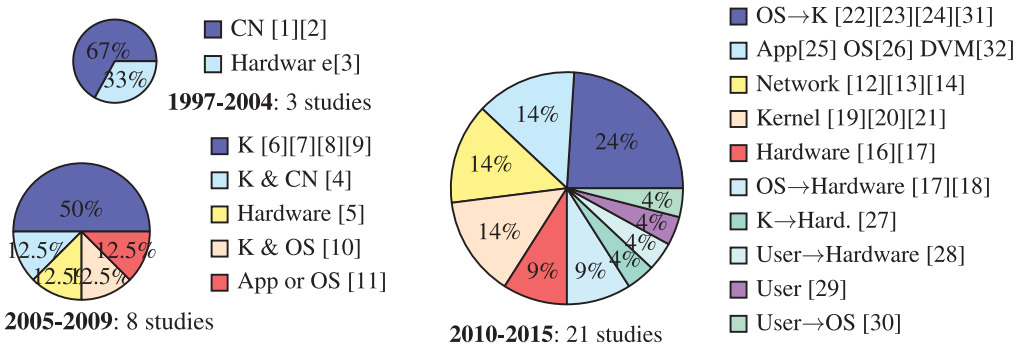


Fig. 4. Dynamic analysis studies based on different Android architectural layers. (CN = Cell Network, K = Kernel)

#	Paper	11	[Ongtang et al. 2009]	22	[Felt et al. 2011]
1	[Moreau et al. 1996]	12	[Yan and Yin 2012]	23	[Tam et al. 2015b]
2	[Samfat and Molva 1997]	13	[Vidas et al. 2014]	24	[Dini et al. 2012]
3	[Jacoby 2004]	14	[Amamra et al. 2012]	25	[Zheng et al. 2012]
4	[Miettinen et al. 2006]	15	[Zaddach et al. 2014]	26	[Xiang et al. 2014]
5	[Nash et al. 2005]	16	[Amos et al. 2013]	27	[Shabtai et al. 2012]
6	[Cheng et al. 2007]	17	[Enck et al. 2010]	28	[Bugiel et al. 2011]
7	[Becher and Freiling 2008]	18	[Yan and Yin 2012]	29	[Xu et al. 2012]
8	[Becher and Hund 2008]	19	[Bläsing et al. 2010]	30	[Chen et al. 2013]
9	[Miettinen et al. 2006]	20	[Burguera et al. 2011]	31	[Backes et al. 2014]
10	[Bose et al. 2008]	21	[Andrus et al. 2011]	32	[Li et al. 2014]

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 5). On the other hand, if the analysis was to reside in a lower layer, say, the kernel, then it would increase security but make it more difficult to intercept app data and communications. To overcome this weakness, there are several methods to fill the semantic gap, that is, recreating OS/app semantics from a lower observation point such as the emulator [Garfinkel and R. 2003; Tam et al. 2015b]. Details of in-the-box, out-of-the-box, and virtualization can be found later in the article, specifically in Sections 3.2.1–3.2.3.

To better understand the progression of dynamic analysis for Android see Figure 4. Here we attempt to illustrate the number of different architectural layers (e.g., hardware, kernel, app, or OS) being studied in dynamic analysis frameworks from 1997 to 2015. One interesting trend is the increasing amount of multi-layered analyses, which increases the number of unique and analyzable features but with increased overheads. Different analysis environments are also represented here, including emulators, real devices, and hybrids of both [Vidas et al. 2014]. Again, because the malware is running during analysis, the choice of environment is more complicated. In 2013, Obad was the first Android malware to detect emulated environments and *choose* not to exhibit malicious behaviors [Unuchek 2013]. Despite this, most analyses still implement emulators (discussed in Section 6).

For stimulating applications, the DynoDroid [Machiry et al. 2013] system was developed by using real user interactions for analysis; it collected user activities, such as tapping the screen, long pressing, and dragging, in order to find bugs in Android apps. Alternatively, hybrid solutions, like EvoDroid [Mahmood et al. 2014], use static and

dynamic analysis to explore as much of the application code as possible in the fewest number of executions. Besides increasing code coverage, user interactions with apps may also be analyzed for malware detection. By crowdsourcing scenarios, PuppetDroid [Gianazza et al. 2014] captured user interactions as stimulation traces and reproduced the UI interactions to stimulate malicious behaviors during dynamic analysis. This is based on the assumption that similar user interactions patterns can be used to detect malicious apps, as malware are often repackaged code or variants of each other (i.e., a malware family).

3.2.1. 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 Dalvik VM. For example, DIVILAR [Zhou et al. 2014] inserts hooks into the Android internals, that is, Dalvik VM, to run apps modified against repackaging. Furthermore Mockdroid [Beresford et al. 2011] modified the OS permission checks to revoke system accesses at runtime. 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 application leaves the analysis open to being attacked or bypassed, for example, with native code or reflection [Xu et al. 2012]. It is possible to increase transparency by hiding processes or loaded libraries, but this is impossible to achieve from the user space alone. Additional downfalls to editing the OS or Dalvik are (1) necessary modifications to multiple Android OS versions, (2) more potential software bugs, and (3) the replacement of the Dalvik just-in-time compiler with an ahead-of-time compiler (ART [Vitas 2013]). Therefore, while in-guest methods already require moderate to heavy modifications between most Android OS versions, with the complete change from the Dalvik runtime to the ART runtime, many in-guest analysis need fundamental changes to adapt. Alternatively, kernel-level frameworks would grant the framework a higher privilege level than user-level apps, increasing transparency and security, unless the malware gained root privileges via a root exploit. Although high-level semantics are more difficult to analyze out-of-the box, this method can provide greater portability across different Android OS versions, as there is more stability in the lower architecture layers.

3.2.2. Out-of-the-Box Analysis. VM-based analyses, like traditional methods, 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. For example, sandboxing native code (i.e., non-Java code compiled to run with a Android Central processing unit (CPU)) in the future may add further protection to Android devices [Afonso et al. 2016]. 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 include the additional cameras, GPS, or accelerometer. While the mobile emulator MobileSandbox [Becher and Freiling 2008] works for both Windows and Android, most other systems like Andrubis [Weichselbaum et al. 2012], DroidScope [Yan and Yin 2012], CopperDroid [Tam et al. 2015b], and [Winter et al. 2012; Frenzel et al. 2010] are purely Android emulators. In particular, these were built on top of short for Quick Emulator (QEMU), an open-source CPU emulator available for ARM hardware.

Unfortunately, malware can, and has, countered emulation by detecting false, nonreal, environments and can stop or misdirect the analysis. For example, multiple personalities can be used to fool detection systems. There are many samples of traditional

Layer*	Accessible Data of Interest
User	User interface inputs
Apps, OS	IPC/ RPC, APIs, security frameworks
Kernel, Hypervisor	Dddr space, network, syscalls, virt registers, data types/constructors/fields
Hardware	Battery usage, file access, CPU

*Lower layers found lower on the table.

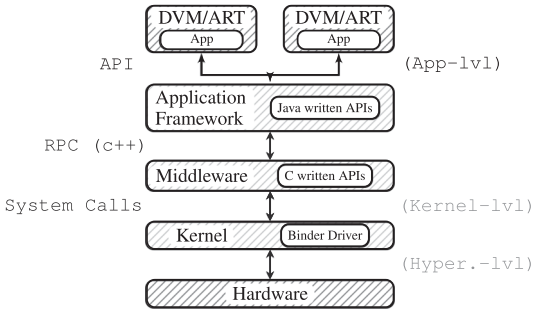


Fig. 5. Data and sandboxing available at all Android architectural layers.

PC malware that do exactly this, and more mobile malware are now exhibiting similar levels of sophisticated VM-awareness (details in Section 5). While it was accepted that out-of-the-box analysis meant that fewer high-level semantic data are available, it was previously believed that recreating high-level behaviors, such as IPC/ICC, was impossible outside the box. This was proven false by the CopperDroid framework [Tam et al. 2015b]. Furthermore, with CopperDroid's agnostic approach to the Android internals, it is able to switch between Android OS versions seamlessly, including the new 5.0 version running ART.

3.2.3. 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 [Andrus et al. 2011]), additional virtualization can be introduced at the kernel or hypervisor levels. Highly privileged kernel- or hypervisor-level (either bare-metal or hosted) sandboxing are less susceptible to corruption and, as seen in Figure 5, provide easier access to kernel data such as system calls [Bugiel et al. 2011; Becher and Hund 2008]. The negative of virtualization, and some emulators, is that the isolation introduces a discontinuity between the data seen by the analysis and high-level OS data. Such semantic gaps are reconstructable with virtual machine introspection (VMI). However, the Android Dalvik VM complicates VMI, as two-level VMI might be necessary.

If implemented, then 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, Heiser [2008] was one of the first to analyze the security benefits of hypervisors in embedded (e.g., mobile) devices. Unfortunately, the majority of on-shelf ARMs cannot currently support pure-virtualization,[1] and so alternative solutions have relied on other methods, such as para-virtualization or hardware extensions, to achieve similar affects. *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, Hwang et al. [2008] successfully created a secure hypervisor, or virtual machine monitor, on an ARM processor. In contrast, *pure-virtualization* (i.e., hardware virtualization) utilizes built-in processor

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

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 [Varanasi and Heiser 2011]. In 2012, Frenzel et al. [2010] used an ARM TrustZone processor extension to achieve effects similar to full virtualization, and, in 2013, Smirnov et al. [2013] 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.

3.3. Hybrid Analysis

By combining static and dynamic analysis, hybrid methods can increase robustnesses, monitor edited apps, increase code coverage, and find vulnerabilities. For example, Backes et al. [2013] and Chen et al. [2013] statically inserted hooks into functions (i.e., sensitive APIs) that provided runtime data for dynamic policy enforcement. Similarly, Ongtang et al. [2009] governed static permission assignments and then dynamically analyzed Android inter-process communications, as dictated by its policies. Although unable to analyze ICC, that is, IPC, Harvester [Rasthofer et al. 2016] can obtain important runtime data via a hybrid, static and dynamic, method.

Hybrid malware detectors like in Bläsing et al. [2010] 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 [Zheng et al. 2012], EvoDroid [Mahmood et al. 2014], and [Spreitzenbarth et al. 2013] use static analysis to find all possible activity paths before guiding the dynamic analysis through them. A5 [Vidas et al. 2014] also employed a similar hybrid analysis for detection, triggering intents found in the code in order to examine all paths of execution for malicious behaviors. A5 also utilized both real devices and emulators (one or the other) in their experiments. Concolic testing, a mixture of static and dynamic analysis, has also been used to uncover malicious information leaks in Android apps [Anand et al. 2012].

4. MALWARE ANALYSIS APPROACHES FOR ANDROID

This section provides detailed descriptions of various analysis techniques. While most are used both statically and dynamically, several are unique to one or the other (see Table III).

4.1. Analysis Techniques

4.1.1. Network Traffic. As we discover in our analysis in Section 5, most apps, normal and malicious, require network connectivity. In Zhou and Jiang [2012b], 93% of collected Android malware samples made network connections to a malicious resource. Additionally, Sarma et al. [2012] analyzed 150,000 Android applications in 2012 and found that 93.38% of malicious apps required network access while only 68.50% of normal apps did so. Similarly, in Hein and Myo [2016], permissions of 2,000 apps were analyzed to find that over 93% of malicious applications requested network connectivity. This demonstrates that network access is requested by most apps but particularly by the malicious ones. Alternatively, 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.

Table III. Android Malware Analysis Techniques Used by Static and Dynamic Methods

	Network Traffic	APIs	System Calls	Dependency Graphs	Features
Dynamic	destination & packets [Shabtai et al. 2012; Bugiel et al. 2011; Wei et al. 2012b]	hooks and so on. Xu et al. [2012]; Yan and Yin [2012]	[Burguera et al. 2011; Grace et al. 2012; Tam et al. 2015b]	[Anand et al. 2012; Azim and Neamtiu 2013]	[Backes et al. 2013; Davis et al. 2012]
Static	hard coded info	decompiling [Amamra et al. 2012; Arp et al. 2014; Chen et al. 2013; Zheng et al. 2012]	×	[Au et al. 2012; Kim et al. 2012; Huang et al. 2014; Yang et al. 2015; Mahmood et al. 2014]	[Hanna et al. 2013; Felt et al. 2011]

	Function Call Monitoring	Taint	IPC	Hardware
Dynamic	[Li et al. 2014; Becher and Freiling 2008; Distefano et al. 2010; Xu et al. 2012; Ravindranath et al. 2012]	[Enck et al. 2010; Gibler et al. 2012; Enck et al. 2010]	[Tam et al. 2015b; Xu et al. 2012; Ongtang et al. 2009; Bugiel et al. 2011]	[Kim et al. 2008; Buennemeyer et al. 2008; Nash et al. 2005; Jacoby 2004; Amos et al. 2013]
Static	×	[Yang et al. 2013; Arzt et al. 2014; Rasthofer et al. 2014; You et al. 2015]	[Yang et al. 2015; Octeau et al. 2013; Li et al. 2015; Wei et al. 2014]	Manifest [Arp et al. 2014]

Frameworks studying network communications have been implemented on both real and emulated devices [Shabtai et al. 2012; Bugiel et al. 2011; Wei et al. 2012b], as well as cell networks, which is computationally easier on individual mobile devices but must protect communication channels from attacks [Sandeep Sarat 2007; Jin and Wang 2013; Moreau et al. 1996; Samfat and Molva 1997; Burguera et al. 2011; Lever et al. 2013]. As a new area of research, it is still unclear how different the challenges are between mobile malware detection and traditional malware detection via network analysis. However, as shown in these studies, for botlike behaviors and leaked data, network analysis seems an effective method both traditional PCs and mobiles devices.

4.1.2. 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 the 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 [Chen et al. 2013; Zheng et al. 2012] and Aurasium [Xu et al. 2012] dynamically monitor these APIs for app policy enforcement and to discover UI triggers. Furthermore, if a private interface has no corresponding public API, then it can still be invoked with reflection—the ability an object has to examine itself. Library and system APIs can also be studied in conjunction [Yan and Yin 2012] and, once extracted, APIs can also be used to classify malware, as shown in Amamra et al. [2012].

4.1.3. System Calls. System-level APIs are highly dependent on Android hardware, that is, ARM. The ARM Instruction Set Architecture (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 [Becher and Freiling 2008; Hwang et al. 2008] and Android devices [Burguera et al. 2011; Grace et al. 2012; Tam et al. 2015b]. And while these are based on low-level information, it is still possible to reconstruct high-level semantic behaviors using data from system call analysis.

4.1.4. 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 analyzed for similarities such as plagiarism [Crussell et al. 2012]. Conversely, in control dependency graphs, an edge exists if the execution trigger of one state depends on the value in another state. For example, ScanDal [Kim et al. 2012] builds, and analyses, control flow graphs (CFG) based on sensitive data returned by APIs to discover information leaks. Similarly, Yajin Zhou [2013] also uses CFGs to detect information leaks but utilizes content providers instead of APIs. DroidSIFT [Zhang et al. 2014], on the other hand, creates 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 [Au et al. 2012] combines all call graphs from the Android framework components for a full, flow-sensitive analysis, and Pegasus [Chen et al. 2013] constructs permission event graphs to abstract the context in which events fire. Multiple flow analysis can also be used together to search for malicious background actions [Felt et al. 2011; Grace et al. 2012]. To make these frameworks scalable, graphs must remove all redundancies to avoid path explosions as more paths require more computations.

4.1.5. Features. Feature-based analysis extracts and studies sets of features from decompiled apps in order to enforce policies, understand API permissions, and detect code reuse through feature hashing (e.g., Juxtapp [Hanna et al. 2013]). To enforce security policies, hooks can be inserted at key points for later dynamic monitoring [Backes et al. 2013; Davis et al. 2012]. Conversely, to identify which permissions an API requires, Felt et al. [2011] ran different combinations of extracted content providers and intents. Besides analyzing the actual feature, like which APIs were triggered, feature frequency analysis is also often used to see how many times certain features are found, that is, multiple executions of the same API. The primary downside of feature-based analysis is it cannot reveal the context (i.e., when or how) in which a permission was triggered [Felt et al. 2011].

4.1.6. Function Call Monitoring. By dynamically intercepting function calls, such as library APIs, frameworks can analyze both single calls and sequences of calls to reconstruct behaviors 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 analyze Android behaviors. While it does require modifications to the Dalvik VM and may not work on Android 5.0 (e.g., with ART), the method requires relatively light modifications and has been used on versions 4.0–4.2 [Li et al. 2014].

4.1.7. Information Flow. 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 analyze information being moved, or copied, to new locations is *taint analysis*. Hence, traditional taint analysis has been frequently used to find vulnerabilities in Windows [Kang et al. 2011]. Analyzing tainted data allows one to track how data propagate 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, that is, 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 arrive at a sink, different procedures can then be run depending on the data, source, and sink. Typically, taint analysis method is used to detect leaked data, like in TaintDroid and AndroidLeaks [Enck et al. 2010; Gibler et al. 2012]. Specifically, TaintDroid performs dynamic taint analysis on application-level messages and VM-level variables, while AndroidLeaks uses a mapping of API methods and permissions as the sources and sinks in a data-flow analysis.

Alternatively, FlowDroid [Arzt et al. 2014] implemented both object and flow-sensitive taint analysis to consider the lifecycle 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, short for Sources and Sinks (SUSI) [Rasthofer et al. 2014], 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.

Broadly speaking, 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, You et al. [2015] analyzed 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, this 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 [Huang et al. 2015]. Using the same static decompiliation methods as FlowDroid (i.e., Soot and Dexpler), Dflow was used to understand context -sensitive information flows and DroidInfer for type inference analysis. 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) more so than related works such as FlowDroid.

4.1.8. Inter-Process Communications Analysis. Within the Android OS, apps rely on IPC and remote procedure calls (RPC) to carry out most tasks. These channels use Binder, a custom implementation of the OpenBinder protocol that allows Java processes (e.g., apps) to use remote objects methods (e.g., services) as if they were local methods. Thus analyzing IPC/RPC can provide essential Android-level insights. While CopperDroid [Tam et al. 2015b] does this dynamically, there have been static methods tracking the movement of intents within IPC, that is, ICC [Li et al. 2015; Yang et al. 2015]. As data can be passed though various channels like IPC, they are often analyzed for information flow. In one static study, Epicc [Octeau et al. 2013] created and analyzed a

control-flow super graph to detect ICC information leaks. While Epicc relied on Soot for the majority of its needs, Amandroid used a modified version of dexdump (i.e., dex2IR) to study inter-component data flows [Wei et al. 2014]. 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., libraries), which leads to a higher accuracy but at a performance cost. Particularly for Android, analyzing ICC/IPC is essential for understanding and detecting stealth behaviors [Huang et al. 2014] and leaked information [Li et al. 2015] 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 apps analyzed by Li et al. [2015] used IPC and malware noticeably leaked more data via IPC than benign apps.

4.1.9. Hardware Analysis. Several studies monitor the hardware status for abnormal behavior through app power signatures [Kim et al. 2008] and power/CPU consumption [Buennemeyer et al. 2008; Nash et al. 2005; Jacoby 2004]. Since 2010 (see Figure 4), most dynamic analyses that extracted hardware-based features also analyzed additional layers and features. Furthermore, since devices like the camera can only be accessed by system calls, they are rarely analyzed on a hardware level. The framework STREAM [Amos et al. 2013] collects data regarding system components like `cpuUser`,`cpuIdle`, `cpuSystem`, `memActive`, and `memMapped`. STREAM gains this information via APIs from its own installed app and 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 analyzing the Android Manifest of an APK.

4.1.10. Android Application Metadata. Application market metadata are the information users see prior to downloading and installing an app. Such data include the app's description, requested permissions, rating, and developer information. Since app metadata are not a part of the APK itself, we do not categorize it as a static or dynamic feature. In WHYPER [Pandita et al. 2013], the app permissions were acquired through the market and Natural Language Processing 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 (address book, calendar, and record audio). Similarly, Teufl et al. [2014] used sophisticated knowledge discovery processes and lean statistical methods to analyze Google Play metadata. Nonetheless, this study also stressed that metadata analysis should be used to complement other analyses. The app metadata they fed to their machine-learning algorithms included the last time modified, category (e.g., game), price, description, permissions, rating, and number of downloads. Additional metadata included creator ID (i.e., developer ID), contact email, contact website, promotional videos, number of screenshots, promo texts, package name, installation size, version, and the app title.

4.2. Feature Selection

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

4.2.1. Selection Reasoning. As mentioned previously, Android applications must be granted permissions in order to perform specific actions. Therefore, many studies such as VetDroid [Zhang et al. 2013] and DroidRanger [Zhou et al. 2012] analyze permission usage because of this reasoning. Similarly, DREBIN [Arp et al. 2014] analyzes

intents, components, and APIs in addition to permissions, as they provide additional permission- and usage-based features for more fine-grained results. One method for feature selection, therefore, is to understand the Android system and hypothesize that a set of features will provide the most reliable malware analysis or detection. Alternatively, new or largely unused feature sets may be explored to confirm whether these hypotheses were correct or to discover new, novel, solutions. Alternatively, feature ranking and selection algorithms may be used to choose a subset of all available features.

4.2.2. Feature Ranking Algorithms. Identifying the ideal feature set can be done with pre-existing algorithms [Jensen and Shen 2008]. Such algorithms use various mathematical calculations to rank all the possible features in the dataset. More details on datasets themselves can be found in Section 6. 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 [Hyo-Sik and Mi-Jung 2013]. One study [Shabtai and Elovici 2010] used feature ranking algorithms to select feature subsets from 88 features (i.e., top 10, 20, and 50). Comparably, Shabtai et al. [2012] analyzed 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 to choose from. Similarly, in Yerima et al. [2014], the authors collected 2,285 Android apps and extracted over 22,000 features. Using selection algorithms, sets of the top features were then used for analysis.

4.3. Building on Analysis

Section 3 has provided a study on a diverse set of Android analysis approaches to obtain detailed behavioral profiles [Enck et al. 2010; Zheng et al. 2012; Wei et al. 2012a; Au et al. 2012; Yan and Yin 2012; Tam et al. 2015b; Anand et al. 2012; Wei et al. 2012b] and assess the malware threat [Felt et al. 2011; Enck et al. 2011; Felt et al. 2011; Zhou and Jiang 2012b; Rastogi et al. 2013; Gomez et al. 2013; Zheng et al. 2013a; Jing et al. 2014]. These can be further developed to build classification or clustering frameworks [Zhang et al. 2014; Schmidt et al. 2009a; Zhou and Jiang 2012b; Grace et al. 2012; Rasthofer et al. 2014], policy frameworks [Ongtang et al. 2009; Distefano et al. 2010; Dietz et al. 2011; Xu et al. 2012; Davis et al. 2012; Backes et al. 2013], and malware detectors. The primary difference between classification and clustering is that classification generally has a set of predefined classes and the objective is to find which class a new object, or malware sample, belongs to. Conversely, clustering groups unlabeled objects together by seeking similarities.

With these frameworks to build on top of, it is possible to detect Android malware [Sandeep Sarat 2007; Becher and Hund 2008; Schmidt et al. 2009a; Burguera et al. 2011; Shabtai et al. 2012], policy violations such as information leaks [Bugiel et al. 2011; Kim et al. 2012; Yajin Zhou 2013], colluding apps [Marforio et al. 2012], and even repackaged or plagiarized apps [Crussell et al. 2012; Zhou et al. 2012]. Most malware detection methods are anomaly based [Shabtai et al. 2012] (e.g., defining normal and abnormal attribute sets), misuse based [Yajin Zhou 2013] (e.g., identifying specific malicious actions), or signature based (e.g. semantic or bytecode) [Crussell et al. 2012]. Accurately defining “abnormal” and “malicious” becomes essential. Furthermore, once detected, it is important to classify the threat for proper mitigation, family identification, and so new malware (e.g., zero day malware [Grace et al. 2012]) can be dealt with properly.

With the increasing amount of malware each year, scalability and automated classifying (or clustering) are also essential as malware flood app markets. In one study, it was shown that over 190 application markets host varied amounts of malware [Vidas

and Christin 2013]. Traditionally, the output of a classifier is either binary (i.e., the sample is either malicious or benign) multi-class (i.e., a sample can belong to one of many malware families or types). Furthermore, as classifiers normally compute vectorial data, features for study must be mapped to a vector space that the classifier can compute. Several general methods for inputing data into different classifiers include a binary representation, feature frequency, and by representing the states and/or transitions of a control or data flow graph. The difference between binary representation and feature frequency is that, for binary representation, a 1 is used for features the samples have and 0 otherwise, while feature frequency counts the number of times the feature was seen in a sample.

One of the more popular classifiers used for Android malware has been support vector machines, but many more are available (e.g., decision trees, naive bayes, k-Nearest Neighbors (kNN), random forest) and should be explored to find the most suitable fit to the features and desired task. In terms of scalability, manual efforts [Zhou and Jiang 2012b] will not scale, and sometimes accuracy is sacrificed for scalability (see Section 6). To keep accuracy high but improve its scalability, different filters or simplification methods can be used. For instance, DNADroid [Crussell et al. 2012] implemented several filters on their graphs to automatically reduce the search space and improve scalability with little cost.

5. EVOLUTION OF MALWARE TACTICS

As mentioned throughout the article, there are several kinds of obfuscation and VM-detection methods used by both traditional and mobile malware to obstruct analysis. In this article, we place static obfuscation techniques into several tiers; trivial transformations, transformations that hinder static analysis, and transformations that can prevent static analysis (e.g., anti-disassembly).

5.1. Trivial Layout Transformations

Trivial transformations require no code or bytecode level changes and mainly deter signature-based analysis. Unique to the Android framework, unzipping and repackaging APK files is a trivial form of obfuscation that does not modify any data in the manifest. This is because when repackaging the new app, it is signed with custom keys instead of the original developer's keys. Therefore, 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. Android APK dex files may also be decompiled, as previously shown in Figure 3(a), and reassembled. We are unaware of any studies decompiling ART oat or odex files as of early 2015. Once disassembled, components may be re-arranged 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.

5.2. Transformations That Complicate Static Analysis

While some static techniques are resilient to obfuscations, each technique is vulnerable to a specific obfuscation method. 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 and, depending on its construction, structural analysis is vulnerable to layout, data, and control obfuscation.

5.2.1. Data Obfuscation. This method alters APK data, such as the Manifest's package name. 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., C or C++ code compiled to run with a specific processor) is normally accessed via the Java native interface (JNI), but malicious encrypted native exploits can also be stored within the APK itself. 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. Similar forms of obfuscation have roots in traditional PC practices [Collberg et al. 1997].

5.2.2. Control Flow Obfuscation. This method deters call-graph analysis with call indirections: moving method calls without altering semantics. For example, a method can be moved to a previously non-existent method that then calls the original method. Alternatively, code reordering also obfuscates an application's flow. Programming languages are also often compiled into more expressive language, such as virtual machine code. This is the case with Java, as Java bytecode possesses the `goto` instruction while normal Java does not. Bytecode instructions can then be scrambled with `goto` instructions inserted to preserve 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, the breaking of functions into multiple smaller 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 This is known as interweaving classes. Last, Android allows for a few unique transformations by renaming or modifying non-code files and stripping away debug data (i.e., anti-debugging), such as source file names, source file line numbers, and local parameters [Rastogi et al. 2013].

5.3. Transformations That Prevent Static Analysis

These transformations have long been the downfall of static analysis frameworks for traditional analyses [Moser et al. 2007] and mobile malware analysis [Rastogi et al. 2013; Hoffmann et al. 2013]. Unless also a hybrid solution, no static framework can fully analyze Android applications using full bytecode encryption or Java reflection. 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.

For Android APKs, the bulk of essential code would be 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. This is possible as Java reflection allows objects to examine and modify itself. Thus, 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 analyzing it impossible. Cryptography, another useful tool for obfuscation, can be used by the app developer prior installation or at runtime with the use of Android *crypto APIs*.

Similarly, the use of *dynamically loaded code* cannot be analyzed statically and may difficult to analyze dynamically, depending on the technique. This mechanism loads a library into memory at runtime, hence the static difficulties, in order to retrieve the addresses of library functions and variables. Functions can then be executed to achieve

the desired effect. While utilizing libraries my be a benign action, dynamically loading code *is* a practical and effective form of obfuscation.

5.4. Anti-Analysis and VM-Aware

With the rapid growth of Android malware, sophisticated anti-analysis 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), requiring user input, and timing attacks like QEMU scheduling (i.e., measuring emulated scheduling behaviors), all of which have been implemented by Petsas et al. [2014] to evade cutting-edge detection tools. DenDroid, a real-world Trojan discovered in 2014, is capable of many malicious behaviors but will not exhibit them if it detects emulated environments such as Google Bouncer [Dilger 2014]. Another malware family, AnserverBot, detects and stops on-device mobile anti-virus software by randomly restarting their processes. The malware Android.hehe also has a split personality and acts benignly when the device IDs (e.g., International Mobile Equipment Identity (IMEI)) and `Build` strings indicate that it is running in an emulated environment [Hitesh 2014].

Other ways to deter analysis, but not necessarily detect VMs, is to make the app UI intensive, execute at "odd" times (i.e., midnight or a day after installation), require a network, or require the presence of another app. For example, the malware CrazyBirds will only execute if the application AngryBirds had also been installed and played with at least once. Additional obfuscation methods to deter dynamic analysis are data obfuscation (e.g., encryption), misleading information flows (e.g., You et al. [2015]), mimicry, and function indirections.

5.5. Statistics for Android Malware Evolution

In this subsection, we present our study on how Android malware has evolved to avoid analysis and as a threat in terms of permissions from 2010 onwards to late 2014. Our 2010–2012 dataset is made of 5,560 Android malware samples provided by the DREBIN project [Arp et al. 2014], including those previously studied in the Android Malware Genome Project [Zhou and Jiang 2012a]. The older 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 by a well-established AV vendor.[2] The analysis itself was primarily based on the Androguard tool [Desnosi and Gueguen 2012]. Thus our script could compile data on how many malware in our dataset used techniques like native code (i.e., `is_native_code`) from 2010 to 2015.

Android Malware Obfuscation: Overall, we automatically analyzed more than 9,300 Android malware samples to understand how the malware threat evolved in terms of used dynamically loaded code, Java reflection, native code invocation, crypto APIs, and top used permissions. Table IV shows the permission rankings found in our analyses. We then examined the implication of such trends on the state-of-the-art techniques 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 6. Such features hinder the effectiveness of static analysis and call for further research on robust hybrid or dynamic analysis development [Zhang et al. 2013; Yan and Yin 2012].

[2]Due to confidentiality agreements, we cannot redistribute Android malware samples provided by the McAfee AV vendor, but we can share their metadata to allow sample lookups and replicate our findings.

Table IV. Rank Variations of Top 10 Android Permission Requests from 2010 to 2014

Permission Ranking 2010-2011	Permission Ranking 2012
R1. INTERNET (96.6%)	R1. INTERNET (97%)
R2. READ_PHONE_STATE (90.5%)	R2. (+3) ACCESS_NETWORK_STATE (92%)
R3. VIBRATE (67%)	R3. VIBRATE (89%)
R4. WRITE_EXTERNAL_STORAGE (67.2%)	R4. (+6) ACCESS_FINE_LOCATION (84.5%)
R5. ACCESS_NETWORK_STATE (67.2%)	R5. (-3) READ_PHONE_STATE (90.5%)
R6. SEND_SMS (58.11%)	R6. (+1) WAKE_LOCK (80.9%)
R7. WAKE_LOCK (50%)	R7. (+2) ACCESS_WIFI_STATE (59%)
R8. RECEIVE_BOOT_COMPLETED (48%)	R8. (-4) WRITE_EXTERNAL_STORAGE (67.2%)
R9. ACCESS_WIFI_STATE (46.6%)	R9. (+5) ACCESS_CO*_LOCATION (48%)
R10. ACCESS_FINE_LOCATION (43%)	R10. (+8) FACTORY_TEST (40.9%)
Permission Ranking 2013	**Permission Ranking 2014**
R1. INTERNET (97.7%)	R1. INTERNET (98.7%)
R2. ACCESS_NETWORK_STATE (95.6%)	R2. ACCESS_NETWORK_STATE (98.3%)
R3. (+2) READ_PHONE_STATE (94.2%)	R.3 READ_PHONE_STATE (96.2%)
R4. (-1) VIBRATE (92.6%)	R4. VIBRATE (93.7%)
R5. (+2) ACCESS_WIFI_STATE (88.6%)	R5. (+1) WAKE_LOCK (92.5%)
R6. WAKE_LOCK (85.9%)	R6. (-1) ACCESS_WIFI_STATE (92.1%)
R7. (-3) ACCESS_FINE_LOCATION (82.1%)	R7. ACCESS_FINE_LOCATION (86.8%)
R8. WRITE_EXTERNAL_STORAGE (70.6%)	R8. (+1) FACTORY_TEST (81.6%)
R9. (+1) FACTORY_TEST (67.2%)	R9. (-1) WRITE_EXTERNAL_STORAGE (78.8%)
R10. (-1) ACCESS_COARSE_LOCATION (57%)	R10. ACCESS_CO*_LOCATION (63.7%)

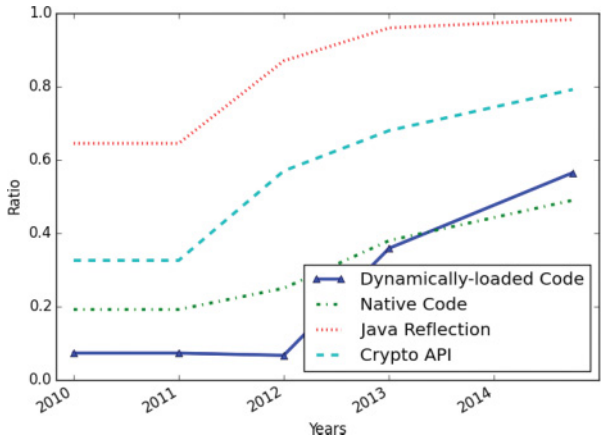


Fig. 6. Evolution of Android malware using dynamically loaded code, native code invocations, reflection, and crypto APIs.

Although dynamic analysis is more robust against the use of dynamically loaded code and Java reflection, its effectiveness is often reduced by its limited code coverage. Recent works, such as by Anand et al. [2012], Vidas et al. [2014], and Gianazza et al. [2014], 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 6). Similarly, Figure 6 shows a constant increase in the use of native code, which calls for further research in the development of techniques able to transparently analyze low-level semantics as well as high-level Android semantic seamlessly [Tam et al. 2015b].

Permission Usage and Malware Threat: Shifting to permission usage within our dataset, a reasonable indicator of the growing influence (i.e., threat) of malware, the INTERNET permission was the most requested, followed by READ_PHONE_STATE (e.g., access to IMEI, IMSI, phone number). As seen in Table IV, their popularity has fluctuated a few positions the first few years but eventually stabilized. Furthermore, even though 82% of all apps read device ID and 50% collect physical locations, malware are even more likely to gather such data. For example, malware are 8 times more likely to steal SIM card data [McAfee 2014]. There are many ways to misuse this leaked user information, such as determining the user's location and differentiating between real devices and emulators (details in Section 6). To collect geographical data, the malware we analyzed became increasingly interested in location-based permissions (COARSE and FINE, as seen in Table IV). We also noted the prevalence of the SEND_SMS permission, although it lessened over the years due to Google's efforts and thus is omitted from Table IV. Despite this, it was found that SMS malware have increased over 3 times since 2012; are a top concern in the U.S., Spain, and Taiwan; and can both generate revenue for attackers and steal bank SMS tokens to hack bank accounts [McAfee 2014].

In Table IV, 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 [Lindorfer et al. 2014]. 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 0.23% of malware in 2010 but by 24.8% in 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, and other similar actions.

We also witnessed several new permissions being requested across these 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 starting to become popular with malware include USE_CREDENTIALS and AUTHENTICATE_ACCOUNTS, which were categorized as dangerous by Google, as they could greatly aid in privilege escalation. INSTALL_PACKAGES, added in 2011, is another dangerous permission, as it allows malware to install other packages to gain more privileges, spread the infection, or make it harder to eradicate. Partly due to the introduction of more dangerous permissions,[3] the percentage of malware in our dataset requesting such permissions increased from 69% in 2010 to 79% by 2014. Again, 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 in Section 6, other studies on the Android permission system evolution have also shown it growing larger, more coarse grained, and with a higher percentage of dangerous permissions.

6. DISCUSSION

Smartphones are currently the top personal computing device, and trends show that this is unlikely to change with over 2.5 billion mobile shipments made by early 2015 [Gartner 2015]. Of these shipped smartphones, Android is by far the most popular smartphone OS and has attracted a growing number of dangerous malware [Securelist 2013; McAfee 2015].

[3]Google maintains a list of dangerous permissions at http://developer.android.com/guide/topics/security/permissions.html.

To better understand the current malware threat, we use this survey on Android malware analysis and detection methods to assess their effectiveness. We then suggest the next logical steps for future research against malware and make 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 is 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.

6.1. 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 analysis frameworks and anti-virus solutions. By amassing and analyzing various Android malware techniques and Android malware analysis frameworks, this article has identified several risks that should motivate continuous research efforts in certain directions. These research directions are discussed later in this section, after assessing today's mobile security effectiveness.

6.1.1. 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, although not spread evenly across countries, is 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. As a recap, despite low Android malware infections in some geographical areas like the U.S. [Securelist 2013; Lever et al. 2013], the overall global infection rate is more concerning. For example, Truong et al. [2013] has estimated a 26–28% infection rate worldwide based on real device data, and McAfee has estimated a 6–10% infection rate using Android devices running their security solutions. Like biological viruses, it is also dangerous to ignore developing malware families in other app markets or countries, as there may be future cross infection. Furthermore, when considering that the majority of new malware are undetected by antivirus products, discussed further below, it is highly plausible that actual infection rates are higher than reported. To reduce malware infections, malware markets need to be able to both accurately vet submitted applications and remove available malware as soon as they have been identified or detected by themselves or by an external party. Ideally, users should also 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 applications in some locations. Online application malware and virus detectors and on-device 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 lifetime [Felt et al. 2011]. While only one known malware sample attacked rooted phones in 2011, by the following year, more than one-third the malware analyzed by Zhou and Jiang [2012b] 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% [La Polla et al. 2013; Felt et al. 2011] of the samples, and user information harvesting, a top security issue in 2011 [Felt et al. 2011],

is still a current issue with 51% of malware samples exhibiting this behavior [McAfee 2014].

6.1.2. Weaknesses in Analysis Frameworks. Many frameworks today are unable to analyze dynamically loaded code and are susceptible to at least one kind of obfuscation (see Tables V and VI). This is significant and, within our own experiments in Section 5, 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, drive-by-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 [Poeplau et al. 2014; Fedler et al. 2013]. 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 [Mulliner et al. 2014]. Native-based attacks can also be used on at least 30% of the million apps Andrubis analyzed as they were vulnerable to web-based attacks by exposing native Java objects [Lindorfer et al. 2014]. Despite this, as can be seen in Tables V and VI, 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 analyzed malware (e.g., failed during decompiling).

6.1.3. Weaknesses in AV Products. To evaluate AV products, Zhou and Jiang [2012b] 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 5, 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 [Poeplau et al. 2014]. Unfortunately, the inner workings of Google Bouncer and similar systems are not available but can still be evaluated. In 2013, DroidChameleon [Rastogi et al. 2013] submitted automatically obfuscated Android apps to ten popular AV products and found all ten vulnerable to trivial transformations, the lowest of the three transformations "tiers". Approximately 86% of apps also use repackaging [Zhou and Jiang 2012b]. This is significant as at least 43% of the malware signatures are not based on code-level artifacts 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), then they can deter 90% of popular AV products [Rastogi et al. 2013]. Half of Android apps also use Java reflection to access API calls, which is a top-tier obfuscation method [Felt et al. 2011]. In 2012, ADAM [Zheng et al. 2013a] showed results similar to DroidChameleon even when analyzing 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 detection rates further from 16.5% to 42.8%, implying that higher-tiered obfuscations 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 after ADAM, Vidas and Christin [2013] found that AV detection rates fell to 0–32%.

The framework AndroTotal [Maggi et al. 2013] can also be used to analyze a malware with multiple mobile AV products to compare their results. In 2014, Morpheus [Kazdagli et al. 2014] 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

Table V. Android Malware Analysis Frameworks; Superscript "M" for Malicious, "B" for Benign, "GP" for Google Play

Year	Framework	Method	Samples	Sample Selection	Scalable	Sturdiness
2012	Aurasium [Xu et al. 2012]	sandbox (dynamic) detect API misuse	third party	3491^{b} (99.6% detect), 1260^{m} (99.8% detect)	low overhead	× native code, java refl. × sandbox transparency
2012	PScout [Au et al. 2012]	perm. spec. from OS source code & APK + stim. (UI fuzzing)	GP	1,260 chosen for API coverage	—	× unfeasible paths = false mappings
2012	AppGuard [Backes et al. 2013]	app rewriter + dynamic inline ref. monitoring + stimulation	GP, SlideMe	25,000 apps tested for robustness (stimulation)	low overhead	× no callee-site rewriting ✓ java refl., dy. loaded code
2012	DroidScope [Yan and Yin 2012]	dynamic + virt. + reconstruct OS & Java-level semantics	GP	7 bench. (efficiency & capability) + 2^{m}	taint 11×- 34× slow↓	✓ Java, JNI, ELF × limited code coverage
2012	I-ARM-DROID [Davis et al. 2012]	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 [Zheng et al. 2012]	statically find activity paths + dynamic to find triggers	—	19 wild apps (7 fam.) w/ UI triggered mal.	6/7 <1.5 mins	× native code × cannot reveal hidden UI
2013	CopperDroid [Tam et al. 2015b]	VM-based dynamic analysis + stimulation	several sources	$1{,}200^{m}$ (49 families) 400^{m} (13 families)	∽10min/ app	✓ Java, JNI, ELF ✓ ∽ 25% more behaviors
2013	Jin et. al Jin and Wang [2013]	software-defined network traffic monitoring	—	4 mobiles 100 IPs	∽746k response/s	× encrypted traffic ✓ monitors traffic from all OSs
2013	ContentScope [Yajin Zhou 2013]	static path-sensitive data-flow + dynamic exec confirmation + classify leak/pollution	markets (mult.)	62,519 apps (3,018 vul.) from Feb. 2012	-	× false +'s (static & start errors) × manual classification
2013	Contest [Anand et al. 2012]	concolic app testing (generate event sequence for auto. tests)	—	5 open-source apps	∽hour/app	✓ lessens path explosion × only handles tap events
2013	Droid Analytics [Zheng et al. 2013b]	multi-lvl (method, class, payload) op code signatures (static)	markets, web	$148k^{b}$ $2k^{m}$ (234 families)	∽70s/app	✓ repackaged code obfuscation × logic obfuscation
2013	Pegasus [Chen et al. 2013]	static + runtime policy monitoring + API/permission event graphs	—	152^{m}, 117^{b}	80% 0.5h, max 5.6h	✓ captures event fire context × detects obfu but still vuln.
2013	ProfileDroid [Wei et al. 2012b]	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 [Hoffmann et al. 2013]	static (smali) auto and optional manual	GP	free apps: $136k^{b}$, $6k^{m}$	<10s/ app	× reflection (no backtrack) × runtime info (native code)
2013	VetDroid [Zhang et al. 2013]	dynamic permission usage + reconstruct fine-grained actions	GP	32 categories top $1.2k^{b}$ apps	2min/ app	— slow↓ 32% on device × native code, java code
2014	Rasthofer et al. [2014]	dy + taint + machine learning + API feat.	Virus Share	$11k^{m}$ apps with API data leaks	SVN, QP-prob.	✓ supports all Android OSs — not tested w/ obfuscation
2014	RiskMon [Jing et al. 2014]	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	[Poeplau et al. 2014]	edit DVM + control flow graph (method) + dyn. code loading	GP	popular free 1.6k 2012–Aug 2013	relies on white list	× code executed in default app config × custom integrity checks
2014	A5 [Vidas et al. 2014]	static execution paths via Activities + dy. network ID sigs.	public source	1,260 malicious apps	avg. 149s/app	✓ attempts vm transparency × dynamically loaded intents
2015	DroidSafe [Gordon et al. 2015]	static information flow + hooks + calls that start activities	real-world apps	24 modified apps for hooking	<222s per analysis	✓ avoid irrelevant classes × dynamically loaded code

Table VI. Android Malware Detection Frameworks; Superscripts "M" Malicious, "B" Benign, and "P" for Google Play

Year	Framework	Method	# Apps	Sample Selection	Result	False +/-	Scalable	Sturdiness
2011	Andromaly [Shabtai et al. 2012]	machine learn + real-time monitor hardware	4	maliciously self-developed	detect all malware	low/ ?	perf. 10% ↓	× misses "quick" actions
2011	Crowdroid [Burguera et al. 2011]	dynamic system call logs + k-means cluster	5	3-developed, 2-real	100%-self, 93%-real	Yes/ -	NP-hard	× needs cloud connection — scalable in comparison
2012	Droid MOSS [Yajin Zhou 2013]	fuzzy hash + static sigs + dynamic comparison	200	random (6 world markets)	5–13% repackaged	10%/ 10%	—	× assumes legit apps × incomplete list of shared libs
2012	RiskRanker [Grace et al. 2012]	study native code, encryption, dynamic code loading	118k	mult. markets (end 2011), 29 families	detect 322 zero-day malware	Yes (?/?)	all in 4 days	× downloaded exploits × tests small behavior set × obfuscation/encryption
2012	ScanDal [Kim et al. 2012]	static + sensitive APIs + sources/sinks	90	9 free pop, random type July'11	detect 11 leaks	18% /?	83s-49m	✓ simple reflection × native code, refl., obf.
2012	DroidMat [Wu et al. 2012]	static + perms & API + components	238^{m} 1500^{b}	Contagio&GP (50^{b} apps)	quick and accurate detection	0.4%/ 12.6%	Y	✓ 50% faster than Androguard × native code, refl., obf.
2013	AppIntent [Yang et al. 2013]	static + symbolic exec. + event-space reduction	1750	1000^{b} GP 750^{m}	detect 582, sym. 358	164/ low	symb. <2hrs	✓ see user vs. background × native code
2013	AppProfiler [Rosen et al. 2013]	static + map API to behavior	80k	15 diverse & popular apps	detect ∽59% behaviors	16%/ 15%	500 a/ day	✓ see user vs. background × obfu. class/pkg names, native code
2013	MAMA [Sanz et al. 2013]	extract Manifest features + machine learning	333^{m} 333^{b}	max coverage / diversity, 2011	best detects 94%	best 5% /?	—	✓ wide app coverage × Internet/piggy payloads
2013	PiggyApp [Zhou et al. 2013]	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 [Zhang and Yin 2013]	static bytecode + program slices	16	vulnerable apps	patch vul apps	0%/ ?	most <60s	✓ device shadowing/patches — app size +16-45% — app slowdown 2%
2014	Droid Barrier [Almohri et al. 2014]	hidden shells w/ own proc. + credentials	$\backsim 400^{m}$	3 malware fams	36.7% use hidden shell	—	preform. penalty <13%	✓ isolated in kernel mode × kernel level & embedded attacks
2014	Apposcopy [Feng et al. 2014]	static taint+control/ data flow+semantic sig	$1k^{m}$ $8k^{P}$	in the 8k there were 6^{m}	classify family	10%/ 0.2%	not instant	✓ low level obfuscation × native code
2014	AsDroid [Huang et al. 2014]	static dex to jar + intent propagation/-correlation + ICC call chains	128	free pop. apps (GP, Contagio, 3rd Party)	model stealthy behaviors		28% / 11%	× flow obfu., native code, refl. × only textual UI
2014	DroidInfer [Huang et al. 2015]	static + jimple + taint analysis w context	22^{m} 144^{b} 39	free pop. apps (GP, Contagio, DroidBench)	data flow in logs & network	16%/ —	∽2 mins	— partial ICC flows — +2GB, edits libraries ✓ Does well with DroidBench
2015	AppContext [Yang et al. 2015]	Soot + Dexpler + Extended call graphs	202^{m} 633^{b}	GP	context-based detection		∽12% / 5%	— some dynamic code, refl. × Pscout's drawbacks (relies on Pscout mappings)

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 [Felt et al. 2011]), are significantly more successful than lower tiers [Maiorca et al. 2015]. Besides being heavily obfuscated against static analysis, sophisticated malware is also bypassing dynamic analyses like Google Bouncer by detecting emulated environments.

6.1.4. 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 [Lookout 2010], researchers lacked a solid, standard dataset to work with. Many

instead wrote their own malware to assess their systems [Shabtai et al. 2012]. Others collected and shared samples with website crawlers, such as Contagio [2014]. These approaches, however, yielded limited datasets, hindering thorough system evaluations. In 2012, the MalGenome project [Zhou and Jiang 2012b] attempted to fix that, as it contained 1,260 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 threefold.

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 [Arp et al. 2014], a collection of 5,560 malware from 179 different families collected between August 2010 and October 2012, but considering the continuing increase in malware (400% from 2012 to 2013 [Symantec 2013]), and all the new sophisticated malware after 2012 (e.g., Oldboot, Android.HeHe), a more complete and up-to-date dataset is necessary [McAfee 2013, 2014]. For reasons we will explain later in this section, it is also essential to have a diverse dataset with samples from a range of years, app categories, popularity, markets, among others.

6.1.5. 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 a simple, open-source, popular, reasonably sized OS, such as Android. Therefore, reliable and portable Android analysis frameworks may be even more essential. For example, there are already several smart TVs and watches powered by Android (i.e., Android Wear watches that communicate with the user's phone) on the market for public consumption. Efforts have also been made to adopt the Android operating system for satellites, espresso makers, game controllers, and refrigerators [Vance 2013]. If the IoT were to adopt smaller, altered versions of the Android OS, then 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 added security would be even more effective, if done in conjunction with improved application market vetting methods.

6.2. Mobile Security Effectiveness

To evaluate the present status of Android malware analysis and detection frameworks, this article provides Tables V and VI. These provide details on framework methods (e.g., static or dynamic), sample selection process, scalability, accuracy, and sturdiness.

6.2.1. Analyzed Datasets. As mentioned previously in Section 4, sample selection is essential as different markets, social circles, and geographic locations are often infected by different malware and in different amounts [Zhou et al. 2012; Juniper 2013; Securelist 2013; Lever et al. 2013; McAfee 2014]. Despite this, many studies only use one app source and either choose several apps per category (e.g., games, business) or select apps that best suit their research needs (see Tables V and VI). For example, SmartDroid [Zheng et al. 2012] chose a small set of malware triggered by UI to test its system specifically for revealing UI-based triggers. Ideally, however, malware samples should be chosen from several families to provide a more diverse set of behaviors, including evasion techniques, with which to test the system. Hence, for most cases, a diverse, representative dataset is desired. In actuality, however, obtaining a truly representative dataset can be a real challenge.

AppProfiler [Rosen et al. 2013] discovered that popular Google Play apps exhibited more behaviors, and were more likely to monitor the hardware, than an average app.

This is significant, as many studies, such as VetDroid, I-ARM-DROID, and ScanDal, only analyzed popular apps. While a dataset of only free, popular apps may provide more malicious behaviors to analyze, the selection would not be a reasonable representation of the Android markets as a whole. Similarly, a significant number of studies only analyze 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. It could be argued that popular apps affect more users and are therefore more essential for research. However, the difference between free/paid samples should be at least considered when choosing a dataset. Furthermore, applications selected from markets should represent several categories, as each category entails a unique functionality and set of behaviors for study.

In the future, up-to-date datasets should continue to be expanded by incorporating new samples from multiple sources to provide more globally representative, and diverse, datasets. If used correctly, specialized datasets may also be gathered for benchmarking (e.g., DroidBench), and testing types of obfuscation (e.g., DroidChameleon). Among other uses, datasets would also be highly useful to identify specific weakness or traits in analyses, detection, and classification techniques.

6.2.2. Scalability, Accuracy, and Portability. Scalability 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 analyzed 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, and 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 [Zhang et al. 2014].

Tables V and VI attempt to make note of any performance statistics or scalability information. They also attempt to base each framework's sturdiness on several key points, made previously concerning native code, Java reflection, VM-awareness, and obfuscation. With Tables V and VI, we discovered that several systems were able to detect, but not analyze, samples with such traits. Furthermore, these traits often contributed to their false positives/negatives. An encouraging number of frameworks such as Apposcopy [Feng et al. 2014] are making efforts to overcome limitations like low levels of obfuscation but still do not cope with higher ones. Portability is also essential so malware can be analyzed on multiple Android OS versions, as they have different vulnerabilities, and to minimize the window of vulnerability whenever a new Android version is released.

6.2.3. Significant Changes in Android. One of the most recent significant changes to Android has been the switch from the Dalvik runtime to the ART runtime. This was introduced in Android version 5.0 in 2015. Moderate changes to the operating system itself are introduced with each Android version which, as seen in Table VII, happens frequently (i.e., more often than traditional operating systems). Changes to the kernel have also been made, albeit less frequently, with the introduction of Android 2.x and 4.x [Tam et al. 2015a]. Ideally, solutions should be agnostic to the parts of Android that may frequently change; 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 tune with specific aspects of Dalvik runtime internals. It is possible that no more drastic changes will be made to the Android OS, but ideally frameworks and techniques should be resilient or easily adaptable to changes within Android. The benefit of this is high portability across all Android versions, possible Android variants applied to the future IoT, and possibly even other platforms.

Table VII. 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						
Doughnu		1.6						
Eclair		2.0.[0,1]	2.1					
Froyo			2.2	2.2.[1,2,3]				
Gingerbread			2.3.[0,1]	2.3.[2,3,4,5,6,7]				
Honeycor				3.[0,1,2], 3.2.[1,2,3,4]	3.2.[5,6]			
Icecream Sandwich				4.0.[0,1,2,3]				
Jelly Bean					4.1.[0,1,2], 4.2, 4.2.1	4.2.2, 4.3.[0,1]		
KitKat						4.4.[0,1,2]	4.4.[3,4]	
Lollipop							5.0.[0,1,2]	5.1.[0,1]

6.3. Future Research Directions

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 and contemporary malware [Securelist 2013; Symantec 2013; McAfee 2015; Vidas and Christin 2013]. We therefore suggest the following areas for future research.

6.3.1. Hybrid Analysis and Multi-leveled Analysis. Static solutions are beginning to harden against trivial obfuscations [Feng et al. 2014], but many apps, and most malware, are already using higher levels of obfuscation [Felt et al. 2011; Securelist 2013]. As recent static systems are still effective, and scalable, we suggest that, in the cases where obfuscation (e.g., native code, reflection, encryption) is detected, dynamic analysis can be used in conjunction for completion. Alternatively, dynamic solutions inherently have less code coverage but can use static analysis to guide analyses through more paths [Zheng et al. 2012; Spreitzenbarth et al. 2013; Mahmood et al. 2014] or use apps exercisers like MonkeyRunner, manual input, or intelligent event injectors [Azim and Neamtiu 2013; Machiry et al. 2013; Mahmood et al. 2014].

Hybrid solutions could therefore combine static and dynamic analysis in ways that their added strengths mitigate each other's weakness. For example, the Harvester [Rasthofer et al. 2016] tool can reduce obfuscation generated by encrypted strings and reflective methods with its hybrid methods. It also seems beneficial to develop multi-level systems, as it often provides more, and 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. Parallel processing could also greatly enhance multi-level analyses and provide faster detection systems [Dini et al. 2012]. The downside of this multi-level methods, however, is it can cause large additional overhead, decrease transparency, increase chances of code bugs, and may be less portable.

6.3.2. 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 behaviors. There are several benefits to dynamic out-of-the-box solutions, considering the launch of ART [Vitas 2013], like being able to cope with multiple 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 analyze Android-level behaviors and dynamic network behaviors [Tam et al. 2015b] and can be used to stop certain root exploits [Vidas and Christin 2014]. While hybrid solutions and smarter stimulations (e.g., IntelliDroid, a static and dynamic API-based input generator [Wong and Lie 2016]) would greatly increase code coverage, different approaches should be further researched based on malware trends. For example, while manual input is normally not scalable, crowdsourcing [Gianazza et al. 2014] may be an interesting approach. However, zero-day malware will introduce complications as time is needed to create and collect user input traces.

Code coverage also introduces an interesting question on whether malware tend to use "easily" chosen paths to execute more malicious behavior 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 that may be beneficial is identifying and understanding subsets of malware behavior through path restrictions (e.g., remove permissions or triggers like user UI or system events) to see which behavior equates to whic 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. This would be much more difficult to implement if any modifications were made to the Dalvik VM or the OS to accommodate for high-level analyses but feasible with out-of-the-box analyses.

6.3.3. 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 and interesting [Zaddach et al. 2014]. Newer, more sophisticated, malware from 2014 and 2015 are becoming increasingly aware of emulated environments, but achieving a perfect emulator is, unfortunately, unfeasible. Things such as a timing attack, where certain operations are timed for discrepancies, are still open problems for traditional malware as well and are difficult to fool.

Furthermore, malware such as DenDroid and Android.hehe do not just detect their emulated environments but often hide their malicious behaviors 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 the serial number of the device [Petsas et al. 2014]. It is also possible to fingerprint and identify particular emulated environments, such as different dynamic analysis frameworks, via the device performance features aforementioned [Maier et al. 2014]. One solution to this problem would be to use real devices in all dynamic experiments. However, this makes analyzing 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.

Here we would also like to propose combining real devices and emulators as a new hybrid solution, where real devices pass necessary values to emulators to enhance their transparency. Data from a real device can also be slightly and randomly altered in order to generate information for multiple emulators. This would ideally reduce the cost and speed of experiments while revealing more malicious behaviors. A similar hybrid device method has proved to be effective for analyzing embedded systems' firmware [Zaddach et al. 2014], and it would be very interesting to see if it would also work for Android malware detection and analysis and how effective it would be against VM-aware malware. As an alternative to virtualization, it would also be interesting to see if splitting the kernel, where untrusted system calls are directed to

the hardened kernel code, can be applied to Android. 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 [Kurmus and Zippel 2014]. Last, we look forward to new technology, such as the new ARM with full virtualization support, and more explorations into ART and its new challenges.

7. CONCLUSION

This article studied a wide range of Android malware analysis and detection frameworks, illustrating changing trends in their methods. This article also discussed Android malware's ability to obstruct analysis and avoid detection, including its roots in traditional malware when applicable. By analyzing both threats and solutions, this article evaluated the effectiveness of several current analysis and detection methods in order to understand, and suggest, several areas for future research in more scalable, portable, and accurate manners for Android. This differs from previous surveys studying mobile security in general, Android malware attacks only, and more general Android security.

REFERENCES

Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupe, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. 2016. Going native: Using a large-scale analysis of Android apps to create a practical native-code sandboxing policy. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

Hussain M. J. Almohri, Danfeng (Daphne) Yao, and Dennis Kafura. 2014. DroidBarrier: Know what is executing on your Android. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*.

A. Amamra, C. Talhi, and J. Robert. 2012. Smartphone malware detection: From a survey towards taxonomy. In *Malicious and Unwanted Software (MALWARE)*.

B. Amos, H. Turner, and J. White. 2013. Applying machine learning classifiers to dynamic Android malware detection at scale. In *Wireless Communications and Mobile Computing Conference (IWCMC)*.

Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Foundations of Software Engineering (FSE)*.

Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. 2011. Cells: A virtual mobile smartphone architecture. In *ACM Symposium on Operating Systems Principles (SOSP)*.

Apple. 2015. iOS developer library. Retrieved from https://developer.apple.com/library/ios/navigation/.

Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. 2014. DREBIN: Effective and explainable detection of Android malware in your pocket. In *Network and Distributed System Security Symposium*.

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM Programming Language Design and Implementation*.

Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, Phillipa Gill, and David Lie. 2011. Short paper: A look at smartphone permission models. In *ACM Security and Privacy in Smartphones and Mobile Devices (SPSM)*.

Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android permission specification. In *ACM Computer and Communications Security (CCS)*.

Schmidt Aubrey-Derrick and A. Sahin. 2008. *Malicious Software for Smartphones*. Technical Report. Universität Berlin.

Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of Android apps. In *ACM Object Oriented Programming Systems Languages (OOPSLA)*.

Michael Backes, Sven Bugiel, Sebastian Gerling, and Philipp von Styp-Rekowsky. 2014. Android security framework: Extensible multi-layered access control on Android. In *Annual Computer Security Applications Conference*.

Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. 2013. AppGuard—fine-grained policy enforcement for untrusted Android applications. In *Data Privacy Management (DPM)*.

Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. 2009. A view on current malware behaviors. In *USENIX Large-scale Exploits and Emergent Threats (LEET)*.

M. Becher, F. C. Freiling, J. Hoffmann, T. Holz, S. Uellenbeck, and C. Wolf. 2011. Mobile security catching up? Revealing the nuts and bolts of the security of mobile devices. In *IEEE Security and Privacy (S&P)*.

Michael Becher and Felix C. Freiling. 2008. Towards dynamic malware analysis to increase mobile device security. In *Sicherheit*.

Michael Becher and Ralf Hund. 2008. Kernel-level interception and applications on mobile devices. Technical Report. Department for Mathematics and Computer Science, University of Mannheim; TR-2008-003. http://ub-madoc.bib.uni-mannheim.de/1933/.

Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. 2011. MockDroid: Trading privacy for application functionality on smartphones. In *Mobile Computing Systems and Applications (HotMobile)*.

BlackBerry. 2013. Architecture and data flow overview. Retrieved from https://help.blackberry.com/en/bes10/10.2/.

T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak. 2010. An Android application sandbox system for suspicious software detection. In *Malicious and Unwanted Software (MALWARE)*.

Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. 2008. Behavioral detection of malware on mobile handsets. In *ACM Mobile Systems, Applications, and Services (MobiSys)*.

Rodrigo Branco, Gabriel Barbosa, and Pedro Neto. 2012. Scientific but not academical overview of malware anti-debuggin, anti-disassembly and anti-VM technologies. Blackhat USA.

T. K. Buennemeyer, T. M. Nelson, L. M. Clagett, J. P. Dunning, R. C. Marchany, and J. G. Tront. 2008. Mobile device profiling and intrusion detection using smart Batteries. In *Hawaii International Conference on System Sciences (HICSS)*.

Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. 2011. Practical and lightweight domain isolation on Android. In *Security & Privacy in Smartphones & Mobile Devices (SPSM)*.

Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. 2011. Crowdroid: Behavior-based malware detection system for android. In *ACM Security and Privacy in Smartphones and Mobile Devices (SPSM)*.

Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. 2013. MAST: Triage for market-scale mobile malware analysis. In *ACM Security and Privacy in Wireless and Mobile Networks (WiSec)*.

Kevin Zhijie Chen, Noah Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Tom Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Song. 2013. Contextual policy enforcement in Android applications with permission event graphs. In *Network and Distributed System Security Symposium (NDSS)*.

Jerry Cheng, Starsky H. Y. Wong, Hao Yang, and Songwu Lu. 2007. SmartSiren: Virus detection and alert for smartphones. In *ACM Mobile Systems, Applications, and Services (MobiSys)*.

Christian Collberg, Clark Thomborson, and Douglas Low. 1997. A taxonomy of obfuscating transformations. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.9852.

Contagio. 2014. Contagio. Retrieved from http://contagiodump.blogspot.com/.

Jonathan Crussell, Clint Gibler, and Hao Chen. 2012. Attack of the clones: Detecting cloned applications on aAndroid markets. In *European Symposium on Research in Computer Security (ESORICS)*.

B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. 2012. I-ARM-Droid: A rewriting framework for in-app reference monitors for Android applications. In *IEEE Mobile Security Technologies (MoST)*.

Anthony Desnosi and Geoffroy Gueguen. 2012. Android: From reversing to decompilation. In *Black Hat Abu Dhabi*.

Michael Dietz, Shashi Shekhar, Dan S. Wallach, and Anhei Shu Yuliy Pisetsky. 2011. QUIRE: Lightweight provenance for smart phone operating systems. In *USENIX Security (SEC)*.

Daniel Eran Dilger. 2014. New Android RAT infects Google play apps. Retrieved from http://appleinsider.com/articles/14/03/07/new-android-rat-infe cts-google-play-apps-turning-phones-into-spyware-zombies.

Gianluca Dini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. 2012. MADAM: A multi-level anomaly detector for Android malware. In *Mathematical Methods, Models, and Architectures for Computer Network Security*.

Toralv Dirro. 2011. Straight from the anti-malware labs. Retrieved from http://www.mcafee.com/uk/resources/reports/rp-mobile-security-consumer-trends.pdf.

Alessandro Distefano, Antonio Grillo, Alessandro Lentini, and Giuseppe F. Italiano. 2010. SecureMyDroid: Enforcing security in the mobile devices lifecycle. In *ACM Cyber Security and Information Intelligence Research (CSIIRW)*.

Joshua Drake, Zach Lanier, Collin Mulliner, Pau Oliva Fora, Stephen A. Ridley, and Georg Wicherski. 2014. *Android Hacker's Handbook* (1st ed.). Wiley Publishing.

Ken Dunham. 2009. *Mobile Malware Attacks & Defense*. Syngress.

William Enck. 2011. Defending users against smartphone apps: Techniques and future directions. In *Information Systems Security Association (ISSA)*.

William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX Operating Systems Design and Implementation (OSDI)*.

William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. 2011. A study of Android application security. In *USENIX Security (SEC)*.

F-Secure. 2013. Android accounted for 79% of all mobile malware in 2012, 96% in q4 alone. Retrieved from http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q4_2012.pdf.

P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan. 2015. Android security: A survey of issues, malware penetration, and defenses. In *IEEE Communications Surveys Tutorials*.

Rafael Fedler, Marcel Kulicke, and Julian Schütte. 2013. Native code execution control for attack mitigation on Android. In *ACM Security and Privacy in Smartphones and Mobile Devices (SPSM)*.

Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *ACM Computer and Communications Security (CCS)*.

Adrienne Porter Felt, Serge Egelman, and David Wagner. 2012. 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)*.

Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. 2011. A survey of mobile malware in the wild. In *ACM Security and Privacy in Smartphones and Mobile Devices (SPSM)*.

Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based detection of Android malware. In *ACM Foundations of Software Engineering (FSE)*.

Torsten Frenzel, Adam Lackorzynski, Alexander Warg, and Hermann Hrtig. 2010. ARM TrustZone as a virtualization technique in embedded systems. In *OSADL Real-Time Linux Workshop (RTLWS)*.

Tal Garfinkel and R. Mendel. 2003. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*. 191–206.

Gartner. 2015. Devices by operating system and user type. Retrieved from http://www.gartner.com/newsroom/id/3010017.

Andrea Gianazza, Federico Maggi, Aristide Fattori, Lorenzo Cavallaro, and Stefano Zanero. 2014. PuppetDroid: A user-centric UI exerciser for automatic dynamic analysis of similar Android applications. *ACM CoRR*. abs/1402.4826. http://arxiv.org/abs/1402.4826.

Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. 2012. AndroidLeaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Trust and Trustworthy Computing (TRUST)*.

Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. RERAN: Timing- and touch-sensitive record and replay for android. In *ACM International Conference on Software Engineering*.

Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. 2015. Information-flow analysis of Android applications in DroidSafe. In *Network and Distributed System Security Symposium*.

Alexander Gostev and Denis Maslennikov. 2009. Mobile malware evolution: An overview. Retrieved from http://www.viruslist.com/en/analysis?pubid=204792080.

Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. 2012. RiskRanker: Scalable and accurate zero-day android malware detection. In *ACM Mobile Systems, Applications, and Services (MobiSys)*.

Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. 2013. Juxtapp: A scalable system for detecting code reuse among android applications. In *Detection of Intrusions and Malware and Vulnerability*.

Gernot Heiser. 2008. The role of virtualization in embedded systems. In *Isolation and Integration in Embedded Systems*.

Dharmdasani Hitesh. 2014. Android.HeHe: Malware disconnects phone calls. Retrieved from http://www.fireeye.com/blog/technical/2014/01/Android-shehe-malware-now-disconnects-phone-calls.html.

Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. 2013. Slicing droids: Program slicing for smali code. In *ACM Symposium on Applied Computing (SAC)*.

Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In *ACM International Conference on Software Engineering (ICSE)*.

Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and precise taint analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*.

Joo-Young Hwang, Sang bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. 2008. Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. In *IEEE Consumer Communications and Networking Conference (CCNC)*.

Ham Hyo-Sik and Choi Mi-Jung. 2013. Analysis of Android malware detection performance using machine learning classifiers. In *Cybercrime and Trustworthy Computing (CTC)*.

InformationWeek. 2014. Cybercrime black markets grow up. Retrieved from http://www.informationweek.com/cybercrime-black-markets-grow-up/d/d-id/1127911.

Grant A. Jacoby. 2004. Battery-based intrusion detection. In *IEEE Global Communications (GLOBECOM)*.

Richard Jensen and Qiang Shen. 2008. *Computational Intelligence and Feature Selection: Rough and Fuzzy Approaches.* Wiley-IEEE Press.

Xuxian Jiang. 2012. An evaluation of the application ("app") verification service in Android 4.2. Retrieved from http://www.cs.ncsu.edu/faculty/jiang/appverify/.

Ruofan Jin and Bing Wang. 2013. Malware detection for mobile devices using software-defined networking. In *GENI Research and Educational Experiment Workshop (GREE)*.

Yiming Jing, Gail-Joon Ahn, Ziming Zhao, and Hongxin Hu. 2014. RiskMon: Continuous and automated risk assessment of mobile applications. In *ACM Data and Application Security and Privacy (CODASPY)*.

Juniper. 2013. Networks 3^{rd} annual mobile threats report March 2012 through March 2013. Retrieved from http://www.juniper.net/us/en/local/pdf/additional-resources/jnpr-2012-mobile-threats-report.pdf.

Min Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. Network and distributed system security symposium, (NDSS). *The Internet Society*.

Mikhail Kazdagli, Ling Huang, Vijay Reddi, and Mohit Tiwari. 2014. Morpheus: Benchmarking computational diversity in mobile malware. In *Hardware & Architectural Support for Security & Privacy (HASP)*.

Hahnsang Kim, Joshua Smith, and Kang G. Shin. 2008. Detecting energy-greedy anomalies and mobile malware variants. *ACM Mobile Systems, Applications, and Services (MobiSys)*. (2008).

Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. 2012. ScanDal: Static analyzer for detecting privacy leaks in Android applications. In *IEEE Mobile Security Technologies (MoST)*.

Mudge Kingpin. 2001. Security analysis of the palm operating system and its weaknesses against malicious code threats. In *USENIX Security*.

Tero Kuittenin. 2013. Google play app revenue rockets to more than half of iOS. Retrieved from http://bgr.com/2013/09/20/google-play-app-revenue-ios-august/.

Anil Kurmus and Robby Zippel. 2014. A tale of two kernels: Towards ending kernel hardening wars with split kernel. In *ACM Computer and Communications Security (CCS)*.

M. La Polla, F. Martinelli, and D. Sgandurra. 2013. A survey on security for mobile devices. *IEEE Communications Surveys Tutorials (COMST)*.

E. Lagerspetz, Hien Thi Thu Truong, S. Tarkoma, and N. Asokan. 2014. MDoctor: A mobile malware prognosis application. In *IEEE Conference on Distributed Computing Systems Workshops (ICDCS)*.

Charles Lever, Manos Antonakakis, Reaves, Patrick Traynor, and Wenke Lee. 2013. The core of the matter: Analyzing malicious traffic in cellular carriers. In *Network and Distributed System Security Symposium (NDSS)*.

Juanru Li, Wenbo Yang, Junliang Shu, Yuanyuan Zhang, and Dawu Gu. 2014. InDroid: An automated online analysis framework for Android applications. In *Crisis Intervention Team (CIT)*.

Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. 2015. IccTA: Detecting inter-component privacy leaks in android apps. In *ACM International Conference on Software Engineering (ICSE)*.

Tung Liam. 2014. Modded firmware may harbour worlds first Android bootkit. Retrieved from http://www.zdnet.com/modded-firmware-may-harbour-worlds-first-android-bootkit-7000025665/.

Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. 2014. ANDRUBIS-1,000,000 apps later: A view on current Android malware behaviors. In *Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*.

Lookout. 2010. Security alert: Geinimi, sophisticated new Android trojan found in wild. Retrieved from https://blog.lookout.com/blog/2010/12/29/geinimi_trojan/.

Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for Android apps. In *ACM Foundations of Software Engineering (FSE)*.

Federico Maggi, Andrea Valdi, and Stefano Zanero. 2013. AndroTotal: A flexible, scalable toolbox and service for testing mobile malware detectors. In *ACM Security and Privacy in Smartphones and Mobile Devices (SPSM)*.

Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented evolutionary testing of android apps. In *Foundations of Software Engineering (FSE)*.

Dominik Maier, Tilo Mller, and Mykola Protsenko. 2014. Divide-and-conquer: Why Android malware cannot be stopped. In *Availability, Reliability and Security (ARES)*.

Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. 2015. Stealth attacks: An extended insight into the obfuscation effects on Android malware. In *Computers & Security (JCS)*.

Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. 2012. Analysis of the communication between colluding applications on modern smartphones. In *Annual Computer Security Applications Conference (ACSAC)*.

McAfee. 2013. Threats report. Retrieved from http://www.mcafee.com/uk/resources/reports/rp-quarterly-threat-q1-2013.pdf.

McAfee. 2014. Mobile security report. Retrieved from http://www.mcafee.com/uk/resources/reports/rp-mobile-security-consumer-trends.pdf.

McAfee. 2015. Labs threats report. Retrieved from http://www.mcafee.com/uk/resources/reports/rp-quarterly-threat-q1-2015.pdf.

Joseph Menn. 2011. Smartphone shipments surpass PCs. Retrieved from http://www.ft.com/cms/s/2/d96e3bd8-33ca-11e0-b1ed-00144feabdc0.html.

M. Miettinen, P. Halonen, and K. Hatonen. 2006. Host-based intrusion detection for advanced mobile devices. In *Advanced Information Networking and Applications (AINA)*.

Yves Moreau, Peter Burge John Shawe-taylor, Christof Stoermann, Siemens Ag, and Chris Cooke Vodafone. 1996. Novel techniques for fraud detection in mobile telecommunication networks. In *Association for the Advancement of Artificial Intelligence (AAAI)*.

A. Moser, C. Kruegel, and E. Kirda. 2007. Limits of static analysis for malware detection. In *Annual Computer Security Applications Conference (ACSAC)*.

Collin Mulliner, William Robertson, and Engin Kirda. 2014. VirtualSwindle: An automated attack against in-app billing on Android. In *ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*.

D. C. Nash, T. L. Martin, D. S. Ha, and M. S. Hsiao. 2005. Towards an intrusion detection system for battery exhaustion attacks on mobile computing devices. In *IEEE Pervasive Computing and Communications (PerCom)*.

Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *USENIX Security (SEC)*.

M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. 2009. Semantically rich application-centric security in Android. In *Annual Computer Security Applications Conference (ACSAC)*.

Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards automating risk assessment of mobile applications. In *USENIX Security (SEC)*.

Bogdan Petrovan. 2015. Google is now manually reviewing apps. Retrieved from http://www.androidauthority.com/google-now-manually-reviewing-apps-submitted-to-play-store-594879/.

Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage against the virtual machine: Hindering dynamic analysis of Android malware. In *European System Security Workshop*.

Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute this! Analyzing unsafe and malicious dynamic code loading in Android applications. In *Network and Distributed System Security Symposium (NDSS)*.

M. La Polla, F. Martinelli, and D. Sgandurra. 2013. A survey on security for mobile devices. *IEEE Communications Surveys Tutorials* 15, 1 (2013), 446–471. DOI:10.1109/SURV.2012.013012.00028

Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A machine learning approach for classifying and categorizing Android sources and sinks. In *Network and Distributed System Security Symposium (NDSS)*.

Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting runtime values in Android applications that feature anti-analysis techniques. In *23nd Annual Network and Distributed*

System Security Symposium (NDSS). San Diego, California, USA. http://www.internetsociety.org/sites/default/files/blogs-media/harvesting-runtime-values-android-applications-feature-anti-analysis-techniques.pdf.

Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. 2013. DroidChameleon: Evaluating Android anti-malware against transformation attacks. In *ACM Special Interest Group on Security, Audit and Control (SIGSAC)*.

Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. 2012. AppInsight: Mobile app performance monitoring in the wild. In *Operating Systems Design and Implementation (OSDI)*.

The Register. 2013. Earn 8,000 a month with bogus apps from Russian malware factories. Retrieved from http://www.theregister.co.uk/2013/08/05/mobile_malware_lookout/.

Sanae Rosen, Zhiyun Qian, and Z. Morely Mao. 2013. AppProfiler: A flexible method of exposing privacy-related behavior in Android applications to end users. In *ACM Conference on Data & Application Security & Privacy (CODASPY)*.

Ethan Rudd, Andras Rozsa, Manuel Gunther, and Terrance Boult. 2016. A survey of stealth malware: Attacks, mitigation measures, and steps toward autonomous open world solutions. CoRR. abs/1603.06028. http://arxiv.org/abs/1603.06028.

Didier Samfat and Refik Molva. 1997. Idamn: An intrusion detection architecture for mobile networks. *IEEE Journal on Selected Areas in Communications (J-SAC)* 15, 7 (Sept. 1997), 1373–1380. DOI:10.1109/49.622919

Andreas Terzis Sandeep Sarat. 2007. On the detection and origin identification of mobile worms. In *ACM Workshop on Rapid Malcode (WORM)*.

Chit La Pyae Myo Hein and Khin Mar Myo. 2016. Characterization of malware detection on Android application. *Genetic and Evolutionary Computing: Proceedings of the Ninth International Conference on Genetic and Evolutionary Computing*, Thi Thi Zin, Jerry Chun-Wei Lin, Jeng-Shyang Pan, Pyke Tin, and Mitsuhiro Yokota (Eds.). Vol. 1. Springer International Publishing, Yangon, Myanmar, 113–124. DOI:10.1007/978-3-319-23204-1_13

Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. 2012. Android permissions: A perspective combining risks and benefits. In *Symposium on Access Control Models & Technologies*.

A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K. A. Yuksel, S. A. Camtepe, and S. Albayrak. 2009a. Static analysis of executables for collaborative malware detection on Android. *IEEE International Conference on Communications (ICC)*.

A.-D. Schmidt, J. H. Clausen, A. Camtepe, and S. Albayrak. 2009b. Detecting symbian OS malware through static function call analysis. In *Malicious and Unwanted Software (MALWARE)*.

Securelist. 2013. Mobile malware evolution: 2013. Retrieved from https://www.securelist.com/ en/analysis/204792326/Mobile-Malware-Evolution-2013.

Asaf Shabtai and Yuval Elovici. 2010. Applying behavioral detection on Android-based devices. In *Mobilware*.

Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. 2012. "Andromaly": A behavioral malware detection framework for Android devices. *Journal of Intelligent Information Systems (JIIS)* 38, 1 (2012), 161–190. DOI:10.1007/s10844-010-0148-x

SlideME. 2013. SlideME Android apps market: Download free & paid Android application. Retrieved from http://slideme.org/.

Alexey Smirnov, Mikhail Zhidko, Yingshiuan Pan, Po-Jui Tsao, Kuang-Chih Liu, and Tzi-Cker Chiueh. 2013. Evaluation of a server-grade software-only ARM hypervisor. In *IEEE Conference on Cloud Computing (CLOUD)*.

Sophos. 2012. Angry birds malware—Firm fined 50,000 for profiting from fake Android apps. Retrieved from http://nakedsecurity.sophos.com/2012/05/24/angry-birds-malware-fine/.

Sophos. 2014. Feejar-B. Retrieved from http://www.sophos.com/en-us/threat-center/threat-analyses/viruses-and-spyware/Andr Feejar-B.aspx.

Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. 2013. Mobile-sandbox: Having a deeper look into android applications. In *ACM Symposium on Applied Computing (SAC)*.

Tim Strazzere. 2014. The new NotCompatible. Retrieved from https://blog.lookout.com/blog/2014/11/19/notcompatible/.

G. Suarez, J. E. Tapiador, P. Peris-Lopez, and A. Ribagorda. 2014. Evolution, detection and analysis of malware for smart devices. *IEEE Communications Surveys Tutorials (COMST)*.

Sufatrio, Darell J. J. Tan, Tong-Wei Chua, and Vrizlynn L. L. Thing. 2015. Securing Android: A survey, taxonomy, and challenges. *ACM Computing Survey*.

Symantec. 2013. Mobile adware and malware analysis. Retrieved from http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/madware_and_malware_analysis.pdf.

Symantec. 2014. The future of mobile malware. Retrieved from http://www.symantec.com/connect/blogs/future-mobile-malware.

Kimberly Tam, Nigel Edwards, and Lorenzo Cavallaro. 2015a. Detecting Android malware using memory image forensics. In *Engineering Secure Software and Systems (ESSoS) Doctoral Symposium*.

Kimberly Tam, Salahuddin Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015b. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *Network and Distributed System Security Symposium (NDSS)*.

Techcrunch. 2013. Android accounted for 79 alone, says f-secure. Retrieved from http://techcrunch.com/2013/03/07/f-secure-android-accounted-for-79-of-all-mobile-malware-in-2012-96-in-q4-alone/.

Peter Teufl, Michaela Ferk, Andreas Fitzek, Daniel Hein, Stefan Kraxberger, and Clemens Orthacker. 2014. Malware detection by applying knowledge discovery processes to application metadata on the Android market (Google play). *Journal Security and Communication Networks (SCN)*.

Hien Thi Thu Truong, Eemil Lagerspetz, Petteri Nurmi, Adam J. Oliner, Sasu Tarkoma, N. Asokan, and Sourav Bhattacharya. 2013. The company you keep: Mobile malware infection rates and inexpensive risk indicators. *ACM Computing Research Repository (CoRR)*.

Roman Unuchek. 2013. The most sophisticated Android trojan. Retrieved from http://www.securelist.com/en/blog/8106/The_most_sophisticated_Android_Trojan.

Ashlee Vance. 2013. Behind the "Internet of Things" is Android. Retrieved from http://www.bloomberg.com/bw/articles/2013-05-29/behind-the-internet-of-things-is-android-and-its-everywhere.

Prashant Varanasi and Gernot Heiser. 2011. Hardware-supported virtualization on ARM. In *APSys*.

Timothy Vidas and Nicolas Christin. 2013. Sweetening android lemon markets: Measuring and combating malware in application marketplaces. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*.

Timothy Vidas and Nicolas Christin. 2014. PREC: Practical root exploit containment for Android devices. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*.

Timothy Vidas, Jiaqi Tan, Jay Nahata, Chaur Lih Tan, Nicolas Christin, and Patrick Tague. 2014. A5: Automated analysis of adversarial Android applications. In *ACM Security and Privacy in Smartphones and Mobile Devices (SPSM)*.

Timothy Vidas, Daniel Votipka, and Nicolas Christin. 2011. All your droid are belong to us: A survey of current android attacks. In *USENIX Conference on Offensive Technologies (WOOT)*.

Marko Vitas. 2013. ART vs Dalvik. Retrieved from http://www.infinum.co/the-capsized-eight/articles/art-vs-dalvik-introducing-the-new-android-runtime-in-kit-kat. (2013).

Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. AmAndroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. *Computer & Communications Security (CCS)*.

Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. 2012a. Permission evolution in the android ecosystem. In *Annual Computer Security Applications Conference (ACSAC)*.

Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. 2012b. ProfileDroid: Multi-layer profiling of android applications. In *ACM Mobile Computing and Networking (MobiCom)*.

Lukas Weichselbaum, Matthias Neugschwandtner, Martina Lindorfer, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. 2012. Andrubis: A tool for analyzing unknown android applications. Retrieved from http://blog.iseclab.org/2012/06/04/andrubis-a-tool-for-analyzing-unknown-android-applications-2/.

Johannes Winter, Paul Wiegele, Martin Pirker, and Ronald Tögl. 2012. A flexible software development and emulation framework for ARM TrustZone. In *International Conference on Trustworthy Systems (INTRUST)*.

Michelle Wong and David Lie. 2016. Going native: Using a large-scale analysis of Android apps to create a practical native-code sandboxing policy. In *Network and Distributed System Security Symposium (NDSS)*.

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

Cui Xiang, Fang Binxing, Yin Lihua, Liu Xiaoyi, and Zang Tianning. 2014. AirBag: Boosting smartphone resistance to malware infection. In *Network and Distributed System Security Symposium (NDSS)*.

Meng Xu, Chengyu Song, Yang Ji, Ming-Wei Shih, Kangjie Lu, Cong Zheng, Ruian Duan, Yeongjin Jang, Byoungyoung Lee, Chenxiong Qian, Sangho Lee, and Taesoo Kim. 2016. Toward engineering a secure

android ecosystem: A survey of existing techniques. *ACM Comput. Surv.* 49, 2 (Aug. 2016), 38:1–38:47. DOI:10.1145/2963145

Rubin Xu, Hassen Saïdi, and Ross Anderson. 2012. Aurasium: Practical policy enforcement for Android applications. In *USENIX Security (SEC)*.

Xuxian Jiang Yajin Zhou. 2013. Detecting passive content leaks and pollution in Android applications. In *Network and Distributed System Security Symposium (NDSS)*.

Lok Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *USENIX Security (SEC)*.

Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. 2015. AppContext: Differentiating malicious and benign mobile app behaviors using context. In *International Conference on Software Engineering*.

Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. 2013. AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection. In *ACM Computer and Communications Security (CCS)*.

Suleiman Y. Yerima, Sakir Sezer, and Gavin McWilliams. 2014. Analysis of Bayesian classification-based approaches for Android malware detection. *IET Information Security (IETIS)*.

Wei You, Bin Lian, Wenchang Shi, and Xiangyu Zhang. 2015. Android implicit information flow demystified. In *Asia Computer and Communications Security (AsiaCCS)*.

Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 2014. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *Network and Distributed System Security Symposium*.

Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-aware android malware classification using weighted contextual API dependency graphs. In *21st ACM Conference on Computer and Communications Security*.

Mu Zhang and Heng Yin. 2013. AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. In *Network and Distributed System Security Symposium (NDSS)*.

Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. 2013. Vetting undesirable behaviors in Android apps with permission use analysis. In *Computer & Communications Security*.

Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. 2012. SmartDroid: an automatic system for revealing UI-based trigger conditions in Android applications. *ACM SPSM*.

Min Zheng, Patrick P. C. Lee, and John C. S. Lui. 2013a. ADAM: An automatic and extensible platform to stress test Android anti-virus systems. In *Detection of Intrusions and Malware and Vulnerability (DIMVA)*.

Min Zheng, Mingshen Sun, and John C. S. Lui. 2013b. DroidAnalytics: A signature based analytic system to collect, extract, analyze and associate Android malware. *ACM Computing Research Repository (CoRR)*.

Wu Zhou, Zhi Wang, Yajin Zhou, and Xuxian Jiang. 2014. DIVILAR: Diversifying intermediate language for anti-repackaging on Android platform. In *ACM Data and Application Security and Privacy (CODASPY)*.

Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. 2013. Fast, scalable detection of "piggybacked" mobile applications. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*.

Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. Detecting repackaged smartphone applications in third-party Android marketplaces. In *ACM Conference on Data & Application Security & Privacy (CODASPY)*.

Yajin Zhou and Xuxian Jiang. 2012a. Android malware genome project. Retrieved from http://www.malgenomeproject.org/.

Yajin Zhou and Xuxian Jiang. 2012b. Dissecting Android malware: Characterization and evolution. *IEEE S&P*.

Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Network and Distributed System Security Symposium (NDSS)*.

Received May 2015; revised September 2016; accepted November 2016