Toward Engineering a Secure Android Ecosystem: A Survey of Existing Techniques

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, Georgia Institute of Technology

The openness and extensibility of Android have made it a popular platform for mobile devices and a strong candidate to drive the Internet-of-Things. Unfortunately, these properties also leave Android vulnerable, attracting attacks for profit or fun. To mitigate these threats, numerous issue-specific solutions have been proposed. With the increasing number and complexity of security problems and solutions, we believe this is the right moment to step back and systematically re-evaluate the Android security architecture and security practices in the ecosystem. We organize the most recent security research on the Android platform into two categories: the software stack and the ecosystem. For each category, we provide a comprehensive narrative of the problem space, highlight the limitations of the proposed solutions, and identify open problems for future research. Based on our collection of knowledge, we envision a blueprint for engineering a secure, next-generation Android ecosystem.

CCS Concepts: • **Security and privacy → Mobile platform security**; *Malware and its mitigation*; Social aspects of security and privacy

Additional Key Words and Phrases: Android, mobile malware, survey, ecosystem

ACM Reference Format:
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, Article 38 (August 2016), 47 pages.
DOI: http://dx.doi.org/10.1145/2963145

1. INTRODUCTION

Android security has been in the spotlight ever since the first Android-powered phone debuted in October 2008. As Android grows into the most popular mobile operating system by global market share, Android-targeted attacks continue to rise in both number and complexity [Svajcer 2014; Zhou and Jiang 2012].

At the same time, the demand for quality Android device security is increasing. Security sensitive applications (apps) such as online shopping, mobile banking, and

This work is supported by the National Science Foundation (grant DGE-1500084), Office of Naval Research (grant N000141512162), Defense Advanced Research Projects Agency (contract DARPA-15-15-TC-FP-006), and Electronics and Telecommunications Research Institute (contract MSIP/IITP[B0101-15-0644]).

Authors' addresses: M. Xu, C. Song, Y. Ji, M. Shih, K. Lu, C. Zheng, R. Duan, Y. Jang, B. Lee, C. Qian, S. Lee, and T. Kim, School of Computer Science, Georgia Institute of Technology, Klaus Advanced Computing Building, 266 Ferst Dr NW, Atlanta GA, United States; emails: {meng.xu, csong84, yang.ji, mingwei.shih, kjlu, cong, ruian, yeongjin.jang, blee, chenxiong, sangho, taesoo}@gatech.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2016 ACM 0360-0300/2016/08-ART38 $15.00

DOI: http://dx.doi.org/10.1145/2963145

personal healthcare are gaining ever more popularity. Meanwhile, thanks to its openness and extensibility, Android is reaching further than smartphones, appearing in smart TVs, car navigation systems, and home automation systems. As a result, it is also considered one of the most promising platforms for the growing Internet-of-Things ecosystem. Based on these facts, one can easily predict the future security landscape of the Android arena: more valuable and more numerous targets for attackers, spawning more powerful and sophisticated malware.

Motivated by the urgent need to prepare a secure Android platform, we believe that now is the right moment to step back and systematically re-evaluate the Android security architecture and the security practices in the ecosystem. In the past few years, many issues were identified and a multitude of defensive techniques were proposed to solve them. However, due to the scale and complexity of the Android ecosystem, each research work generally focuses on only one particular problem. Lacking a holistic blueprint to guide refinement of the overall ecosystem, we are motivated to analyze, categorize, and evaluate proposed solutions and to shed light on a way to envision the next-generation Android ecosystem.

In this article, we survey the Android related research and development efforts presented in top conferences and journals.[1] Without a loss of generality, we themed the survey with a focus on Android malware attacks and defenses, defining malware as any hostile or intrusive instrument attackers might leverage to achieve their goals. Note that malware can take practically any form, such as rootkit exploiting kernel vulnerabilities, malicious web domain abusing improper uses of Secure Sockets layer (SSL), or simply repackaging of a popular Android app. Malware can achieve multiple goals, including but not limited to intrusive advertising (adware) or privacy compromising (spyware). Therefore, discussion around malware provides broad coverage on a variety of Android security topics. Based on this insight, we organize this article based on two key areas where the focus of offensive and defensive techniques lies on:

(1) **Android software stack:** where malware tries to exploit system weaknesses or design errors to penetrate and execute intended actions. In this area, once malware reaches a device, it could exploit vulnerabilities in Android OS to acquire root privilege, or exploit flaws in the permission model to fool the system. It might also abuse features such as dynamic code loading to mount the attack or use side channels and covert channels.
(2) **Android ecosystem:** where malware tries to evade app review/detection, attract downloads, or find alternative distribution channels to reach end users. In this area, it is common to see an attacker imitate the appearance of a popular app or even repackage it in order to trick naive users into installing his/her malicious app. The malware might also use obfuscation techniques to hide its exploitive intent and evade malware detection practices.

The rest of the article is organized as follows: Section 2 contains background knowledge on Android platform security architecture and security practices in the current ecosystem: the necessary pieces in understanding the rest of this article. Sections 3–7 describe the offensive and defensive techniques on the Android software stack. Sections 8–11 describe the offensive and defensive techniques on the Android ecosystem. In Section 12, we present our views on issues that the Android platform may face in the near future. We also discuss how to prepare Android for the Internet-of-Things (IoT) trend and demands for increased privacy. Based on these survey results, we explain our envisioned landscape for the next-generation Android ecosystem in Section 13, where antimalware techniques are deployed at every core participant's side.

[1] The complete list of sources of surveyed papers are presented in Section A.

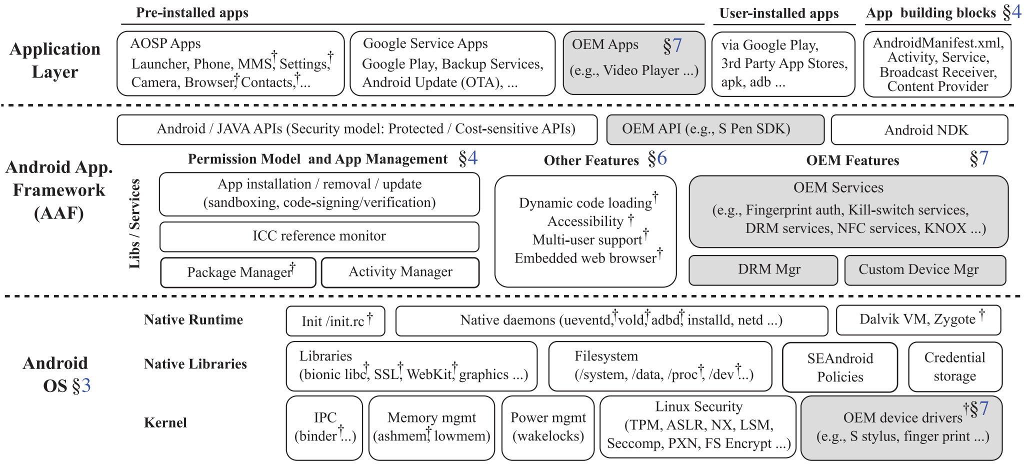


Fig. 1. Overview of Android software stack in terms of security. Components introduced by OEMs are shaded, and components for which researchers have previously identified vulnerabilities are marked with †. Section 2.1 describes the details of each layer and component with pointers to the corresponding sections.

2. UNDERSTANDING ANDROID SECURITY

This section explains our views on current Android security architecture. We first provide a comprehensive overview of its componentwise layered design, with a focus on security-related components; then we discuss current security practices in the Android ecosystem. This section also serves as an entry point to find corresponding sections of interest.

2.1. Android Platform Security Architecture

In the layered architecture of Android platform, security of components at the upper layer are built on those at the lower layers. In contrast to the legacy notion that Android security relies exclusively on the Android Open Source Project (AOSP) [Android Developers 2016b], our proposed architecture in Figure 1 considers the larger ecosystem, including Original Equipment Manufacturers (OEMs), carriers, and Google. It also sheds light on these entities' roles and relations in constructing a secure Android system.

2.1.1. Android Operating System. The Linux kernel is the foundation of the whole software stack. Android implements the application-level sandbox by leveraging Linux's Discretionary Access Control (DAC). By assigning a unique `uid` to each app, Android isolates individual apps within a `uid`-based process boundary. Therefore, an app cannot interact with other apps by default and can only access resources in its own sandbox (e.g., own files). Similarly, each system resource (e.g., network, sound, etc.) is assigned a unique `gid`: to grant an app access to a particular resource, the app's `uid` is added to the resource's `gid` group. Although many Android apps are running in the Dalvik Virtual Machine (VM), the VM does not provide additional sandboxing like the Java VM does, so the only security boundary of an Android app is the DAC-based application sandbox.

Attacks at this layer mainly focus on breaking the DAC sandbox by exploiting particular kernel vulnerabilities, while defensive techniques focus on hardening the kernel to either eliminate the vulnerability or reduce impact when exploits occur. Detailed descriptions of attacks and defenses of this layer are discussed in Section 3.

2.1.2. Android Application Framework (AAF). Abstracted from the Linux DAC model, in order to provide apps fine-grained accesses to resources (such as GPS or contacts), Android implements its permission model at the AAF. To gain such permissions, the developer first declares the resources required for his app; then, the user approves the declared permissions upon app installation. Researchers have explored many security issues with such a permission model and have proposed various new models to solve them as discussed in Section 4.

Besides issues in the permission model, another common attack vector for Android app isolation schemes are the use of side channels and covert channels. These attacks and particularly their mitigations have not been studied thoroughly in Android; thus, such attacks will be emerging threats for new platforms like the IoT. We study their eligibility, potential threats, and possible mitigation in Section 5.

In addition to the permission model, AAF provides a rich set of features such as dynamic code loading and accessibility. Unfortunately, retrofitting new features into the existing Android security model often introduces vulnerabilities. We approach them systematically in Section 6.

2.1.3. Application Layer. Android apps, whether pre-installed or user-installed, generally adhere to a modularized design to facilitate component reuse. An app must be divided into `Activities`, `Services`, `BroadcastReceivers`, and `ContentProviders`, which are held together by an `AndroidManifest.xml` file. The majority of app components communicate through `Intent`, the default Intercomponent Communication (ICC) channel, and share their data with other apps using Content Provider, which can be queried in an SQL-like way. App developers also have the freedom to build apps in native code and enjoy the rich services provided by the Android framework, such as cryptography and SD Card access. If, however, developers fail to utilize these services or adhere to the aforementioned principles either by mistake or carelessness, they leave their apps and users open to attack. A handful of studies have been done regarding this issue, as we describe in Section 4 and Section 6.

2.1.4. Device Fragmentation. In Figure 1, components in gray boxes reflect the impact of the fragmentation of Android devices. These components are sometimes highly customized by device distributors like OEMs and carriers. However, such customization frequently introduces new security issues, as explained in Section 7.

2.2. Security Practices Across the Ecosystem

Compared to traditional desktop systems, mobile platforms like Android generally have more diversified participants, which in turn brings new features into the ecosystem. For example, one feature common in all the major mobile ecosystems is the app store model, which has only recently started to emerge in desktop systems (e.g., Windows 10 and Mac OS). Such features bring both complications and opportunities to Android platform security.

2.2.1. Core Participants. Out of the hundreds of stakeholders involved in the Android ecosystem, we focus on four entities: users, developers, app stores, and the Open Handset Alliance (Google, OEMs, and carriers), since they are the core participants of the Android ecosystem. The interactions between these participants are essentially flows of software/apps and revenue. We visualize these interactions in Figure 2.

When an attacker (particularly, a malware writer) is considered in the ecosystem (also shown in Figure 2), the interaction graph suggests two goals the ecosystem should achieve: (1) preventing malware from reaching the end user and (2) preventing profit from flowing to the attacker.

2.2.2. Malware Defense Practices. To achieve the goals in taming malware attacks, we have identified the following practices that can be employed by different entities in order to collectively make the ecosystem more secure.

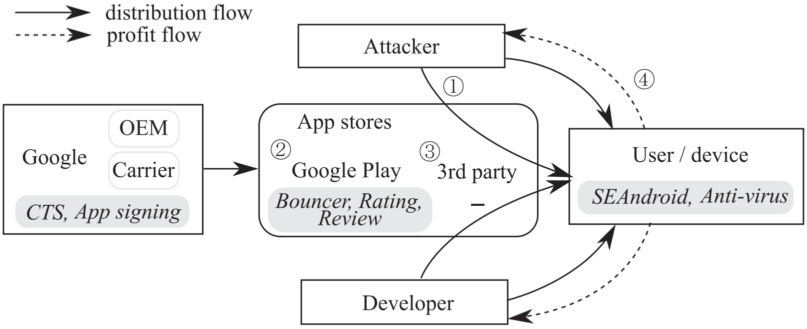


Fig. 2. Current security practices (shaded) in Android ecosystem, and interactions among core participating entities.

Table I. Security Practices in the Current Android Ecosystem. We Categorize Our Approaches in Terms of the Incentives Relationship in Figure 2

Description	Category	Current Practices
① Disincentivize attackers to publish malware to trusted app stores	Incentive elimination (Section 11)	Developer rewarding and profit-sharing schemes
② Prevent malware from being distributed from trusted app stores —Active malware detection —Passive malware detection	Malicious behavior detection (Section 8) Repackaging detection/prevention (Section 9) Sociological methods (Section 10)	Bouncer for Google Play App rating and user comments
③ Prevent malware from being distributed from other sources	Infection channel cut-off (Section 10)	N.A.
④ Prevent intended benefits flow to attacker —The system is hardened to mitigate exploitation of unknown vulnerabilities	System hardening (Sections 3–7)	SEAndroid, capability system Antimalware apps

—**Malicious behavior detection:** To detect the malicious behaviors of apps, Google Play introduced *Bouncer* [Poeplau et al. 2014], a malware scanning service to detect malicious in-store and prestore apps. Researchers have also explored a variety of new techniques (Section 8) to this end.

—**Repackaging detection and prevention:** Taming app repackaging is crucial to a successful and secure ecosystem, which has attracted much research in this area (Section 9).

—**Infection channel cut-off:** Unlike the current open app distribution model that allows app downloading from alternative stores, researchers have proposed to cut off malware distribution from app stores and untrusted sources (Section 10).

—**Incentive elimination:** Android's profit model involves sharing revenue between developers and the app store (e.g., Google Play and Amazon App Store). Such a model indirectly undermines illegal malware by suppressing malware writers' incentives (Section 11).

Table I shows how each practice might be adopted by the core participants described in Figure 2.

Table II. Past Vulnerabilities in the Android Platform

				Vulnerable Component		
Nickname	**CVE or ID**	**Release (platform)**	**Cause of Vulnerability**	**Linux**	**Driver**	**Daemon**
asroot	2009-2692	08/2009 (≤2.2)	Null pointer dereference	socket	-	-
exploid	2009-1185	07/2010 (≤2.1)	Incorrect input validation	-	-	udev
RAtC	2010-EASY	10/2010 (≤2.2)	Incorrect error handling	-	-	adbd
Zimperlich	2010-EASY	12/2010 (≤2.2)	Incorrect error handling	-	-	zygote
KITNO	2011-1149	01/2011 (≤2.2)	Incorrect sharing of resources	-	-	init
psneuter	2011-1149	01/2011 (≤2.2)	Incorrect sharing of resources	-	-	init
GingerBreak	2011-1823	04/2011 (2.1-2.3.3)	Incorrect input validation	-	-	vold
Zergrush	2011-3874	10/2011 (2.2-2.3.6)	Buffer overflow	-	-	vold
levitator	2011-1350,1352	11/2011 (2.3-2.3.5)	Improper bound check	-	PowerVR	-
mempodroid	2012-0056	01/2012 (4.0-4.0.4)	Improper permission check	mem_write	-	-
bin4ry	OSVDB 94059	09/2012 (4.0-4.0.4)	Symlink attack	-	-	adbd
diaggetroot	2012-4220,4221	11/2012 (2.3-4.2)	Integer overflow	-	diagchar	-
-	2013-2094	06/2013 (2.2-4.3)	Integer overflow	perf	-	-
FramaRoot	2013-6282	04/2014 (2.x-4.x)	Missing checks	get/put_user	-	-
TowelRoot	2014-3153	06/2014 (4.0-4.4)	Use-after-free	futex	-	-
GiefRoot	2014-4321,4322	12/2014 (4.0-4.4)	Missing checks	-	camera	-
PingPongRoot	2015-3636	08/2015 (≥4.3)	Use-after-free	net	-	-

3. SYSTEM PRIVILEGE ESCALATION

Preventing users or high-level apps from acquiring system privilege (i.e., `root` privilege) is a primary security assumption on the Android platform. If this assumption is broken, many fundamental protection mechanisms (e.g., the Android permission model) cannot guarantee the security they intend, since these mechanisms are built upon this assumption.

In this section, we describe attacks and defenses related to system privilege escalation on Android. Although this topic is well studied in traditional Linux-based systems, these studies are not directly transferable to the Android domain due to the amount of customization Android introduced to overcome various hardware restrictions in the mobile environment, for example, limitations in memory, battery, and computing power, as well as heterogeneity in hardware drivers. Therefore, unlike traditional Linux-based systems, Android is unique in that high-level apps are running on top of not only Linux Kernel but also Android customization, as illustrated in Figure 1. As such, this architectural design exposes wider attack surfaces than traditional Linux-based systems.

To clearly show where the current Android security stands against system privilege escalation attacks, Table II illustrates various attacks against the Linux kernel, customized drivers, or system daemon programs (Section 3.1), and further indicates whether each attack can be stopped using state-of-the-art mitigation techniques (Section 3.2). We also want to highlight one major behavioral difference between Linux users and Android users: the latter, even inexperienced users, may intentionally root their devices, making privilege escalation prevention even more complicated (Section 3.3).

3.1. Vulnerabilities and Attack Surfaces

In general, all components of the Android OS layer can be targeted by system privilege escalation attacks because they are running with system privileges and attackers can gain desired system privilege by exploiting them. Specifically, the attack surfaces include Android Linux Kernel and Android system components and can further be classified, based on the source of vulnerabilities, as shown in the following.

3.1.1. Mainline Linux Kernel. Core operating system services of Android (e.g., scheduling, `mem_write`, `socket`) are supported by the mainline Linux kernel. Thus, Android can suffer from the vulnerabilities found in the mainline Linux Kernel related to these services. For example, Mempodroid [Freeman 2012], inspired by its Linux counterpart Mempodipper [Donenfeld 2012], exploits the same `mem_write` vulnerability to gain root access in the Android kernel. TowerRoot [geohot 2014] abuses a recently found vulnerability in the `futex` system call to root the various Android devices.

3.1.2. Native Daemons. Native daemons have always been an attractive target for subverting the Android system, largely because most of them are run with system privilege, and tend to contain legacy bugs from the low-level programming language they are using. Many severe attacks against the native runtime have been reported: RAtC [thesnkchrmr 2011] and Zimperlich [Sebastian 2011] acquire root privilege with fork bombs to prevent `adbd` and `zygote`, respectively, from dropping root privilege; psneuter [Choong 2012] and KITNO [AVO 2011] abuse an implementation bug in `init` to change the system settings to prevent `adbd` from dropping its root privilege; Zergrush [F-Secure 2011b] and GingerBreak [F-Secure 2011a] exploit `vold`; and bin4ry [vuldb.com 2013] exploits the `adbd` daemon.

3.1.3. Third-Party Drivers. Android relies on hardware manufacturers to provide custom drivers, many of which are close-sourced and are implemented with little concern for security. For example, `levitator` exploits bugs in the PowerVR SGX driver and `diagroot` exploits bugs in the Qualcomm diagnostic driver to mount a privilege escalation attack.

3.2. Mitigation Techniques

Current mitigation techniques adopted by Android fall into two categories: (1) kernel and native code hardening techniques that make it more difficult to compromise the kernel, native libraries, and native runtime; and (2) SEAndroid, which confines the capabilities of native daemons.

3.2.1. Hardening Kernel and Native Code. Kernel hardening techniques are designed to make kernel vulnerabilities more difficult to exploit. Unfortunately, despite a large amount of existing work, most kernel hardening techniques have not been adopted by the Android kernel due to their ineffectiveness or performance overhead. The only documented adoptions to the Android kernel are `dmesg_restrict` and `kptr_restrict` that prevent leaking kernel addresses. Samsung Knox provides *Real-time Kernel Protection* to prevent malicious modification or injection to the kernel code [Azab et al. 2014], but it is not available in the mainline Android kernel. Since native code (e.g., native libraries and native runtime of Android), is prone to memory corruption vulnerabilities, a range of protection techniques have been employed to improve the system's security: for example, eliminating vulnerabilities (`safe_iop`, `format-security`), preventing control-flow hijacking (stack cookies, NX, ASLR, etc.), and restricting system's policies (e.g., restricting `READ_LOGS` access and defaulting `umask` to `Ø77`).

3.2.2. SEAndroid. Because the Android application sandbox is built upon Linux's DAC, exploiting any daemon with root privilege may compromise the security of the entire system. To mitigate such threats, SEAndroid [Smalley and Craig 2013] (enabled in the *enforce mode* from Android 5.0) was introduced to provide Mandatory Access Control (MAC). By enforcing MAC, SEAndroid is able to (1) stop critical steps of exploits, such as disallowing the creation of the `NETLINK` socket by user shells or apps; and (2) prevent abuse of root privileges (e.g., `setuid`) even if the daemon is compromised, that is, minimizing the damages caused by granting only minimum privileges needed by those daemons. FlaskDroid [Bugiel et al. 2013] further extends SEAndroid by supplying it with an efficient and flexible policy language that is tailored to the specifics of Android middleware semantics. In addition, EASEAndroid [Wang et al. 2014a] introduces an SEAndroid analytic platform to automatically analyze and refine SEAndroid policy.

When deploying mitigations for mobile devices, the trade-off between mitigation effectiveness and performance overhead is of particular concern. We observed that, likely due to the unique resource constraints (e.g., battery and RAM) of the mobile environment, adoption of proposed mitigation techniques in the Android OS depends more on the performance overhead incurred rather than soundness of the proposed techniques.

3.3. User Voluntary Rooting

Besides unintended attacks, rooting a device can also be a voluntary behavior from users with various motivations, for example, removing OEM pre-installed apps, enabling tethering, or just for fun. To facilitate this demand, a variety of root providers begin to offer root as a service (e.g., Root Genius [Team 2016]) and many convenient one-click root methods operate by exploiting one or more vulnerabilities described in Table II or even zero-day vulnerabilities. Such behaviors create more complications for the Android ecosystem, such as (1) how to ensure the zero-day vulnerabilities are not abused by malware writers [Zhang et al. 2015], (2) how to safely unroot the device without leaving security loopholes, and (3) how to protect rooted devices, as rooting breaks the Trust Computing Base (TCB) of many proposed solutions for addressing higher level security issues in Section 4, as shown in Zhang et al. [2014b].

These questions are largely unaddressed, as rootkit users are assumed to take full responsibility of the consequences. However, this assumption may not hold. According to Ludwig [2013], 494 nonmalicious rootkits are installed per million installs from Google Play, whose users are unlikely to have sufficient security awareness to mitigate potential security threats introduced by rooting. Moreover, certain OEMs provide users with customizable bootloaders to ease the burden of loading customized kernels [HTC Corporation 2016], allowing more rooted Android devices. Both of them indicate that protecting rooted devices is a pressing research problem. Shao et al. [2014a] proposes a kernel hooking approach to mediate the `su` requests. Whenever an app issues a privileged request, that is, a system call, it is first captured by `RootGuard`, which then checks against its policy database to determine whether the request should be allowed or denied. To the best of our knowledge, at the time of writing, this is the only work that targets protecting rooted Android phones and we believe more research should follow.

3.4. Open Problems

We have observed the following open issues and emerging trends over these years in Android system privilege escalation prevention.

3.4.1. Performance Optimization for Hardening Techniques. Due to the long cycle of security updates in the Android ecosystem, exploit mitigation techniques are critical to ensure a robust TCB. Several open source projects are available to further harden the Linux kernel, such as grsecurity [Open Source Security, Inc. 2016]; however, porting them to Android may face a major obstacle—performance overhead—which is particularly sensitive for mobile devices due to their limited computation power and battery.

3.4.2. New Hardware Security Features. Besides porting existing hardening techniques to the Android kernel, introducing new hardware features can be an alternative approach. Given the openness of mobile device configuration and OEMs' incentive for differentiation, it is promising to search for new mobile hardware features that could boost the security in Android OS. For example, ARM TrustZone establishes and isolates a secure and nonsecure world with hardware support. Security-critical services can be designed to run only in the secure world, while normal services run in the nonsecure world. As shown in TrustZone-based Real-time Kernel Protection (TZ-RKP) [Azab et al. 2014], the entire (TCB of the Android OS can be protected with this hardware feature.

3.4.3. Control Flow Integrity Guarantee. Control Flow Integrity (CFI) [Abadi et al. 2005; Davi et al. 2012] is a strong technique for preventing the increasingly sophisticated control flow hijacking attacks. In essence, a CFI policy requires that program execution must follow a predetermined Control Flow Graph (CFG), which is generally statically determined by analyzing the program source code or binary files. Unfortunately, a critical issue with CFI is that the statically computed CFG is usually too conservative and allows too many targets. This fundamental issue has been exploited to bypass CFI protections [Davi et al. 2014; Goktas et al. 2014].

A better approach is to protect the integrity of data or pointers that have an impact on the program control flow, for example, code pointers [Kuznetsov et al. 2014]. Another problem indicated in Table II is that many system privilege escalation attacks do not rely on control flow hijacks at all. Instead, they use data-only attacks [Chen et al. 2005]. We expect to see more efforts toward preventing such attacks.

3.4.4. Logic Bugs. As seen in Table II, logic bugs, such as incorrect input validation, symlink attack, and incorrect error handling, are the main vulnerabilities exploited for rooting. However, general detections or defenses against these bugs are still missing. To combat them, more efforts are expected to both abstract these bugs, and propose general protections.

3.4.5. Adoption of Capability System. A capability system can effectively restrict daemons' capabilities. Specifically, it can allow daemon programs to start with requested privileges and later drop privileges when no longer needed. A prominent example is `seccomp-bpf`, a long developed mainline Linux sandboxing facility with high efficiency, which is also widely used in practice. Some Android communities are experimenting on porting `seccomp-bpf` to Android, which can be tracked in Chromium Dev Community [2012]. In many cases, a capability system can be considered as a lightweight framework for SELinux with a focus on practicality and performance. However, SELinux has finer granularity and more mechanisms for monitoring and mediating access controls.

3.4.6. Policy-Agnostic Security Infrastructure. The industry is adopting policy-based approaches to harden Android. For example, people are tuning the SEAndroid policies to provide better protections. In the meantime, the academic community is having second thoughts about policy-based solutions such as SEAndroid [Backes et al. 2014; Heuser et al. 2014]. They argue that hard-wiring a specific security model into Android not only impairs its practicality and maintainability in a fragmented environment, but also precludes many other security extensions. As an alternative, they propose to hook throughout the Android OS and build security APIs on top of the hooks, which can be further leveraged to generate various security extensions discussed in Section 3.2 and Section 4. The performance and practicality of these approaches is yet to be tested in industry.

4. PERMISSION MODEL

On top of the application sandbox is the Android permission model, which is directly exposed to developers and users as a mechanism for mediating apps' accesses to system resources. This section discusses issues in the Android permission model and proposed solutions for mitigating these problems (Table III).

4.1. Issues

4.1.1. Incorrect Assignment. Under the current permission model, developers are responsible for claiming permissions for their apps. Since most developers are security-unaware, they are likely to overclaim permissions in order to ensure a smooth user experience, that is, an app runs well in all situations, and hence, ignoring the *Principle*

Table III. Summary of Previous Work on Improving the Android Permission Model. ①–⑥ Corresponds to the Six Categories of Solutions Discussed in Section 4.2

Type	Solution	Description	Required Modification
①	Stowaway	Extract permission-to-API map through API testing.	-
	PScout	Extract permission-to-API map from Android source code.	-
	WHYPER	Ensure description-to-permission fidelity.	-
	AutoCog	Ensure description-to-permission fidelity.	-
②	Apex	Intercept sensitive API calls and filter against predefined policies.	App Installer, ICC Monitor
	CRePE	Intercept sensitive API calls and filter against predefined policies.	ICC Monitor
	AppFence	Supply sensitive API calls with mock data/reject network transmission.	Dalvik VM, Resourec Manager
③	Constroid	Data-centric access control by policies.	ContentProvier, ICC Monitor
	Dr. Android	Sensitive API drop-in replacement.	App Repackaging
	Aurasium	In-app `libc` interposition.	App Repackaging
④	AdDroid	Run advertisements as system services.	System Service, Libc
	AdSplit	Split ad library and app into separate processes.	App Repackaging
	LayerCake	Provide in-app privilege separation through process.	View Object
	Compac	Provide component-level permission assignment.	ICC Monitor, Kernel
	FlexDroid	Provide in-app privilege separation through call stack tracing	Kernel, Dalvik VM, Libc
⑤	IPC Inspection	Permission check across the whole API access call chain.	ICC Monitor
	SORBET	Permission check across the whole API access call chain.	ICC Monitor, Kernel
	XManDroid	Permission check across the whole API access call chain.	App Installer, ICC Monitor
	Quire	Permission check across the whole API access call chain.	ICC Monitor, Kernel
	Saint	Allow developers to define policies on interface access.	App Installer, ICC Monitor
⑥	Aquifer	Allow developers to attach policies on data shared.	ICC Monitor, Kernel
	Jia et al.	Allow developers to attach policies on data shared.	ICC Monitor, Kernel
	Maxoid	Allow developers to attach policies on data shared.	ICC Monitor, Kernel

of Least Privilege (PoLP) [Felt et al. 2011b]. A previous work [Felt et al. 2011a] shows that 35.8% out of 900 Android apps analyzed are overprivileged due to developers' overclaiming. Multiple problems can arise from this incorrect assignment issue. For example, once the app is exploited, the overclaimed permissions will be available to attackers as well, making complex attacks easier. Overclaiming permissions without corresponding functionalities will also result in higher risk ratings for the app when being scrutinized by automatic detection tools such as Google *Bouncer* [Poeplau et al. 2014], which leads to longer review time or even app rejection.

4.1.2. Usability. Current Android permission model (prior to Android Marshmallow) relies on users to grant permissions to apps but fails to provide enough flexibility to users for addressing their own privacy requirements. Specifically, users have to either grant all the requested permissions to an app or decline to install it at all. This causes three problems: 1) Users have no choice but to grant redundant permissions in order to use an app. For example, even if users think the `INTERNET` permission is not necessary for a calculator app, they must grant this permission in order to install and use it. 2) Once the app is installed, it can abuse permissions without any restriction. For example, an app may send an unlimited number of text messages as long as it is granted the `SEND_SMS` permission. 3) There is no way to enforce runtime context-aware policies. For example, users may want to enforce a policy that disallows all access to personal identifiable data when the device is connected to public WiFi. Starting from Android Marshmallow, and also found in customized versions of Android OS like Cyanogenmod [CyanogenMod Team 2016] and Blackphone [Silent Circle 2016], the system supports runtime permissions toggling, resembling the approach taken by iOS platform and partially solves these problems. Users have the flexibility to grant and revoke dangerous permissions after app installation and app developers can also prompt users for permission at runtime, all handled by the `PackageManager` service. However, for backward compatibility, apps targeting previous platforms still follow the current permission model by default. We anticipate that it will take one or two years before the new permission model is widely adopted by developers, as was seen in the transition from Android 2.X to Android 4.X [OpenSignal Inc. 2015].

4.1.3. Granularity. Researchers have argued that the Android permission model is currently too coarse-grained to ensure that an app behaves exactly the same as expected and agreed by users. For example, the `INTERNET` permission allows an app to access any domain address. This gives malicious apps an opportunity to launch attacks, for example, a malicious app that claims to offer Google Map service may talk to `maps.google.com` in the foreground, and leak location information to an attacker's server in the background.

Additionally, the current permission model grants permissions to all components in an app, despite the fact that they may come from various developers. This allows one component to abuse permissions required by another component [Paul Pearce et al. 2012]. For example, an advertisement (Ad) library may abuse the `VIBRATE` permission that is actually needed by the main functional component. Likewise, the main component can also abuse the `LOCATION` permission required by the Ad library.

4.1.4. Transitivity. The Android component model and the `intent`-based ICC allow apps to expose services (e.g., read location and send SMS) or data (e.g., pictures, recording) to other components or apps through legal interfaces. The design is intended to promote modularization and code re-use in app development; however, it also presents a challenge on both developers and users.

Developers should write robust and secure interfaces that only accept intents from apps with required permissions. Failure to do so results in *confused deputy attack* [Felt et al. 2011c] where the deputy app fails to check whether the calling app has the credentials to use their permission-protected interfaces. For users, permission transitivity makes it harder to foresee how an app might use permissions from other apps, which enables *collusion attack* [Marforio et al. 2011] where two or more malicious apps with distinct but limited permissions collaborate to effectively generate a joint set of permissions.

4.1.5. Authority. The implied assumption in the current Android permission model is that the end user is the final authority for making permission granting decisions. However, researchers have began to challenge this assumption, that is, whether normal users are capable of or care enough to determine which permissions are appropriate for an app even if they are given sufficient flexibility in permission granting period. A previous survey [Felt et al. 2012b] shows that only 3% of Internet survey and 24% of laboratory study participants can successfully understand and grant permissions, while as many as 42% of users are unaware of permissions at all.

4.2. Solutions

4.2.1. Permission Claim Check. To address the incorrect assignment problem, as Google fails to provide clear mappings between permissions claiming and how claimed permissions are actually being used, several tools are designed to fill the gap. Stowaway [Felt et al. 2011a] utilizes the static analysis to determine the set of API calls an app uses and then maps API calls to permissions. Permission over-claim can be determined by comparing the permissions used with the permissions claimed in the manifest file. Similarly, PScout [Au et al. 2012] applies static analysis to Android source code to construct the API call to permission mappings. The mappings can provide more complete information than the existing Android's permission system documentation. In addition to finding mappings between API calls and permissions, WHYPER [Pandita et al. 2013] and AutoCog [Qu et al. 2014], analyze the app description using natual language processing (NLP) techniques to automatically assess description-to-permission fidelity. These two works intend to help both users and developers better understand why an app needs specific permissions and check for permission over-claim.

4.2.2. Privacy/Context Awareness. Privacy/context-aware permission management enhances usability by ensuring a smooth user experience. Such mechanisms are usually realized in the form of policy enforcement. By hooking into the ICC monitor, it is possible to intercept sensitive API calls and make decisions based on pre-defined policies. Apex [Nauman et al. 2010] provides a modified app installer that allows a user to selectively grant permissions to an app as well as limit the resource usage at the install-time. CRePE [Conti et al. 2010] introduces a context-related policy enforcement system, which allows a user to define context-related policies at both install-time and runtime. AppFence [Hornyack et al. 2011] achieves privacy control by allowing a user to either replace private data with shadow data to prevent privacy misuse, or reject transmission of on-device only data over the network. As the permission granting mechanism gets more complex, Felt et al. [2012a] propose a set of guidelines for platform designers to decide the most appropriate permission-granting mechanism for a sensitive API call.

4.2.3. Permission Decomposition. Coarse-grained permissions can be decomposed into multiple fine-grained permissions to help enforce the PoLP. For example, INTERNET permission can be deconstructed into domain-based permission like INTERNET(google.com), which is already the practice of Chrome Apps and Extensions [Google Inc. 2016b]. Constroid [Schreckling et al. 2012] modifies Android middleware to provide finer-grained data-centric access control policies on top of the Android permission framework. Unlike implementations that require framework or system modification, Aurasium [Xu et al. 2012] and Dr. Android & Mr. Hide [Jeon et al. 2012] perform permission decomposition using app repackaging. Aurasium attaches user-level sandboxing and monitoring code to the repackaged app so that the finer-grained policy can be enforced. For example, a user can define a network policy to allow or disallow the connection to a specific IP address. Similarly, Dr. Android & Mr. Hide replaces the original permission system with Mr. Hide, an Android service that provides finer-grained permissions, by app repackaging. In case if an application owns the INTERNET permission, Mr. Hide introduces a new permission InternetURL(d), which only grants access to the domain d.

4.2.4. Assignment Decomposition. Recall that in the current permission model, once an app has a permission, all of its components and libraries share the same privilege. This problem is particularly obvious in the case of using advertisement libraries. Since advertisement libraries are usually developed by third parties, it may be risky to allow a third-party library to share the same set of permissions with the app. Tools like Brahmastra [Bhoraskar et al. 2014] have found various cases of privacy violations in third-party libraries.

To address this problem, several works [Paul Pearce et al. 2012; Roesner and Kohno 2013; Shekhar et al. 2012] propose assigning separate permissions to the in-app advertisement component. Compac [Wang et al. 2014b] further generalizes the idea by proposing a component-level permission assignment approach, such that each in-app component only gets the minimum set of permissions needed for preserving the app's functionalities. FlexDroid [Seo et al. 2016] achieves in-app privilege separation and empowers app developers with fine-grained access control for third-party libraries. It relies on call stack traces collected at both Dalvik VM and native levels and is suitable for sophisticated libraries.

4.2.5. ICC Tracing. ICC tracing aims to solve the transitivity issue. Solutions [Bugiel et al. 2011, 2012; Dietz et al. 2012; Felt et al. 2011c] in this direction usually monitor both sensitive API calls and ICC. By keeping track of the communication channels between apps, an API call is blocked if any app in the call chain lacks the required permission to access it. This is used to effectively tamper permission escalation attacks including the confused deputy attack and the collusion attack.

Moreover, Saint [Ongtang et al. 2009] demonstrates that it is also possible to reinforce the security check by allowing developers to define policies to protect exposed app interfaces, for example, all conditions in the policy must be satisfied before another app can access the interface.

4.2.6. Decentralized Information Flow Control. While ICC tracing only checks the permission before sensitive APIs are called, Aquifer [Nadkarni 2012] and Jia et al. [2013] aim to enforce information flow integrity even after data leaves the app. Both works require developers to tag policies on the data that flows into another app, and instrument the system to enforce the policies. Similarly, Maxoid [Xu and Witchel 2015] modifies the Android system to enforce information flow policies between apps, but it focuses on secrecy and integrity for both the invoking app and the invoked app.

4.3. Open Problems

Despite the extensive research on improving the existing permission model, few solutions have been adopted in practice. We believe the following open issues contribute to the low adoption rate:

4.3.1. Granularity v.s. Intuitiveness. Although finer granularity helps to enforce the PoLP, the permission requester and granter might be over-burdened by the higher complexity. As a result, the more types of permissions are introduced, the less intuitive the permission model becomes. User-driven access control [Roesner et al. 2012] provides a new direction for permission model design which might fundamentally remove this trade-off.

4.3.2. Permission Granting Authority. Another unaddressed problem in the permission granting mechanism is whether users have the capability to make the right decisions. To deal with this problem, one possible solution is to delegate permission granting to the system or third-parties with expertise; while at the same time, the flexibility can be kept for expert users to adjust permissions, much like the proposed work for a desktop environment [Kushman and Katabi 2010].

5. SIDE CHANNELS AND COVERT CHANNELS

All operating systems are subject to side-channel attacks as well as information leakage via covert channels, and Android is no exception. By legitimately observing the behavior patterns of shared resources, an attacker is able to infer sensitive information (side-channel) or use these patterns as a communication channel with another app (covert channel), allowing him/her to bypass both the low-level application sandbox and high-level permission model. Table IV summarizes the existing side-channel attacks and mitigations along with their effectiveness and adoptability for Android.

5.1. Shared Hardware Resources

While sensors in smartphones enable valuable interactions, they can also be abused to infer inputs such as passwords [Al-Haiqi et al. 2013;Aviv et al. 2013; Deshotels 2014;

Table IV. Summary of Side-Channel Attacks. Hardware based Side-Channel Includes 2 Channels: Accelerometer and The Emerging CPU Cache Targeting the Sensitive Inputs, Crypto Key and Kernel Addresses; Most Mitigation Methods are Effective but Barely Adoptable in Android. The Exploit on `/proc` File System Represents the Software based Side-Channel

Type	Channel	Target	Attacks	Mitigation
HW	Accelerometer	Screen taps, PIN	Sensor based	Finer-grained permission Reducing sampling frequency
HW	CPU Cache	Crypto key	Crypto key recovery	Side-channel resistant crypto algorithm Crypto co-processor w/ AES instruction
HW	CPU Cache	Kernel address map	Breaking kernel ASLR	Normalizing page fault handling time Isolation of user and kernel cache use Disabling high-precision timer (e.g., `rdtsc`)
SW	`/proc` file system	Illegal user data retrieval	UI State inference	Access restriction to `/proc` file system Adding indicator on sensitive screen page

Miluzzo et al. 2012; Schlegel et al. 2011; Xu et al. 2009]. Notably, the motion sensor that is mounted beneath the screen, is capable of sniffing users' interactions with the device by analyzing micro changes in device motion and positioning. For example, TapPrints [Miluzzo et al. 2012] can infer the location of taps on touch screens and subsequently recover English characters. An attacker can use the accelerometer to track the tilt of a mobile device, then he/she can recover PIN code of the lockscreen [Aviv et al. 2013]. Soundcomber [Schlegel et al. 2011] selectively records conversations that carry sensitive information including credit card numbers, end then xtracts and transmits the information to another application via either an overt or covert channel. A general problem in the reseach of hardware-based side channels is the lack of end-to-end practical attack cases. For example, gaining knowledge about the victims' PIN code or lockscreen patterns are not likely to benefit attackers if they lack physical access to victims' devices. An even harder problem is how to infiltrate the phone first to mount the appropriate sniffing malware and send out the recorded information. Corresponding mitigations are mainly to limit the sensor usages. For example, mitigations in Aviv et al. [2013], Miluzzo et al. [2012], and Schlegel et al. [2011] rely on the Android's permission control scheme to limit the use of motion sensors and the microphone.

5.2. Shared Software Resources

Because certain software resources are shared across processes, activity signatures such as CPU, memory, or network usage patterns can be retrieved from the accessible `/proc` file system, which can be further analyzed to infer ongoing activities in other processes. For example, Zhou et al. [2013] is able to infer the identity, health information, and location of an Android user by monitoring network usage via `/proc`. Also, Chen et al. [2014b] is able to infer changes of UI state in another app based on the variations of shared memory size in `/proc`, and inject a fake login page seamlessly to steal login credentials. Applying access control on `/proc` filesystem could effectively mitigate this side-channel. However, given that Android is an open mobile system designed to facilitate resource sharing, more complete side-channel mitigations are pressing.

5.3. Covert Channel

As shown in Gasior and Yang [2012], seemingly normal operational variances can be abused to stealthily transmit data between apps, which effectively bypasses the permission model and kernel sandbox. Known covert channels take advantage of vibration, screen brightness, volume settings and file lock statuses to transmit bits to a listening application. Also, Marforio et al. [2012] shows a number of other cover channels in Android, such as intent type, UNIX socket discovery, threads enumeration, automatic intents, and free disk space. The method in Deshotels [2014] even leverages the audio channel to transmit ultrasonic sound which can hardly be noticed by the human ear.

Blocking covert channels in a highly compacted mobile device is challenging due to the large amount of shared resources and numerous ways to encode data. Generally, mitigation requires multiple levels of isolation of hardware and software resources and a well-controlled sensor usage policy. Unfortunately, such methods are impractical in mobile device designs that pursues efficiency over security.

5.4. Open Problems

5.4.1. Emerging Cache Peeping Threat. Many cache-based timing attacks originally found in the desktop or server environment [Bangerter et al. 2011; Hund et al. 2013; Zhang et al. 2012] have become feasible in Android as a result of the wide adoption of the Linux scheduler and multi-core CPU in smartphones. Cache Games [Bangerter et al. 2011] observes the cache access of CPU to infer cryptographic keys during encryptions and decryptions. Timing side-channel attack [Hund et al. 2013] inferred the privileged kernel address by probing the CPU cache usage shared by kernel and user code. Cache partitioning [Page 2005] and side-channel resistant cryptographic algorithms (e.g., Tromer et al. [2010]) flatten the cache usage variance thus blocking these eavesdropping. However, due to the system performance overhead, these methods have not been not widely adopted in desktop Linux, let alone Android.

5.4.2. Blocking Side-Channels. Most side-channels could be avoided if shared resources were no longer used or at least used exclusively when needed. For software shared resources, interleaving the accesses to software stack by different apps can effectively block the side-channel. However, for hardware shared resources, such exclusion is more expensive in mobile OS at the stake of critical downgrade of performance and power efficiency.

5.4.3. Protecting Sensitive Data/Operation. Future workarounds may consider additional protection methods for sensitive inputs if the side-channel cannot be blocked. For example, Intel proposed a two-factor authentication solution that requires a transient One-Time-Password generated from its IPT (Identity Protection Technology) [Intel Corporation 2016] to be attached along with the user password at every login. This scheme may prevent an attacker from completing the login even if the victim's password is inferred. Alternatively, a context-aware noise (e.g., a deliberate vibrating when user taps PIN) may disturb the performance of sensors during a user's interaction with sensitive information.

6. FEATURE ABUSES

This section describes several Android features that have been (or could be) abused to carry out attacks. They include 1) dynamic code generation and loading, 2) Java-Native interface, 3) assistive technologies, 4) multi-user support, 5) embedded web browser and 6) the new ART runtime. These features can be used as attack vectors mostly because they were introduced after the security model was designed. They often require workarounds of existing security models (e.g., for the sake of usability), or intentionally compromise security assumptions to achieve their design goals.

6.1. Dynamic Code

In Android, developers are allowed to load the code for an app dynamically. Developers can load `JAR` files or shared libraries (i.e., `.so` files) from remote sources at runtime by using `DexClassLoader` and `System.loadLibrary()`. This feature gives developers great flexibility in maintaining their apps. For example, apps can self-update by downloading a new JAR file without going through the official channel: Google Play Store. A similar problem is dynamic code generation such as the just-in-time (JIT) compilation adopted by Dalvik VM to improve performance.

However, both features open a hole on the security model of the Android ecosystem: verification of the code signature. Android's default security policy is to only allow the code to run if it is signed by a verified developer. Nonetheless, no signature check is performed on downloaded or generated code; therefore, attackers can abuse both features to inject malicious code into benign apps [Poeplau et al. 2014]. Abusing dynamic code loading can also help bypass Google's app review process [Kim 2015] because dynamically loaded code cannot be correctly tested at the review time (it can be changed later), or attackers could hide their malicious code at review time and make the logic available when the end-user actually downloads and runs the application. One way to protect generated code cache is to ask developers to follow the best practices published in official document that help ensure the integrity of the generated code cache. However, relying on developers' discretion might not be enough, and a more sound approach should be studied. Luckily, such a problem has been well studied in web browsers and potential solutions such as Homescu et al. [2012] can be ported.

To mitigate the threat of dynamic code loading, it is possible to enforce that the signature of code being loaded must match the original app's. However, this approach will preclude the convenience of loading code developed by others. A workaround is to use a system-wide whitelist for acceptable dynamic libraries and to employ multiple independent verification services to provide and update the whitelist [Poeplau et al. 2014].

6.2. Native Code

In addition to Java code that runs on Dalvik VM or Android Runtime (ART), Android supports execution of native code through the Java Native Interface (JNI). Developers can implement their code as JNI for overcoming the limitations of Android Runtime, such as memory cap and performance loss. Code implemented in JNI runs in the application sandbox of Android system. Protections provided by the baseline Linux system layer; such as UID/GID based access control, SELinux, etc., will still work.

However, JNI introduces several security implications into the system. First, unlike Java, a type-safe language, JNI code is prone to vulnerabilities. Traditional attacks on stack/heap buffer-overflow or problems with dangling pointers can be applied to JNI code. Second, despite of plethora of application analysis tools and researches on security mechanisms such as finer-grained access control and dynamic analysis in Android, few support JNI due to the general assumption that JNI is not widely used in Android apps. However, this assumption would not hold in practice; although the portion of code is small in application, use of JNI is prevalent [Afonso et al. 2016]. For this reason, exclusion of JNI code creates holes in applying such mechanisms, that is, proposed security is not guaranteed on the system.

As mitigation attempts, isolating the app-specific native code from the rest of the system is not impossible to implement. A successful example is Native Client (NaCl) used in the Chrome Browser. Robusta has successfully isolated the native code from Java Virtual Machine (JVM) in the traditional OSes. Continues on this direction, NativeGuard [Sun and Tan 2014] achieves third-party native library isolation by running native code in a separate app and [Afonso et al. 2016] further provides a way to automatically set privileges on the isolated native libraries.

6.3. Accessibility

To support easy access of devices, Android implements various accessibility features such as text-to-speech screen reader, voice commander (Google Now), etc. Android also allows third-party apps to access these features through the `AccessibilityService` class [Android Developers 2016a]. However, Jang et al. [2014] discovered that these features are powerful enough to construct an input/output subsystem on the operating systems (including Android) which could be used to completely bypass the permission model (both at middleware and kernel layer) and the application sandbox. By abusing these features, an application could not only intervene in user I/O, but also read app states from UIs and take control of the app by sending inputs that are in direct violation of the isolation.

To protect against attacks that exploit accessibility, A11y [Jang et al. 2014] suggests three layers of protection. First, a finer-grained access control on `AccessibilityService` is required. A simple solution would be to separate the privileges of the a11y library into two sets: one for reading the content of other apps and one for interacting and controlling other apps. For example, a screen reader will only be allowed to read the content of other apps, but not launch other apps. Second, assistive technologies such as Google Now voice commander should verify that the commands come from the authorized user, possibly through voice recognition or other techniques. Finally, apps should be aware of the source of the input (whether from a11y libraries or the real device), and react accordingly for security-sensitive UI contents. However, as this issue is not unique to Android and the suggested mitigations are non-trivial to implement, additional research is required in this area.

6.4. Multi-User Support

Starting from Android 4.2, support for multiple users has been added as a new feature. Unfortunately, researchers have found a significant number of vulnerabilities from their systematic evaluation of multi-user support [Ratazzi et al. 2014]. The root cause of these vulnerabilities is that this new feature is inconsistent with existing protections, especially in accessing shared resources and system-wide configurations.

For instance, all users have full access to WiFi settings and these settings are shared among users. Moreover, the current implementation requires three additional logins to guarantee a complete removal of processes that belong to the current user. Until then, these processes remain running in the background after user switch. This also introduces new privacy and security problems.

6.5. Embedded Web Browser

WebView [Android Developers 2016c] is a feature that allows for an app to create its own browser or to display rich web content. It also enables effortless porting of existing web-based apps to the mobile world. Frameworks such as PhoneGap [Adobe Systems, Inc. 2016], Apache Cordova [The Apache Software Foundation 2016] and Titanium [Appcelerator Inc. 2016] allow developers to write mobile apps just as web apps. In 2013, 8.5% (59,354 among 691,517 apps) of apps were developed using WebView-based framework [Viennot et al. 2014].

However, the convenience of *WebView* is achieved at the cost of compromising the general security principles in web browsers, such as isolation of JavaScript (JS) runtime, same origin policy (SOP), etc. Consequently, numerous security problems have been discovered. WebView provides the `addJavascriptInterface` method as a attempt to isolate Java and JS contexts, allowing JS to access selected Java classes and their public methods of the parent app. However, this feature also creates holes in both the sandbox for Android app and the browser sandbox for webapp, as demonstrated in Luo et al. [2011]. For example, lacking SOP, malicious JS from an untrusted web site can easily abuse this feature to attack the parent app and access sensitive resources. Since Android 4.2, instead of exposing every public method of selected Java classes, only public methods annotated with `JavascriptInterface` can be accessed from JS context. However, this only works if developers voluntarily enforce this feature and manually annotate their source code.

In addition to attacks through the Java-JS interface, researchers have identify different vulnerabilities in Android *WebView* feature. Fahl et al. [2012] details an SSL stripping attack against WebView. Missing visual cue for SSL certificate, Luo et al. [2012] illustrated UI redressing attack. And another attack is developed based on use of OAuth in mobile settings under WebView [Chen et al. 2014a].

Mitigation techniques generally try to bring back the missing security features. Yu and Yamauchi [2013] proposes restricting JS access to sensitive Android APIs. However, this does not address the potential leakage of sensitive data stored in the app. Bifocals [Chin and Wagner 2013] proposes to expose the Java interface only to domains in a whitelist, which is specified by developers and acknowledge by users. Shin et al. [2013] tries to bring back the missing visual security cues. And [Roesner and Kohno 2013] proposes a systematic way to protect embedded user interface, include WebView.

6.6. ART

Starting with Android 5.0, Android replaced Dalvik VM with a new runtime environment called ART [Google Inc. 2016c], which uses ahead-of-time (AOT) compilation to compile Dalvik bytecode to native code during app installation. Therefore, in ART, only part of the app code is executed in JIT mode (e.g., dynamic loading) while the majority runs in native mode. Developing a new compiler to translate Dalvik bytecode to machine code is a bug-prone process; Kyle et al. [2015] and Anestis [Bechtsoudis 2015] were able to fuzz ART to find a number of bugs. Some of the bugs that happen at the compiling phase cause the compiler to crash, while other bugs are triggered at runtime when the generated native code is executed, among which bugs like arbitrary memory read/write, null pointer dereference, etc., are more serious. In addition, ART also exposes a new attack surface by allowing attackers to manipulate the compiled native code. For example, the native code related to boot image provides attackers with a large number of gadgets to perform Return Oriented Programming (ROP) attacks [Corelan Team 2014]. Also, Paul [Sabanal 2015] successfully developed user mode rootkits by taking advantage of ART's mechanisms to replace framework and application code.

7. DEVICE FRAGMENTATION

Device fragmentation is a unique issue in the Android ecosystem. Software-wise, Google has released 147 builds of Android images spanning from v1.6 to v6.0 through February 2016 [Google Inc. 2016d]. Hardware-wise, the number of Android devices released by OEMs, for example, Samsung and HTC, has exceeded 24,093 [OpenSignal Inc. 2015]. Due to the complexity in deploying Android updates to various devices, compared with the stock AOSP image, security guarantees for end-users tend to be less consistent or even weakened.

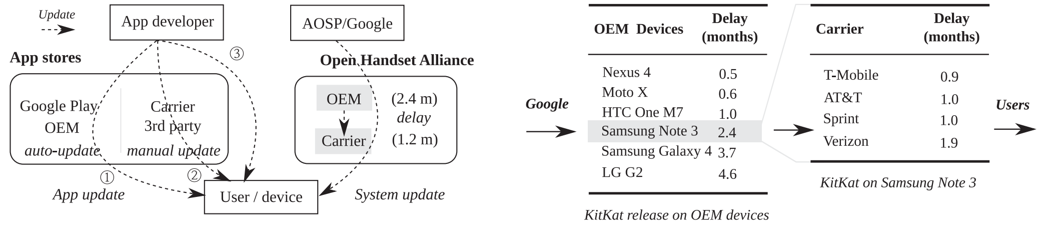
7.1. Security Implications of OEM Customization

To differentiate from other OEM vendors, each OEM customizes its own Android image based on the AOSP. Typical customizations include (1) hardware support (e.g., radio daemon for other modem chips); (2) system UI redesign (e.g., TouchWiz UI from Samsung); and (3) custom apps installation (e.g., S-Health in Samsung Galaxy S4).

Unfortunately, these customizations often weaken existing security mechanisms at both the app layer and the OS layer. At the app layer, OEM-customized or preloaded apps may introduce vulnerabilities that leak users' private information or enable permission escalation attacks [Wu et al. 2013; Grace et al. 2012]. It is also possible that an AOSP app is removed to fit a new device model by the OEM while references to the app still remain inside the OS. Aafer et al. [2015] shows how to exploit this customization to bypass the Android permission model. OEMs might also publish their SDKs to build device/brand-specific apps; for example, Samsung publishes a customized KNOX SDK that opens access to critical APIs such as TrustZone. It is unclear whether such

Table V. Exploits with `symlink` Attack Targeting OEM Devices. For Example, TacoRoot Exploits a World-Writable Recovery Log File `/data/data/recovery/log`. Attacker Can `symlink` it to `/data/local.prop`, Reboot to Recovery-Mode and Write a New Log to Set `ro.kernel.qemu=1`, which Yields the Root Privilege

Manufacturer	Model	Release	Symlink Target	Symlink Destination
HTC	All before release	2011/01	/data/local.prop	/data/data/recovery/log
Motorola	Droid 3	2011/08	/data	/data/local/12m
ASUS	Transformer Prime	2012/01	/data/local.prop	/data/sensors/AMI304_Config.ini
Samsung	Infuse 4G	2012/01	/data	/data/data/.drm/.wmdrm/sample.hds
LG	Spectrum & Intuition	2012/09	/data/local.prop	/data/vpnch/vpnc_starter_lock
Sony	Tablet S	2012/02	/data/{system/packages.list, local.prol}	/log/AndroidRadio.txt



OEM Devices	Delay (months)
Nexus 4	0.5
Moto X	0.6
HTC One M7	1.0
Samsung Note 3	2.4
Samsung Galaxy 4	3.7
LG G2	4.6

Carrier	Delay (months)
T-Mobile	0.9
AT&T	1.0
Sprint	1.0
Verizon	1.9

(a) System and app update procedure

(b) Update delay introduced by various OEM and carriers

Fig. 3. For system update, (1) Google first releases the base update for AOSP; (2) OEM vendors customize Google's update for their devices; (3) with the update from OEM vendors, carriers again test it for all their carrier-specific devices. Cunningham [2014] summarizes system update delay, which is presented in (b). For app update, (1) developers push the update to Google Play and OEM app stores (①), or other app stores (②), and then app stores will distribute the new version to end-users and install it automatically. (2) Alternatively, developers can perform in-app updating, but it requires user's confirmation to download and install updates (③).

customized SDKs have security flaws. At the OS layer, OEM vendors often make mistakes in assigning proper privileges to system or device-related files, allowing attackers to bypass the Android permission model through low-level system calls [Zhou et al. 2014a] or to launch system privilege escalation attacks via crafted symbolic link (`symlink`), as summarized in Table V.

7.2. Security Update

System and app updates play a key role in maintaining the security of Android devices. However, the complicated and lengthy updating process exposes end-users to critical security threats that should have be prevented by prompt update procedures.

7.2.1. System Update. Currently, the process for updating the Android system involves three independent entities—Google, OEMs, and carriers—in delivering new patches to end-users. For this reason, the current update procedure often takes more than three months, as shown in Figure 3. This lengthy procedure makes users' devices vulnerable to known and preventable attacks. Another unexpected Pileup attack presented in Xing et al. [2014] shows a way to exploit the vulnerabilities in the Android system update process: a malicious app can strategically declare a set of privileges and attributes on a low-version OS and wait until it is upgraded to escalate its privileges on the new OS.

7.2.2. App Update. To distribute new app updates, developers either push new versions to the app stores, or build an in-app updating function (see Figure 3). When updating an app, a common security issue for ② and ③ lies in where the new download APK file is stored. If users or developers choose to store it in external storage, such as SD Cards, a malicious app with write permission of SD Cards can easily tamper with the downloaded APK file and inject malicious code into the update [Tao 2014]. Similar to system update, app update may also suffer from delays, particularly when distributed via third party app stores. To reduce the risks in app update, a timely update from Google Play or OEM app stores is recommended and the updated APK file is preferably kept in the internal storage.

7.3. New Model for Security Updates

The update procedure could be redesigned to eliminate the unnecessary attack window due to a lengthy security update. For example, one immediate action to take is to use separate channels to deliver security updates and functional updates (i.e., redesign the update metaprocess). Unlike functional upgrades, security patches are often self-contained and can be pushed directly from Google instead of passing through OEM vendors and carriers. In practice, Google is already separating out core functionalities from AOSP such as WebViews to allow them to be updated without vendor involvement [Toombs 2014]. In terms of update mechanisms, adopting hot-patching and dynamic/live update schemes used in server environments like ksplice [Arnold and Kaashoek 2009] or kGraft [SUSE 2016] will help minimize the latency period of known attacks for end-users. We acknowledge that Google could not possibly manage or be responsible for security updates for all the thousands of device models that exist, hence, the search for a new model that caters to the fragmentation of the Android ecosystem remains an open problem.

8. PRIVACY LEAK AND MALWARE DETECTIONS

Privacy disclosure and malware detections are essential components to enhance security of the Android ecosystem. In this section, we survey recent researches in these areas, analyze their limitations, and identify remaining open problems.

8.1. Detecting Privacy Disclosure

Many researches shown in Table VI have reported the pervasiveness of privacy disclosures[2] in Android apps. We categorize current analysis approaches into three types: static, dynamic, and hybrid dataflow analysis.

8.1.1. Dynamic Dataflow Analysis. Under this concept, privacy disclosure detection is transformed into a dataflow tracing problem: finding viable traces from predefined source APIs (the ones that read private data) to sink APIs (the ones that send private data out). Dynamic dataflow analysis is performed while the app is being executed on real devices or emulated environments, therefore, it is highly resistant to code polymorphism, for example, Java reflection and code encryption (see Section 6.1).

Past projects using dynamic dataflow analysis have demonstrated precise detection results. TaintDroid [Enck et al. 2010] is the first dynamic analysis engine for Android apps. It performs taint tracking to precisely analyze how private data is obtained and released at runtime. In achieving this, it pioneers an efficient and elegant way to handle taint storage. It also defines taint propagation rules on Dalvik instructions across API calls. As TaintDroid handles taint analysis of Dalvik instructions across API calls at runtime, it is resistant to Java reflection and code encryption. In addition, TaintDroid can be loaded into real devices, allowing for realtime monitoring of actual hardware and sensors. These advantages have pushed TaintDroid to be used widely in Android app behavior analysis. However, TaintDroid cannot support the latest Android ART

[2] Privacy disclosure means that private data is disclosed outside the device, which could be either legitimate or malicious; whereas privacy leak means private data is leaked for malicious purposes, which is a subset of privacy disclosure.

Table VI. Categorization of Proposed Algorithms for Detecting Privacy Disclosure of Apps

Analysis	Projects	Description	Bypassable?
Dynamic	TaintDroid	Dynamic tracking on Dalvik VM	Taint cleanse through control dependence or side channels
	NDroid	Dynamic tracking on native code (JNI)	
	Capper	Tracking with Dalvik bytecode rewriting	
	VetDroid	Permission-based discovery of taint sources and sinks	
	SuSi	Machine learning for discovery of taint sources and sinks	
Static	CHEX	Intercomponent (ICC) awareness	Code encryption, Java reflection, dynamic code loading, etc.
	Epicc	Reduce ICC to IDE problem	
	FlowDroid	Lifecycle awareness	
	Amandroid	ICC & lifecycle awareness	
	IccTA	ICC & lifecycle awareness	
	EdgeMiner	Model implicit control flow through Android framework	
	DroidSafe	Model Android specific features with “stubs”	
Hybrid	AppIntent	Event-constrained symbolic exec	Any technique in static or dynamic.
	SmartDroid	Directional dynamic execution	
	Intellidroid	Event-focused API reachability analysis	
	Harvester	Execution on sliced app components	

runtime, deployed since Android L. Complementing TaintDroid, ARTDroid [Costamagna and Zheng 2016] could be extended to implement the dynamic taint analysis in the ART runtime in the future. ARTDroid is a hooking framework on Android ART runtime without modifications to both the Android system and the app’s code.

It is worth noting that besides the execution engine, the effectiveness of dynamic dataflow analysis relies on two more important components: (1) data source and sink definition, and (2) input generation and test driving. Although both components can be complemented with manual effort, human involvement is certainly impractical for scalability reasons. In terms of automated source/sink discovery, VetDroid [Zhang et al. 2013] leverages the predefined Android permissions for automation. To be specific, it automatically marks the information returned by permission-backed function calls as tainted. SuSi [Rasthofer et al. 2014] uses machine learning techniques to automatically identify data source and sinks in Android APIs with a comprehensive feature set including API method name, return value type, class name, etc. We postpone the discussion on Android app automation tools to Section 8.4.

Several follow-up works are proposed on the dataflow analysis engine as well: NDroid [Qian et al. 2014] provides a complementary mechanism for taint-tracking information flows through JNI. It interfaces with TaintDroid’s tracking logic on the Dalvik VM side and, in the native context, maintains taint storage using shadow registers and memory maps. NDroid tracks taints by hooking functions through `QEMU`. To reduce the relatively high runtime overhead of TaintDroid, (32% measured by Enck et al. [2010]), Capper [Zhang and Yin 2014] proposes to instrument the app instead of the Android system in incorporating taint-tracking logic. It employs a byte code rewriting approach to insert code in to the original app codebase in order to keep track of private information and detect data leakage. Capper claims to have better runtime performance than TaintDroid.

8.1.2. Static Dataflow Analysis. Although modern static dataflow analysis systems have employed many techniques to improve their analysis precision on Java programs, we cannot directly port these static analysis systems to the Android platforms, as Android introduces many unique programming paradigms that need to be handled correspondingly, including:

—Event-driven system. Android is an event-driven system. The control flow of an app is determined by events, and there are many callbacks for system-event handling, for example, UI interaction and location update, which pose significant challenges in building a precise control flow graph.

—Runtime intercomponent communications. Since an app often consists of multiple components with various entry points, complex ICCs flows, including both intracomponent and intercomponent control and dataflows, should be considered. Statically building the control flow graph among components poses a challenge to static analysis techniques.

CHEX [Lu et al. 2012] proposes a static dataflow analysis system to detect component hijacking vulnerabilities in Android. To capture dataflows in multiple components, CHEX first finds all app-splits (an app-split consists of all code segments reachable from an entry point) and then permutes the identified app-splits to find intercomponent dataflows. However, Android OS defines an ordering of lifecycle events for all components in an app. For example, a component can only be stopped or paused if it is started, and may later be resumed. CHEX does not consider such lifecycle; instead, it enumerates all possible app-split orderings, which may introduce a severe imprecision. Epicc [Octeau et al. 2013] reduces the detection of ICC to an instance of the Interprocedural Distributive Environment (IDE) problem. But it also has the same limitation as CHEX. FlowDroid [Arzt et al. 2014] only performs intracomponent analysis and IccTA [Li et al. 2015] extends FlowDroid to analyze intercomponent dataflows. Both IccTA and Amandroid [Wei et al. 2014] focus on ICC privacy leaks and model Android-specific features such as component lifecycle, intent, and callbacks in a precise manner. Such a fine-grained modeling significantly reduces both false positives and false negatives. EdgeMiner [Cao et al. 2015] further improves the modeling of the Android framework with API summaries that describe implicit control flow transitions through the Android framework. DroidSafe [Gordon et al. 2015] represents the latest development in Android static dataflow analysis. It integrates many Android-specific features such as native methods, event callbacks, component lifecycles, etc., into its AOSP model and abstracts them with simplified "stubs" that are accurate enough for point-to and dataflow analysis. It also employs many heuristics to statically model ICC. One drawback, however, is its dependence on manual specifications, which can be error-prone and inflexible with system updates.

In general, static dataflow analysis techniques do not exhibit the low code coverage problem. However, they may run the risk of high false positives, as these techniques tend to conservatively overapproximate point-to targets or model program inputs. In addition, they cannot resist code encryption or Java reflection [Arzt et al. 2014; Lu et al. 2012]. Unfortunately, these features are popular in Android apps, either for self-protection or for performance improvements.

8.1.3. Hybrid Program Analysis. Hybrid analysis is a natural direction to balance efficiency, scalability, and accuracy in identifying privacy disclosure. The basic idea is to use static analysis to narrow down the scope of code pieces to be examined at runtime, and then perform dynamic analysis on the identified code pieces.

AppIntent [Yang et al. 2013] uses static analysis to identify relevant code sections to execute. At runtime, AppIntent exhaustively run dynamic symbolic execution to fully explore all behaviors. SmartDroid [Zheng et al. 2012] is similar to AppIntent, except that it only handles UI events and ensures that the app follows a specific path at runtime. Intellidroid [Wong and Lie 2016] generalizes the targeted execution problem by defining methods of interest as a set of APIs and then extracts the events required to reach these APIs. Harvester [Rasthofer et al. 2016] complements static analysis tools by computing runtime values and feeding them to these tools to reduce uncertainties (e.g., to resolve reflection targets). It performs runtime value extraction by first generating a reduced app that contains the value-of-interest and then dynamically executing and extracting the value.

8.2. Detecting Privacy Leakage

It is noteworthy that sensitive dataflows detected by the aforementioned works are not necessarily suspicious or malicious, as most of them are actually necessary to the apps' functionalities and should be allowed. Judging the legitimacy of detected privacy disclosures usually requires domain knowledge, and thus is hard to be automated. To the best of our knowledge, only a few papers aim at differentiating suspicious privacy leaks from legitimate ones, and they can be categorized into two classes:

8.2.1. User Interaction Check. The intuition is that users' interactions (e.g., users' consent on the disclosure of location information) should present before private data is disclosed. Livshits and Jung [2013] implements a graph algorithm to place mediation prompts to ask for users' consent if no user interaction is found. Similarly, AppIntent [Yang et al. 2013] aims to match the sequence of Graphical User Interface (GUI) manipulations with the sequence of events that trigger the private data access and disclosure. AppIntent considers the detected privacy disclosure legitimate only when the user intention, for example, clicking of "send" button to send a Short Messaging Service (SMS) message, is found in the extracted event graph.

8.2.2. Peer App Voting. User-device interaction analysis for privacy disclosure legitimacy may incur high false negatives, since an interaction does not always mean "intention." Another approach is to extract the "standard/common privacy disclosures" from functionally similar peer apps, and then compare the suspicious privacy disclosures with the extracted "standard set." If the disclosure is a privacy leak, it will generally not be included in the "standard set" since its peer apps do not have such leaks. AAPL [Lu et al. 2015] provides a framework to automatically infer privacy disclosures legitimacy based on peer voting mechanism. As the peer apps are selected based on similar functionalities, detection of privacy leaks that are not functionally required may have high false rates, for example, detection device ID and phone number.

According to AAPL, about 67% of detected privacy disclosures are in fact legitimate. With an automated approach to differentiate legitimate privacy disclosures from suspicious ones in demand, we strongly hope more researchers can contribute to this direction in the future.

8.3. Identifying Malicious Behavior

As malicious apps pose increasing threats to the Android ecosystem, a series of malicious app detection mechanisms has been proposed. Based on the detection methodologies, we categorize them into four types: execution-based detection, model checking, WYSIWYX, and machine learning, as shown in Table VII.

8.3.1. Execution-Based Detection. Abnormal behavior detection using sequences of system calls has been successfully applied on the intrusion detection domain, as the sequence of system calls executed by the program is a good indicator between normal and abnormal behaviors. Higher level semantics such as ICC through Binder IPC/RPC

Table VII. Categorization of Algorithms for Identifying Malicious Behavior of Apps

Detection	Projects	Description	Bypassable?
Execution	DroidScope/CopperDroid	Execute/monitor apps in an emulator	Emulator detection techniques
	NJAS/Boxify	Sandbox interaction between apps and Android/OS	Sandbox escape
Model Checking	Pegasus	Permission Event Graph based checking	Code encryption and reflection
	AppContext	Check security-sensitive behavior contexts	
WYSIWYX	WHYPER/AutoCog	Compare permissions to app description	Ambiguous/fake description
	CHABADA/ACODE	Compare API calls to app description	
	AsDroid	Evaluate app behaviors based on UI texts	Dynamic UI/code loading
Machine Learning	Hao et al.	Check permission requests and app categories	Feature manipulation
	DroidAPIMiner	Check API calls, packages, and parameters	
	Drebin	Check permissions, APIs, and network activity	
	DroidSIFT	Check contextual API dependency graph	
	MUDFLOW	Check sensitive data accesses and usage	

or Dalvik VM traces are additional features that Android app analysis techniques can leverage to improve detection accuracy.

Following this intuition, two QEMU-based solutions, DroidScope [Yan and Yin 2012] and CopperDroid [Kimberly et al. 2015] are proposed to both capture OS-level event sequences (system calls) and higher level semantics. However, malicious apps can use emulator detection techniques to prevent itself from exhibiting malicious behaviors inside an emulator [Kirat et al. 2014].

Another line of research focuses on achieving the same goal by sandboxing apps inside a real Android device instead of whole system emulation/virtualization. NJAS [Bianchi et al. 2015] builds an app sandbox with `ptrace`-based syscall interposition. It also hooks `Binder` IPC calls to mediate interaction with the Android framework. Boxify [Backes et al. 2015] leverages the `isolated process` feature introduced in Android 4.3 to sandbox a target app and also completely mediates its interaction with the framework. In addition, Boxify can launch multiple apps in the same sandbox, and hence, could potentially capture collusion attacks that requires cooperation of multiple independent apps. Both works are capable of fine-grained information recording without the need of kernel or Android framework modification. However, sandbox-based solutions always suffer from attacks that attempt to escape them, as shown in the Chrome sandbox case [Fisher 2015]. Because the sandbox itself is usually highly privileged, compromising the sandbox yields more benefits to the attackers than compromising a normal app.

8.3.2. Model Checking. The general idea on model checking is to treat the Android platform as an "event-driven" system and model the interactions of an app with the platform. Checking these interactions will help catch some abnormal behaviors, for example, SMS is sent without user's consent (i.e., click the "send" button).

Pegasus [Chen et al. 2013] proposes Permission Event Graph (PEG) to model check an app's behaviors. PEG is an abstraction to the interactions between the Android event system, permissions, and API calls in a given app. With such graph, the analyst can specify policies that model the legitimate behaviors an app should exhibit. By model checking the graph with these policies, abnormal behaviors that violate the policy will be uncovered. A nonnegligible limitation with model checking-based detection is that manually specified policies are hard to be precise and complete, since an app's normal behaviors could be diverse in different contexts.

Another approach, AppContext [Wei et al. 2015], models malicious apps according to two contextual observations. First, malicious behaviors are usually triggered not by UI events but by system events to evade from user attentions (*activation conditions*). Second, malicious behaviors are usually triggered only when specific environmental conditions (e.g., location and time) are satisfied to evade detection systems (*guarding conditions*). AppContext extracts such conditions for security-sensitive behaviors from apps. Although the model checking looks promising, code polymorphism can be used to hide malicious behaviors and evade detection. For example, dynamic code loading is widely used for enhancing app flexibility and compatibility, but the loaded code cannot be reliably modeled statically. In terms of Java reflection, some reflection cases can be easily modeled using constant propagation, while others need actual execution to analyze.

8.3.3. WYSIWYX. **W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou e**X**ecute is an intuitive policy aiming to ensure that the actual app behaviors should be consistent to users' perceptions: a functionality not stated or implied in the app description should not be allowed. Checking WYSIWYX is usually done by text analytics with the help of recent development of Natural Language Processing (NLP).

WHYPER [Pandita et al. 2013] and AutoCog [Qu et al. 2014] discussed in Section 4.2 extract permissions *advertised* in the app's description, and compare them with permissions actually requested. CHABADA [Gorla et al. 2014] and ACODE [Watanabe et al. 2015] first reduce the app descriptions into a set of keywords/topics and then measure whether the actual APIs used in the app confirm with these topics. AsDroid [Huang et al. 2014] first extracts and analyzes UI text to infer the *criteria behaviors* of the apps, and then matches them with the actual app behaviors to uncover contradicted or stealthy activities. However, WYSIWYX approaches are vulnerable to fake description and dynamic UI text manipulation.

8.3.4. Machine Learning-Based Detection. To achieve effectiveness and scalability of Android malware detection, machine learning-based approaches are well explored and promising. The basic idea is to fist statically or dynamically analyze the app to gather a predefined set of features, and then learn a detection model based on given datasets consisting of malicious apps and benign apps. For example, Hao et al. [2012] builds probabilistic generative models using requested permissions and app categories to rank their risks. DroidAPIMiner [Aafer et al. 2013] considers package-level information as well as API call sequences and parameters to distinguish malicious apps. The state-of-the-art system, Drebin [Arp et al. 2014], not only uses the explained features, but also considers other features such as intents, requested hardware components, and network addresses. Its detection rate is up to 94% with a false positive rate of only 1%. DroidSIFT [Zhang et al. 2014c] extracts a weighted contextual API dependency graph to represent program semantics (i.e., feature set) and further uses graph similarity metrics to uncover potential malware variants. MUDFLOW [Avdiienko et al. 2015] detects malware based on the intuition that malicious apps treat sensitive data differently from benign apps, and hence, can be captured by identifying abnormal dataflow.

However, it is noteworthy that (1) the accuracy of machine learning-based detection is highly dependent on the quality of datasets used for training; and (2) the detection may be easily bypassed if the attacker can figure out how the features are combined for malware indication. Two reports on machine learning-based Android malware detection [Allix et al. 2014; Roy et al. 2015] both show that the detection results are significantly biased when fed with different malware datasets for training. Also, it is suggested that security research should not produce approaches or techniques that are not in line with reality, for example, the detection results of Drebin, which is derived from a "predefined" sample of 5,560 malware may be biased from the ones derived from other app datasets that are crawled from real app markets. So, we hope to see in the future that machine learning-based malware detection approaches can choose app datasets that are consistent with both reality and other competitive approaches as training and evaluation sets, for example, the top 1,000 apps that are most commonly installed by users.

8.4. Test Automation Tools

The effectiveness of dynamic analysis in detecting privacy disclosure or malicious behaviors relies significantly on test automation tools to drive the app. Therefore, a large number of techniques has been proposed to better automate Android app testing.

AndroTest [Shauvik et al. 2015] provides a comprehensive summary and comparison of existing Android testing tools. Based on exploration strategy, current testing methods are categorized into three classes: random, model-based, and systematic. Dynodroid [Aravind et al. 2013] is based on random exploration, but its exploration technique is claimed to be more efficient than Monkey, the default testing tool that comes with the Android SDK. MobiGuitar [Amalfitano et al. 2015] dynamically builds a model of the app under testing by crawling it from a starting state. AppsPlayground [Rastogi et al. 2013], A3E [Tanzirul and Iulian 2013], SwiftHand [Wontae et al. 2013], and PUMA [Shuai et al. 2014] are all similar to MobiGuitar with different static analysis and exploration strategies.

Acteve [Saswat et al. 2012] is a concolic-testing tool that symbolically tracks events from the point where they are generated in the Android framework up to the point where they are handled in the app. In contrast, Evodroid [Mahmood et al. 2014] relies on evolutionary algorithms to generate relevant inputs. Harvester [Rasthofer et al. 2016] provides new insights on automated test driving. Different from other approaches that start path finding from program entry points and explore down to the point-of-interest, Harvester first performs *backword slicing* from the point-of-interest and extracts the minimal code required to trigger the target from a program entry point, hence, significantly improving the efficiency of app testing.

8.5. Open Problems

8.5.1. False Alerts. All proposed algorithms for identifying privacy leakage suffer from false positives: incorrectly reporting legitimate privacy disclosures as suspicious or malicious leakages [Livshits and Jung 2013; Yang et al. 2013], which usually dominate the detection results [Lu et al. 2015]. Unfortunately, humans have little patience for false alerts. No warning will be attended after malware detectors report a few incorrect alerts. To be effective and practical, future solutions should consider such false alerts seriously.

8.5.2. Malicious Behavior Triggering. A fundamental problem on test automation tools is how to guarantee that all malicious behaviors can be triggered during testing. In this sense, merely measuring the code/path coverage might not be sufficient as malware can easily hide malicious behaviors deep inside the program, for example, executing malicious activities after 1 hour of app launching (time bomb) or only when no debugger is attached (logic bomb). Another common scenario is: if the app starts with a login screen, none of the test automation tools we surveyed will be able to process further without valid credentials. Furthermore, if malicious behaviors or privacy leakage only manifests after the login activity, it will never be triggered during testing. However, obtaining login credentials inevitably involves human interaction, which compromises scalability.

Harvester [Rasthofer et al. 2016] sheds light on this problem based on the assumption that the path that leads to a predefined set of potential malicious behaviors (e.g., send SMS or make phone calls) generally has no data dependence on the antianalysis techniques used. Therefore, through precise *program slicing*, the aforementioned time bomb and logic bomb can be sliced out, preserving only code sections that trigger the target behavior.

8.5.3. Analyzing Native Code. A majority of proposed algorithms do not assume the use of native code in Android apps, which is hardly true in the current Android ecosystem. Although NDroid [Qian et al. 2014] proposes a new taint-tracking algorithm that can identify dataflows in the presence of native code, it still suffers from high false negative and incurs nonnegligible runtime overheads. Native code must also be considered in developing new detection and analysis algorithms due to its wide presence in real-world apps.

8.5.4. Strong Adversary. Due to the diverse features of Android malware, no detection technique can be a panacea, and many of them can be bypassed if attackers correspondingly adjust their offensive techniques. (1) Many antitaint analysis techniques have been developed in order to cleanse taints during dynamic tracing, as reported in Sarwar et al. [2013]. (2) Static data analysis cannot resist intentional code obfuscation techniques, such as using code encryption, dynamic code loading, Java reflection, and JNI. (3) WYSIWYX is an intuitive policy but is hard to enforce as extracting the *criteria behaviors* is nontrivial. (4) Model checking-based algorithms usually require manual specifications of policies, which are neither precise nor complete. (5) Even though using machine learning seems an attractive direction, we still need a systematic way of collecting training and evaluation data to avoid biases [Allix et al. 2014].

8.5.5. Place of Detection. Given the plethora of privacy violation and malware detection tools, it is a natural question on where these tools should be deployed. On-device detection, restricted by CPU power, memory, and battery, might suffer from low precision, while cloud-based detection might suffer from high communication overhead and late response. Devising a hybrid approach will be beneficial to the Android community.

9. APP REPACKAGING

App repackaging is prevalent among Android malware; according to Zhou and Jiang [2012], 86% of malware samples are repackaged versions of benign apps.

The pervasiveness of app repackaging among malware can be explained in three ways: (1) A repackaged app can boost its infection rate leveraging the victim app's popularity. (2) It can also achieve stealthiness by preserving original app functionalities. (3) It is technically easy to repackage an Android app (unless it is heavily obfuscated, which few developers can do or will do). All these properties help malware writers meet their motivations directly (e.g., stealing credit card numbers) or indirectly (e.g., bumping ad's click counts).

Recent years have witnessed many solutions to tame this problem, and in general, these solutions can be classified into two categories: repackaging detection and repackaging prevention.

9.1. Repackaging Detection

All repackaging detection algorithms surveyed in this article follow a general procedure: (1) features of the target app, such as logics and UI, are extracted and deterministically transformed into a special representation format, a.k.a., birthmarks, and (2) a subsequent comparison between two app birthmarks determines the similarity between two apps. Table VIII summarizes existing works. As shown in Table VIII, we perform a subjective comparison on their performance using two criteria: transformation resilience and scalability.

Transformation resilience measures how an algorithm might be defeated if an attacker uses one of the following common evasion techniques: ① point-of-interest minor modifications, ② control flow changes, ③ data dependency changes, and ④ heavy obfuscation (encryption).

Table VIII. Repackage Detection Techniques. In the Resilience Column, ①–④ Denotes Obfuscation Techniques Discussed in Section 9. In the Scalability Column, *S* Denotes DEX Bytecode Size; *C* Denotes Program Complexity, in Number of Methods [Crussell et al. 2012, 2013a; Desnos and Gueguen 2012; Hanna et al. 2012], or Number of Activities and Intents [Zhang et al. 2014a]; *P* Denotes Parameter of the Detection Technique

Solution	App representation	Similarity comparison	Resilience
DroidMOSS	Hash of opcodes block	Edit distance	Low, ①
DNADroid	Data dependence graph	Subgraph isomorphism	Medium, ①–③
AnDarwin	Data dependence graph	Subgraph isomorphism	Medium, ①–③
Androguard	Regex string of CFG	Normalized compression distance	Low, ① ②
PiggyApp	Regex string of CFG	Normalized compression distance	Low, ① ②
Centroid	Centroid of CFG	Method centroid distance	Low, ① ②
DroidSim	Component-based CFG	Jaccard coefficient	Low, ① ②
Juxtapp	Feature hashing	Jaccard similarity metric	Low, ① ②
ViewDroid	View-event graph	Subgraph isomorphism	Strong, ①–④
ResDroid	View-event graph and statistics	Hierarchical Clustering	Strong, ①–④

The general observation is that sequence-based approaches are less resilient to code obfuscation techniques as code streams do not contain any higher level semantic knowledge, and hence are more likely to be defeated with control flow changes such as method restructuring, statement re-ordering, or dead code insertion. Program Dependency Graph (PDG) based approaches generally have better resilience, but they can still be defeated by data dependency changes, for example, massive aliasing of variables. User-interaction-based approach leverages the user-interaction/event-driven nature of Android apps and is more resilient to code-level obfuscations.

DroidMOSS [Zhou et al. 2012] applies fuzzy hashing to each app's opcodes to generate the birthmark. For each sequence of opcodes, it partitions them into smaller chunks and aggregates the hash of each chunk to get the final hash. It then measures the similarities of two apps using a custom formula with both hash values as input. Similarly, Androguard [Desnos and Gueguen 2012] uses several standard similarity metrics to hash app functions and basic blocks for comparison. Juxtapp [Hanna et al. 2012] characterizes apps through k-grams of opcodes and feature hashing and clusters corresponding bitvectors to identify app repackaging. PiggyApp [Zhou et al. 2013b] is designed to detect piggybacked apps, a special kind of repackaged app with code injected into victim apps. It first deconstructs an app into modules according to their dependency relationship and then constructs a fingerprint for the primary module by collecting various features, such as requested permissions and Android API calls.

Desnos [2012] uses Normalized Compression Distance (NCD) to compare app similarity according to method signatures, including external API usages, exceptions, and CFG. Potharaju et al. [2012] proposes an approach to detect plagiarized apps baesd on symbol tables and method-level AST fingerprints. This approach can handle obfuscation techniques that mangle symbol tables or insert random methods with no functionality. DroidSim [Sun et al. 2014] utilizes Component-Based Control Flow Graph (CB-CFG) to quantify the similarity between apps. DNADroid [Crussell et al. 2012] constructs a PDG for each method and performs subgraph isomorphism comparison on PDGs after filtering out unnecessary methods. To speed up DNADroid and AnDarwin Crussell et al. [2013a] splits PDGs into connected components (i.e., semantic blocks), each of which will be represented by a semantic vector containing the number of specific types. After that, it employs a Locality Sensitive Hashing (LSH) algorithm to identify code clones that have similar semantic vectors. Crussell et al. [2013b] proposes a novel approach that uses the centroid of control dependency graph to measure similarity between methods in order to detect cross-market app clones.

Besides the aforementioned works that analyze program logic, Zhang et al. proposes ViewDroid [Zhang et al. 2014a] that first constructs a feature view graph and then applies subgraph isomorphism to measure similarity between two apps. ResDroid [Shao et al. 2014b] follows a similar approach, it extracts information about UI elements with minor granularity differences. Besides, Shao et al. [2014b] leverages the fact that attackers often use code loading for app repackaging—simply packing bytecode into JNI at deliver time and recovering bytecode at runtime—and presents a tool to dump the memory at runtime. The analysis is done on the runtime bytecode, which solves the problem of heavy obfuscation to some extent.

We also observed evolution of repackage detection techniques, from simply porting traditional approaches for desktop application repackaging detection (such as DroidMOSS) to leveraging Android-specific features such as events and user interactions (such as ViewDroid).

9.2. Repackaging Prevention

Besides detection, solutions have been proposed to tame app repackaging in a more fundamental way.

9.2.1. Embedding Watermark Into the App Executable. AppInk [Zhou et al. 2013a] presents a toolkit that can (1) embed a runtime watermark in the app by instrumenting its source code and (2) generate a complementary verification app that can automatically extract and verify the embedded watermark. Developers publish the watermarked app to app stores and the verification app is requested on demand to prove the originality.

9.2.2. Ensuring App Authenticity Verification without Relying on CA. AppIntegrity [Vidas and Christin 2013] modifies the current app signing and verification procedures to achieve a simple authentication protocol. Instead of embedding the verification key in the app package, it suggests app developers place the verification key behind the DNS that is within his/her control and use the package name as a hint to direct users to his/her DNS. However, this approach is prone to man-in-the-middle attacks and lacks practicality as not every app developer has Domain Name System (DNS) service or prefers to name their app packages in this way.

9.2.3. Making Repackaging Technically Hard. For example, DIVILAR [Zhou et al. 2014b] obfuscates Dalvik bytecode with a randomized virtual instruction set, and translates its obfuscated code with a customized interpreter at runtime. Such an approach significantly increases the bar for reverse engineers, but considering limited resources in mobile, this approach may result in a high performance penalty.

9.3. Open Problems

9.3.1. Repackaging Detection Algorithms Prone to Code Obfuscation. For repackaging detection, all the algorithms we surveyed rely on static analysis, thus are prone to obfuscation techniques such as code encryption and dynamic code loading (see Section 6 and Section 8). More critically, all surveyed algorithms perform analysis only on DEX bytecode, limiting its effectiveness in practice where 16% apps have native code embeded [Qian et al. 2014]. One option to solve these problems is to leverage dynamic features such as UI changes, event sequences, or other runtime invariants. The intuition is that, since fake apps usually want to preserve or mimic the appearance and the user experience of the original apps, dynamic features may be more resilient.

9.3.2. Repackaging Prevention Algorithms Lack Deployability. For repackaging protection, none of the techniques proposed are readily deployable due to high performance penalty incurred or drastic changes made to the app publish/download process. Therefore, the quest for efficient repackaging prevention techniques is still open for research.

9.3.3. Malicious Behavior Extraction. Since most malware is repackaging benign apps, an intuition would be to extract out the added component to improve malware analysis accuracy or efficiency. For example, it would be beneficial to separate the privacy invasive functionalities in a benign app from the malicious behaviors of the malicious part. However, such a goal could be difficult due to potentially unlimited ways a malware might use to repackage a benign app. Hence, automated malicious behavior extraction remains an open problem in both app repackaging and malware detection areas.

10. INFECTION CHANNELS CUT-OFF

This section describes how Android malware finds their ways into users' devices from an ecosystem perspective and compares research proposals to cut off these infection channels.

Due to the openness of the Android app distribution environment, malware can be distributed from either trusted app stores like Google Play or other sources such as web forums or alternative app stores. We survey the existing and potential mechanisms that could prevent malware distribution through both channels.

10.1. Trusted App Stores

Many trusted app stores already utilize some vetting mechanisms to minimize malware distribution. For example, Google Play uses *Bouncer* [Poeplau et al. 2014] to prevent malicious apps from getting into the store. Another mechanism is app rating and reviews. Although app rating is not generally viewed as a security mechanism, app rating/reviewing can be an effective way for users to alert the app store and other users.

10.2. Untrusted Sources

Various techniques can entice users to download an app from untrusted sources: search engine optimization [Gu 2014], in-app promotion [Apvrille 2014], phishing [Naraine 2012], and drive-by attacks [Lookout, Inc. 2012a]. Malware can also be silently installed through `adb` if a user connects his/her device to a compromised computer [Kassner 2014]. To defeat this threat, stock Nexus devices provide an option to prevent installing apps from untrusted sources. Users can also install antivirus apps to protect them against *known* malware.

10.3. Open Problems

10.3.1. Inefficiencies in User Feedback. Although user feedback like app review/rating might be an effective sociological mechanism to minimize malware distribution from an app store, it is plagued by two factors:

—App reputation is subject to rating optimization services where app developers pay the service providers to help boost their apps' reputation rating by false downloads or false comments, as shown by AppWatcher [Xie and Zhu 2015].

—User feedback cannot be used to discover zero-day malware as feedback is only effective with a large sample size, that is, number of users downloading the malware. The malware may have already achieved its popularity goal at this stage.

Therefore, how to leverage user feedback and other nontechnical solutions in infection channel cut-off is a promising research problem, which might require interdisciplinary efforts from humanity, physiology, etc.

10.3.2. Collective App Vetting. For trusted app stores, one possible way to enhance the vetting process is to invite independent parties, especially security-oriented parties such as antivirus vendors, to jointly analyze and endorse/rate apps. In this case, users can get more assurance from security experts about the apps they download.

Table IX. Summary of Popular Monetization Schemes by Malware

Monetization Scheme	Description	Financial Benefit
Toll fraud	Premium rate number billing via SMS or call	Direct
Click fraud	Imitating user's clicks in pay-per-click ads	Direct
Pay-per-installation	Payment per app installation (e.g., up to $1 USD)	Direct
Ad profit hijacking	Replacing developer's ID to reroute ad income	Direct
Ad network hijacking	Replacing the underlying ad provider	Direct
Adware	Harvesting ad views	Direct
Mining	Mine virtual currency on user's devices	Direct
IMEI/IMSI stealing	Stealing device identity (e.g., imitate or unblock)	Indirect
Spyware	Stealing personal information (e.g., contacts)	Indirect
Man-in-the-mobile	Stealing random tokens (e.g., mTAN for Bank app)	Indirect
Ranking poisoning	Requesting fake search queries to boost rank	Indirect

10.3.3. Cloud-Based Malware Scanning. Without strong DRM (Digital Rights Management) protection, it is impossible to prevent malware infection through untrusted sources. However, given DRM is against the open nature of the Android ecosystem, the best way to prevent malware infection through untrusted sources is probably to warn users [Willis 2013] of potential threats by allowing the system to submit the downloaded app to cloud services for repackage check [Lindorfer et al. 2014], reputation check [Rajab et al. 2013], etc.

11. INCENTIVE ELIMINATION

Incentive elimination is an underresearched area that lacks direction and quantitative studies. It also requires creative approaches to counter malware writers' incentives. This section describes our perspectives on potential research opportunities to reduce the incentives for malware creation.

11.1. Understanding Attacker's Monetization Schemes

Attackers create Android malware for various reasons, but the primary incentive is financial return. Such an incentive exists because the current Android landscape provides many malicious monetization opportunities that have emerged into a business considered as malware-as-a-service. Table IX summarizes some of the monetization schemes identified in recently discovered Android malware samples based on industry reports [Chien 2011; Lookout, Inc. 2012b][3]

We believe that as long as these schemes exist, it is highly likely that new (and potentially more sophisticated) malware will be created to meet attackers' monetary incentives. From this perspective, a complete elimination of monetization schemes is the best cure for malware issues in the Android ecosystem.

However, elimination of those incentives usually requires multiple parties' collective effort, not only just the core participants of the Android ecosystem. For example, in order to prevent search engine poisoning or click fraud, search engine or ad providers need to employ techniques that make it difficult to generate legitimate search or click requests automatically, which by itself is another hot research topic [Lu et al. 2011]. As another example, to stop man-in-the-mobile attacks, instead of using SMS, financial institutions should require users to use a stand-alone hardware-based token generator in a two factor authentication scheme. Fortunately, such incentive elimination efforts also align with their own interests.

Meanwhile, we do not deny the fact that many incentives are hard to be eliminated, at least for now, such as virtual currency mining and invasive advertisements. Therefore,

[3] Kumar and Kaur [2015] provides some examples on how to monetize stolen IMEI numbers, which might not be obvious based on the Symantec and Lookout report.

although it is worthwhile for society to make an effort in eliminating these incentives, it may not solve all malware issues.

11.2. Creating Legitimate Monetization Channels

While blocking illegal monetization channels is important, we believe creating legitimate channels is equally important. This is a general solution the Android ecosystem can offer to eliminate or at least reduce most of the incentives mentioned in Section 11.1.

The argument is that, in general, monetary gains from mobile malware become significant only if there is a large infection rate, which implies that the app itself is popular enough to attract a large number of downloads. If the ecosystem has legitimate schemes to reward developers who can create apps with such popularity, they will have no incentive to create malware to achieve the same goal.

Currently, the app store model of Android provides legitimate ways to reward developers who can create popular apps: profit sharing through (in-)app purchases. For developers of free apps, in-app advertisement is a common way to get compensation. However, in-app advertisement might impair user experience especially when developers opt for high-commission advertisement libraries, which in turn, usually use more invasive (or even malicious) ways in order to display advertisements. We believe more creative and diversified awarding schemes need to be designed to compensate developers of free apps.

12. ANDROID SECURITY OUTLOOK

Android is reaching further than smartphones. Emerging trends such as Internet-of-Things and digital identity are making Android security a more serious concern, as wide adoption of the Android platform gives attackers more incentives, and system compromisation leads to more serious damages. This section discusses new practices and attack surfaces, with the goal of soliciting more research in these areas.

12.1. IoT

12.1.1. Home Automation. IoT enables the connections of virtually any personal devices or appliance such as refrigerator, TV, light switches, doorlock, etc. Backed by the *Thread* work group [Rockman 2014] (initiated by Google, Samsung, and ARM), Android has been selected as the potential standard operating system for IoT, particularly for home automation [Vance 2013]. And Google has opened its own IoT platform `Brillo` with an Android-based embedded OS [Google Inc. 2016a]. However, Android must be heavily customized to drive IoT especially in terms of security. For instance, the home automation protocol requires authentications between devices. Further, Android lacks a fine-grained access control mechanism for individual apps to manage external IoT devices. For example, an app having a Bluetooth permission can access arbitrary Bluetooth-enabled IoT devices without any per-device permission checks [Naveed et al. 2014]. Thus, fine-grained access control mechanisms for external devices (e.g., Dabinder [Naveed et al. 2014] and SEACAT [Demetriou et al. 2015]) are necessary to protect IoT devices from malicious apps.

12.1.2. Cyber Physical System (CPS). Android is not limited to personal uses but also reaches further to CPS, by replacing traditional embedded systems. A previous work [Lei et al. 2013] shows a sensor-based voice message theft attack on mobile CPS. As Android is widely used in mobile CPS ranging from a handheld device for fire fighters, to a main controller of unmanned vehicle, to a remote controller of military arsenal, its security has became more serious than ever.

12.1.3. Digital Identity. The presence of mobile devices is one popular way to authenticate its owner; for example, Google Authenticator is a popular app used for the two-factor authentication of user login [Estourgie and Poll 2013]. Moreover, with Near-Field Communication (NFC), smart cards, and digital payment systems all stored on the device, Android plays an important role as digital identity, and accordingly, it requires thorough and provable protection schemes in the future.

12.2. Potential Massive Attacks

12.2.1. USB. In public places like airports, cafes, and libraries, users may plug their mobile devices into USB plugs. However, due to the multipurposes nature of USB ports (i.e., charging the battery, media exchange, and debugging), connecting the device to unknown USB ports exposes the device to the risk of arbitrary app installation by exploiting vendor-specific customization [Pereir et al. 2014].

12.2.2. NFC and Bluetooth. NFC and other similar proximity-based communication channels such as Bluetooth LE have been deployed to various Android devices. Gaining popularity in using NFC for Google Wallet or Android Beam, or pairing Bluetooth to various peripherals, a large number of blackbox fuzzing tools have been built in order to identify potential vulnerabilities [Miller 2012; Soto 2005]. Considering its communication proximity of 3–10cm, these attacks can be launched when the attacker approaches the device. Although it is a common practice that the majority of the NFC interactions only happens when the device screen is on and unlocked, attackers can easily find the attack windows such as during commuting time or near a store checkout counter.

12.2.3. WiFi. Mobile data usually does not come in unlimited volume or at high speed, therefore smartphone users tend to connect their phones to public WiFi without much care. When connected to an insecure Access Point (AP), DNS or Man-In-The-Middle attacks become more critical security issues compared to when connected in a typical desktop or server environment [Chaskar 2009], as mobile devices generally lack software/hardware components to defend such attacks. For example, with rogue WiFi, an attack that typically happens on desktop environment has found its way to mobile devices [Silver et al. 2014].

12.2.4. Baseband. Mobile devices are generally subject to exploits against the baseband chip, which is in charge of processing data transmitted between terminal and cellular base stations. One popular baseband processor from Qualcomm runs the custom RT kernel, REX, but without any standard protection schemes like ASLR and DEP [Delugre 2011; Weinmann 2012]. Omission of security schemes enables an attacker to unlock phones. Moreover, the whole software stack could be exposed if the device connects to a rogue base station [Weinmann 2012]. The mitigation to this exploit demands a mutual authentication scheme between the device and station.

12.2.5. Silent Drive-By Installation. Recent reports have shown possibilities of silent malware installation without any user interaction on the part of the victim by exploiting vulnerabilities on older versions of Android. Stagefright [Drake 2015] exploits Android's `libStageFright` component that processes downloaded videos without users' consent. The rendered MMS image from the SMS app enables a worm to penetrate silently. Fortunately, SEAndroid policies on recent Android versions have mitigated this attack. However, a more dangerous case have been revealed where a ransomware is silently installed by simply visiting an attacker controlled website [Constantin 2016]. Specifically, the JavaScript code on the website first exploits a vulnerability in `libxslt` and then drops the payload: `TowelRoot` kernel exploit. After the device is compromised, a ransomware is silently downloaded and installed. No user interaction is involved throughout the whole process.

12.3. Privacy

12.3.1. Ad Library. Advertisement is key to the Android market, especially in the profit model. Section 4 points out finer-grained granularity assignment should be employed to separate ad library and app. But private data access for ad library is still a problem. The new Android ecosystem should respect the privacy of mobile users, unlike in the current Android model where each app keeps personal information and does not give any control (or opt-out option) to device owners. One solution is to provide a central storage as a "personal vault" and regulate apps' accesses to the vault via well-specified APIs.

12.3.2. Persistent Monitoring. Ranging from fitness monitoring bands to health care accessories, Android becomes a central place to record personal, privacy-sensitive information. In particular, the always-on nature of these devices raises a question of having persistent threats of leaking the collected data [Naveed et al. 2014]. As Android-based devices become deeply insinuated to human's life, a naive bug can lead to serious privacy violation.

12.3.3. BYOD. As enterprises use Android for their business or in workplaces, employees are usually required to carry multiple devices to isolate business activities from personal uses. However, carrying multiple devices is inconvenient and Bring Your Own Device (BYOD) gains popularity as employees can use a single device but protect business logic from personal uses purely by software [Andrus et al. 2011; Morrow 2012]. As 24% employees use mobiles to access or store business data, new security mechanisms that protect business data and also isolate user's personal activity need a fair amount of research in order to provide provable and strong guarantees (perhaps, via new hardware) of their protections in a single software stack. Currently, KNOX [Samsung Electronics 2014] and Android for Work [Google Inc. 2016e] are the leading industry solutions that enable BYOD.

13. TOWARD NEXT-GENERATION ANDROID ECOSYSTEM

Based on our systematization of knowledge in improving the security of the Android ecosystem, we carefully envision the landscape of the next-generation Android ecosystem, as well as how existing and future Android security researches can fit into the landscape, as shown in Figure 4.

13.1. New Features

We believe that at the core of the next-generation Android ecosystem should be collaboration and openness. To be specific, the new ecosystem should have three distinctive features: (1) leveraging collaborative efforts, (2) promoting the openness of security-related data, and (3) providing egalitarian roles to participating entities. With these features, the proposed ecosystem can solicit more independent and diverse parties in the Android ecosystem, such as Antivirus (AV) vendors that can provide scalable malware scanning and research institutes that can test and evaluate the latest developments of detection and mitigation techniques with real-world samples. We illustrate this collaboration and openness effort with a few examples.

13.1.1. Service APIs. Unlike the current ecosystem that largely depends on Google's initiative and effort on security enhancement, our new model proposes a collaborative means to solicit more participants to the Android ecosystem security. Through the open access to the *Service APIs* infrastructure, various entities, including AV vendors and research institutes, can now contribute to the security evaluation of uploaded apps. Following the principle of open source security, which states more bugs can be caught by more eyes, we believe this openness in our new ecosystem can enhance

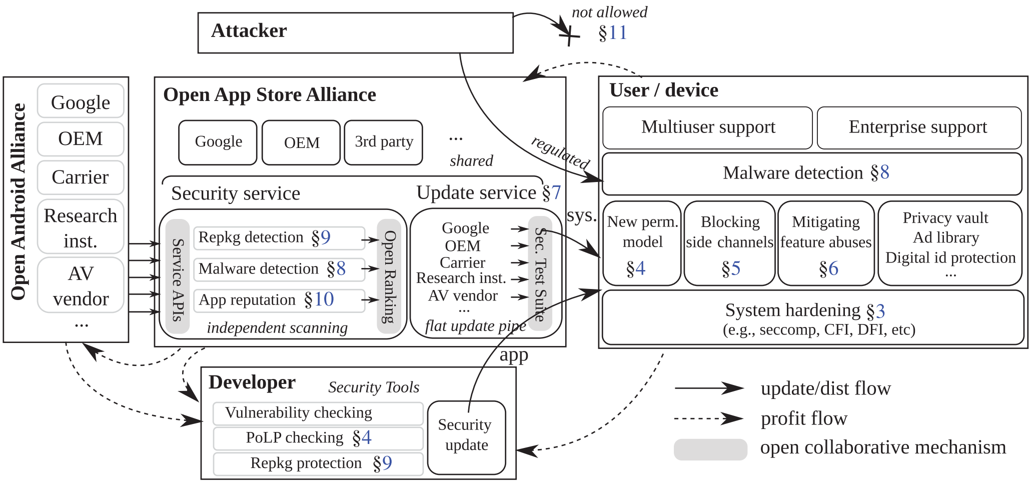


Fig. 4. Our proposal of the next-generation Android ecosystem. The new model considers security in every aspect of interactions between participating entities, by promoting openness, collaborative efforts, and egalitarian roles of participants. Also, it embodies the proposed research projects—hardening Android OS, developer's security tools, and algorithms—in the app store.

the overall security to an unprecedented level. The *Service APIs* can either be developed and maintained by Google or the Open Handset Alliance, as an official initiative or by independent third parties following a community-driven model and interface with related parties in the ecosystem, as demonstrated by the success of VirusTotal [VirusTotal Team 2012].

13.1.2. Open Ranking. Furthermore, the security scores evaluated by independent participants will be accumulated as an *Open Ranking*, which is shared across the ecosystem and will ultimately be consumed by end-users. This cumulative security ranking will open up new opportunities in improving the closed reputation system in the current Android app market. For example, we cannot only consolidate scores from multiple independent parties, but also consider the time domain or history of each app.

13.1.3. Security Test Suite. Since the new ecosystem promotes the egalitarian roles of participants, each player, including carriers, developers, or even AV vendors, can initiate new security updates streamlining to end users without reliance on other parties, hence, bypassing the lengthy update procedure (as shown in Figure 3) and dramatically shrinking avoidable vulnerability windows of known attacks. However, to enable this ideal update scenario, patch effectiveness and compatibility must be thoroughly tested before being released to end users. A new patch should neither break the functionalities required by law (e.g., calling emergency numbers without the presence of SIM card) nor compromise existing security mechanisms (e.g., violating the invariants defined by SEAndroid policies). To address this problem and enable a faster update procedure, the new ecosystem should provide *Security Test Suite*, which is similar to Compatibility Test Suite in AOSP in its form with a focus on testing the effectiveness and compatibility of each security update against as many device models as possible, even if the update is not developed by Google, OEM, or carriers. After a security patch is certified by the *Security Test Suite*, it can be delivered directly to end users from any entity (e.g., AV vendors, security researchers, etc.) instead of soley relying on Google and OEM to distribute the patch.

13.2. Overall Protection

Our new Android ecosystem also embodies the state-of-the-art research projects, as shown in Figure 4, particularly in hardening the devices security (S.1–S.5), in detecting malicious apps at app store (A.1–A.3), and in assisting developers with security tools (D.1–D.3).

13.2.1. System Hardening on User's Device. On-device hardening techniques should consider (1) potential security holes due to device fragmentation; (2) increasing demands on *private vault*, a conceptual private store to regulate accesses to personal information (e.g., health and fitness info); (3) new separation/isolation mechanism for personal and enterprise uses; (4) countermeasures against feature abuse; and (5) side/covert-channel.

13.2.2. Detecting Malware in the App Store. To support *Service APIs* and *Open Ranking*, the ecosystem requires groundbreaking algorithms to tackle current problems, including (1) repackaging detection, (2) malware detection, and (3) app rating. Our new ecosystem is flexible enough to embody prospective algorithms and experimental research proposals, as a field test.

13.2.3. Security Tools for Developers. In the current Android ecosystem, mistakes by app developers often directly impair the security of end-users (e.g., leaking private information). Considering the lack of security tools for app developers, our new ecosystem incorporates various techniques to assist developers in avoiding trivial, but security-critical mistakes on the course of app development: for example, enforcing PoLP and checking component hijacking vulnerabilities. Repackage prevention service is also provided to ensure app developers can harvest full credits for their original work.

14. CONCLUSION

In this article, we give a comprehensive narrative of the research landscape of Android security, with an analysis of security issues and solutions in the Android software stack and the Android ecosystem. We identify a number of intensively researched methodologies taking real effects such as kernel hardening (Section 3), dynamic or static program analysis (Section 8), and app repackaging detection (Section 9).

However, more fundamental problems are revealed from our evaluation of existing solutions: redesigning the permission model (Section 4), re-engineering system and app update procedures (Section 7), devising new algorithms to analyze user intention for malware detection (Section 8), preventing app repackaging attacks (Section 9), and eliminating attackers' monetization schemes (Section 11). In the meantime, emerging security threats such as the baseband attack and new practices like BYOD deserve more research attention (Section 12). We believe that solutions to these issues are indispensable and pressing for a secure ecosystem in the future.

Based on our findings, we propose a holistic approach at the level of the ecosystem, composed of three distinct parts: (1) building the *Service API* infrastructure to encourage diverse participants to the Android ecosystem security playground; (2) sharing and consolidating security-related data through an *Open Ranking* mechanism; and (3) promoting egalitarian roles among participants by sharing the *Security Test Suite* and flattening the update flow. We also present guidelines for validating how existing and future solutions fit into Android ecosystem security. We believe this work will open up new opportunities to facilitate state-of-the-art security research into Android, enhancing its security for the advent and growth of the IoT.

APPENDIX

A. PAPER SOURCES

Table X. A List of Major Sources of the Surveyed Papers

Abbr.	**Full Name**	**#**
Oakland	IEEE Symposium on Security and Privacy	7
CCS	ACM Conference on Computer and Communications Security	22
Security	USENIX Security Symposium	19
NDSS	Network and Distributed System Security Symposium	20
ACSAC	Computer Security Applications Conference	7
ASIACCS	ACM Symposium on Information, Computer and Communication Security	4
CODASPY	ACM Conference on Date and Application Security and Privacy	6
DIMVA	Conference on Detection of Intrusions and Malware and Vulnerability Assessment	2
DSN	International Conference on Dependable Systems and Networks	1
WOOT	USENIX Workshop on Offensive Technologies	2
ISC	Information Security Conference	1
SOUPS	ACM Symposium on Usable Privacy and Security	2
HOTSEC	SENIX Conference on Hot Topics in Security	1
SPSM	ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices	3
WISEC	ACM Conference on Security and Privacy in Wireless and Mobile Networks	4
MOST	Mobile Security Technologies	1
ESORICS	European Symposium on Research in Computer Security	4
BLACKHAT	Black Hat Conventions	3
OSDI	Symposium on Operating Systems Design and Implementation	3
SOSP	ACM Symposium on Operating Systems Principles	1
EUROSYS	European Conference on Computer Systems	2
MOBISYS	ACM International Conference on Mobile Computing Systems	2
ICSE	International Conference on Software Engineering	5
FSE	ACM SIGSOFT Symposium on the Foundations of Software Engineering	3
ASE	IEEE/ACM International Conference on Automated Software Engineering	1
WODA	International Workshop on Dynamic Analysis	1
PLDI	ACM SIGPLAN Conference on Programming Language Design and Implementation	1
OOPSLA	ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications	2
ESSOS	International Symposium on Engineering Secure Software and Systems	1
IFIPSEC	International Conference on Systems Security and Privacy Protection	1
FPS	International Symposium on Foundations and Practice of Security	1
CMS	International Conference on Communications and Multimedia Security	1
WISA	International Workshop on Information Security Applications	1
IMPS	Innovations in Mobile Privacy and Security	1
ESSOS	International Symposium on Engineering Secure Software and Systems	1
ACISP	Australasian Conference on Information Security and Privacy	1
WEBAPPS	USENIX Conference on Web Application Development	1
SIGMETRICS	ACM International Conference on Measurement and Modeling of Computer Systems	1
VEE	ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments	1
EUC	IEEE International Conference on Embedded and Ubiquitous Computing	1
ICEEI	International Conference on Electrical Engineering and Informatics	1
HICSS	Hawaii International Conference on System Science	1
SAC	ACM Symposium on Applied Computing	2
Others	Technical Reports, Workshop Presentations, Online Documents, etc.	69
Total		215

REFERENCES

Yousra Aafer, Wenliang Du, and Heng Yin. 2013. DroidAPIMiner: Mining API-level features for robust malware detection in android. In *Proceedings of the 9th International Conference on Security and Privacy in Communication Networks (SecureComm)*. Springer, Sydney, NSW, Australia, 163–182.

Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiaoyong Zhou, Wenliang Du, and Michael Grace. 2015. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. ACM Press, Denver, Colorado, 1248–1259.

Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*. ACM, 340–353.

Adobe Systems, Inc. 2016. PhoneGap. (Feb. 2016). http://phonegap.com.

Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, 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 2016 Annual Network and Distributed System Security Symposium (NDSS).* The Internet Society, San Diego, CA, 51:1–51:15.

Ahmed Al-Haiqi, Mahamod Ismail, and Rosdiadee Nordin. 2013. On the best sensor for keystrokes inference attack on android. *Procedia Technology* 8 (2013), 947–953.

Kevin Allix, Tegawendé François D. Assise Bissyande, Jacques Klein, and Yves Le Traon. 2014. *Machine Learning-Based Malware Detection for Android Applications: History Matters!* Technical Report. University of Luxembourg.

Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. 2015. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software* 32, 5 (Sept. 2015), 53–59.

Android Developers. 2016a. Android—AccessibilityService. (Feb. 2016). http://developer.android.com/reference/android/accessibilityservice/AccessibilityService.html.

Android Developers. 2016b. Android Security Overview. (Feb. 2016). https://source.android.com/security.

Android Developers. 2016c. WebView. (Feb. 2016). http://developer.android.com/reference/android/webkit/WebView.html.

Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. 2011. Cells: A virtual mobile smartphone architecture. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 173–187.

Appcelerator Inc. 2016. Appcelerator Titanium SDK. (Feb. 2016). http://www.appcelerator.com/titanium/titanium-sdk.

Axelle Apvrille. 2014. New Drive-By Download Android Malware. (Oct. 2014). http://blog.fortinet.com/xbrk post/new-drive-by-download-Android-malware.

Machiry Aravind, Tahiliani Rohan, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 18th European Software Engineering Conference (ESEC)/21st ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. ACM, 224–234.

Jeff Arnold and M. Frans Kaashoek. 2009. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th European Conference on Computer Systems (EuroSys)*. ACM, 187–198.

Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. 2014. Drebin: Efficient and explainable detection of android malware in your pocket. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS).* The Internet Society, San Diego, CA, 49:1–49:12.

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 *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 259–269.

Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. Pscout: Analyzing the android permission specification. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*. ACM Press, Raleigh, NC, 217–228.

Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering (ICSE).* IEEE Computer Society, Austin, TX, 426–436.

Adam J. Aviv, Benjamin Sapp, Matt Blaze, and Jonathan M. Smith. 2013. Practicality of accelerometer side channels on smartphones. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC)*. ACM, 41–50.

AVO. 2011. KillingInTheNameOf ashmem. (Jan. 2011). http://androidvulnerabilities.org/vulnerabilities/KillingInTheNameOf%5Fpsneuter%5Fashmem.

Ahmed Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS).* ACM Press, Scottsdale, Arizona, 90–102.

Michael Backes, Sven Bugiel, Sebastian Gerling, and Philipp von Styp-Rekowsky. Android security framework: Extensible multi-layered access control on android. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)*. ACM, 46–55.

Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. 2015. Boxify: Full-fledged app sandboxing for stock android. In *Proceedings of the 24th USENIX Security Symposium (Security)*. USENIX Association, 691–706.

Endre Bangerter, David Gullasch, and Stephan Krenn. 2011. Cache games: Bringing access-based cache attacks on AES to practice. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland)*. IEEE Computer Society, 490–505.

Anestis Bechtsoudis. 2015. Fuzzing Objects d'ART—Digging Into the New Android L Runtime Internals. (2015). https://census-labs.com/media/Fuzzing%5FObjects%5Fd%5FART%5Fhitbsecconf2015ams%5FWP.pdf.

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

Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. NJAS: Sandboxing unmodified applications in non-rooted devices running stock android. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*. ACM, 27–38.

Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. 2011. *XmAndroid: A New ANdroid Evolution to Mitigate Privilege Escalation Attacks*. Technical Report TR-2011-04. Technische Universität Darmstadt.

Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. 2012. Towards taming privilege-escalation attacks on android. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, CA, 19:1–19:18.

Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. 2013. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Proceedings of the 22th USENIX Security Symposium (Security)*. USENIX Association, Washington, DC, 131–146.

Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. 2015. EdgeMiner: Automatically detecting implicit control flow transitions through the android framework. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, CA, 8:1–8:15.

Gopinath K. N. Hemant Chaskar. 2009. All You Wanted to Know About WiFi Rogue Access Points. (2009). http://www.rogueap.com/rogue-ap-docs/RogueAP-FAQ.pdf.

Eric Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. 2014a. OAuth demystified for mobile application developers. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*. ACM Press, Scottsdale, Arizona, 892–903.

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 *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, CA, 28:1–28:19.

Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. 2014b. Peeking into your app without actually seeing it: UI state inference and novel android attacks. In *Proceedings of the 23rd USENIX Security Symposium (Security)*. USENIX Association, San Diego, CA, 1037–1052.

Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. 2005. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium (Security)*. USENIX Association, 12–26.

Eric Chien. 2011. *Motivations of Recent Android Malware*. Technical Report. Symantec Corporation.

Erika Chin and David Wagner. 2013. Bifocals: Analyzing webview vulnerabilities in android applications. In *Proceedings of the 14th International Workshop on Information Security Applications (WISA)*. Springer, 138–159.

Allen Choong. 2012. Rooting Android Manually. (March 2012). https://allencch.wordpress.com/2012/03/14/rooting-android-manually/.

Chromium Dev Community. 2012. Issue 166704: Security: Use a seccomp-bpf Sandbox on Android. (Dec. 2012). https://code.google.com/p/chromium/issues/detail?id=166704.

Lucian Constantin. 2016. Malvertising Attack Silently Infects Old Android Devices with Ransomware. (2016). http://www.itworld.com/article/3060191/malvertising-attack-silently-infects-old-android-devices-with-ransomware.html.

Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. 2010. CRePE: Context-related policy enforcement for android. In *Proceedings of the 13th Information Security Conference (ISC)*. Springer, 331–345.

Corelan Team. 2014. State of the ART: Exploring the New Android KitKat Runtime. (2014). https://www.corelan.be/index.php/2014/05/29/hitb2014ams-day-1-state-of-the-art-exploring-the-new-android-kitkat-runtime/.

Valerio Costamagna and Cong Zheng. 2016. ARTDroid: A virtual-method hooking framework on android ART runtime. In *Proceedings of the 2016 Innovations in Mobile Privacy and Security (IMPS)*. Springer, 24–32.

Jonathan Crussell, Clint Gibler, and Hao Chen. 2012. Attack of the clones: Detecting cloned applications on android markets. In *Proceedings of the 17th European Symposium on Research in Computer Security (ESORICS)*. Springer, 37–54.

Jonathan Crussell, Clint Gibler, and Hao Chen. 2013a. AnDarwin: Scalable detection of semantically similar android applications. In *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS).* Springer, Egham, UK, 182–199.

Jonathan Crussell, Clint Gibler, and Hao Chen. 2013b. Scalable semantics-based detection of similar android applications. In *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS).* Springer, Egham, UK, 182–199.

Andrew Cunningham. 2014. Android's Update Woes. (Aug. 2014). http://arstechnica.com/gadgets/2014/08/to-solve-androids-update-woes-google-should-look-to-the-pc/.

CyanogenMod Team. 2016. Cyanogenmod. (Feb. 2016). http://www.cyanogenmod.org.

Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2012. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS).* The Internet Society, San Diego, CA, 18:1–18:17.

Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Conference on Security Symposium.* USENIX Association, 401–416.

Guillaume Delugre. 2011. Reverse Engineering a Qualcomm Baseband. (2011). http://events.ccc.de/congress/2011/Fahrplan/attachments/2022%5F11-ccc-qcombbdbg.pdf.

Soteris Demetriou, Xiaoyong Zhou, Muhammad Naveed, Yeonjoon Lee, Kan Yuan, XiaoFeng Wang, and Carl A. Gunter. 2015. What's in your dongle and bank account? Mandatory and discretionary protection of android external resources. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS).* The Internet Society, San Diego, CA, 7:1–7:15.

Luke Deshotels. 2014. Inaudible sound as a covert channel in mobile devices. In *Proceedings of the 2014 USENIX Workshop on Offensive Technologies (WOOT)*. USENIX Association, 16:1–16:9.

Anthony Desnos. 2012. Android: Static analysis using similarity distance. In *Proceedings of the 45th Hawaii International Conference on System Science (HICSS)*. IEEE Computer Society, 5394–5403.

Anthony Desnos and Geoffroy Gueguen. 2012. New "Open Source" Step in Android Application Analysis. (Nov. 2012). https://androguard.googlecode.com/files/pacsec2012.pdf.

Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. QUIRE: Lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Conference on Security (SEC'11)* USENIX Association, 23:1–23:16.

Jason A. Donenfeld. 2012. Linux Local Privilege Escalation via SUID /proc/pid/mem Write. (Jan. 2012). https://git.zx2c4.com/CVE-2012-0056/about/.

Joshua Drake. 2015. Stagefright: Scary Code in the Heart of Android. (Aug. 2015).

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 *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI).* USENIX Association, Vancouver, Canada, 393–407.

Raoul Estourgie and Erik Poll. 2013. *Analysis of Android Authenticators*. B.S. thesis. Radboud Universiteit Nijmegen.

F-Secure. 2011a. Exploit Description Exploit:Android/GingerBreak. (April 2011). https://www.f-secure.com/v-descs/exploit%5Fandroid%5Fgingerbreak.shtml.

F-Secure. 2011b. Exploit Description Exploit:Android/Zergrush. (Oct. 2011). https://www.f-secure.com/v-descs/exploit%5Fandroid%5Fzergrush.shtml.

Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why eve and mallory love android: An analysis of android SSL (in)security. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS).* ACM Press, Raleigh, NC, 50–61.

Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011a. Android permission demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS).* ACM Press, Chicago, Illinois, 627–638.

Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdatta Akhawe, David Wagner, and others. 2012a. How to ask for permission. In *Proceedings of the 7th USENIX Conference on Hot Topics in Security (HotSec)*. USENIX Association, 7:1–7:6.

Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011b. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX Conference on Web Application Development (WebApps)*. USENIX Association, 7:1–7:12.

Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. 2012b. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the 8th ACM Symposium on Usable Privacy and Security (SOUPS)*. ACM, 3:1–3:12.

Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. 2011c. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium (Security)*. USENIX Association, 22:1–22:16.

Dennis Fisher. 2015. Google Fixes Sandbox Escape in Chrome. (May 2015). https://threatpost.com/google-fixes-sandbox-escape-in-chrome/112899/.

Jay Freeman. 2012. mempodroid Details. (Aug. 2012). https://github.com/saurik/mempodroid.

Wade Gasior and Li Yang. 2012. Exploring covert channel in android platform. In *2012 International Conference on Cyber Security (CyberSecurity)*. IEEE Computer Society, 173–177.

geohot. 2014. towelroot by geohot. (June 2014). https://towelroot.com/.

Enes Goktas, Elias Athanasopoulos, Herbert Bos, and Gerogios Portokalidis. 2014. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*. IEEE Computer Society, San Jose, CA, 575–589.

Google Inc. 2016c. ART and Dalvik. (Feb. 2016). https://source.android.com/devices/tech/dalvik.

Google Inc. 2016a. Brillo. (Feb. 2016). https://developers.google.com/brillo.

Google Inc. 2016b. Chrome Extension—Declare Permissions. (Feb. 2016). https://developer.chrome.com/extensions/declare%5Fpermissions.

Google Inc. 2016d. Codenames, Tags, and Build Numbers. (Feb. 2016). https://source.android.com/source/build-numbers.html.

Google Inc. 2016e. Put Android to work. (Feb. 2016). https://www.android.com/work.

Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. 2015. Information-flow analysis of android applications in DroidSafe. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, CA, 6:1–6:16.

Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM Press, Hyderabad, India, 1025–1035.

Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. 2012. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, CA, 20:1–20:15.

Lion Gu. 2014. *The Mobile Cybercriminal Underground Market in China*. Technical Report. Trend Micro.

Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. 2012. Juxtapp: A scalable system for detecting code reuse among android applications. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. Springer, 62–61.

Peng Hao, Gates Chris, Sarma Bhaskar, Li Ninghui, Qi Yuan, Rahul Potharaju, Nita-Rotaru Chrisina, and Molloy Ian. 2012. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*. ACM Press, Raleigh, NC, 241–252.

Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. 2014. ASM: A programmable interface for extending android security. In *Proceedings of the 23rd USENIX Security Symposium (Security)*. USENIX Association, San Diego, CA, 1005–1019.

Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2012. Librando: Transparent code randomization for just-in-time compilers. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*. ACM Press, Raleigh, NC, 993–1004.

Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. 2011. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*. ACM Press, Chicago, Illinois, 639–652.

HTC Corporation. 2016. HTCDev Unlock Bootloader. (Feb. 2016). http://www.htcdev.com/bootloader.

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 *Proceedings of the 36th International Conference on Software Engineering (ICSE).* ACM Press, Hyderabad, India, 1036–1046.

Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*. IEEE Computer Society, 191–205.

Intel Corporation. 2016. Intel Identity Protection Technology. (Feb. 2016). http://ipt.intel.com.

Yeongjin Jang, Chengyu Song, Simon P. Chung, Tielei Wang, and Wenke Lee. 2014. A11y attacks: Exploiting accessibility in operating systems. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS).* ACM Press, Scottsdale, Arizona, 103–115.

Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. 2012. Dr. Android and Mr. Hide: Fine-grained permissions in android applications. In *Proceedings of the 2nd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM).* ACM Press, Raleigh, NC, 3–14.

Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. 2013. Run-time enforcement of information-flow properties on android. In *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS).* Springer, Egham, UK, 775–792.

Michael Kassner. 2014. Droidpak: A Sneak Attack on Android Devices via PC Malware. (Feb. 2014). http://www.techrepublic.com/blog/it-security/droidpak-a-sneak-attack-on-android-devices-via-pc-malware/.

Eunice Kim. 2015. Creating Better User Experiences on Google Play. (March 2015). http://android-developers.blogspot.com/2015/03/creating-better-user-experiences-on.html.

Tam Kimberly, J. Khan Salahuddin, Fattori Aristide, and Cavallaro Lorenzo. 2015. CopperDroid: Automatic reconstruction of android malware behaviors. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS).* The Internet Society, San Diego, CA, 9:1–9:15.

Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2014. BareCloud: Bare-metal analysis-based evasive malware detection. In *Proceedings of the 23rd USENIX Security Symposium (Security).* USENIX Association, San Diego, CA, 287–301.

Krishan Kumar and Prabhpreet Kaur. 2015. Vulnerability detection of international mobile equipment identity number of smartphone and automated reporting of changed IMEI number. *International Journal of Computer Science and Mobile Computing* 4 (May 2015), 527–533.

Nate Kushman and Dina Katabi. 2010. Enabling configuration-independent automation by non-expert users. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI).* USENIX Association, Vancouver, Canada, 223–236.

Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code pointer integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 147–163.

Stephen Kyle, Hugh Leather, Björn Franke, Dave Butcher, and Stuart Monteith. 2015. Application of domain-aware binary fuzzing to aid android virtual machine testing. In *Proceedings of the 2015 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. ACM, 121–132.

Lingguang Lei, Yuewu Wang, Jian Zhou, Daren Zha, and Zhongwen Zhang. 2013. A threat to mobile cyber-physical systems: Sensor-based privacy theft attacks on android smartphones. In *Proceedings of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE Computer Society, 126–133.

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 *Proceedings of the 37th International Conference on Software Engineering (ICSE).* IEEE Computer Society, Austin, TX, 280–291.

Martina Lindorfer, Stamatis Volanis, Alessandro Sisto Matthias Neugschwandtner, Elias Athanasopoulos, Federico Maggi, Christian Platzer, Stefano Zanero, and Sotiris Ioannidis. 2014. AndRadar: Fast discovery of android applications in alternative markets. In *Proceedings of the 11th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. Springer, 51–71.

Benjamin Livshits and Jaeyeon Jung. 2013. Automatic mediation of privacy-sensitive resource access in smartphone applications. In *Proceedings of the 22th USENIX Security Symposium (Security).* USENIX Association, Washington, DC, 113–130.

Lookout, Inc. 2012a. Security Alert: Hacked Websites Serve Suspicious Android Apps (NotCompatible). (May 2012). https://blog.lookout.com/blog/2012/05/02/security-alert-hacked-websites-serve-suspicious-Android-apps-noncompatible.

Lookout, Inc. 2012b. *State of Mobile Security 2012*. Technical Report. Lookout, Inc.

Kangjie Lu, Zhichun Li, Vasileios Kemerlis, Zhenyu Wu, Long Lu, Cong Zheng, Zhiyun Qian, Wenke Lee, and Guofei Jiang. 2015. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS).* The Internet Society, San Diego, CA, 19:1–19:15.

Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS).* ACM Press, Raleigh, NC, 229–240.

Long Lu, Roberto Perdisci, and Wenke Lee. 2011. SURF: Detecting and measuring search poisoning. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS).* ACM Press, Chicago, Illinois, 467–476.

Adrian Ludwig. 2013. Android: Practical Security from the Ground Up. (Oct. 2013). https://docs.google.com/presentation/d/1YDYUrD22Xq12nKkhBfwoJBfw2Q-OReMr0BrDfHyfyPw.

Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. 2011. Attacks on WebView in the android system. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*. ACM, 343–352.

Tongbo Luo, Xing Jin, Ajai Ananthanarayanan, and Wenliang Du. 2012. Touchjacking attacks on web in android, iOS, and windows phone. In *Proceedings of the 5th International Symposium on Foundations and Practice of Security (FPS)*. Springer, 227–243.

Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. ACM, 599–609.

Claudio Marforio, Aurélien Francillon, Srdjan Capkun. 2011. *Application Collusion Attack on the Permission-Based Security Model and Its Implications for Modern Smartphone Systems*. Technical Report. ETH Zurich.

Claudio Marforio, Hubert Ritzdorf, A. Francillon, and Srdjan Capkun. 2012. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*. ACM, 51–60.

Charlie Miller. 2012. Exploring the NFC attack surface. (Aug. 2012).

Emiliano Miluzzo, Alexander Varshavsky, Suhrid Balakrishnan, and Romit Roy Choudhury. 2012. TapPrints: Your finger taps have fingerprints. In *Proceedings of the 10th ACM International Conference on Mobile Computing Systems (MobiSys)*. ACM, 323–336.

Bill Morrow. 2012. BYOD security challenges: Control and protect your most sensitive data. *Network Security* 2012, 12 (Dec. 2012), 5–8.

Adwait Pravin Nadkarni. 2012. Workflow Based Information Flow Control (IFC) in Modern Operating Systems. (2012).

Ryan Naraine. 2012. Android Drive-by Download Attack via Phishing SMS. (Feb. 2012). http://www.zdnet.com/blog/security/Android-drive-by-download-attack-via-phishing-sms/10422.

Mohammad Nauman, Sohail Khan, and Xinwen Zhang. 2010. Apex: Extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM, 328–332.

Muhammad Naveed, Xiaoyong Zhou, Soteris Demetriou, XiaoFeng Wang, and Carl A. Gunter. 2014. Inside job: Understanding and mitigating the threat of external device mis-bonding on android. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS).* The Internet Society, San Diego, CA, 15:1–15:14.

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 *Proceedings of the 22th USENIX Security Symposium (Security).* USENIX Association, Washington, DC, 543–558.

Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. 2009. Semantically rich application-centric security in android. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. IEEE Computer Society, 340–349.

Open Source Security, Inc. 2016. grsecurity features. (Feb. 2016). https://grsecurity.net/features.php.

OpenSignal Inc. 2015. Android Fragmentation Report. (Aug. 2015). http://opensignal.com/reports/2015/08/android-fragmentation.

Dan Page. 2005. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. (2005). http://eprint.iacr.org/2005/280.

Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards automating risk assessment of mobile applications. In *Proceedings of the 22th USENIX Security Symposium (Security)*. USENIX Association, Washington, DC, 527–542.

Adrienne Porter Felt Paul Pearce, Gabriel Nunez, and David Wagner. 2012. AdDroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM, 71–72.

Andre Pereir, Manuel Eduardo Correia, and Pedro Branda. 2014. USB connection vulnerabilities on android smartphones: Default and vendors' customizations. In *Proceedings of the 15th International Conference on Communications and Multimedia Security (CMS)*. Springer, 19–32.

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 *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, CA, 46:1–46:16.

Rahul Potharaju, Andrew Newell, Cristina Nita-Rotaru, and Xiangyu Zhang. 2012. Plagiarizing smartphone applications: Attack strategies and defense techniques. In *Proceedings of the 2012 International Symposium on Engineering Secure Software and Systems (ESSoS)*. Springer, 106–120.

Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin T. S. Chan. 2014. On tracking information flows through JNI in android applications. In *Proceedings of the 44th International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, 180–191.

Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. AutoCog: Measuring the description-to-permission fidelity in android applications. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*. ACM Press, Scottsdale, Arizona, 1354–1365.

Moheeb Abu Rajab, Lucas Ballard, Noé Lutz, Panayiotis Mavrommatis, and Niels Provos. 2013. CAMP: Content-agnostic malware protection. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, CA, 24:1–24:17.

Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A machine-learning approach for classifying and categorizing android sources and sinks. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, CA, 42:1–42:15.

Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, CA, 55:1–55:15.

Vaibhav Rastogi, Yan Chen, and William Enck. 2013. AppsPlayground: Automatic security analysis of smartphone applications. In *Proceedings of the ACM Conference on Data and Applications Security and Privacy (CODASPY)*. ACM Press, San Antonio, 209–220.

Paul Ratazzi, Yousra Aafer, Amit Ahlawat, Hao Hao, Yifei Wang, and Wenliang Du. 2014. A systematic security evaluation of android's multi-user framework. In *Proceedings of the Mobile Security Technologies (MoST)*. IEEE Computer Society, 9:1–9:10.

Simon Rockman. 2014. Google Nest, ARM, Samsung Pull Out Thread to Strangle ZigBee. (July 2014). http://www.theregister.co.uk/2014/07/15/google%5Fnest%5Fthread%5Fprotocol/.

Franziska Roesner and Tadayoshi Kohno. 2013. Securing embedded user interfaces: Android and beyond. In *Proceedings of the 22th USENIX Security Symposium (Security)*. USENIX Association, Washington, DC, 97–112.

Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan. 2012. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*. IEEE Computer Society, San Francisco, CA, 224–238.

Sankardas Roy, Jordan DeLoach, Yuping Li, Nic Herndon, Doina Caragea, and Xinming Ou. Experimental study with real-world data for android app security analysis using machine learning. ACM, 81–90.

Paul Sabanal. 2015. Hiding Behind ART. (Aug. 2015).

Samsung Electronics. 2014. White Paper: An Overview of Samsung KNOX 2.0. (March 2014). http://www.samsung.com/ca/business-images/resource/white-paper/2014/03/Samsung%5FKNOX%5Ftech%5Fwhitepaper%5FFinal%5F140220-0.pdf.

Golam Sarwar, Olivier Mehani, Roksana Boreli, and Dali Kaafar. 2013. *On the Effectiveness of Dynamic Taint Analysis for Protecting Against Private Information Leaks on Android-based Devices*. Technical Report. NICTA.

Anand Saswat, Naik Mayur, Jean Harrold Mary, and Yang Hongseok. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. ACM, 59:1–59:15.

Roman Schlegel, Kehuan Zhang, Xiaoyong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. 2011. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 1:1–1:17.

Daniel Schreckling, Joachim Posegga, and Daniel Hausknecht. 2012. Constroid: Data-centric access control for android. In *Proceedings of the 27th ACM Symposium on Applied Computing (SAC)*. ACM, 1478–1485.

Sebastian. 2011. Zimperlich Sources. (Feb. 2011). http://c-skills.blogspot.com/2011/02/zimperlich-sources.html.

Jaeback Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, and Insik Shin. 2016. FlexDroid: Enforcing in-app privilege separation in android. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, CA, 53:1–53:15.

Yuru Shao, Xiapu Luo, and Chenxiong Qian. 2014a. RootGuard: Protecting rooted android phones. *Computer* 47 (June 2014), 32–40.

Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. 2014b. Towards a scalable resource-driven approach for detecting repackaged android applications. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*. ACM, 56–65.

Roy Choudhary Shauvik, Gorla Alessandra, and Alessandro (Alex) Orso. 2015. Automated test input generation for android: Are we there yet? In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 429–440.

Shashi Shekhar, Michael Dietz, and Dan S. Wallach. AdSplit: Separating smartphone advertising from applications. In *Proceedings of the 21st USENIX Conference on Security Symposium (Security'12)* USENIX Association, 553–567.

Dongwan Shin, Huiping Yao, and Une Rosi. 2013. Supporting visual security cues for webview-based android apps. In *Proceedings of the 28th ACM Symposium on Applied Computing (SAC)*. ACM, 1867–1876.

Hao Shuai, Liu Bin, Nath Suman, G. J. Halfond William, and Ramesh Govindan. 2014. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th ACM International Conference on Mobile Computing Systems (MobiSys)*. ACM, 204–217.

Silent Circle. 2016. Blackphone 2 and Silent OS. (Feb. 2016). https://www.silentcircle.com.

David Silver, Suman Jana, Dan Boneh, Eric Chen, and Collin Jackson. 2014. Password managers: Attacks and defenses. In *Proceedings of the 23rd USENIX Security Symposium (Security)*. USENIX Association, San Diego, CA, 449–464.

Stephen Smalley and Robert Craig. 2013. Security enhanced (SE) android: Bringing flexible MAC to android. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, CA, 9:1–9:18.

Carlos A. Soto. 2005. A Menu of Bluetooth Attacks. (July 2005). http://gcn.com/articles/2005/07/20/a-menu-of-bluetooth-attacks.aspx.

Mengtao Sun and Gang Tan. 2014. NativeGuard: Protecting android applications from third-party native libraries. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. ACM Press, Oxford, UK, 165–176.

Xin Sun, Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie. 2014. Detecting code reuse in android applications using component-based control flow graph. In *Proceedings of the 29th International Conference on Systems Security and Privacy Protection (IFIPSEC)*. Springer, 142–155.

SUSE. 2016. Live Kernel Patching with kGraft. (Feb. 2016). https://www.suse.com/promo/kgraft.html.

Vanja Svajcer. 2014. *Sophos Mobile Security Threat Report 2014*. Technical Report. Sophos, Ltd.

Azim Tanzirul and Neamtiu Iulian. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 24th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, Indianapolis, IN, 641–660.

Chengkai Tao. 2014. *Android App Update Flaw Affects China-Based Users*. Technical Report. Trendmicro.

Root Genius Team. 2016. Root Genius. (Feb. 2016). http://www.shuame.com/en/root.

The Apache Software Foundation. 2016. Apache Cordova. (Feb. 2016). http://cordova.apache.org.

thesnkchrmr. 2011. RageAgainstTheCage. (March 2011). https://thesnkchrmr.wordpress.com/2011/03/24/rageagainstthecage/.

Cody Toombs. 2014. [Lollipop Feature Spotlight] WebView Is Now Unbundled From Android And Free To Auto-Update From Google Play. (Oct. 2014). http://www.androidpolice.com/2014/10/19/lollipop-feature-spotlight-webview-now-unbundled-android-free-auto-update-google-play.

Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology* 23, 1 (2010), 37–71.

Ashee Vance. 2013. Behind the'Internet of Things' Is Android and It's Everywhere. (2013). http://www.businessweek.com/articles/2013-05-29/behind-the-internet-of-things-is-Android-and-its-everywhere.

Timothy Vidas and Nicolas Christin. 2013. Sweetening android lemon markets: Measuring and combating malware in application marketplaces. In *Proceedings of the ACM Conference on Data and Applications Security and Privacy (CODASPY)*. ACM Press, San Antonio, TX, 197–208.

Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A measurement study of google play. In *Proceedings of the 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. ACM, 221–233.

VirusTotal Team. 2012. VirusTotal. (Sept. 2012). https://www.virustotal.com/en/documentation/.

vuldb.com. 2013. Google Android 4.0 debug mode /data/local.prop privilege escalation. (June 2013). https://vuldb.com/?id.9059.

Ruowen Wang, William Enck, Douglas Reeves, Xinwen Zhang, Peng Ning, Dingbang Xu, Wu Zhou, and Ahmed M. Azab. 2014a. EASEAndroid: Automatic policy analysis and refinement for security enhanced android via large-scale semi-supervised learning. In *Proceedings of the 23rd USENIX Security Symposium (Security)*. USENIX Association, San Diego, CA, 351–366.

Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. 2014b. Compac: Enforce component-level access control in android. In *Proceedings of the ACM Conference on Data and Applications Security and Privacy (CODASPY)*. ACM Press, San Antonio, TX, 25–36.

Takuya Watanabe, Mitsuaki Akiyama, Tetsuya Sakai, and Tatsuya Mori. 2015. Understanding the inconsistencies between text descriptions and the use of privacy-sensitive resources of mobile apps. In *Proceedings of the 11th ACM Symposium on Usable Privacy and Security (SOUPS)*. USENIX Association, 241–255.

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. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*. ACM Press, Scottsdale, Arizona, 1329–1341.

Yang Wei, Xiao Xusheng, Andow Benjamin, Li Sihan, Xie Tao, and Enck William. 2015. AppContext: Differentiating malicious and benign mobile app behaviors using context. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Austin, TX, 303–313.

Ralf-Philipp Weinmann. 2012. Baseband attacks: Remote exploitation of memory corruptions in cellular protocol stacks. In *Proceedings of the 2012 USENIX Workshop on Offensive Technologies (WOOT)*. USENIX Association, 12–21.

Nathan Willis. 2013. Tizen Content Scanning and App Obfuscation. (June 2013). http://lwn.net/Articles/553676.

Michelle Y. Wong and David Lie. 2016. IntelliDroid: A targeted input generator for the dynamic analysis of android malware In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, CA, 54:1–54:15.

Choi Wontae, Necula George, and Sen Koushik. 2013. Guided GUI testing of android apps with minimal restart and approximate learning. In *Proceedings of the 24th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, Indianapolis, IN, 623–640.

Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. 2013. The impact of vendor customizations on android security. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*. ACM Press, Berlin, Germany, 623–634.

Zhen Xie and Sencun Zhu. 2015. AppWatcher: Unveiling the underground market of trading mobile app reviews. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. ACM, 10:1–10:11.

Luyi Xing, Xiaorui Pan, Rui Wang, Kan Yuan, and XiaoFeng Wang. 2014. Upgrading your android, elevating my malware: Privilege escalation through mobile OS updating. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*. IEEE Computer Society, San Jose, CA, 393–408.

Nan Xu, Fan Zhang, Yisha Luo, Weijia Jia, Dong Xuan, and Jin Teng. 2009. Stealthy video capturer: A new video-based spyware in 3G smartphones. In *Proceedings of the 2nd ACM Conference on Wireless Network Security (WiSec'09)*. ACM, 69–78.

Rubin Xu, Hassen Saïdi, and Ross Anderson. 2012. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX Security Symposium (Security)*. USENIX Association, Bellevue, WA, 539–552.

Yuanzhong Xu and Emmett Witchel. 2015. Maxoid: Transparently confining mobile applications with custom views of state. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*. ACM, 26:1–26:16.

Lok Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Security Symposium (Security)*. USENIX Association, Bellevue, WA, 569–584.

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 *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*. ACM Press, Berlin, Germany, 1043–1054.

Jing Yu and Toshihiro Yamauchi. 2013. Access control to prevent attacks exploiting vulnerabilities of webview in android OS. In *Proceedings of the 11th IEEE International Conference on Embedded and Ubiquitous Computing*. IEEE Computer Society, 1628–1633.

Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. 2014a. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. ACM Press, Oxford, UK, 25–36.

Hang Zhang, Dongdong She, and Zhiyun Qian. 2015. Android root and its providers: A double-edged sword. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. ACM Press, Denver, Colorado, 1093–1104.

Mu Zhang and Heng Yin. 2014. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM Press, 259–270.

Yingqian Zhang, Michael K. Reiter, Ari Juels, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*. ACM Press, Raleigh, NC, 305–316.

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 *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*. ACM Press, Berlin, Germany, 611–622.

Zhongwen Zhang, Yuewu Wang, Jiwu Jing, Qiongxiao Wang, and Lingguang Lei. 2014b. Once root always a threat: Analyzing the security threats of android permission system. In *Proceedings of the 19th Australasian Conference on Information Security and Privacy (ACISP)*. Springer, 354–369.

Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014c. Semantics-aware android malware classification using weighted contextual API dependency graphs. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*. ACM Press, Scottsdale, Arizona, 1105–1116.

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. In *Proceedings of the 2nd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*. The Internet Society, Raleigh, NC, 93–104.

Wu Zhou, Zhi Wang, Yajin Zhou, and Xuxian Jiang. 2014b. DIVILAR: Diversifying intermediate language for anti-repackaging on android platform. In *Proceedings of the ACM Conference on Data and Applications Security and Privacy (CODASPY)*. ACM Press, San Antonio, TX, 199–210.

Wu Zhou, Xinwen Zhang, and Xuxian Jiang. 2013a. AppInk: Watermarking android apps for repackaging deterrence. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM Press, Hangzhou, China, 1–12.

Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. 2013b. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the ACM Conference on Data and Applications Security and Privacy (CODASPY)*. ACM Press, San Antonio, TX, 185–196.

Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the ACM Conference on Data and Applications Security and Privacy (CODASPY)*. ACM, 317–326.

Xiaoyong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A. Gunter, and Klara Nahrstedt. 2013. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*. ACM Press, Berlin, Germany, 1017–1028.

Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. 2014a. The peril of fragmentation: Security hazards in android device driver customizations. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*. IEEE Computer Society, San Jose, CA, 409–423.

Yajin Zhou and Xuxian Jiang. 2012. Dissecting android malware: Characterization and evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*. IEEE Computer Society, San Francisco, CA, 95–109.

Received April 2015; revised May 2016; accepted June 2016