# Securing Android: A Survey, Taxonomy, and Challenges

SUFATRIO, DARELL J. J. TAN, TONG-WEI CHUA,
and VRIZLYNN L. L. THING, Institute for Infocomm Research, Singapore

Recent years have seen a global adoption of smart mobile devices, particularly those based on Android. However, Android's widespread adoption is marred with increasingly rampant malware threats. This article gives a survey and taxonomy of existing works that secure Android devices. Based on Android app deployment stages, the taxonomy enables us to analyze schemes that share similar objective and approach and to inspect their key differences. Additionally, this article highlights the limitations of existing works and current challenges. It thus distills the state of the art in Android security research and identifies potential research directions for safeguarding billions (and keep counting) of Android-run devices.

Categories and Subject Descriptors: D.4.6 [**Security and Protection**]: Invasive Software (e.g., viruses, worms, Trojan horses); K.6.5 [**Security and Protection**]: Invasive Software (e.g., viruses, worms, Trojan horses)

General Terms: Security

Additional Key Words and Phrases: Android, Android security, mobile security, malware mitigation

**ACM Reference Format:**
Sufatrio, Darell J. J. Tan, Tong-Wei Chua, and Vrizlynn L. L. Thing. 2015. Securing Android: A survey, taxonomy, and challenges. ACM Comput. Surv. 47, 4, Article 58 (May 2015), 45 pages.
DOI: http://dx.doi.org/10.1145/2733306



## 1. INTRODUCTION

Recent years have seen a global widespread adoption of smart mobile devices (i.e., smartphones and tablets). This trend is expected to continue, with the total worldwide smartphone subscriptions projected to grow from 1.9 billion in 2013 to 5.6 billion in 2019 [Ericsson 2013]. Android, a mobile OS from Google, currently holds the biggest market share in the global smartphone market [Wikipedia 2015]. It was estimated that there were 1.9 billion Android devices in use in 2014 [Gartner, Inc. 2014].

The proliferation of Android's user base, unfortunately, has also made the devices become prominent targets of malware attacks [Cisco 2014; U.S. Dept. of Homeland Security 2013]. To address this serious concern, many security researchers have actively proposed numerous schemes to safeguard Android devices against security breaches. Although Android security is a relatively new field, it is a very active and attractive one. In fact, this field has produced a large number of published works in the past several years. They propose numerous schemes with their different objectives, techniques, and degree of required modifications on Android middleware and the underlying Linux kernel, as well as claims, limitations, and varying results.

Authors' addresses: Sufatrio, D. J. J. Tan, T.-W. Chua, and V. L. L. Thing, Institute for Infocomm Research, 1 Fusionopolis Way, #21-01 Connexis, Singapore 138632; emails: {sufatrio, jjdtan, twchua, vriz}@i2r.a-star.edu.sg.

DOI: http://dx.doi.org/10.1145/2733306

This article aims to distill the latest landscape in Android security field by surveying, classifying, characterizing, and fairly contrasting existing works. Given the volume of previous works done in the field, this article certainly cannot cover all entries. Therefore, we choose to review major works from 2009 until toward the end of 2014, particularly those appearing in high-tier security conferences/journals, or ones with novel and significant contributions. Our goal is to systematize these works by means of a taxonomy, which makes clear how the whole body of work in Android security all fits together. We also aim to point out, in each taxonomy leaf node, the contributions of relevant existing works, remaining challenges, and promising research opportunities.

As will be elaborated more in Section 3, we classify the existing works into a taxonomy based on five deployment stages of Android applications (henceforth called *apps*), starting from their development, then their availability on app market(s), and subsequently their installation, execution, and possible setting modification on a user's device. Using this stage-based approach, the taxonomy enables us to easily group related prior works and analyze them together, because all schemes under a single taxonomy node address a common problem. In addition, they face the same challenges since they operate under the same constraints found in a particular app deployment stage. As a result, we can more easily pinpoint each proposed scheme's unique differentiator(s), strengths, and limitations relative to one another.

The breadth, depth, and timeliness of our survey and analysis coverage here, in our view, have not been provided by prior survey works (see also Section 2.3). By comprehensively covering major works to the current date, we thus position this article as the latest point of reference for researchers who are new to the field and experts alike. We also hope that it can help spur many upcoming solutions, which will bring forth novelties unto existing schemes and further safeguard a multitude of Android devices.

The remainder of this article is organized as follows. Section 2 first gives a brief background on Android security, summarizes its weaknesses, and surveys prior works that review Android security mechanisms and malware behavior. Section 3 presents our taxonomy. Sections 4 through 8 survey and comparatively analyze existing schemes, which are classified under the first five main categories of the taxonomy. Section 9 summarizes the remaining challenges and discusses promising directions for future research. Section 10 concludes this article.

## 2. BACKGROUND: ANDROID SECURITY, WEAKNESSES, AND EXISTING SURVEY WORKS

Before presenting our taxonomy in the next section, we first summarize in this section important notions on the Android security model and enumerate its weaknesses. These weaknesses are important, and we will refer to them later as the root causes behind the problems addressed by the proposed existing schemes. Subsequently, we review prior works that survey proposed security measures and malware behavior.

### 2.1. Android and Its Security Measures

Android is an open-source, Linux-based mobile OS from the Open Handset Alliance, which is led by Google. Android apps are written in Java and compiled to Dalvik bytecode (.dex), which is a bytecode format designed for Android. In addition to Java code, an app may contain native libraries, which are invoked from the Java code through the Java Native Interface (JNI). All files belonging to an app are packaged and then signed as a single APK file. To make app distribution easy to mobile users, app markets host third-party apps that can be downloaded into a device. Besides Google Play (formerly known as Android Market) as the official Android app market,

a number of alternative markets are available. Once installed on a device, an app runs as an instance of a Dalvik virtual machine (DVM).[1]

An Android app internally consists of multiple app components. There are four different types of app components, namely activity, service, broadcast receiver, and content provider. Intercomponent communication (ICC) is performed using *intent*, which is a messaging object that contains the destination component's address or action string, and possibly data. Besides facilitating unicast-based ICC between two components, an intent is also used to deliver a broadcast to multiple interested broadcast receivers. The Android system itself delivers various broadcasts for system events, such as upon completion of system boot-up. Unlike regular Java programs that have a single entry point, Android apps can have multiple entry points. Android app developers write their code by overriding the lifecycle methods of app components. The Android framework interacts with different app components independently and calls a component's lifecycle methods based on the app execution environment.

Android OS deploys various security measures. Two main measures are app sandboxing and the Android permission model. The former provides app isolation and containment by taking advantage of Linux access control and process protection mechanisms. The latter restricts an app's capability by regulating sensitive API calls that access Android-protected resources. Other deployed security measures include app signing to verify that different apps come from the same developer, as well as app component encapsulation that restricts access to a component. A widely cited paper by Enck et al. [2009b] gives an overview of Android and its security aspects. A review by Shabtai et al. [2010b] explains the aforementioned security measures. Six [2011], Misra and Dubey [2013], and Drake et al. [2014] also cover various security aspects of Android.

### 2.2. Security Weaknesses of Android

We summarize the limitations, which constitute the weaknesses, of Android security model and its app-market ecosystem as follows:

$W_1$: Android adopts an open market model with less restrictive app installation:
(1) There exist numerous alternative app markets, which impose no or limited app vetting process [Zhou et al. 2012b].
(2) There is little background check on app issuers [Oberheide and Miller 2012].
(3) There exist avenues to installing apps without involving app markets, such as using the `adb` tool from a connected computer. Although this gives Android users more freedom to install apps obtained from nonmarket sources, it represents an additional malware entry point, particularly to less security-aware users.

As a result, there is a higher probability for Android users to inadvertently install malicious apps compared to those on other walled-garden–based platforms.

$W_2$: Android employs an install-time and rather coarse-grained permission model:
(1) The install-time permission is on an all-or-nothing basis: a user must grant *all* permissions requested by an app or the app's installation will not proceed.
(2) The consequences of a set of requested permissions may not be fully understood by Android users, who tend to simply approve the permissions [Felt et al. 2012].
(3) App developers tend to bloat their requested permissions [Felt et al. 2011a].
(4) Android permissions may not be sufficiently fine grained. For example, it cannot enforce domain-based Internet access or partial selective access to sensitive resources [Jeon et al. 2012].

[1]Android version 4.4. introduces a new virtual machine called *ART*. ART enhances DVM, among other things, by means of ahead-of-time compilation and improved garbage collection.

(5) There is a lack of runtime permission revocation,[2] control (e.g., substitution of accessed private information [Zhou et al. 2011; Hornyack et al. 2011]), and monitoring/auditing tools.

Hence, once the user installs an app on his or her device, the app can run and use, or misuse, all of its granted permissions. There are no user-accessible mechanisms to limit or monitor any uses of the permissions.

$W_3$: Android apps are relatively easier to reverse-engineer compared to native apps in a desktop environment, such as Windows and Unix executables, because hardware-independent Dalvik bytecode files retain a great deal of information of the original Java sources [Nolan 2012]. In addition, UI layouts and string literals of an app are typically stored as separate resource files in XML format. Thus, Android apps can be subject to app repackaging, including that for malware injection [Zhou et al. 2012a; Hanna et al. 2012].

$W_4$: here is a lack of separation or isolation mechanism for third-party libraries, such as advertisement and analytics (A&A) libraries, that are contained within apps [Pearce et al. 2012]. Therefore, third-party libraries cannot be prevented from abusing the granted permissions of their host apps. Conversely, ill-behaving host apps may tamper with the libraries, such as by performing improper ad display or click fraud.

$W_5$: Android lacks a configurable, runtime ICC control for the following purposes:

(1) To prevent an app from accessing any open interfaces of another app, despite the former having obtained the required permissions at its install time [Chin et al. 2011; Felt et al. 2011c]. Combined with weakness $W_6$ that follows, this lack of runtime interapp access control can lead to data leakage and confused deputy problems discussed in Section 5.4 and 7.3.1, respectively.

(2) To prevent an app from intercepting an intent broadcast, and possibly stopping its propagation afterward [Chin et al. 2011]. By intercepting system-event broadcasts, a malicious app is able to stealthily intercept important system events that contain sensitive information, such as an incoming call or SMS.

(3) To isolate apps and prevent them from communicating via ICC and other shared channels [Bugiel et al. 2011a, 2012]. The presently unrestricted ICC among apps in Android can be exploited by colluding apps as elaborated later in Section 7.3.2.

$W_6$: App developers could be unfamiliar with the subtle aspects of Android ICC [Enck et al. 2009b; Chin et al. 2011; Zhou and Jiang 2013], which may lead to unintentional exposure of apps' private interfaces and data.

$W_7$: An app can invoke native code through JNI. This may cause security issues due to widely known memory corruption bugs in low-level languages (e.g., C or C++) [Sun and Tan 2014]. Similar to $W_4$, third-party native libraries are able to abuse the granted permissions of their host app. In addition, native code may be intentionally employed by malware writers to evade Android-level analysis techniques and monitoring tools.

$W_8$: An Android device has several identifiers that can be used as a unique device ID, such as IMEI, hardware serial number, or Android system ID [Bray 2011]. Since Android devices are prone to private information leakage (see Section 7.1), if this device ID is also leaked, external parties can track the user easily.

Furthermore, there exist potential security weaknesses outside of the Android middleware that could compromise the security of an Android device, namely

[2]Android version 4.3 has a hidden feature known as App Ops. It lets the user selectively disable app permissions that have been granted at install time. Ultimately, Google removed it on Android version 4.4.2, citing that this experimental feature could cause compatibility issue with existing apps.

$W_9$: Potential security weaknesses or vulnerabilities at the Linux layer, such as Linux kernel and its native libraries [Loscocco and Smalley 2001; Zhou et al. 2013c].
$W_{10}$: Potential vulnerabilities associated with device manufacturers' customization and preinstalled apps [Wu et al. 2013; Grace et al. 2012a; Zhou et al. 2014].

### 2.3. Existing Survey of Android Security and Malware Behavior

This section looks at prior works that survey proposed security enhancements to Android or analyze Android malware behavior. On works that also propose taxonomies of Android security enhancements, we highlight their differences with ours.

Shabtai et al. [2010b] give a taxonomy of threats to the Android framework and perform a qualitative risk assessment on the threats. Vidas et al. [2011] survey Android's attack vectors and propose their taxonomy of attacks.

Enck [2011] gives a very good survey of various research works in smartphone security up to 2011, of which most are on Android. The work classifies existing solutions into two main categories, namely protection systems and application analysis. These two categories are further divided into several subcategories, in which observations and limitations are then discussed. The categorization in the work, however, seems not to be specifically geared toward system comparison. The defined subcategories at the same level are based on either the techniques employed (e.g., rule-driven policy approach), the approaches taken (e.g., high-level policy approach), or security objectives targeted (e.g., mocking sensitive information). For instance, IPC Inspection [Felt et al. 2011c], which prevents confused deputy problem, falls under the high-level policy approach. Meanwhile, XManDroid [Bugiel et al. 2011a], which prevents collusion attacks, is grouped under the rule-driven policy approach. The connection between the two systems in preventing privilege escalation attacks therefore cannot be easily seen. We view that our taxonomy groups existing systems better, thus providing clearer connections and easier comparisons among them. Additionally, we update Enck [2011] by covering many later works between 2011 and 2014 in the fast-growing Android security field.

In their comprehensive survey work, Zhou and Jiang [2012] characterized 1,260 malware samples belonging to 49 different families, which appeared between August 2010 and October 2011. Based on their experimentation with four mobile antivirus systems, they concluded that the systems were lagging behind with detection rates between 20.2% and 79.6%. In November 2012, Jiang [2012] evaluated Google's app verification service, which was introduced in Android 4.2 to check for the installations of potentially harmful apps. Using the same malware samples gathered by Zhou and Jiang [2012], Jiang found that the verification service gave only a low detection rate of 15.32%. In addition, there exists concern over low resistance levels of antivirus solutions against transformation attacks as reported by Rastogi et al. [2013b] and Zheng et al. [2012].

Schlegel et al. [2011] thoroughly analyze Soundcomber, a sophisticated malware with limited permissions that is able to leak sensitive high-value data based on its audible surroundings. Felt et al. [2011b] analyze the incentives behind 46 malware samples on three mobile platforms, of which 18 samples are on Android. An additional description of Android malware behavior can also be found in Jiang and Zhou [2013]. The Android Malware Genome Project (http://www.malgenomeproject.org) and Contagio Mobile (http://contagiominidump.blogspot.com) are two widely used Android malware repositories, which are made freely available to the security community.

## 3. TAXONOMY OF EXISTING SECURITY SCHEMES ON ANDROID

This section presents our taxonomy of existing security solutions on Android. Although there are many ways of classifying existing works, our taxonomy is geared toward grouping together related works that share common objectives and characteristics to yield clear category formation and easier comparative analysis.

In examining a large number of existing works, we found that it is more relevant to compare the works based on their common security goal. Deeper analysis further reveals that a common security goal is aimed at addressing a particular problem under the same operative constraints faced in a specific deployment stage of an app. Hence, we develop our taxonomy by structuring the related body of work around an Android app's deployment stages—that is, the stages in the lifecycle of an app. We identify the following five app deployment stages in the Android ecosystem:

(1) *App development*: Since apps can be reverse-engineered and subsequently be tampered or repackaged, app developers therefore may want to harden their apps prior to releasing them for *tampering/repackaging prevention or deterrence*.
(2) *App availability on app markets*: From a user's viewpoint, apps that are hosted on app markets are untrusted. In other words, they are possibly malicious or are benign but could be vulnerable to attacks when they run on his or her device. The corresponding security goal at this stage is thus *untrusted app or market analysis*.
(3) *App installation on a device*: When installing an app, a user may want to ensure that the app complies to a locally enforced security policy. The corresponding security goal here is therefore *install-time app checking*.
(4) *App execution on a device*: Considering that the standard Android security mechanisms are deemed insufficient to prevent attacks [Shabtai et al. 2010b; Vidas et al. 2011; Zhou and Jiang 2012], there is a compelling need to perform *continuous runtime monitoring* on apps to enforce stronger policy or detect malicious behavior.
(5) *App security setting modification on a device*: At times, the users may significantly modify security settings in which an app operates on a device. Alternatively, installed apps may get updated. Hence, *rechecking of installed apps* is necessary.

We make the five preceding security goals as the five main categories at the first level of our taxonomy tree, which is shown in Figure 1. Works in categories 2 and 4 constitute most of the surveyed works, highlighting the two focus areas of current research efforts in Android security, namely app analysis and runtime-based platform protection. We further classify these works based on their more specific goals, and subsequently based on either the techniques employed, levels of software stack on which they operate, or subgoals. Table I lists the surveyed solutions and characterizes their main properties.

Sections 4 through 8 survey and analyze existing works in the five main categories, respectively. Throughout these category-area discussions, several category-level comparison tables are given to contrast similar works in more detail.

## 4. APP-HARDENING SYSTEMS (CATEGORY 1)

As stated by weakness $W_3$ defined in Section 2.2, Dalvik bytecode is vulnerable to reverse engineering. As a result, software tampering and software piracy (i.e., app repackaging) represent serious threats to Android [Zhou et al. 2012a]. Consequently, protecting released apps by means of watermarking, tamper proofing, and obfuscation [Collberg and Thomborson 2002] is of high interest to Android app developers.

To make reverse engineering harder, Android SDK comes with a code obfuscation tool called ProGuard.[3] Commercial code obfuscation solutions are also available in the market, which implement various obfuscating transformations [Collberg et al. 1997].

To deter app repackaging, Zhou et al. [2013a] propose AppInk, a graph-based dynamic watermarking scheme for Android apps. AppInk takes as inputs the source code of an app together with a watermark value, such as a number or string. AppInk then generates a new watermark-embedded app and its Robotium-based companion app.

[3]http://developer.android.com/tools/help/proguard.html.

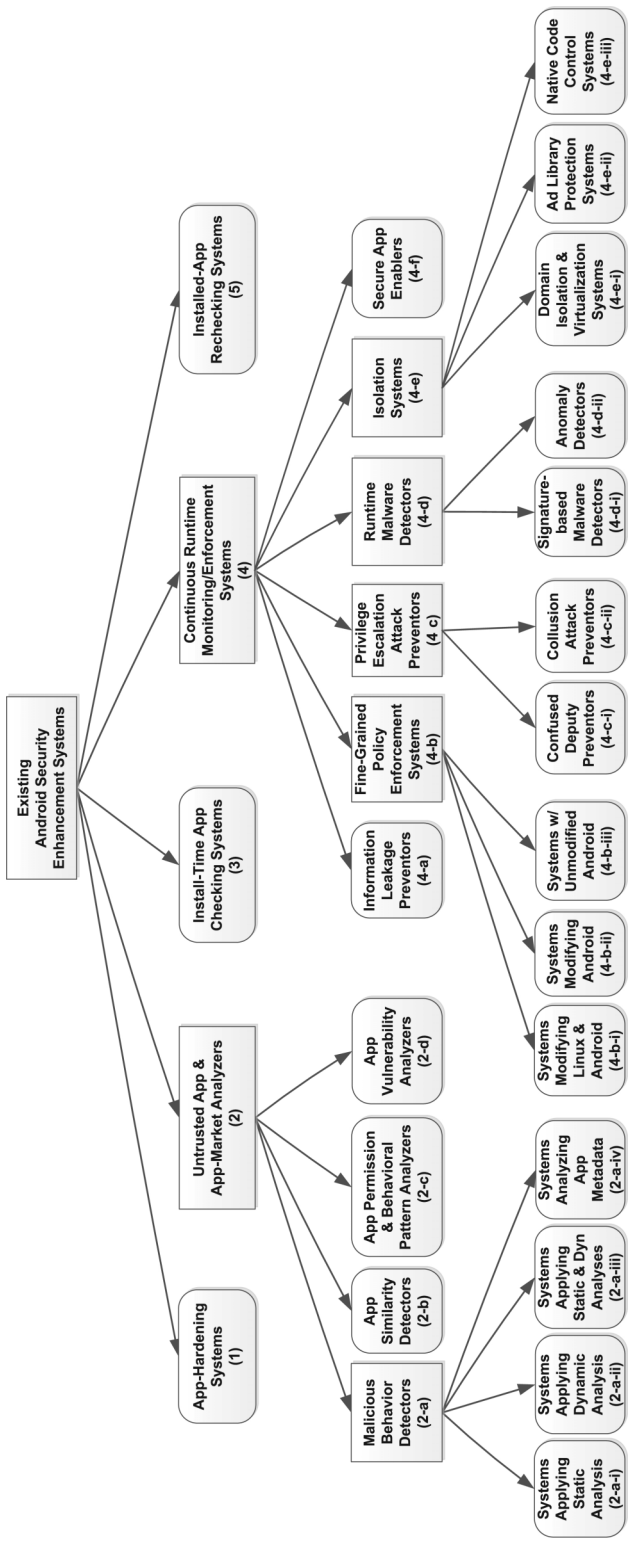


Fig. 1. Proposed taxonomy of existing security solutions on Android.

Table I. List of the Surveyed Systems and Their Main Properties

| System | Category | Technique(s) Used | Required Modifications | Rooting Needed? | Off-Device Execution | Other Tools Utilized | Publicly Available? |
|---|---|---|---|---|---|---|---|
| ***App-Hardening Systems (Category 1):*** | | | | | | | |
| AppInk [Zhou et al. 2013a] | 1 | Dynamic watermarking | App | N.A. | PC (rewriting & verification) | Robotium, ANTLR | No |
| ***Untrusted App/App-Market Analysis (Category 2):*** | | | | | | | |
| RiskRanker [Grace et al. 2012b] | 2-a-i | Static analysis | N.A. | N.A. | PC (analysis) | | No |
| SCanDroid [Fuchs et al. 2009] | 2-a-i | Static analysis | N.A. | N.A. | PC (analysis) | WALA | Source code |
| FlowDroid [Arzt et al. 2014] | 2-a-i | Static analysis | N.A. | N.A. | PC (analysis) | Dexpler, Soot, Spark, Heros | Source code |
| Amandroid [Wei et al. 2014] | 2-a-i | Static analysis | N.A. | N.A. | PC (analysis) | dexdump | Source code |
| AppsPlayground [Rastogi et al. 2013a] | 2-a-ii | Dynamic analysis | Android,[a] kernel[a] | Yes[a] | PC (emulator) | TaintDroid | Source code |
| VetDroid [Zhang et al. 2013b] | 2-a-ii | Dynamic analysis | Android,[a] kernel[a] | Yes[a] | PC (emulator) | TaintDroid | No |
| DroidScope [Lok and Yin 2012] | 2-a-ii | Dynamic analysis | Android,[a] kernel[a] | Yes[a] | PC (emulator) | QEMU | Source code |
| DroidRanger [Zhou et al. 2012b] | 2-a-iii | Static & dynamic analyses | Android,[a] kernel[a] | Yes[a] | PC (analysis & emulator) | | No |
| Mobile-Sandbox [Spreitzenbarth et al. 2013] | 2-a-iii | Static & dynamic analyses | Android[a] kernel[a] | Yes[a] | PC (analysis & emulator) | aapt tool, baksmali, TaintDroid, DroidBox, ltrace, monkey-runner | No |
| Pegasus [Chen et al. 2013] | 2-a-iii | Static & dynamic analyses | No | No | PC (analysis, rewriting, emulator) | Soot | No |
| AppIntent [Yang et al. 2013] | 2-a-iii | Static & dynamic analyses | Android[a] kernel[a] | Yes[a] | PC (analysis & emulator) | ded, Soot, JavaPathfinder, InstumentationTestRunner | No |
| Peng et al. [2012] | 2-a-iv | Probabilistic modeling | N.A. | N.A. | PC (analysis) | | No |
| WHYPER [Pandita et al. 2013] | 2-a-iv | NLP | N.A. | N.A. | PC (analysis) | | No |
| DroidMOSS [Zhou et al. 2012a] | 2-b | Fuzzy hashing | N.A. | N.A. | PC (analysis) | baksmali, keytool | No |
| Juxtapp [Hanna et al. 2012] | 2-b | Feature hashing | N.A. | N.A. | PC (analysis) | djb2 hash, Hadoop | No |
| DNADroid [Crussell et al. 2012] | 2-b | PDG isomorphisms | N.A. | N.A. | PC (analysis) | WALA, VF2 algorithm | No |
| PiggyApp [Zhou et al. 2013b] | 2-b | Nearest neighbor clustering | N.A. | N.A. | PC (analysis) | | No |
| AnDarwin [Crussell et al. 2013] | 2-b | PDG & clustering | N.A. | N.A. | PC (analysis) | dex2jar, WALA, LSH, MinHash | No |

(Continued)

Table I. Continued

| System | Category | Technique(s) Used | Required Modifications | Rooting Needed? | Off-Device Execution | Other Tools Utilized | Publicly Available? |
|---|---|---|---|---|---|---|---|
| [Enck et al. 2011] | 2-c | Static analysis | N.A. | N.A. | PC (analysis) | ded, Soot, Fortify SCA | Source code (for ded) |
| Stowaway [Felt et al. 2011a] | 2-c | Static analysis | N.A. | N.A. | PC (analysis) | Randoop, Dedexer, ComDroid | *(site down)* |
| PScout [Au et al. 2012] | 2-c | Static analysis | N.A. | N.A. | PC (analysis) | Soot | Source code |
| Bartel et al. [2014] | 2-c | Static analysis | N.A. | N.A. | PC (analysis) | Soot | No |
| ComDroid [Chin et al. 2011] | 2-d | Static analysis | N.A. | N.A. | PC (analysis) | Dedexer | No |
| WoodPecker [Grace et al. 2012a] | 2-d | Static analysis | N.A. | N.A. | PC (analysis) | baksmali, adb | No |
| CHEX [Lu et al. 2012] | 2-d | Static analysis | N.A. | N.A. | PC (analysis) | DexLib, WALA | No |
| ContentScope [Zhou and Jiang 2013] | 2-d | Static analysis | N.A. | N.A. | PC (analysis) | baksmali | No |
| SEFA [Wu et al. 2013] | 2-d | Static analysis | N.A. | N.A. | PC (analysis) | baksmali, PScout | No |
| Epicc [Octeau et al. 2013] | 2-d | Static analysis | N.A. | N.A. | PC (analysis) | Heros, Spark, Soot, dare | Source code |
| ***Install-Time Checking (Category 3):*** | | | | | | | |
| Kirin [Enck et al. 2009a] | 3 | Rule-based system | Android | Yes | No | | Source code |
| ***Continuous Runtime Monitoring/Enforcement Systems (Category 4):*** | | | | | | | |
| TaintDroid [Enck et al. 2010; Enck et al. 2014] | 4-a | Dynamic taint analysis | Android | Yes | No | | Source code |
| BayesDroid [Tripp and Rubin 2014] | 4-a | Bayesian-based privacy leak determination | Android | Yes | No | TaintDroid | No |
| MockDroid [Beresford et al. 2011] | 4-a | Resource-access mocking | Android | Yes | No | | Source code |
| TISSA [Zhou et al. 2011] | 4-a | Resource-access mocking | Android | Yes | No | | No |
| AppFence [Hornyack et al. 2011] | 4-a | Dynamic taint analysis, resource-access mocking | Android | Yes | No | TaintDroid, TEMA project | Source code |
| LP-Guardian [Fawaz and Shin 2014] | 4-a | Location-access regulation | Android | Yes | No | | No |
| SELinux+Android [Shabtai et al. 2010a] | 4-b-i | MAC | Kernel | Yes | No | SELinux | No |
| SEAndroid [Smalley and Craig 2013] | 4-b-i | MAC | Android, kernel | Yes | No | SELinux | Source code |
| FlaskDroid [Bugiel et al. 2013] | 4-b-i | MAC | Android, kernel | Yes | No | SEAndroid | Source code |
| Android Security Module [Heuser et al. 2014] | 4-b-i | Reference monitor interface hooks | Android, kernel | Yes | No | | Source code |
| Apex [Nauman et al. 2010] | 4-b-ii | Rule-based policy | Android | Yes | No | | Source code |

(Continued)

Table I. Continued

| System | Category | Technique(s) Used | Required Modifications | Rooting Needed? | Off-Device Execution | Other Tools Utilized | Publicly Available? |
|---|---|---|---|---|---|---|---|
| CRêPE [Conti et al. 2010, 2012] | 4-b-ii | Rule-based policy | Android | Yes | No | | No |
| Porscha [Ongtang et al. 2010] | 4-b-ii | Rule-based policy, secure delivery | Android | Yes | No | | No |
| Jeon et al. [2013] | 4-b-iii | Binary rewriting | App | No | App-market (rewriting) | (no information) | No |
| Aurasium [Xu et al. 2012] | 4-b-iii | Binary rewriting | App | No | Aurasium (rewriting) | apktool | Web-based |
| Dr. Android & Mr. Hide [Jeon et al. 2012] | 4-b-iii | Binary rewriting | App | No | PC (analysis & rewriting) | Redexer, apktool, API mapping [Felt et al. 2011a] | No |
| I-ARM-Droid [Davis et al. 2012] | 4-b-iii | Binary rewriting | App | No | PC (rewriting) | smali/baksmali, API mapping [Felt et al. 2011a] | No |
| Invivo rewriting [Bartel et al. 2012] | 4-b-iii | Binary rewriting | App | No | No | dex2jar, Soot, ASM | No |
| AppGuard [Backes et al. 2013] | 4-b-iii | Binary rewriting | App | No | No | dexlib lib (smali), monkey | APK file |
| [Kantola et al. 2012] | 4-c-i | Heuristics for exporting components | Android | Yes | No | ComDroid | No |
| Saint [Ongtang et al. 2009, 2012] | 4-c-i | Rule-based policy | Android | Yes | No | | No |
| IPC Inspection [Felt et al. 2011c] | 4-c-i | Permission reduction | Android | Yes | No | Dedexer (for app analysis) | No |
| QUIRE [Dietz et al. 2011] | 4-c-i | Call-chain propagation | Android, OS | Yes | No | | No |
| XmanDroid [Bugiel et al. 2011a] | 4-c-ii | ICC & channel control | Android | Yes | No | | No |
| Bugiel et al. [2012]] | 4-c-ii | ICC & channel control | Android, kernel | Yes | No | XmanDroid | No |
| KBTA IDS [Shabtai et al. 2010c] | 4-d-i | Misuse detection | Android, kernel | Yes | No | | No |
| Paranoid Android [PortoKalidis et al. 2010] | 4-d-i | Misuse detection | Android, kernel | Yes | Detection server | ClamAV, dynamic taint analysis | No |
| Andromaly [Shabtai et al. 2012] | 4-d-ii | Anomaly detection | Android, kernel | Yes | No | Machine learning techniques | No |
| Crowdroid [Burguera et al. 2011] | 4-d-ii | Anomaly detection | Kernel | Yes | Detection server | strace | No |

(Continued)

Table I. Continued

| System | Category | Technique(s) Used | Required Modifications | Rooting Needed? | Off-Device Execution | Other Tools Utilized | Publicly Available? |
|---|---|---|---|---|---|---|---|
| TrustDroid [Bugiel et al. 2011b] | 4-e-i | Domain isolation | Android, kernel | Yes | No | TOMOYO Linux, XManDroid | No |
| AirBag [Wu et al. 2014] | 4-e-i | Domain isolation | Android, kernel | Yes | No | | No |
| L4Android [Lange et al. 2011] | 4-e-i | Virtualization | Kernel (L4Linux) | Yes | No | Fiasco.OC, L4R3, L4Linux, Karma | Source code |
| AdDroid [Pearce et al. 2012] | 4-e-ii | New advertisement API | Android, App | Yes | No | | No |
| AdSplit [Shekhar et al. 2012] | 4-e-ii | App transparency & provenance support | Android, App | Yes | No | QUIRE | No |
| AFrame [Zhang et al. 2013a] | 4-e-ii | Embedded activity | Android, App | Yes | No | | No |
| LayerCake [Roesner and Kohno 2013] | 4-e-ii | Embedded UI interfaces | Android, App | Yes | No | | Source code |
| NativeGuard [Sun and Tan 2014] | 4-e-iii | Native code isolation | App | No | PC (rewriting) | apktool, objdump | No |
| MalloDroid [Fahl et al. 2012] | 4-f | Static analysis | N.A | No | PC (analysis) | AndroGuard | No |
| SMV-Hunter [Sounthiraraj et al. 2014] | 4-f | Static & dynamic analyses | Android[a] | Yes[a] | PC (analysis & emulator) | | No |
| [Falh et al. 2013] | 4-f | Secure SSL handling | Android, App | Yes | No | | Source code |
| Morbs [Wang et al. 2013] | 4-f | Cross-origin policy | Android, App | Yes | No | | Source code |
| CryptoLint [Egele et al. 2013] | 4-f | Static analysis | N.A | No | PC (analysis) | AndroGuard | No |
| PatchDroid [Mulliner et al. 2013] | 4-f | Patch application | Android, App | Yes | Cloud | | No |
| AppSealer [Zhang and Yin 2014] | 4-f | Static analysis | Android, App | Yes | PC (analysis) | Soot | No |
| ***Installed-App Rechecking (Category 5):*** | | | | | | | |
| Pyandrazzi [Kennedy et al. 2013] | 5 | App repackaging, dynamic analysis | Android[a] | Yes[a] | PC (repackaging & emulator) | apktool, monkeyrunner, AndroidViewClient, logcat | No |

[a]The required modifications or rootings are performed on Android emulators instead of on real mobile devices.

The companion app can later be presented to a verifying party to trigger the executions of the embedded code segments inside the main app and recover the watermark value.

Research activity in this category seems to be limited. Some researchers instead attempt to address app repackaging problem by detecting app similarity as surveyed in Section 5.2. Given the need to protect Android apps, we can thus expect to see more upcoming works in this area in the near future, which bring software protection measures [Collberg and Thomborson 2002] into the Android development environment.

## 5. UNTRUSTED APP AND MARKET ANALYZERS (CATEGORY 2)

Systems in this category share a common important goal of analyzing untrusted apps on Android markets. Although the analyses can be done by users who are interested in installing the apps, it is more practical for them to be done by the respective markets or app-reviewing systems. These entities usually have the required knowledge and scalable resources to perform comprehensive, multimodal assessments on the apps. We divide the systems into the following four subcategories based on their subgoals.

### 5.1. Malicious Behavior Detectors (Category 2-a)

Systems in category 2-a aim to uncover any potentially malicious behavior of untrusted apps in an automated and scalable fashion. Typically, malicious apps in Android attempt to mount (i) cost-incurring attacks, such as sending SMSs or making calls to premium numbers, or (ii) privacy infringing attacks, such as leaking the user's phone-related information (e.g., contacts, messages, and call logs) or information derived from various available sensors (e.g., position sensors, camera, and microphone). We divide the systems in category 2-a based on their main approaches as follows.

*5.1.1. Systems Applying Static Analysis (Category 2-a-i).* A static analyzer inspects an app without actually executing it. Since it analyzes an app's whole source or recovered code, the analyzer can achieve a high code coverage. However, it lacks the actual execution path and relevant execution context. Moreover, it faces challenges in the presence of code obfuscation as well as dynamic code loading [Poeplau et al. 2014].

Performing an accurate static analysis on an event-driven GUI-based Android app requires an analyzer to satisfactorily deal with the following Android-specific issues [Arzt et al. 2014; Yang et al. 2013]: (i) the app has multiple entry points; (ii) there may be multiple asynchronously running app components; (iii) there exist numerous callbacks due to the app's lifecycle states, as well as listeners to both GUI and system events; and (iv) app components may accept both intra- and interapp ICC. In addition, there exist additional challenges posed by Java reflection and native code (weakness $W_7$) [Grace et al. 2012b]. Unlike Dalvik bytecode analysis, native binary code analysis has long been known to be challenging due to the lack of higher-level semantics, and it falls beyond the scope of our survey. Yet there exists work that aims to regulate native code execution in Android, which is surveyed later in Section 7.5.3.

RiskRanker [Grace et al. 2012b] analyzes an app in two major stages. In the first stage, it aims to discover nonobfuscated executions that invoke (i) known root, (ii) illegal cost creation, and (iii) privacy violation attacks. To detect known root exploits, RiskRanker scans any included ELF native binary files of the app for root exploit signatures. To find illegal cost creation, it applies a static analysis to check if a cost-creation operation is reachable from any callback methods that do not usually involve user interaction. Finally, for privacy violation detection, RiskRanker extends its reachability algorithm with slicing techniques to check if an information-sending operation is connected to any personal information readings. In the second stage, RiskRanker employs a set of heuristics to uncover app behavior related to encryption or dynamic code loading, which could evade the first stage. On its experiment with 118,318 apps

from 15 app markets, RiskRanker successfully uncovered 718 malware samples (in 29 families), among which 322 are zero-day (in 11 families). One main limitation of RiskRanker is its use of rather basic heuristics in dealing with encryption and code loading, which is prone to evasion attacks.

Fuchs et al. [2009] formalize Android app components and constructs to enable a dataflow analysis of Android apps using a previously developed language-based security model. Based on the formalism, a WALA[4]-based tool called *SCanDroid* can determine if the flow of data through a set of apps is consistent with confidentiality- or integrity-related specifications that are deduced from the apps' manifest files. No experimentation results are, however, reported on SCanDroid.

FlowDroid [Arzt et al. 2014] performs a flow-, context-, object-, and field-sensitive static taint analysis on Android apps. It models Android app's lifecycle states and handles taint propagation due to callbacks and UI objects. It also utilizes SuSi [Rasthofer et al. 2014], a machine-learning based technique to automatically identify the sources and sinks of sensitive information in the Android API. Using an accompanying benchmark tool for evaluating taint analyzers called *DroidBench*, FlowDroid is shown to achieve better results than two commercial tools, AppScan Source and Fortify SCA, with 86% precision and 93% recall. FlowDroid, however, does not aim to infer if a data transmission is most likely intended by the user.

Although FlowDroid applies an interprocedural dataflow analysis, it does not track dataflows across different app components that communicate by means of ICC. To achieve an accurate intercomponent dataflow analysis, Wei et al. [2014] devised Amandroid. While building its interprocedural control flow graph (ICFG), Amandroid calculates, at each program point, the points-to information of all objects in a flow- and context-sensitive manner. The ICFG improves the ones employed by FlowDroid as well as Epicc (see Section 5.4) not only by inferring ICC call parameters using the computed points-to facts but also by resolving ICC call targets and linking ICC sources to the call targets. Given the resulting ICFG whose nodes contain the computed points-to information, Amandroid constructs the data dependence graph. Various security analyses can then be conducted simply by querying the built graphs. Amandroid is shown to be able to discover data leaks, including those on OAuth token, and data injection attacks.

Other static analysis systems that also apply dynamic analysis are covered later in category 2-a-iii. Table II compares all systems in category 2 that apply static analysis techniques.

*5.1.2. Systems Applying Dynamic Analysis (Category 2-a-ii).* Dynamic analysis is conducted by executing an app, on either a real or virtual execution environment such as the Android Virtual Device (AVD), and observing the app as it is running. Although various dynamic analysis techniques and tools abound in the literature [Egele et al. 2012], conducting dynamic analysis on Android apps faces novel challenges, such as Android's managed resources, Binder-based ICC, and event triggers [Zhang et al. 2013b].

In our taxonomy, we differentiate between systems applying dynamic analysis on apps that are hosted on a market and systems that continuously monitor running apps on a device as the mobile user interacts with them (surveyed in category 4-d). The former faces more challenges due to the following reasons:

—There is a time limit on how an app can be executed and observed.
—There is a need to simulate system events and the user's GUI responses.
—Malicious apps may attempt to detect the employed Android virtualization or emulation systems and avoid launching their payloads. Vidas and Christin [2014] discuss techniques that detect the presence of Android virtualization or emulation systems.

[4]http://wala.sourceforge.net/wiki/index.php/Main_Page.

Table II. Comparison of Systems Performing Static Analysis in Category 2

| System | Analysis Goal | IR Format | Techniques Employed |
|---|---|---|---|
| ***Malicious Behavior Detectors (Category 2-a):*** | | | |
| RiskRanker [Grace et al. 2012b] | Dangerous behavior detection (known and zero-day) | Disassembled .dex (no specific information given) | Symbolic execution, points-to analysis, reachability analysis using slicing; checks on calls to crypto methods, dynamic code loading, secondary package invocation, native code execution |
| SCanDroid [Fuchs et al. 2009] | App confidentiality & integrity property validation | Java source code or bytecode | App reasoning based on a language-based security model |
| FlowDroid [Arzt et al. 2014] | Privacy leak detection | Jimple (derived by Dexpler) | Static taint analysis, on-demand alias analysis |
| Amandroid [Wei et al. 2014] | Intercomponent dataflow analysis | Disassembled .dex (derived by a modified dexdump) | Points-to analysis, ICC-call correlation, data dependence analysis |
| DroidRanger [Zhou et al. 2012b] | Malicious behavior detection (known and zero-day) | (No information given) | Permission analysis, API-call sequence analysis; checks on native code execution, dynamic code loading |
| Mobile-Sandbox [Spreitzenbarth et al. 2013] | Malicious behavior detection | Smali code (derived by baksmali) | Checks on dangerous methods, statically coded URL strings, calls to encryption libraries; collect information on timers and broadcasts as event triggers for dynamic analysis |
| Pegasus [Chen et al. 2013] | App interaction's temporal order validation | Java bytecode (derived by its translation module) | Points-to analysis, model checking, rewriting of reflective calls |
| AppIntent [Yang et al. 2013] | Privacy leak detection | Java bytecode (derived by ded) | Guided symbolic execution, data constraint solver |
| ***App Permission & Behavior Pattern Analyzers (Category 2-c):*** | | | |
| Enck et al. [2011] | App behavior analysis | Recovered Java source code (derived by ded & Soot) | Structural and semantic analyses using Fortify SCA |
| Stowaway [Felt et al. 2011a] | App's maximum set of permission derivation | Disassembled .dex files (derived by Dedexer) | API call identification, content-provider access permission analysis, intent permission analysis (using ComDroid [Chin et al. 2011]) |
| PScout [Au et al. 2012] | Permission mapping generation | Recovered Java bytecode (derived by Soot) | Permission check identification, backward reachability analysis |
| Bartel et al. [2014] | Permission mapping generation | Recovered Java bytecode (derived by Soot) | String analysis, service redirection, service identity inversion, service initialization, entry point initialization |

(Continued)

Table II. Continued

| System | Analysis Goal | IR Format | Techniques Employed |
|---|---|---|---|
| | ***App Vulnerability Analyzers (Category 2-d):*** | | |
| ComDroid [Chin et al. 2011] | ICC-based vuln. discovery | Disassembled .dex files (derived by Dedexer) | Intent-based dataflow analysis, unintentional publicly exposed component analysis |
| WoodPecker [Grace et al. 2012a] | ICC-based vuln. discovery on preloaded apps | Smali code (derived by baksmali) | Possible path construction, symbolic path simulation |
| CHEX [Lu et al. 2012] | ICC-based vuln. discovery (as a dataflow problem) | Static Single Assignment (derived by DexLib & additional translation) | Component entry-point identification using call-graph analysis, reachability analysis on data dependence graph, split & permutation dataflow summary calculations |
| Content-Scope [Zhou and Jiang 2013] | ICC-based vuln. discovery on content providers | Smali code (derived by an extension of baksmali) | Reachability analysis, contraint generation using dataflow analysis, constraint solver application |
| SEFA [Wu et al. 2013] | ICC-based vuln. discovery on vendor customization | Smali code (derived by baksmali) | App provenance analysis using call-graph similarity analysis, reachability analysis including using cross-app path discovery |
| Epicc [Octeau et al. 2013] | ICC-based vul. discovery (as an IDE problem) | Recovered Java source code (derived by dare & Soot) | Interprocedural CFG analysis, points-to analysis |

—The detection systems are additionally expected to be scalable to assess an enormous amount of available apps.

AppsPlayground [Rastogi et al. 2013a] performs a TaintDroid (see Section 7.1)-based dynamic taint tracing, API monitoring, and kernel-level monitoring. To reduce the chances of sandbox detection, AppsPlayground sets realistic device-related identifiers and data. To realize a comprehensive execution coverage, it implements event triggering and intelligent execution techniques. These execution techniques perform heuristics-based efficient exploration of an app's GUI model that is created from its window and widget features. With all of these mechanisms, AppsPlayground achieves a code coverage of 33%.

VetDroid [Zhang et al. 2013b] analyzes an app's permission use—that is, how permissions are used by an app to access sensitive resources and how the resources are subsequently utilized. To identify the callsites where the app requests sensitive resources, VetDroid intercepts all calls to the Android API and synchronously monitors permission check information from the Android permission enforcement system. In this way, it manages to generate a more accurate permission mapping than Stowaway and PScout (surveyed in Section 5.3). To keep track of where the acquired resources are subsequently used, VetDroid leverages TaintDroid and implements a permission-based taint analysis. In analyzing 1,249 top apps in Google Play, it managed to find more privacy leaks than TaintDroid. Additionally, VetDroid can point out the causes of the leaks by tracing the context of permission usage within the analyzed apps.

There exist works that focus on devising a robust dynamic analysis platform. One example is DroidScope [Lok and Yin 2012], which is a virtual machine introspection (VMI)-based platform that reconstructs both OS- and Java-level semantic views. Since QEMU-based DroidScope inspects a running app from its separate protection domain, it thus offers a more robust analysis environment, as it remains safe even when monitoring a root-exploit type of malware.

Many other systems applying dynamic analysis opt to additionally apply static analysis techniques, and they are discussed in category 2-a-iii next.

*5.1.3. Systems Applying Static and Dynamic Analyses (Category 2-a-iii).* Given the known respective limitations of static and dynamic analysis techniques, many proposed systems opt to perform both types of analyses to complement each other.

Google Play deploys a system called *Bouncer*, which automatically scans for malicious apps. Details of how Bouncer works are not publicly available, yet there exist reports from security researchers who made attempts to infer how it might work [Oberheide and Miller 2012; Percoco and Schulte 2012]. Oberheide and Miller [2012] found out that Bouncer seemed to perform dynamic analysis on an app by running it on a QEMU-based emulated Android environment for only about 5 minutes. They also found that it was rather easy to check that the app was running inside an emulator as opposed to a real phone. Additionally, Bouncer appeared to apply static analysis as a sample app containing a pathname string `/system/bin` was blocked.

DroidRanger [Zhou et al. 2012b] aims to evaluate the health of Google Play and four representative alternative Android markets based on the number of malicious apps contained among the available apps. It applies two main techniques: permission-based behavioral profiling to detect known malware, and heuristics-based filtering and dynamic execution monitoring to detect zero-day malware.

Mobile-Sandbox [Spreitzenbarth et al. 2013] also combines static and dynamic analyses. For static analysis, it first analyzes the permissions requested by an app. Then, it converts the app's Dalvik bytecode to smali code using baksmali[5] and looks for

[5]https://code.google.com/p/smali.

dangerous methods, statically-coded URL strings, and calls to encryption libraries. Information on timers and broadcasts as event triggers are also collected to improve code coverage during its dynamic analysis stage. In its dynamic analysis, Mobile-Sandbox logs runtime information at the following three levels: (i) Dalvik-level monitoring using TaintDroid and a customized version of DroidBox, (ii) native code monitoring using `ltrace`, and (iii) network traffic monitoring into a PCAP file. To simulate user interaction, Mobile-Sandbox utilizes the monkeyrunner tool.[6] External events, such as incoming calls or SMS messages, are also simulated so as to trigger potentially malicious behavior.

One important problem with the surveyed analyzers so far is that they cannot differentiate whether an app's behavior is in fact intended by the user. Pegasus [Chen et al. 2013] detects malicious behavior that violates the temporal properties of safe interactions between an app and the Android event system. It can thus detect, for instance, if an operation is invoked without the prerequisite GUI-based interaction that indicates the user's consent. Pegasus specifies permissible interactions between events and an app as temporal logic formulae. It constructs a graph that abstracts the interactions between events and the app's permission usage. Using model checking, Pegasus seeks to find a counterexample of the stipulated safety properties. Additionally, a dynamic analysis through app rewriting is performed to deal with possible Java reflection cases.

AppIntent [Yang et al. 2013] checks if a data transmission by an app is intended by the user. It employs a symbolic execution to produce a sequence of UI interactions and data inputs that can lead to data transmission. To deal with the known path-explosion problem faced by symbolic execution, AppIntent first runs a static taint analysis to generate an event-space constraint graph. This graph identifies all possible paths over all events raised by the app's lifecycle callbacks, GUI interactions, and system events that can lead to data transmission. Using this constraint graph, the symbolic execution now operates on a much more bounded search space. Given the produced events and data inputs, a human analyst can utilize the developed dynamic analysis platform to confirm if the transmission is indeed user intended.

*5.1.4. Systems Analyzing App Metadata (Category 2-a-iv).* Besides an app's recovered code and runtime behavior, its accompanying metadata is a source of useful information that can be used to infer the app's capability, and therefore potential behavior. Several works analyze the metadata of untrusted apps to assess their security risks.

Peng et al. [2012] employ probabilistic generative models to *relatively* rank the risks posed by a set of apps based on the permissions they request. By applying the scheme, one would know, for instance, that a particular app is among the highest 5% risk of all apps under the Weather category in an app market. Based on an evaluation using 200,000 apps from Google Play, naive Bayes with informative priors is the examined model that performs most satisfactorily. Although interesting and useful, this approach may fall short in highlighting a high risk carried by malicious apps that utilize a single but critical permission. Furthermore, given its comparative risk ranking approach, the scheme would need to periodically run to keep track of very dynamic app markets.

WHYPER [Pandita et al. 2013] is a natural language processing (NLP)-based system that analyzes the permission list and textual English description of an app to infer why the app possibly requires a permission. Although WHYPER is shown to achieve significantly better accuracy than keyword-based search, considerable challenges remain for it to provide a reliable, automated app risk assessment. Nevertheless, WHYPER highlights the importance of requiring app developers to provide adequate explanation on the permissions requested by their apps.

[6]http://developer.android.com/tools/help/monkeyrunner_concepts.html.

*5.1.5. Discussions on Works in Category 2-a.* The proposed systems in category 2-a utilize various techniques to assess untrusted apps. Yet the main challenge remains on how to accurately model Android app's behavior and malicious actions so that app analyzers can detect apps that contain existing malware samples, their variants, or zero-day samples in a scalable manner. Such modeling should also take into account possible user, system, and interapp interactions. The analysis systems additionally must be resistant against evasion attacks and produce acceptably low false negatives. We will discuss several common problems and possible enhancements that could address them.

One main issue with performing static analysis on Android apps is the availability of reverse engineering tools that can generate an Intermediate Representation (IR) suitable for accurate analyses. As can be seen in Table II, different schemes utilize different tools and IRs. One direction is thus to derive an IR, and its corresponding conversion tools, that allows various established static analysis tools to be utilized. In addition, the existing analysis tools should also be extended to take into account the semantics specific to Android apps and Android execution environments.

As mentioned earlier, achieving high code coverage is very important to dynamic analysis systems. Information on event triggers that can be inferred by static analysis techniques, such as those derived by Mobile-Sandbox [Spreitzenbarth et al. 2013], is very useful to increase this coverage. Likewise, a comprehensive UI event generator tool needs to be available to automatically simulate user responses. Otherwise, owing to the absence of proper app contexts, fuzzing-based interactions can be limited and the automation may fail to explore critical execution paths. Other than AppsPlayGround, PUMA [Hao et al. 2014] and Brahmastra [Bhoraskar et al. 2014] are two recently proposed UI event–generating systems. PUMA is a programmable UI automation framework that separates analysis logic from exploration logic. Brahmastra, on the other hand, is specifically aimed at invoking third-party code. Rather than performing GUI exploration, it applies static analysis to discover execution paths that invoke third-party code, then rewrites an app to direct its execution into that code. Last, an analysis system in which an app is executed needs to be made as realistic as possible to avoid evasion attempts as analyzed by Vidas and Christin [2014].

### 5.2. App Similarity Detectors (Category 2-b)

Due to Android weaknesses $W_3$ and $W_1$, Android apps are prone to illegal repackaging. The repackaging can be done for software piracy, license evasion, or malware injection. With millions of apps available on various app markets, an accurate, fast, and scalable technique for app similarity detection is therefore required.

DroidMOSS [Zhou et al. 2012b] applies a fuzzy hashing technique to the list of opcodes used by an app to generate a fingerprint of that app. It then computes an edit-distance–based similarity score between two fingerprints to measure similarity of the two corresponding apps. An evaluation with DroidMOSS on six third-party Android markets revealed a worrisome fact that 5% to 13% of the hosted apps are repackaged from those in Google Play, which are assumed to be original.

Juxtapp [Hanna et al. 2012] similarly works on code sequence of an app. From a code sequence within each basic block of an app, it generates all $k$-grams of opcodes (and necessary differentiating arguments) and then applies a Bloom-filter–based feature hash to produce the app's vector representation. By applying Jaccard similarity metric on the vectors, Juxtapp determines the similarity either between any two apps by means of a pairwise code containment or among a set of apps by means of clustering.

To detect app similarity in the face of substantial code modifications, DNADroid [Crussell et al. 2012] compares the program dependency graphs (PDGs) of two different apps. It utilizes WALA to generate a PDG for each method in every class of the two

apps. In a PDG, each node represents a statement, whereas each edge shows a data dependency between statements. Given two sets of PDGs belonging to the two apps, DNADroid looks for interset subgraph isomorphisms to deduce their similarity. Hence, in essence, it attempts to find similar methods in a pair of apps from a data dependency viewpoint.

One limitation of the three preceding systems is the need to perform pairwise comparisons, which hampers their scalability in dealing with an enormous amount of released apps. Instead of detecting a more general problem of app similarity, PiggyApp [Zhou et al. 2013b] detects piggybacked apps, where an original app (carrier) is injected with additional code (rider). PiggyApp first performs a module decoupling to partition an app's code into primary and nonprimary modules. Then, it produces a feature vector of the app's primary module. A vantage point tree (VPT)-based metric space is subsequently constructed so that the problem of detecting similar primary modules is cast into a nearest neighbor searching problem. By employing triangle inequality-based VPT pruning, PiggyApp is able to perform a search with $O(n \log n)$ time complexity.

AnDarwin [Crussell et al. 2013] extends DNADroid to similarly avoid quadratic pairwise comparison. It performs the following four stages on examined apps. First, AnDarwin represents an app as a set of semantic vectors as follows. For each method in an app, it constructs a PDG, which is then split into connected components. A semantic vector for each connected component is then computed based on the frequencies of contained node types. Second, AnDarwin finds similar code segments by clustering all semantic vectors from all examined apps using locality-sensitive hashing (LSH). Third, unlike previous works that manually exclude common libraries (e.g., A&A libraries), AnDarwin automatically eliminates them by filtering out clusters whose member size exceeds a certain threshold. Finally, AnDarwin performs another round of clustering using MinHash on the surviving features to determine either full or partial app similarity in $O(n \log n)$.

*5.2.1. Discussions.* Detecting app similarity constitutes an important part of an app vetting process. Although app similarity detection is orthogonal to malware detection, the former is useful in revealing the possible presence of injected malware. Hanna et al. [2012] discuss several challenges faced by app similarity detectors in an Android environment. Achieving a high detection accuracy remains a key issue to these detectors, as both false positives and false negatives are possible due to the employed heuristics, approximation algorithms, or filtering processes. Repackaged or injected apps that additionally have been subjected to code transformations [Rastogi et al. 2013b; Zheng et al. 2012; Collberg et al. 1997] must be correctly associated with their respective original apps. Another challenge is how to isolate and remove various commonly found libraries, which act as noise with respect to an app's main functionality. Whereas most works manually remove them prior to applying their similarity algorithms, AnDarwin automatically does so. PiggyApp, meanwhile, deals with this issue by considering only the primary module of an app. A common threat to all of the proposed systems is the possibility of evasion attacks, which attempt to deceive the employed heuristics and algorithms. Last, the presence of native code remains a problem to systems that solely work at the Dalvik bytecode level.

### 5.3. App Permission and General Behavioral Pattern Analyzers (Category 2-c)

Works in this category aim to discover the behavioral patterns of Android apps, including their permission usages.

Enck et al. [2011] conducted a static analysis on 1,100 (50 apps from each 22 category) popular free Android apps. They developed a Dalvik decompiler called *ded*, which

converts a .dex file to Java source code, and make use of Soot as a post-retargeting optimizer tool. The analysis conducted on the recovered code reveals a number of important findings on the apps, including a wide misuse of privacy-sensitive information and an extensive use of A&A libraries.[7] The decompilation of .dex to Java, however, faces several challenges. First, it cannot always recover the Java source code: only approximately 94% of .dex files in the experimentation are convertible.[8] Second, the process is also rather time consuming: it took nearly 20 days to analyze the 1,100 apps.

As permissions are central to app behavior, many works focus on examining Android permission usages and abuses by apps. Barrera et al. [2010] examine 1,100 representative apps to find out how the Android permission model is used in practice. They employ the neural network–based self-organizing map (SOM) algorithm, which is able to derive a two-dimensional visualization of high-dimensional cluster structures. Their SOM-derived visualization graphically shows the correlation among the used permissions and reveals Android permission usage patterns by app developers.

Felt et al. [2011a] present a static analysis tool called *Stowaway*, which can check whether an app is overprivileged. To determine what permissions are actually used by an app, a map between Android API calls and their required permissions is necessary. Felt et al. empirically build a permission map through a combination of feedback-directed fuzzing-based testing, proprietary test-case generation, and manual verification techniques. ComDroid [Chin et al. 2011] (surveyed later) is also used to examine permission-requiring intents. On Android version 2.2, a permission map with 1,259 API calls is derived. An experimentation with 940 apps uncovers that nearly 35.8% of the apps request permission(s) that they never actually use.

As shown by Felt et al., the availability of an accurate permission map is important in analyzing Android apps. Stowaway, however, faces challenges in replicating its permission map generation on different Android versions due to the significant amount of manual effort involved. To overcome this limitation, PScout [Au et al. 2012] performs repeated reachability analyses between API calls and permission checks on a call graph that is constructed from the Android framework's code base. On Android 2.2, PScout manages to derive 17,218 mappings, whereas Stowaway derives only 1,259.

Bartel et al. [2014] also applies static analysis to derive a permission map. Similar to PScout, they use Soot to analyze the Android framework. Whereas PScout's analysis is based on class hierarchy analysis (CHA), Bartel et al. [2014] instead employ a field-sensitive call graph analysis called *Spark*. Their results confirm those of PScout.

Based on analyzing 10 Android images from five smartphone vendors, Wu et al. [2013] find that 85.78% of the examined preloaded apps are overprivileged, of which 66.40% are due to vendor and bundled third-party apps. In their work, they leverage the permission map of PScout by additionally analyzing permissions at both the *system* and *signatureOrSystem* levels. Since the work also analyzes ICC-based vulnerabilities of the Android images, we additionally discuss it in Section 5.4.

*5.3.1. Discussions.* The availability of permission maps derived by Stowaway, PScout, Spark-based static analysis, and VetDroid (see Section 5.1.2) enables various analyses to be done on Android apps [Jeon et al. 2012; Davis et al. 2012]. The uncovered app permission bloat issue [Felt et al. 2011a; Wu et al. 2013], together with the possibility of ICC-based vulnerability (discussed next), allows malicious apps to mount privilege

[7]The results reported by Enck et al. [2011] also include several ICC-based vulnerabilities on the examined apps. Section 5.4 surveys works that specifically analyze ICC-based vulnerabilities of Android apps.

[8]A newer Dalvik decompiler called *Dare* is subsequently proposed by the same research group [Octeau et al. 2012]. Dare improves ded with a retargeting success rate greater than 99% using the same set of 1,100 apps evaluated by Enck et al. [2011].

escalation attacks more easily. One possible solution to this problem is providing developers with an automated tool that can help them infer the necessary permissions for their apps.

### 5.4. App Vulnerability Analyzers (Category 2-d)

Several works focus on statically discovering an app's ICC-based vulnerabilities. The presence of these vulnerabilities may result in data leakage, unauthorized ICC interactions, or privilege escalation attacks.

Chin et al. [2011] point out that unrestricted intent-based ICC mechanisms in Android can allow an attacker to perform unauthorized intent receipt, intent spoofing, or access to an app's public components that are guarded with weak or no permissions. Chin et al. developed ComDroid, a static analysis tool that tracks an app's component and intent usages to detect such vulnerabilities. In the reported experimentation using 20 apps, ComDroid found 34 exploitable vulnerabilities in 12 apps.

Octeau et al. [2013] recast the problem of ICC analysis into an interprocedural distributive environment (IDE) problem, where an efficient and more precise static analysis technique can be employed. The developed tool called *Epicc* connects all interapp components and leverages Soot to solve the specified IDE problem. A conducted evaluation shows that Epicc produces far fewer false positives than ComDroid. One additional nice feature of Epicc is that, when it is used to analyze a set of apps, it can determine other previously analyzed apps that can exploit a found vulnerability of an app. Epicc, however, has limitations due to its simple string analysis and omission of content provider analysis.

WoodPecker [Grace et al. 2012a] aims to discover capability leaks, which occur when permission-based capabilities of an app are accessible by other apps. Instead of analyzing apps available on app markets, the work examines preloaded apps that come in eight popular Android smartphones (i.e., weakness $W_{10}$). Similar to IPC Inspection [Felt et al. 2011c] (discussed in Section 7.3.1), WoodPecker statically determines the reachability of dangerous permissions from an app's public interfaces. The reachability analysis in WoodPecker additionally considers cases of control-flow discontinuity through callbacks. It also looks for capability leaks due to permission inheritance from another app that is signed with the same key, which usually comes from the same developer. Based on the analysis on the preloaded apps, WoodPecker finds that among 13 privileged permissions analyzed, 11 permissions are leaked.

Lu et al. [2012] address the problem of statically finding ICC-based vulnerability by taking a new dataflow perspective. They define the notion of *component hijack*, which occurs when an unauthorized app is able to issue requests to a victim app's public interface(s) to either read sensitive data out or write into its critical data region. The developed system, CHEX, performs a reachability analysis on a data dependency graph and is able to capture various types of vulnerabilities reported by ComDroid and WoodPecker. It evaluated 5,486 popular free apps and found that 254 apps contain hijack-enabling flow (with a true positive rate of 81%).

The preceding systems focus on discovering privilege escalation operations. ContentScope [Zhou and Jiang 2013], in contrast, looks for app vulnerabilities due to unprotected content providers. In Android, a content provider is a passive app component that functions as data storage. ContentScope scans apps for *passive content leaks*, which occur when apps inadvertently allow their private data to be accessible by other apps, and *content pollution*, where an app's private data can be manipulated by other apps. To these ends, ContentScope statically checks whether any exposed interfaces of an app's content providers could lead to operations on the app's private data. Candidate inputs to potential execution paths from this static analysis are then confirmed by dynamic testing. Android 4.2 and later remedy this issue of unprotected

content providers by not publicly exporting content providers by default. Hence, they are immune to the vulnerabilities. Earlier Android versions, however, remain affected by the vulnerabilities.

Similar to WoodPecker, Wu et al. [2013] evaluate the security impact of vendor customizations. They perform an analysis on 10 Android images from five smartphone manufacturers to find out if the preloaded apps are vulnerable to confused deputy attacks or content leaks. One novelty of the conducted analysis compared to WoodPecker and CHEX is its cross-app vulnerability analysis, which aims to find a vulnerable path involving two different apps. Wu et al. found that 6.77% of the preloaded apps are vulnerable. Furthermore, on most of the examined phones, 64.07% to 85.00% of these vulnerabilities stem from vendor customizations.

In addition to ICC-based vulnerabilities, apps may also be vulnerable due to poor security practices by app developers. The misuse of Secure Sockets Layer (SSL)/Transport Layer Security (TLS) [Fahl et al. 2012] and cryptographic library [Egele et al. 2013] are two common examples. We, however, examine these vulnerabilities when discussing secure app enablers later in Section 7.6, because the works reporting the vulnerabilities also propose various measures to strengthen app-layer security.

*5.4.1. Discussions.* The works surveyed earlier make important contributions by statically discovering the possibility of ICC-based vulnerability on Android apps. Together with the runtime systems in category 4-c, they can help prevent privilege escalation attacks. Tools that can assist app developers to discover vulnerabilities in their apps' source code should be made available to them during the development stage. Additionally, future Android versions can impose a stricter permission model of interapp ICC for app developers to properly adhere to.

## 6. INSTALL-TIME APP CHECKING SYSTEMS (CATEGORY 3)

Android presents a user with a list of requested permissions before installing an app. If the user find the permissions objectionable, he or she can choose not to proceed. Felt et al. [2012] empirically examine if the Android permission system is effective at warning users of apps' security risks in terms of the users' attention, permission comprehension, and approval decision behavior. Based on their usability studies, Felt et al. conclude that the Android install-time permission system does not help the majority of users make informed decisions on permission approvals.

To help a user better understand the risks carried by an app installation, Kirin [Enck et al. 2009a] runs on a device and checks the permissions requested by an app against a set of predefined security rules. A Kirin rule is a conjunction of Android permissions that may lead to a potentially dangerous operation. With a system like Kirin, a user can therefore make more informed decisions on whether to proceed or abort an installation. Policy engineering is, however, nontrivial, particularly if it is to be done on Android's rather coarse-grained permissions (weakness $W_2$). Another problem is that at install time, a user may find it hard to tell whether an app genuinely needs the requested permissions to perform its functionality or is really a malware.

We note that systems enforcing fine-grained policy in category 4-b may also restrict new app installations. SEAndroid [Smalley and Craig 2013] can restrict app installations based on a signature- or configuration-based policy. Saint [Ongtang et al. 2009], meanwhile, allows a developer to restrict his or her app's interactions. The focus of these systems, however, lies in monitoring app executions as discussed later in Section 7.2.

## 7. CONTINUOUS RUNTIME MONITORING/ENFORCEMENT SYSTEMS (CATEGORY 4)

We now survey systems that continuously run on a device to either prevent/detect malicious behavior or enforce fine-grained policy.

### 7.1. Sensitive Information Leakage Preventors (Category 4-a)

One of the most serious security concerns in mobile platforms, including Android, is the leakage of private information [Enck et al. 2011; Zhou and Jiang 2012]. As such, many previous works have been proposed to deal with this issue at runtime stage by either regulating accesses to sensitive information or releasing falsified (mocked) information instead. Systems in this category can also be considered as a special case of those in category 4-b, which provide infrastructures for generic permission enforcements.

TaintDroid [Enck et al. 2010, 2014] applies dynamic taint analysis to observe an app's potential privacy-infringing behavior. It marks any data that originates, or is possibly derived, from sensitive sources as tainted. Through its modification of the Dalvik VM, TaintDroid stores and propagates taint tags on variables and performs tracking at the method-, message-, and file level. If there is any tainted data leaving the network interface as the taint sink, TaintDroid will report such events to the user. TaintDroid, however, cannot detect privacy leaks via implicit dataflows. Sarwar et al. [2013] evaluate TaintDroid against implicit dataflow, subversion of benign code, and side-channel attacks. They empirically point out a number of attacks that malware writers can apply to evade the dynamic taint-tracking mechanism of TaintDroid.

Another challenge faced by dynamic taint-tracking systems like TaintDroid is the possibility of high false positives due to the overapproximation in the employed taint-tracking policy. To deal with this, BayesDroid [Tripp and Rubin 2014] employs statistical Bayesian-based reasoning to determine if an information release at a sink really represents a privacy leak. It calculates the likelihoods of both legitimate and illegitimate information releases at a sink based on the similarity between the information about to be released and the corresponding private information stored on the device. Experiments show that BayesDroid generates substantially lower false positives than TaintDroid while still maintaining high true positives.

MockDroid [Beresford et al. 2011] provides the user with an option to decide whether to use real data or the mocked versions of sensitive resources. Similarly, TISSA [Zhou et al. 2011] can report either empty, anonymized, or bogus information according to the app's privacy setting. One appealing feature of both MockDroid and TISSA is that the user can change the privacy setting of an app at any time after the app installation.

AppFence [Hornyack et al. 2011] combines data shadowing as in MockDroid and TISSA, with taint analysis for exfiltration blocking as in TaintDroid. In fact, it extends TaintDroid to track seven additional types of sensitive information. It is also able to detect leakage of data that has been obfuscated, encrypted, or transmitted via SSL. In addition, AppFence performs an extensive evaluation on the user-visible side effects of enforcing its privacy measures. It automates the execution of an app with a script-based GUI testing system and captures the visual difference between the screenshots of the app running with and without its privacy controls.

LP-Guardian [Fawaz and Shin 2014] specifically protects location privacy against tracking, identification, and profiling threats on a per-app basis. It utilizes a decision logic, which also takes into account the user's preference, to determine if it should release location information upon access by an app and on what granularity level. A conducted user study reports that the incurred loss of app functionality is still perceived to be tolerable. This is because the coarse-grained or anonymized location information is supplied to requesting apps mostly in places that are frequently visited by the users, such as their homes or workplaces, where location information needs to be protected and accurate location-based functionality is not usually required.

*7.1.1. Discussions.* The idea of utilizing taint tracking to detect information leakage is appealing. It allows an app to read and process sensitive information but detects any information disclosure to an outside party. Performing such taint tracking, as

implemented in TaintDroid and AppFence, however, requires rather extensive modifications on the Android middleware. It also may yield both false positives and false negatives. In contrast, systems like MockDroid and TISSA simply regulate access to sensitive resources. As such, their implementations can be relatively easier. Additionally, such systems can provide the user with a runtime option on how data falsification should be done on each protected app. In fact, an access-regulation–based privacy protection system can be devised on top of general fine-grained policy enforcement systems in category 4-b. Last, there exists a privacy leak concern due to colluding apps [Marforio et al. 2012], which is discussed later in Section 7.3.3.

### 7.2. Continuous Runtime Policy Enforcement Systems (Category 4-b)

Systems in this category intercept interfaces to privileged operations to continuously enforce stricter policy on all running apps. These enforcements are employed to address the problem of coarse-grained, install-time permission granting in the Android security model (weakness $W_2$). We categorize the systems based on which software-stack level(s) they modify to implement their enforcement mechanisms. A comparison of all of these systems and two other systems from category 4-c, which enforce fine-grained policy to regulate interapp communication, is shown in Table III.

*7.2.1. Systems Modifying Linux Kernel and Android (Category 4-b-i).* The Android middleware runs on top of the Linux kernel. If an attacker is able to compromise the kernel, then all Android-level security protections will be rendered ineffective (weakness $W_9$). Linux is, however, widely known to have a limitation where many high-privileged processes run with the root privilege. It therefore makes sense to harden the Linux kernel with mandatory access control (MAC)-based systems that exercise the principle of least privilege more strictly.

Shabtai et al. [2010a] report an effort, including its encountered challenges, to integrate SELinux [Loscocco and Smalley 2001] into Android. They show how the integrated SELinux can enforce kernel-level access control, which will limit what an attacker can do after successfully exploiting vulnerability(ies) in privileged processes.

A more recent and comprehensive integration effort is reported by Smalley and Craig [2013]. The system, SEAndroid, additionally implements a middleware mandatory access control (MMAC) enforcement layer to strengthen the Android permission model. For instance, it allows for install-time MAC, which can regulate app installations based on the developers' signatures or permission sets.

Bugiel et al. [2013] propose FlaskDroid as a generic kernel and middleware MAC based architecture that can support multiple fine-grained security policies and use cases. FlaskDroid basically implements type enforcement (TE) with a policy language that is designed to capture operational semantics at the Android middleware. Bugiel et al. demonstrate how FlaskDroid can instantiate certain uses cases, such as a system that enforces developers' policies similar to Saint [Ongtang et al. 2009].

Heuser et al. [2014] developed the Android Security Module (ASM) framework. ASM modifies both the Linux kernel and Android middleware to provide a programmable framework for defining new reference monitors for Android. It applies the methodology of the Linux Security Modules (LSM) and TrustedBSD frameworks, both of which have been very successful, to Android. With ASM-equipped Android, security developers can register their authorization module that will handle a callback at every point where an app's operation is about to result in access to protected resources. Hence, a variety of access control models can be supported by ASM without making modification to the Android source code.

Zhou et al. [2013c] report how unprotected Linux shared resources (e.g., procfs- and sysfs-based public directories), which are thought to be harmless, can be used

Table III. Comparison of Systems Enforcing Runtime Fine-Grained Policy in Categories 4-b and 4-c

| System | Policy Purpose | Policy Setter | Policy Setting Time | Modifiable? | Policy Activation Time | Policy Scope |
|---|---|---|---|---|---|---|
| ***Systems Modifying Linux & Android (Category 4-b-i):*** | | | | | | |
| SELinux + Android [Shabtai et al. 2010a] | Kernel-level MAC | Platform | Platform release | By patch | Device's boot time | System |
| SEAndroid [Smalley and Craig 2013] | Kernel- & middleware-level MAC | Platform | Platform release | By patch | Device's boot time | System |
| FlaskDroid [Bugiel et al. 2013] | Kernel- & middleware-level MAC | Platform | Platform release | By patch | Device's boot time | System |
| ***Systems Modifying Android (Category 4-b-ii):*** | | | | | | |
| Apex [Nauman et al. 2010] | Runtime constraints on resource access | User | Install- & runtime | Yes | User-direction (when editing setting) | App |
| CRêPE [Conti et al. 2010, 2012] | Context-based constraints on resource access | User | Install- & runtime | Yes | Upon context changes (due to sensor or user/3rd-party interactions) | App |
| Porscha [Ongtang et al. 2010] | Content delivery & access protection | Content owner | Content release | No | During content delivery & access | Content |
| ***Systems with Unmodified Android (Category 4-b-iii):*** | | | | | | |
| Jeon et al. [2013][a] | Resource access control | User | Runtime | Yes | User direction (when approving alert) | App |
| Aurasium [Xu et al. 2012][a] | Resource access control | User | Runtime | Yes | User direction (when approving alert) | App |
| Dr. Android & Mr. Hide [Jeon et al. 2012][a] | Domain- or field-based resource access control | Developer, Refine-Droid | App development or app rewriting | No | App invocation | App |
| I-ARM-Droid [Davis et al. 2012][a] | Resource access control | User | (i) App rewriting<br>(ii) Runtime | (i) No<br>(ii) Yes | (i) App invocation<br>(ii) User direction (when approving alert) | App |
| In vivo rewriting [Bartel et al. 2012][a] | Resource access control | User | App rewriting | No | App invocation | App |
| AppGuard [Backes et al. 2013] | Resource access control | User | Runtime | Yes | User direction (when editing setting) | App |
| ***Privilege-Escalation Attack Preventors (Category 4-c):*** | | | | | | |
| Saint [Ongtang et al. 2009, 2012] | App-interface access control | Developer | App release | No | App installation & invocation | App |
| XmanDroid [Bugiel et al. 2011a, 2012] | ICC & channel control on colluding apps | Platform | Platform release | By patch | Device's boot time | System |

[a]The binary-rewriting systems are capable of letting the user set his or her app's policy at either (i) app-rewriting time or (ii) app runtime by approving an alert shown on each sensitive operation's first invocation. This table simply lists the enforcement scenario considered in each respective work.

to infer the user's private information. A malicious app with no permissions on a device surprisingly can leak information from popular apps installed on the device. Together with public information available on the apps' corresponding Web sites, the leaked information makes user correlation possible. The authors show how such a zero-permission malicious app can infer the user's identity, location, and visited parts of online sites.

Zhou et al. [2014] also report the presence of file permission underprotection vulnerabilities on important Linux device files due to device driver customization by vendors or carriers. These vulnerabilities allow unauthorized apps, for instance, to take pictures and screenshots on several widely used smartphone models.

*Discussion*. Hardening the underlying Linux kernel is important, and the contributions of incorporating SELinux into Android should be acknowledged. In fact, Android had its sandbox reinforced with SELinux starting from version 4.3. The main known challenge with these MAC-enforcing systems is how their security policies should be engineered to cater to general mobile users or targeted use cases. Moreover, although systems like SEAndroid can enforce MMAC, they may cause conflicts with separately deployed security extensions, such as systems in category 4-b-ii. ASM provides a generic framework for implementing access control models in Android. However, to be usable on a large scale, it must be adopted by Google. The result reported by Zhou et al. [2013c] necessitates stricter access controls on system directories such as procfs and sysfs. Vendors and carriers must also ensure that their customizations will not introduce any vulnerabilities.

*7.2.2. Systems Modifying Android Middleware (Category 4-b-ii).* The following systems modify the Android middleware to implement interception mechanisms so that they can enforce more fine-grained policy models. These systems choose to enforce their policies at the Android middleware since they consider it as the suitable enforcement level, which provides relevant semantics concerning Android app permissions.

Apex [Nauman et al. 2010] imposes runtime constraints on a permission. The constraints can be set, for instance, based on a device's location, the time of the day, and the number of times a resource has been accessed. Hence, a user can specify, for example, the number of SMS messages that can be sent in a day. Apex allows the user to selectively grant permissions to apps either during or after app installation.

CRêPE [Conti et al. 2010, 2012] implements context-based permissions on Android.[9] In CRêPE, a context can be defined based on (i) the status of variables sensed by physical sensors, such as location, time, and light; (ii) additional processing on these data via software sensors; and (iii) local and remote interactions with the user or authorized third parties, respectively. A policy entry, meanwhile, specifies the actions to be taken such as activating or disabling a system resource like the camera. The user can associate contexts with each policy, which can then be applied to apps at either runtime or install time. Since there could be several policies with conflicting actions that are active at the same time, CRêPE performs a conflict resolution by selecting policy(-ies) with the highest priority.

Ongtang et al. [2010] propose Porscha to specifically protect DRM-based contents (e.g., MP3-based MMS, SMS, or email) as they are delivered and subsequently accessed on a device. To secure content delivery, Porscha protects the confidentiality of the contents by using an infrastructure built upon the identity-based encryption (IBE). To secure content access on a device, it enforces access policies by both proxying content channels (e.g., POP3, IMAP, Active Sync) and placing reference monitor hooks within

[9]CRêPE makes a very small modification on the Linux kernel related to communication with iptables to mediate network traffic. Yet it implements almost all of its policy infrastructure at the Android middleware.

the Android's Binder ICC framework. As a result, it can enforce the protected contents to be accessible, for instance, by only authorized phones or for a certain time period.

*Discussion*. Table III also contrasts Apex, CRêPE, and Porscha. One useful common feature of both Apex and CRêPE is their runtime policy setting, which allows for dynamic constraint or context setting by the user. Ensuring the security of the policy setting module, together with its policy database, is therefore very important.

Incorporating contexts into a policy, as implemented by CRêPE, provides transparent policy activation and deactivation. Such automated policy reconfigurations, however, might leave the user confused with regard to which context and associated policies are active at a particular time. To deal with this situation, the user must be able to inspect the active contexts and corresponding policies, and possibly override them if he or she wishes. Porscha secures the delivery and access of protected contents. Its mechanisms are, however, specific to the considered DRM-based scenarios. An alternative option is to provide generic mechanisms at the Android middleware that can also achieve secure content delivery and access by means of a suitable policy.

*7.2.3. Systems Running on Unmodified Android (Category 4-b-iii).* The systems in category 4-b-ii modify the Android framework. Unless the proposed extensions are adopted by either Google, device manufacturers, or telecommunication carriers, rooting of a device by its user is required. To avoid this potential hindrance to seamless mass deployment, several systems opt to enforce their policy on unmodified Android. They take an inline reference monitor (IRM) approach, where an app is rewritten so that the code that monitors important operations is embedded into the app itself.

A proposal by Jeon et al. [2013] suggests a new app-market model where the market adds instrumentation codes into all of the apps that it hosts. In this way, all capability-sensitive APIs are changed into proxy APIs so that the apps' runtime behavior now can be monitored and controlled by the user. However, there is no detailed mention in the (short) work on how the instrumentation and monitoring are done.

Aurasium [Xu et al. 2012] implements a similar concept of an automated app rewriting by an external party. When a user obtains an untrusted app, he or she pushes the app to an Aurasium system to get a hardened repackaged app. This app contains sandboxing code within itself to perform both reference monitoring and policy enforcement. Aurasium performs code interposition at the Android's libc level by rewriting function pointers to the libc library. Accordingly, the inserted code contains two types of code: Aurasium's native library for the low-level interposition and high-level Java code that implements the policy logic. In this way, Aurasium is able to interpose interactions between the app and the OS, with the exception of those due to native code that reimplements the libc functionality. More fine-grained runtime policy can be enforced on rewritten apps, such as by alerting the user for approval of sensitive operations.

In addition to the self-contained rewritten apps just described, there is another possible deployment mode of Aurasium-repackaged apps on a device. This second mode runs a system-wide Aurasium Security Manager module, which centrally handles the policy decisions of all repackaged apps on the device. With this module, policy logic can thus be globally controlled at a single enforcement point. Update of the module instance on the device can also be done more easily. Higher overhead due to ICC is, however, incurred in this mode.

Jeon et al. [2012] propose an app rewriting system that derives and enforces more fine-grained policies on Android apps. Its module, called *RefineDroid*, statically analyzes target apps to infer their permission usages and then generates the fine-grained variants of the permissions. Another module, called *Dr. Android*, rewrites the apps to use these more fine-grained permissions. Dr. Android also modifies the apps' manifest files to remove any existing permissions, including those required by native code, and

then adds the newly derived fine-grained permissions. Alternatively, at their development stage, app developers can directly enforce fine-grained permissions on their apps using a supplied client-side library called *hidelib*. All of the reformatted apps can run on an unmodified Android device enforced by the Mr. Hide module. Mr. Hide runs as a set of Android services, each accessing a single fine-grained permission. A conducted evaluation shows that RefineDroid, in generating the more fine-grained permission variants, produces relatively low false negatives (i.e., its failures to discover fine-grained permissions). RefineDroid's false positives, which quantify fine-grained permissions that are reported used by an app but were not used during testing, are still quite high, however.

Davis et al. [2012] propose I-ARM-Droid as another framework for embedding in-app reference monitors into Android apps. I-ARM-Droid manipulates Dalvik bytecode that is converted to an intermediate format using the baksmali tool. One unique feature of I-ARM-Droid is that custom behavior for each target method can be specified in Java.

Bartel et al. [2012] examine the feasibility of performing a complete on-device (in-vivo) app instrumentation, hence eliminating the need for an external PC to do the app transformation. This in vivo instrumentation system converts an app's Dalvik bytecode into Java bytecode using dex2jar, then instruments the Java bytecode using Soot as well as a more lightweight ASM tool.[10] The authors report their experimentation using a set of 130 apps. Whereas the conversion steps could be successfully done on a PC, only 2.3% apps (using Soot) and 30% apps (using ASM) could be successfully instrumented on a smartphone used in the experimentation. The authors attribute these low conversion success rates to the limited memory capacity of the used smartphone, which oftentimes caused the conversions to crash.

Backes et al. [2013] outline AppGuard, which realizes the idea of a fully automated on-device app rewriting system with a high instrumentation success rate. They report that AppGuard successfully rewrote 99.3% of the tested 25,482 apps, and its implementation had been downloaded by more than 1M users. AppGuard performs its interposition at the Java method level. It rewrites the Dalvik VM's virtual method table, which holds the references to the bytecode of all methods. Hence, similar to Aurasium, AppGuard implements a callee-site rewriting, where the transfers of an app's execution control to the reference monitor occur at the beginning of each target method. This is in contrast to caller-site rewriting systems, which redirect the control from each callsite of target methods within the app code. Due to its interposition at the Java level, AppGuard is able to handle Java reflection and dynamic code loading. It also comes with a policy language called *SOSPoX*, which enables constraint-based specifications on method call executions, control flow changes, and data secrecy. The use of this high-level policy language allows AppGuard to instantiate different use cases, such as revoking the previously granted permissions of the apps.

*Discussion*. App rewriting systems can be differentiated and subsequently analyzed based on their two following key aspects. The first is how the rewriting is performed to interpose privileged operations. The second is concerned with the enforced policy and how the user interacts to configure it. Table III shows how the surveyed systems operate and interact with the user in enforcing their policy. Table IV contrasts several design options of app rewriting operation that are taken by different systems. As can be seen in Table IV, there are different possible approaches to performing app rewriting, of which some are discussed by Davis et al. [2012]. Features of several existing systems are also compared by Backes et al. [2013].

[10]This ASM tool (http://asm.ow2.org), a Java bytecode manipulation and analysis framework, should not be confused with the ASM framework surveyed in Section 7.2.1.

Table IV. Comparison of App-Rewriting Systems (in Category 4-b-iii) on Their Rewriting Process

| System | Code Representation for App Rewriting | Instrumentation Location | Ref. Mon. Location | Java Reflection | Policy Language | Additional Operations |
|---|---|---|---|---|---|---|
| Jeon et al. [2013] | (No information given) | App market | Inline | (No information) | No | |
| Aurasium [Xu et al. 2012] | Disassembled Java bytecode (output of apktool) | Aurasium system | Inline or Aurasium Security Module | Yes | No | Add native code containing Aurasium's libc interposition routine into a rewritten app |
| Dr. Android & Mr. Hide [Jeon et al. 2012] | Dalvik bytecode | External PC | Mr. Hide | Partial | No | Remove an app's existing permissions, derive and declare the more fine-grained permissions |
| I-ARM-Droid [Davis et al. 2012] | Disassembled Dalvik bytecode (output of baksmali) | External PC | Inline | Partial | No | Abort a rewriting process if native code is detected |
| In vivo [Bartel et al. 2012] | Transformed Java bytecode (output of dex2jar, Soot/ASM) | Device (with success rate of 3%–30%) | Inline | No | No | Port Soot/ASM into Android for on-device app rewriting process |
| AppGuard [Backes et al. 2013] | Disassembled Dalvik bytecode (output of extended dexlib/smali) | Device (with success rate of 99.3%) | Inline | Yes | Yes | Warn users if a rewritten app executes untrusted native code, enable multiple use cases |

For the first key aspect, one main choice is how the interposition is performed and at which software level. Hao et al. [2013] give a systematic evaluation on the effectiveness of several app rewriting mechanisms in achieving API-level access control. They also identify a number of potential attacks on the mechanisms. For all rewriting systems, native code remains a big issue to achieving a complete mediation (weakness $W_7$).

Another question is whether the reference monitor should be put inside the rewritten apps or implemented in another separate app. Some researchers view that in-app reference monitor results in easier rewriting and faster execution with no ICC involved [Davis et al. 2012]. However, others argue that a separate monitor service for each privileged operation can facilitate better audit for correctness [Jeon et al. 2012]. To clearly answer this question, deeper analysis and empirical experimentation are needed. Yet all IRM-deploying systems surveyed in this category are more prone to security compromise since the monitors run at the application level. In other words, they are not more powerful than the apps they attempt to protect [Bugiel et al. 2013]. Last, there exists a question of the most amenable form of IR for easy app rewriting. Many systems use smali code that is produced by the baksmali tool, which is also utilized by apktool. A Soot-based rewriting system, meanwhile, utilizes Soot's available IRs, such as the commonly used Jimple. This question warrants an investigation to compare the available IR options in facilitating easy rewriting with a near-perfect success rate.

With regard to policy enforcement as the second key aspect, most of the surveyed systems consider only a simple model of regulating permission-based operations. At runtime, these systems will alert the user whenever a privileged operation is about to be executed by a rewritten app. The user typically has an option to always approve, approve once, or abort the operation. The exception is AppGuard, which comes with its own policy language that can instantiate different use cases. Nonetheless, in principle, it is possible for all surveyed systems to implement various different policies. In fact, both I-ARM-Droid and the in vivo rewriting system have a feature to embed custom behavior into the intercepted methods during their rewriting step. Policy languages, however, bring much more flexibility to supporting various policies.

Notice that Android allows app updates only if the updates are signed by the same developer. Since all surveyed systems that we have analysed modify the original apps and then re-sign the rewritten apps, updates of the original apps need to be properly addressed. The app-rewriting systems need to keep track of the pertinent app market(s) for any app updates and subsequently perform the necessary rewritings and installations of the updates.

### 7.3. Preventing Privilege-Escalation Attacks (Category 4-c)

This section analyzes runtime ICC-regulating systems that prevent privilege-escalation attacks, which are possible due to Android weaknesses $W_5$ and $W_6$. Bugiel et al. [2011a, 2012], who specifically analyze the privilege-escalation attacks, suggest a more elaborate taxonomy for classifying existing solutions in this area. For our purpose of surveying the overall Android security field, we adopt a simpler model that divides the proposed solutions into the two following broad categories.

*7.3.1. Preventing Confused Deputy Problem (Category 4-c-i).* Confused deputy problem occurs when a malicious app (as requester) manages to cause a benign app (as deputy) to execute its privileged API without the user's consent. As discussed earlier, one possible cause of this problem is the unintentional exposure of an app's private components by its developer ($W_6$). One common source of confusion that may cause such exposures is the Android's implicit export of a component in the presence of an intent filter declaration. Kantola et al. [2012] suggest a platform-oriented solution to avoid exporting components that are most likely intended for internal communication. They propose

heuristics for the Android platform to export components in a more conservative manner. The authors also develop a static analysis tool called *IntraComDroid*, which is built on top of ComDroid, to evaluate the proposed heuristics in terms of its backward compatibility and security gain. Similar platform modifications as recommended by Chin et al. [2011] can also help app developers better distinguish between internal and external ICC. In what follows, we focus on systems preventing the confused deputy problem that can still arise despite the absence of unintentionally-exported components of the interacting apps.

Although Saint [Ongtang et al. 2009, 2012] is not specifically devised to deal with ICC-based attacks, it regulates interapp ICC and thus can be useful to prevent the attacks. Saint enforces policies that are supplied by app developers to protect their apps. The policies are defined based on an app's various information or a device's runtime state. To be executed, an ICC must satisfy all applicable runtime policies that are asserted by both the caller and callee apps. If no such policies exist, the ICC is implicitly allowed as Saint adopts a default-allow policy. Thus, the onus lies on app developers to specify all policies that must be satisfied by other interacting apps.

Felt et al. [2011c] propose IPC Inspection to deal with the confused deputy problem upon finding a high prevalence of vulnerable apps. Among the 872 apps that they examined, 320 were found to be at risk. Furthermore, using their static analysis tool, Felt at al. managed to craft attacks on 5 of the 16 apps that come preinstalled with Android 2.2. IPC Inspection works by always reducing a deputy's permissions into the *intersection* of the deputy's permissions and those of its requester. Although this mechanism sounds simple, several issues exist with regard to its enforcement. First, a requester will need more permissions than those required for its own functionality to additionally include the permissions of any deputies that it may call. Such upper-bound permissions, however, may not always be obvious to infer. Even if they are, it will lead to the permission bloat issue. Second, for a deputy that serves multiple requesters, over time its permissions will be reduced into the lowest among its requesters. To prevent this, IPC Inspection takes a *polyinstantiation* approach that works as follows. For a nonsingleton deputy, where multiple app instances are allowed to run on a device, IPC Inspection always launches a new instance of the deputy whenever the requester has lower permissions than the deputy. For a singleton deputy, however, the deputy's permissions would be reduced over time, and it may possibly crash at one point. Consequently, a good exception handling on the deputy is required. Third, the possibility of bidirectional interapp communication conversely requires a deputy to also have the permissions of all of its requesters. Under a request-reply interaction, IPC Inspection intentionally chooses not to reduce the requester's permissions to deal with this issue.

QUIRE [Dietz et al. 2011] takes a different approach to the problem by establishing a provenance-carrying ICC that is based on a call-chain tracking technique. It modifies the Android's Java runtime libraries and Binder ICC to establish a calling context so that an ICC call chain can be formed. With this infrastructure, recompiled QUIRE-aware apps can propagate the call chain context to their deputies. A deputy then has a choice either to operate with the reduced permissions of its requesters or to exercise its full permissions as an intentional deputy. QUIRE thus takes a discretionary, developer-centric approach as opposed to a mandatory, system-centric one adopted by IPC Inspection.

*7.3.2. Preventing Collusion Attacks (Category 4-c-ii).* Privilege-escalation attacks can also be mounted by colluding apps that take advantage of shared covert channels, such as the content provider, file system, and network. XManDroid [Bugiel et al. 2011a] builds a system-centric monitoring mechanism to regulate both ICC and shared channels.

Table V. Comparison of Continuous Runtime Malware Detection Systems in Category 4-d

| System | Feature Set | Detection Scope | Signature/Classifier | Processing Location |
|---|---|---|---|---|
| ***Signature-Based Detectors (Category 4-d-i):*** | | | | |
| KBTA IDS [Shabtai et al. 2010c] | System parameters & events (160) | System | KBTA-based signature | Local |
| Paranoid Android [Portokalidis et al. 2010] | App execution trace | App | (Based on) ClamAV antivirus' signature | Remote |
| ***Anomaly Detectors (Category 4-d-ii):*** | | | | |
| Andromaly [Shabtai et al. 2012] | System parameters & events (88) | App | Several machine learning techniques | Local |
| Crowdroid [Burguera et al. 2011] | System calls | App | 2-means clustering algorithm | Remote |

It observes all communication links between apps and verifies them against a set of policy rules. In addition, it tracks entries written on shared channels according to their writers and filters accesses to these channels. Throughout its operation, XManDroid maintains a system view, which is internally represented as a system graph. It ensures that this graph always complies with the enforced policy. XManDroid is able to detect SoundComber malware [Schlegel et al. 2011]. Yet it requires precise policy engineering to minimize false positives, which were observed in its reported experimentation.

Bugiel et al. [2012] improve XManDroid by additionally validating ICC using an intent-tagging based call chain technique and by enforcing a kernel-level MAC. These system-centric extensions utilize a high-level policy language at the Android level and adopt TOMOYO Linux[11] for access control enforcement at the kernel level. A dynamic policy mapping is built to bridge the semantic gap between policies at the two levels.

*7.3.3. Discussion on Systems Preventing Escalation Attacks.* From the surveyed systems presented, we see two different approaches in preventing the confused deputy problem: an app-level approach as taken by QUIRE and a system-wide mechanism as taken by IPC Inspection. The two approaches face different challenges as discussed earlier.

With its elaborate mechanisms, XManDroid and its extension system can prevent collusion attacks. However, outlining precise security rules for regulating all possible interactions among installed apps on a device is difficult. Moreover, strict policy enforcement may cause some intended interapp ICC to be rejected. Last, Marforio et al. [2012] show that fully preventing privacy leaks by colluding apps is hard. They analyzed a number of overt and covert channels at different system levels and reported some attacks that went undetected by both XManDroid and TaintDroid.

### 7.4. Runtime Malware Detectors (Category 4-d)

We now survey systems that detect malicious activities of installed apps as they are running on a device. By following the tradition in the intrusion detection system (IDS) field, we can divide these systems into signature-based (misuse) detectors, which look for known malware patterns/signatures, and anomaly-based (behavioral-based) detectors, which look for deviation from normal profile.

We note that the dynamic analysis techniques surveyed in Section 5.1 can be adapted into continuous detection systems. In what follows, we survey additional systems that are specifically built to detect malicious activities during an app's runtime execution on a device. Table V summarizes several key properties of these systems.

[11]http://tomoyo.sourceforge.jp/index.html.en.

*7.4.1. Signature-Based Detectors (Category 4-d-i).* Shabtai et al. [2010c] propose an IDS based on knowledge-based temporal abstraction (KBTA) technique. KBTA allows for the reasoning with context-aware temporal abstractions of time-stamped data. In the context of Android security, one can employ KBTA to describe malicious activity patterns, such as "there is an access to SD-card event, followed by a trend in which outbound packet traffic increases, while there is no user activity related to the app." The proposed IDS periodically analyzes 160 features defined from various system parameters and events. At every sampling interval, the IDS creates and ends states, trends, and contexts. An evaluation was done using five self-written apps against several defined KBTA-based attack patterns. A detection rate higher than 94% was reported.

Paranoid Android (PA) [Portokalidis et al. 2010] is proposed as a cloud-based detection framework, where a detection server runs a synchronized replica of a monitored device in a virtual environment. Since this server has more computation resources compared to the device, it can therefore afford to run a complex detection algorithm, possibly multiple ones simultaneously. The implemented PA prototype performs execution tracing on a device, transfer of logs to the server via an encrypted channel, and the server's replaying of the monitored device. To reduce its log size, the PA system records only system calls that introduce nondeterminism, as well as nondeterministic inputs and events. To reduce the overhead of log transfer, it adopts a loose synchronization strategy. The log is transmitted only when a device is awake and connected to the Internet, which is argued by the authors as the time the device is most vulnerable to attacks.

In its reported implementation, the PA's detection server performs periodical file scans using a ClamAV based antivirus system. It also applies dynamic taint analysis to detect various memory corruption attacks. The experimentation shows that during periods of high activity such as browsing, 2.5KiBps of log was produced and up to 30% of battery drop was observed. We note that the main focus on the work is its cloud-based replica monitoring infrastructure rather than deploying specific detection techniques.

*7.4.2. Anomaly Detectors (Category 4-d-ii).* Shabtai et al. [2012] outline Andromaly, which applies machine learning–based anomaly detectors on runtime information collected from a device. An evaluation done using several self-written malware samples show that the naive Bayes technique outperforms other techniques in cross-device testing scenarios with 86% to 88% accuracy.

Crowdroid [Burguera et al. 2011] makes use of a lightweight client module to monitor Linux system calls that are invoked by an app. This client sends the processed output from the `strace` tool to a centralized server. The server then compares the submitted log against a normal profile that is constructed from logs submitted by multiple users who run the same app. Crowdroid therefore adopts a crowdsourcing paradigm, where the normal profile of an app is derived from a large number of users running benign executions of that app. The server performs 2-means, which is a binary partitional clustering algorithm, on the system call frequency vector of each app. However, this detection technique gives rise to false positives, which is also observed in the reported experimentation. It is also susceptible to evasion attacks.

*7.4.3. Discussion on Runtime Malware Detection Systems.* Similar to the systems performing malware detection in category 2-a, detection systems that continuously monitor app executions face the same problem of developing accurate normal profiles or signatures that incur acceptably low false positives. In addition, an efficient detection algorithm is desirable so that a device's performance is not significantly degraded. Last, it is preferable to have detection systems that monitor an app's execution in real time so that they can prevent intrusions rather than simply detect past attacks as in Andromaly, Crowdroid, or PA (due to its loose synchronization).

### 7.5. Isolation Systems (Category 4-e)

We now discuss systems that perform isolation for security reasons.

*7.5.1. Isolation and Virtualization Solutions (Category 4-e-i).* Virtualization has long been known in desktop environments to provide a strong execution-environment separation, thereby improving application protection. A similar but more lightweight solution is domain isolation, where apps are assigned into specific execution domains and communication among domains is restricted. Systems based on virtualization or domain isolation can be further divided into the three subcategories that follow, from a lightweight to a full virtualization approach, as suggested by Bugiel et al. [2011b].

*Multiple domains on a single kernel and single Android middleware*. TrustDroid [Bugiel et al. 2011b] supports multiple execution domains on top of a single Android software stack. In TrustDroid, each domain is associated with a trust level. For a considered usage scenario that separates personal and corporate apps, three domains are defined: system (for preinstalled system apps), trusted (for certificate-protected corporate apps), and untrusted domains (for apps from public sources). Communication between apps in trusted and untrusted domains are not allowed. To restrict interdomain interactions, TrustDroid modifies the following software layers: (i) Android layer, by utilizing XManDroid [Bugiel et al. 2011a] to prevent interdomain ICC and data access; (ii) kernel layer, by using the TOMOYO Linux to enforce MAC on file system and IPC channels; and (iii) network layer, by using netfilter to mediate network traffic.

*Multiple domains on a single kernel and multiple middleware instances*. Systems in this category have a benefit of achieving stronger isolation than TrustDroid surveyed earlier. Wu et al. [2014] developed a system that confines untrusted apps by putting them into a special isolated domain called *AirBag*. An added software layer called *App Isolation Runtime* (AIR) functions as a second Android middleware that manages the untrusted apps within this AirBag domain. However, it still allows the untrusted apps to interact with the standard Android runtime, which hosts trusted apps, when the untrusted apps run in a *normal* execution mode. In this case, the AIR simply proxies related API calls to the standard Android runtime via an authenticated channel. By default, however, the untrusted apps run in an *incognito* mode. In this mode, all private information about the phone or user are falsified by the AIR. Moreover, accesses to sensitive operations are allowed only after user approval is obtained. To support both the standard Android and AirBag runtime environments, the extended kernel multiplexes accesses to system resources. It transparently provides a separate namespace and system resource virtualization for the untrusted apps within the AirBag domain.

*Multiple domains on multiple kernel instances*. A full virtualization approach runs a separate kernel instance for each Android middleware by using a hypervisor or virtual machine monitor (VMM). This approach thus provides the highest degree of isolation.

Lange et al. [2011] outline an Android virtualization framework under the L4Android open-source project.[12] The framework is built upon the microkernel-based Fiasco.OC and the associated runtime environment L4Re. On top of this infrastructure, two possible execution environments for Android are available, namely L4Linux kernel[13] and Karma VMM.

Commercial VMM-based solutions are also available, such as the Mobile Virtualization Platform (MVP) from VMWare [Barr et al. 2010] and OK:Android from Open

[12]http://l4android.org.
[13]http://l4linux.org.

Kernel Labs.[14] MVP is specifically devised to address a Bring Your Own Device (BYOD) scenario. In MVP, an employee runs a virtualized corporate domain in addition to his or her own personal domain. The kernel of this corporate domain, however, runs as a guest OS *on top of* the personal domain's kernel. As a result, the security of the corporate domain depends on that of the host's kernel. Taking another approach, OK:Android provides an OS support package to run multiple Android systems as guest OS instances on top of its OKL4 virtualization platform.

*Discussions on isolation and virtualization works*. From the surveyed systems analyzed earlier, we can see how different virtualization categories provide increasingly stronger isolation with the expense of more processing requirements. We also can see that they offer attractive solutions to app isolation, including solutions to the BYOD problem. Yet there exist issues to be dealt with before proposed solutions can be widely adopted. First, we may need several levels of solutions, from lightweight app isolation to full virtualization solutions, that can run with satisfactory performance to suit different deployment requirements. Second, these solutions need to be easily configured to meet the users' dynamic requirements or different use cases. Third, the solutions must come with an easy-to-use user interface for seamless app, domain, and device administration.

*7.5.2. Advertisement Library Protection Systems (Category 4-e-ii).* Advertisement plays an important role in the Android app ecosystem as a source of revenue of many freely distributed apps [Enck et al. 2011; Shekhar et al. 2012]. In Android, ad libraries invoked by an app share the same process and permissions as the app, thereby causing security threats [Pearce et al. 2012]. This problem arises due to Android's present lack of a separation or isolation mechanism for ad libraries ($W_4$). The targeted goals in this category are therefore [Zhang et al. 2013a]: (i) *process isolation*, to place an app's components related to ad libraries in a separate protection domain from that of the host app; (ii) *permission isolation*, to assign these components a separate set of permissions; (iii) *display/output isolation*, to ensure a proper display of ad libraries; and (iv) *input isolation*, to protect the libraries from fraudulent clicks. Ad library protection can be viewed as a form of app isolation system. However, unlike systems in category 4-e-i that isolate distinct apps, ad protection systems separate and isolate two "subapps" that come packaged as a single app and request a combined set of permissions.

Pearce et al. [2012] propose AdDroid API as a new, unified mobile-ad API. It supports the insertion of ad components into an app's UI, as well as data relay between the app and a newly added AdDroid system service. AdDroid also introduces two new Android permissions, namely `ADVERTISING` and `LOCATION_ADVERTISING`, to be acquired by ad-displaying apps. Google, however, has to adopt AdDroid. In addition, app developers need to update their ad-displaying apps to use the new API.

AdSplit [Shekhar et al. 2012] splits an app and its ad display into two distinct activities, which run within two separate Linux processes. The window for the ad activity is layered just beneath that for the host activity. This host activity contains transparent region(s) where the ad will be displayed and visible to the user. AdSplit also leverages QUIRE [Dietz et al. 2011] to provide provenance evidence of user-generated UI events. As a result, the ad activity can validate that UI clicks occur on its activity region(s) without the need to trust the host activity. The use of the transparency technique and stub library in AdSplit is, however, not without some potential issues [Zhang et al. 2013a].

AFrame (Activity Frame) [Zhang et al. 2013a] implements the notion of browser *iframes* in the context of Android activities. It allows an activity to embed another

[14]http://www.ok-labs.com.

activity and achieves the four aforementioned goals. With AFrame, app developers can still utilize any existing mobile ad SDKs. However, they must rewrite their ad activity to load its code into an AFrame region, which is set aside within the host app's UI layout.

Roesner and Kohno [2013] consider a more general problem of securing cross-app interface embedding. The problem is concerned not only with UI objects from ad libraries but also with social network plugins and access control gadgets. In this scenario, more stringent security requirements than those listed earlier are required, including size manipulation prevention and ancestor redirection. They build LayerCake, which enhances the Android platform to meet the specified requirements. Similar to AFrame, LayerCake implements the notion of iframes, which allows embedded activities to run together. It additionally imposes a stricter model of cross-app interaction regulation.

*7.5.3. Native Code Isolation (Category 4-e-iii).* As the number of Android apps keeps increasing, the use of third-party native libraries, which could be malicious or buggy (weakness $W_7$), is expected to be more prevalent as well [Sun and Tan 2014]. NativeGuard [Sun and Tan 2014] isolates third-party native libraries of an app by rewriting the app and putting its native libraries into a separate second app. This second app runs with lesser privileges in the background and responds to requests from the host app via the Android's ICC mechanism. To achieve transparent cross-app library invocations, proxy libraries are inserted into the host app. Likewise, within the second app, stub libraries are added. Native code isolation by NativeGuard represents a useful measure to control native code in Android, which is a step forward toward adapting various native code control solutions that exist in the Java environment [Sun et al. 2013].

### 7.6. Secure App Enablers (Category 4-f)

In addition to systems surveyed so far, there exist works that aim to provide robust infrastructures for secure app-level operations on Android devices. Although some of these works also report app-level vulnerabilities, we opt to discuss them here since they additionally propose systems or measures to strengthen app security.

Fahl et al. [2012] implement MalloDroid, which statically analyzes apps that make use of SSL/TLS. Among 13,500 tested popular free apps, MalloDroid found that 1,074 (8.0%) of them contain SSL/TLS code that is potentially vulnerable to Man-in-the-Middle (MitM) attacks. MalloDroid, however, still requires manual analysis to ascertain the discovered potential vulnerabilities. To address this limitation, SMV-Hunter [Sounthiraraj et al. 2014] combines both static and dynamic analyses to achieve a fully automated, large-scale detection of SSL-based MitM vulnerabilities.

Fahl et al. [2013] suggest a new usage model of how SSL should be handled in app-oriented environments. They implement their platform-based measures in Android that free app developers from writing their own SSL-related code. Instead, developers only need to set configuration options if they intend to deviate from the standard use case. Using the proposed measures, Fahl et al. show how all SSL use cases can be deployed in a more secure manner.

Noticing a lack of strong origin-based protection in mobile platforms, Wang et al. [2013] analyzed the security risks of cross-origin communications among mobile apps and those between an app and Web services. They find various cross-origin attacks, which can allow an attacker, for instance, to obtain the user's authentication credentials from several tested popular sites. Wang et al. then proposed Morbs, which labels every message with its origin information and allows app developers to specify origin-based policies on their apps using a whitelisting approach. Morbs thus extends a message-provenance mechanism, similar to one in QUIRE [Dietz et al. 2011], to also cover Web origins and protect Web resources.

Poeplau et al. [2014] analyzed potential issues that can come from an app's ability to dynamically load additional code from external sources at runtime. They find that benign apps that perform dynamic code execution may be subject to injection attacks due to their insecure ways of downloading and loading the code. Poeplau et al. then proposed a modification to Android to always check the integrity of loaded code by means of a whitelist-based verification server.

Egele et al. [2013] evaluated the usage of cryptographic APIs by Android apps. They aimed to find out if the apps violate six commonly known usage rules of symmetric and password-based encryption schemes. A developed static analysis tool, called *CryptoLint*, found that 10,327 out of the 11,748 evaluated apps made at least one violation. Egele et al. subsequently proposed steps to avoid the observed insecure practices.

Last, there is always an issue of patching vulnerable apps or OS on Android devices, particularly those running an outdated Android version that is poorly or no longer supported by the manufacturer or mobile carrier. PatchDroid [Mulliner et al. 2013] distributes and applies third-party security patches to Android devices. It employs dynamic in-memory patching techniques to inject patches into the affected running processes. AppSealer [Zhang and Yin 2014] addresses an issue of automatic patch generation on component hijacking vulnerabilities (see also Section 5.4). It first translates an app into a Jimple IR using dex2jar and Soot. Static dataflow analysis, app rewriting, and several optimization phases are then successively performed to generate the patched version of the app. This automatic patch generation, distribution, and application, as performed by AppSealer and PatchDroid, open up a new alternative solution to address the Android fragmentation issue and its security implications.

## 8. INSTALLED-APP RECHECKING SYSTEMS (CATEGORY 5)

At times, installed apps on a device may have their security settings modified due to the following reasons: (i) the apps get updated with different security attributes, such as a new set of permissions, and (ii) the user modifies the apps' security settings. The latter may occur, for instance, when the user uses AppGuard, Apex, or Android App Ops to revoke permissions that have been granted previously. Another example is when an app is moved into a different execution domain, such as when an untrusted app in AirBag (see Section 7.5.1) is put into a normal execution mode. We now turn our focus on systems that reanalyze installed apps following changes to their security settings.

One interesting system in this category is Pyandrazzi [Kennedy et al. 2013], which analyzes how an app would behave if the user revokes previously granted permissions. Hence, while AppGuard, Apex, and Android App Ops implement the revocation feature, Pyandrazzi evaluates how removing individual permissions may impact the app's behavior. Using automatically supplied UI events, Pyandrazzi detects when the app crashes and pinpoints its cause. Pyandrazzi applies its analysis on a desktop computer. The challenge is thus how to make such a system run on a device, or to convey the analysis results to mobile users should a cloud-based analysis system be employed.

In general, to reanalyze the security implications of an app's setting modification, one could extend an on-device solution like Kirin [Enck et al. 2009a] to reapply its analysis on the app following the security setting modifications described earlier.

## 9. REMAINING CHALLENGES AND FUTURE DIRECTIONS

We now mention the main remaining challenges in securing Android devices and possible research directions that may be taken by future work.

### 9.1. App-Hardening Systems (Category 1)

To make reverse engineering on apps more difficult and to deter app repackaging attacks, strong app obfuscation tools need to be made readily available to app developers.

The tools should implement various Java transformations [Collberg et al. 1997], including techniques that are usually employed by advanced malware to protect themselves, such as literal encryption. The tools may also protect all resources of an app, including its string values and UI layouts.

To increase the deterrence against app repackaging attacks, efficient and robust static or dynamic watermarking techniques are always useful.

Given various findings of app vulnerabilities, the availability of static analysis tools that work at the source code level and help identify unsafe programming practices leading to vulnerabilities will benefit app developers. The vulnerabilities to be prevented include those related to confused deputy problem, component hijack, illegal content-provider access, dynamic code loading, insecure SSL handling, and cryptographic API usage. A tool that can statically determine the permissions required by an app should also be provided. In this way, the permission bloat issue, which makes privilege-escalation attacks more damaging, can be prevented.

Last, to provide a stronger app authorship proof, one may wish to explore the idea of deploying certificate-based PKI signing, with trusted CAs as the root of trust. Although certificate-based app signing should not be made mandatory to maintain operational compatibility, it will definitely provide strong authorship assurance on apps that come from established developers.

### 9.2. Untrusted App and Market Analyzers (Category 2)

*9.2.1. Using Static Analysis.* In this area, the goal remains to provide accurate static analysis tools that uncover malicious operations of untrusted apps. The tools must be resistant against transformation attacks and satisfactorily address Android-specific analysis requirements [Arzt et al. 2014; Yang et al. 2013]. Ideally, they should be able to detect zero-day malware samples in addition to known ones. Static analysis on native code may also be attempted by taking into account Android-specific execution patterns, including those related to the Android permission and user interaction models.

Owing to the prevalence of vulnerable apps, static analysis tools that work on Dalvik bytecode to discover vulnerabilities remain necessary. Among various types of vulnerabilities, ICC-based vulnerabilities seem to be the most subtle and prevalent ones. Hence, modeling and formalizing various subtle aspects of ICC, such as its inter-component access control and data sharing, would be useful to make the analysis more grounded.

To detect app repackaging, we may apply existing program similarity detection techniques in the Android context. To be effective, the adapted techniques must take advantage of common Android code patterns and permission usage model. Ideally, not only should the techniques cluster similar apps, they should also give insight into the differences of similar apps within the same cluster. This will allow app derivation and mutation to be observed and investigated.

Last, there is always a need to make static analysis techniques more tractable when applied to Android apps by taking advantage of their Android-specific properties.

*9.2.2. Using Dynamic Analysis.* To dynamically analyze apps, we need a highly realistic analysis platform that is sufficiently resistant against evasion attacks, including those pointed out by Vidas and Christin [2014]. Periodic randomization on private information contained by the emulator is necessary to protect such analysis platforms.

New techniques of performing comprehensive UI exploration are expected. The techniques must increase the code coverage of the current state, possibly by making use of well-suited heuristics, image recognition, or machine learning techniques. Efforts that utilize static analysis to trigger certain execution paths, especially the hardly reachable ones, are crucial to complement the UI exploration–based techniques.

### 9.3. Install-Time App Checking Systems (Category 3) and Rechecking Systems (Category 5)

In these two areas, we require lightweight on-device tools that can detect the security implications of installed apps. The tools must rerun their analyses following app installations, updates, and significant security setting changes. Such tools may apply a lightweight static analysis on an app's Dalvik bytecode. The availability of lightweight static analysis toolkits directly on Android platform will therefore be valuable.

### 9.4. Continuous Runtime Monitoring/Enforcement Systems (Category 4)

*9.4.1. Strengthening Linux Kernel.* To ensure the security of the underlying Linux kernel, constant analysis needs to be done to investigate how the kernel potentially could be compromised. The principle of least privilege needs to be observed. This includes limiting the exposure of shared resources at the kernel level that could be useful to spying malware.

The availability of the ASM framework [Heuser et al. 2014] makes the implementation of Android-level security policy much easier. There should be a follow-up effort to realize a robust and efficient implementation of ASM. One issue is how ASM can be provided on various Android versions, should Google decide not to adopt the approach. In this case, although its impact will be limited, ASM can still be useful for security researchers and analysts to experiment in implementing their security schemes.

*9.4.2. Deploying Effective Fine-Grained Access Control.* As permissions are important, proposals to improve Android permissions are always relevant. The challenge is how to design and subsequently deploy new security improvements that can maintain compatibility with existing apps. For instance, allowing the user to revoke an app's previously granted permissions needs be accompanied by measures that minimize any incompatibility effects. Finally, the need for an effective and easy-to-use policy language that can conveniently derive various desirable use cases still warrants investigation.

*9.4.3. Rewriting an App for Effective Policy Enforcement.* We summarize again several required components of an effective app rewriting system, which still need further investigation and experimentation: (i) an amenable IR and associated tool with a near-perfect conversion rate; (ii) a rewriting scheme that comprehensively interposes sensitive app operations; (iii) a robust reference monitor deployment that may work together with the rewritten apps; and (iv) a convenient mechanism to embed new authorization behavior that implements fine-grained security policy. Furthermore, performing all rewriting processes completely on a device, as achieved by AppGuard [Backes et al. 2013], is always desirable.

*9.4.4. Preventing Privilege Escalation Attacks.* Section 9.1 identifies the need to develop accurate static analysis tools that can help app developers prevent the confused deputy problem. As the next line of defense, platform-oriented measures can be deployed to prevent privilege escalation attacks. In this area, there is a need to analyze how partitioning apps based on their customizable trust levels may help address the issue. The main challenge for systems preventing privilege escalation attacks remains on how to devise an effective policy model and easy-to-use language suitable for limiting ICC. The model should ideally address the concerns investigated by Marforio et al. [2012].

*9.4.5. Detecting Malicious App Execution.* To be effective, an IDS running on an Android device must meet the two following known requirements: (i) an accurate normal profile or signature generationand (ii) an efficient, and preferably real-time, detection algorithm. Satisfactorily meeting these requirements remains an open question.

*9.4.6. Performing App Isolation.* As discussed in Section 7.5.1, high performance and easily configurable virtualization solutions are required for effective domain isolation.

To completely isolate third-party libraries, it is imperative to have a solid framework for separating and containing the libraries. New architectural design that can support clear separation is therefore advocated. Lessons drawn from other similar environments, such as the browser environment, would be very useful. Likewise, effective native library isolation, containment, and monitoring mechanisms are also required.

## 10. CONCLUSION

We have surveyed, categorized, characterized, and comparatively analyzed major existing works in the Android security research field. Our taxonomy and survey of existing solutions have distilled existing efforts in securing Android up to the current date. Our survey additionally highlights the advantages and limitations of the proposed systems, as well as contrasts related systems in achieving their security objectives. With this snapshot of the overall research landscape, we thus hope that the security community can better explore various potential opportunities to further safeguard Android devices, including addressing the identified remaining challenges.

## ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive comments and valuable suggestions that greatly contributed to improving the final version of this article.

## REFERENCES

S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th Conference on Programming Language Design and Implementation (PLDI'14)*. 259–269.

K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. 2012. PScout: Analyzing the Android permission specification. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12)*. 217–228.

M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. 2013. AppGuard—fine-grained policy enforcement for untrusted Android applications. In *Proceedings of the 8th International Workshop on Data Privacy Management (DPM'13)*. 213–231.

K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis. 2010. The VMware mobile virtualization platform: Is that a hypervisor in your pocket? *ACM SIGOPS Operating Systems Review* 44, 4, 124–135.

D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. 2010. A methodology for empirical analysis of permission-based security models and its application to Android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*. 73–84.

A. Bartel, J. Klein, M. Monperrus, K. Allix, and Y. Le Traon. 2012. *Improving Privacy on Android Smartphones through In-Vivo Bytecode Instrumentation*. Technical Report 978-2-87971-111-9. University of Luxembourg, Germany.

A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon. 2014. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing Android. *IEEE Transactions on Software Engineering* 40, 6, 617–632.

A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. 2011. MockDroid: Trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile'11)*. 49–54.

R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall. 2014. Brahmastra: Driving apps to test the security of third-party components. In *Proceedings of the 23rd USENIX Security Symposium*. 1021–1036.

T. Bray. 2011. Identifying App Installations. Retrieved February 10, 2015, from http://android-developers.blogspot.sg/2011/03/identifying-app-installat ions.html.

S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. 2011a. *XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks*. Technical Report TR-2011-04. Technische Universitt Darmstadt, Darmstadt, Germany.

S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. 2012. Towards taming privilege-escalation attacks on Android. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS'12)*.

S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. 2011b. Practical and lightweight domain isolation on Android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'11)*. 51–62.

S. Bugiel, S. Heuser, and A.-R. Sadeghi. 2013. Flexible and fine-grained Mandatory Access Control on Android for diverse security and privacy policies. In *Proceedings of the 22nd USENIX Security Symposium*. 131–146.

I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. 2011. Crowdroid: Behavior-based malware detection system for Android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'11)*. 15–26.

K. Z. Chen, N. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. Wu, M. Rinard, and D. Song. 2013. Contextual policy enforcement in Android applications with permission event graphs. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS'13)*.

E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. 2011. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys'11)*. 239–252.

Cisco. 2014. Cisco 2014 Annual Security Report. Retrieved February 10, 2015, from https://www.cisco.com/web/offer/gist_ty2_asset/Cisco_2014_ASR.pdf.

C. Collberg, C. Thomborson, and D. Low. 1997. *A Taxonomy of Obfuscating Transformations*. Technical Report 148. University of Auckland, Auckland, New Zealand.

C. S. Collberg and C. Thomborson. 2002. Watermarking, tamper-proofing, and obfuscation—tools for software protection. *IEEE Transactions on Software Engineering* 28, 8, 735–746.

M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich. 2012. CRêPE: A system for enforcing fine-grained context-related policies on Android. *IEEE Transactions on Information Forensics and Security* 7, 5, 1426–1438.

M. Conti, V. T. N. Nguyen, and B. Crispo. 2010. CRêPE: Context-related policy enforcement for Android. In *Proceedings of the 13th Information Security Conference (ISC'10)*. 331–345.

J. Crussell, C. Gibler, and H. 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'12)*. 37–54.

J. Crussell, C. Gibler, and H. Chen. 2013. AnDarwin: Scalable detection of semantically similar Android applications. In *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS'13)*. 182–199.

B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. 2012. I-ARM-Droid: A rewriting framework for in-app reference monitors for Android applications. In *Proceedings of Mobile Security Technologies (MoST'12)*.

M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. 2011. QUIRE: Lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Security Symposium*. 347–362.

J. J. Drake, Z. Lanier, C. Mulliner, P. Oliva, S. A. Ridley, and G. Wicherski. 2014. *Android Hacker's Handbook*. Wiley, Hoboken, NJ.

M. Egele, D. Brumley Y. Fratantonio, and C. Kruegel. 2013. An empirical study of cryptographic misuse in Android applications. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*. 73–84.

M. Egele, T. Scholte, E. Kirda, and C. Kruegel. 2012. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys* 44, 2, Article No. 6. DOI:http://dx.doi.org/10.1145/2089125.2089126

W. Enck. 2011. Defending users against smartphone apps: Techniques and future directions. In *Proceedings of the 7th International Conference on Information Systems Security (ICISS'11)*. 49–70.

W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. 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'10)*. 393–407.

W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. 2011. A study of Android application security. In *Proceedings of the 20th USENIX Security Symposium*. 315–330.

W. Enck, M. Ongtang, and P. McDaniel. 2009a. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*. 235–245.

W. Enck, M. Ongtang, and P. McDaniel. 2009b. Understanding Android security. *IEEE Security and Privacy* 7, 1, 50–57.

W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, Y. Jung, P. McDaniel, and A. N. Sheth. 2014. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems* 32, 2, Article No. 5.

Ericsson. 2013. Ericsson Mobility Report. Retrieved February 10, 2015, from http://www.ericsson.com/res/docs/2013/ericsson-mobility-report-november-2013.pdf.

S. Fahl, M. Harbach, T. Muders, L. Baumgartner, B. Freisleben, and M. Smith. 2012. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*. 50–61.

S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith. 2013. Rethinking SSL development in an appified world. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*. 49–60.

K. Fawaz and K. G. Shin. 2014. Location privacy protection for smartphone users. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS'14)*. 239–250.

A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. 2011a. Android permissions demystied. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*. 627–638.

A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. 2011b. Survey of mobile malware in the wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'11)*. ACM, New York, NY, 3–14.

A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. 2012. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the 8th Symposium on Usable Privacy and Security (SOUPS'12)*. Article No. 3.

A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. 2011c. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium*. 331–346.

A. P. Fuchs, A. Chaudhuri, and J. S. Foster. 2009. *SCanDroid: Automated Security Certification of Android Applications*. Technical Report CS-TR-4991. University of Maryland, College Park, Maryland.

Gartner, Inc. 2014. Gartner says worldwide traditional PC, tablet, ultramobile and mobile phone shipments on pace to grow 7.6 percent in 2014. Retrieved February 10, 2015 from http://www.gartner.com/newsroom/id/2645115.

M. Grace, Y. Zhou, Z. Wang, and X. Jiang. 2012a. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS'12)*.

M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. 2012b. RiskRanker: Scalable and accurate zero-day Android malware detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys'12)*. 281–294.

S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. 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'12)*. 62–81.

H. Hao, V. Singh, and W. Du. 2013. On the effectiveness of API-level access control using bytecode rewriting in Android. In *Proceedings of the 8th ACM Symposium on Information, Computer, and Communications Security (ASIACCS'13)*. 25–36.

S. Hao, B. Liu, S. Nath, W. G. J. Halfond, and R. Govindan. 2014. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th International Conference on Mobile Systems, Applications, and Services (MobiSys'14)*. 204–217.

S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi. 2014. ASM: A programmable interface for extending Android security. In *Proceedings of the 23rd USENIX Security Symposium*. 1005–1019.

P. Hornyack, S. Han, J. Jung, S. Schechter, and D. 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'11)*. 639–652.

C. Jeon, W. Kim, B. Kim, and Y. Cho. 2013. Enhancing security enforcement on unmodified Android. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC'13)*. 1655–1656.

J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. 2012. Dr. Android and Mr. Hide: Fine-grained permissions in Android applications. In *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'12)*. 3–14.

X. Jiang. 2012. An Evaluation of the Application ("App") Verification Service in Android 4.2. Retrieved February 10, 2015, from http://www.cs.ncsu.edu/faculty/jiang/appverify/.

X. Jiang and Y. Zhou. 2013. *Android Malware*. Springer, New York, NY.

D. Kantola, E. Chin, W. He, and D. Wagner. 2012. Reducing attack surfaces for intra-application communication in Android. In *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'12)*. 69–80.

K. Kennedy, E. Gustafson, and H. Chen. 2013. Quantifying the effects of removing permissions from Android applications. In *Mobile Security Technologies (MoST)*.

M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. 2011. L4Android: A generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'11)*. 39–50.

K. Y. Lok and H. Yin. 2012. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *Proceedings of the 21st USENIX Security Symposium*. 569–584.

P. Loscocco and S. Smalley. 2001. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (USENIX ATC'01)*. 29–42.

L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. 2012. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12)*. 229–240.

C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun. 2012. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*. 51–60.

A. Misra and A. Dubey. 2013. *Android Security: Attacks and Defenses*. CRC Press, Boca Raton, FL.

C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda. 2013. PatchDroid: Scalable third-party security patches for Android devices. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC'13)*. 259–268.

M. Nauman, S. Khan, and X. 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'10)*. 328–332.

G. Nolan. 2012. *Decompiling Android*. Apress, New York, NY.

J. Oberheide and C. Miller. 2012. Dissecting the Android Bouncer. Summercon. Retrieved February 10, 2015, from https://jon.oberheide.org/files/summercon12-bouncer.pdf.

D. Octeau, S. Jha, and P. McDaniel. 2012. Retargeting Android applications to Java bytecode. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE-20)*. Article No. 6.

D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. 2013. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX Security Symposium*. 543–558.

M. Ongtang, K. Butler, and P. McDaniel. 2010. Porscha: Policy oriented secure content handling in Android. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC'10)*. 221–230.

M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. 2009. Semantically rich application-centric security in Android. In *Proceedings of the 2009 Annual Computer Security Applications Conference (ACSAC'09)*. 340–349.

M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. 2012. Semantically rich application-centric security in Android. *Security and Communication Networks* 5, 6, 658–673.

R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. 2013. WHYPER: Towards automating risk assessment of mobile applications. In *Proceedings of the 22nd USENIX Security Symposium*. 527–542.

P. Pearce, A. P. Felt, G. Nunez, and D. 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'12)*. 71–72.

H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. 2012. Using probabilistic generative models for ranking risks of Android apps. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*. 241–252.

N. J. Percoco and S. Schulte. 2012. Adventures in BouncerLand: Failures of automated malware detection within mobile application markets. Black Hat USA. Retrieved February 10, 2015, from http://media.blackhat.com/bh-us-12/Briefings/Percoco/BH_US_12_Percoco_Adventures_in_Bouncerland_WP.pdf.

S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. 2014. Execute this! Analyzing unsafe and malicious dynamic code loading in Android applications. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS'14)*.

G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. 2010. Paranoid Android: Versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC'10)*. 347–356.

S. Rasthofer, S. Arzt, and E. Bodden. 2014. A machine-learning approach for classifying and categorizing Android sources and sinks. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS'14)*.

V. Rastogi, Y. Chen, and W. Enck. 2013a. AppsPlayground: Automatic security analysis of smartphone applications. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY'13)*. 209–220.

V. Rastogi, Y. Chen, and X. Jiang. 2013b. DroidChameleon: Evaluating Android anti-malware against transformation attacks. In *Proceedings of the 8th ACM Symposium on Information, Computer, and Communications Security (ASIACCS'13)*. 329–334.

F. Roesner and T. Kohno. 2013. Securing embedded user interfaces: Android and beyond. In *Proceedings of the 22nd USENIX Security Symposium*. 97–112.

G. Sarwar, O. Mehani, R. Boreli, and M.-A. Kaafar. 2013. On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices. In *Proceedings of the 10th International Conference on Security and Cryptography (SECRYPT'13)*. 461–467.

R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. 2011. Soundcomber: A stealthy and context-aware sound Trojan for smartphones. In *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS'11)*. 17–33.

A. Shabtai, Y. Fledel, and Y. Elovici. 2010a. Securing Android-powered mobile devices using SELinux. *IEEE Security and Privacy* 8, 3, 36–44.

A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. 2010b. Google Android: A comprehensive security assessment. *IEEE Security and Privacy* 8, 2, 35–44.

A. Shabtai, U. Kanonov, and Y. Elovici. 2010c. Intrusion detection for mobile devices using the knowledge-based, temporal abstraction method. *Journal of Systems and Software* 83, 8, 1524–1537.

A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. 2012. Andromaly: A behavioral malware detection framework for Android devices. *Journal of Intelligent Informations Systems* 38, 1, 161–190.

S. Shekhar, M. Dietz, and D. S. Wallach. 2012. AdSplit: Separating smartphone advertising from applications. In *Proceedings of the 21st USENIX Security Symposium*. 553–567.

J. Six. 2011. *Application Security for the Android Platform: Processes, Permissions, and Other Safeguards*. O'Reilly Media, Sebastopol, CA.

S. Smalley and R. Craig. 2013. Security enhanced (SE) Android: Bringing flexible MAC to Android. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS'13)*. 20–38.

D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan. 2014. SMV-Hunter: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS'14)*.

M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. 2013. Mobile-sandbox: Having a deeper look into Android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC'13)*. 1808–1815.

M. Sun and G. Tan. 2014. NativeGuard: Protecting Android applications from third-party native libraries. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'14)*. 165–176.

M. Sun, G. Tan, J. Siefers, B. Zeng, and G. Morrisett. 2013. Bringing Java's wild native world under control. *ACM Transactions on Information and System Security* 16, 3, Article No. 9.

O. Tripp and J. Rubin. 2014. A Bayesian approach to privacy enforcement in smartphones. In *Proceedings of the 23rd USENIX Security Symposium*. 175–190.

U.S. Dept. of Homeland Security. 2013. Threats to Mobile Devices Using the Android Operating System. Retrieved February 10, 2015, from http://info.publicintelligence.net/DHS-FBI-AndroidThreats.pdf.

T. Vidas and N. Christin. 2014. Evading Android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer, and Communications Security (ASIACCS'14)*. 447–458.

T. Vidas, D. Votipka, and N. Christin. 2011. All your Droid are belong to us: A survey of current Android attacks. In *Proceedings of the 5th USENIX Workshop on Offensive Technologies (WOOT'11)*. 10.

R. Wang, L. Xing, X. Wang, and S. Chen. 2013. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*. 635–646.

F. Wei, S. Roy, X. 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'14)*. 1329–1341.

Wikipedia. 2015. Android (Operating System): Reception—Market Share. Retrieved February 10, 2015, from http://en.wikipedia.org/wiki/Android_(operating_system)#Market_share.

C. Wu, Y. Zhou, K. Patel, Z. Liang, and X. Jiang. 2014. AirBag: Boosting smartphone resistance to malware infection. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS'14)*.

L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. 2013. The impact of vendor customizations on Android security. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*. 623–634.

R. Xu, H. Sadi, and R. Anderson. 2012. Aurasium: Practical policy enforcement for Android applications. In *Proceedings of the 21st USENIX Security Symposium*. 539–552.

Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. 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'13)*. 1043–1054.

M. Zhang and H. Yin. 2014. AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS'14)*.

X. Zhang, A. Ahlawat, and W. Du. 2013a. AFrame: Isolating advertisements from mobile applications in Android. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC'13)*. 9–18.

Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. 2013b. Vetting undesirable behaviors in Android apps with permission use analysis. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*. 611–622.

M. Zheng, P. P. C. Lee, and J. C. S. Lui. 2012. ADAM: An automatic and extensible platform to stress test Android anti-virus systems. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'12)*. 82–101.

W. Zhou, X. Zhang, and X. Jiang. 2013a. AppInk: Watermarking Android apps for repackaging deterrence. In *Proceedings of the 8th ACM Symposium on Information, Computer, and Communications Security (ASIACCS'13)*. 1–12.

W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. 2013b. Fast, scalable detection of 'piggybacked' mobile applications. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY'13)*. 185–196.

W. Zhou, Y. Zhou, X. Jiang, and P. Ning. 2012a. Detecting repackaged smartphone applications in third-party Android marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy (CODASPY'12)*. 317–326.

X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. 2013c. 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'13)*. 1017–1028.

X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. 2014. The peril of fragmentation: Security hazards in Android device driver customizations. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP'14)*. 409–423.

Y. Zhou and X. Jiang. 2012. Dissecting Android malware: Characterization and evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (SP'12)*. 95–109.

Y. Zhou and X. Jiang. 2013. Detecting passive content leaks and pollution in Android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS'13)*.

Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. 2012b. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS'12)*.

Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. 2011. Taming information-stealing smart-phone applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing (TRUST'11)*. 93–107.

Received May 2014; revised January 2015; accepted February 2015