ISR Institute for Software Research
University of California, Irvine

A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Apps

Alireza Sadeghi
Univ. of California, Irvine
alirezs1@uci.edu

Sam Malek
Univ. of California, Irvine
malek@uci.edu

Hamid Bagheri
Univ. of California, Irvine
hamidb@uci.edu

Joshua Garcia
Univ. of California, Irvine
joshua.garcia@uci.edu

January 2016
ISR Technical Report # UCI-ISR-16-1

Institute for Software Research
ICS2 221
University of California, Irvine
Irvine, CA 92697-3455
www.isr.uci.edu

A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Apps

Alireza Sadeghi, Hamid Bagheri, Joshua Garcia and Sam Malek

Institute for Software Research
School of Information and Computer Sciences
University of California, Irvine

Abstract—In parallel with the meteoric rise of mobile software, we are witnessing an alarming escalation in the number and sophistication of the security threats targeted at mobile platforms, particularly Android, as the dominant platform. While existing research has made significant progress towards detection and mitigation of Android security, gaps and challenges remain. This paper contributes a comprehensive taxonomy to classify and characterize the state-of-the-art research in this area. We have carefully followed the systematic literature review process, and analyzed the results of more than 100 research papers, resulting in the most comprehensive and elaborate investigation of the literature in this area of research. The systematic analysis of the research literature has revealed patterns, trends, and gaps in the existing literature, and underlined key challenges and opportunities that will shape the focus of future research efforts.

Index Terms—Taxonomy and Survey, Security Assessment, Android Platform, Program Analysis

1 Introduction

Android, with well over a million apps, has become one of the dominant mobile platforms [28]. Mobile app markets, such as Android Google Play, have created a fundamental shift in the way software is delivered to consumers, with thousands of apps added and updated on a daily basis. The rapid growth of app markets and the pervasiveness of apps provisioned on such repositories have paralleled with an increase in the number and sophistication of the security threats targeted at mobile platforms. Recent studies have indicated mobile markets are harboring apps that are either malicious or vulnerable, leading to compromises of millions of devices.

This is nowhere more evident than in the Android markets, where many cases of apps infected with malwares and spywares have been reported [87]. Numerous culprits are in play here, and some are not even technical, such as the general lack of an overseeing authority in the case of open markets and inconsequential implication to those caught provisioning applications with vulnerabilities or malicious capabilities. The situation is even likely to exacerbate given that mobile apps are poised to become more complex and ubiquitous, as mobile computing is still in its infancy.

• *UCI-ISR-16-1*
January 2016

In this context, Android's security has been a thriving subject of research in the past few years, since its inception in 2008. These research efforts have investigated the Android security threats from various perspectives and are scattered across several research communities, which has resulted in a body of literature that is spread over a wide variety of domains and publication venues. The majority of surveyed literature has been published in the software engineering and security domains. However, the Android's security literature also overlaps with those of mobile computing and programming language analysis. Yet, there is a lack of a broad study that connects the knowledge and provides a comprehensive overview of the current state-of-the-art about what has already been investigated and what are still the open issues.

This paper presents a comprehensive review of the existing approaches for Android security analysis. The review is carried out to achieve the following objectives:

- To provide a basis taxonomy for consistently and comprehensively classifying Android security assessment mechanisms and research approaches;
- To provide a systematic literature review of the state-of-the-art research in this area using the proposed taxonomy;
- To identify trends, patterns, and gaps through observations and comparative analysis across Android security assessment systems; and
- To provide a set of recommendations for deriving

a research agenda for future developments.

We have carefully followed the systematic literature review process, and analyzed the results of more than 100 research papers published in diverse journals and conferences. Specifically, we constructed a comprehensive taxonomy by performing a "survey of surveys" on related taxonomies and conducting an iterative content analysis over a set of papers collected using reputable literature search engines. We then applied the taxonomy to classify and characterize the state-of-the-art research in the field of Android security. We finally conducted a cross analysis of different concepts in the taxonomy to derive current trends and gaps in the existing literature, and underline key challenges and opportunities that will shape the focus of future research efforts. To the best of our knowledge, this study is the most comprehensive and elaborate investigation of the literature in this area of research.

The rest of the paper is organized as follows: Section 2 overviews the Android framework to help the reader follow the discussions that ensue. Section 3 lists the existing surveys that are directly or indirectly related to the Android security analysis. Section 4 presents the research method and the underlying protocol for the systematic literature review. Section 5 presents a comprehensive taxonomy for the Android security analysis derived from the existing research literature. Section 6 presents a classification of the state-of-the-art research into the proposed taxonomy as well as a cross analysis of different concepts in the taxonomy. Section 7 provides a trend analysis of surveyed research, discusses the observed gaps in the studied literature, and identifies future research directions based on the survey results. Section 8 presents the conclusions.

2 Android Overview

This section provides a brief overview of the Android platform and its incorporated security mechanisms and protection measures to help the reader follow the discussions that ensue.

Android Platform. Android is a platform for mobile devices that includes a Linux OS, system libraries, middleware, and a suite of pre-installed applications. Android applications (apps) are mainly written in the Java programming language by using a rich collection of APIs provided by the Android Software Development Kit (SDK). An app's compiled code alongside data and resources are packed into an archive file, known as an APK.[1] Once an APK is installed on an Android device, it runs by using the Android runtime (ART) environment.[2]

1. Android application package.

2. ART is the successor of the Dalvik VM, which was Android's runtime environment until version 4.4 KitKat.

Application Components. Android defines four types of components: *Activity* components that provide a user interface, *Service* components that execute processes in the background without user interaction, *Content Provider* components that provide the capability of data sharing across applications, and *Broadcast Receiver* components that respond asynchronously to system-wide announcement messages.

Application Configuration. The manifest is a mandatory configuration file (AndroidManifest.xml) that accompanies each Android app. It specifies, among other things, the principal components that constitute the application, including their types and capabilities, as well as required and enforced permissions. The manifest file values are bound to the Android app at compile-time, and cannot be modified at run-time.

Inter-Component Communication. As part of its protection mechanism, Android insulates applications from each other and system resources from applications via a sandboxing mechanism. Such application insulation that Android depends on to protect applications requires interactions to occur through a message passing mechanism, called inter-component communication (ICC). ICC in Android is mainly conducted by means of *Intent* messages. Component capabilities are specified as a set of *Intent-Filters* that represent the kinds of requests a given component can respond to. An Intent message is an event for an action to be performed along with the data that supports that action. Component invocations come in different flavors, e.g., explicit or implicit, intra- or inter-app, etc. Android's ICC allows for late runtime binding between components in the same or different applications, where the calls are not explicit in the code, rather made possible through event messaging, a key property of event-driven systems. It has been shown that the Android ICC interaction mechanism introduces several security issues [26]. For example, Intent event messages exchanged among components, among other things, can be intercepted or even tampered, since no encryption or authentication is typically applied upon them [30]. Moreover, no mechanism exists for preventing an ICC callee from misrepresenting the intentions of its caller to a third party [31].

Permissions. Enforcing permissions is the other mechanism, besides sandboxing, provided by the Android framework to protect applications. In fact, permissions are the cornerstone for the Android security model. The permissions stated in the app manifest enable secure access to sensitive resources as well as cross-application interactions. When a user installs an app, the Android system prompts the user for consent to requested permissions prior to installation. Should the user refuse to grant the requested permissions to an app, the app installation is canceled. No dynamic mechanism is provided by Android for

granting permissions after app installation. Besides required permissions, the app manifest may also include enforced permissions that other apps must have in order to interact with this app. In addition to built-in permissions provided by the Android system to protect various system resources, any Android app can also define its own permissions for the purpose of self-protection. Because the Android access control model is at the level of individual apps, there is no mechanism to check the security posture of the entire system. This causes several security issues, such as re-delegation attacks [40] and app collusions [21], which are shown to be quite common in the apps on the market [30], [38].

3 Related Surveys

Related prior surveys can be classified into two thrusts: studies on mobile malware and studies on Android security. We have sought survey papers from both research domains.

Identifying, categorizing and examining mobile malware have been an interesting field of research since the emergence of mobile platforms. Several years before the advent of modern mobile platforms, such as iOS and Android, Dagon et al. [29] provided a taxonomy of mobile malware. Although the threat models were described for old mobile devices, such as PDAs, our article draws certain attributes from this study for the Android security taxonomy that will be introduced in Section 5.

More recently, Felt et al. [39] analyzed the behavior of a set of malware spread over iOS, Android, and Symbian platforms. They also evaluated the effectiveness of techniques applied by the official app markets, such as Apple AppStore and Google playStore, for preventing and identifying such malware. Along the same line, a comprehensive survey on the evolution of malware for smart devices is provided by Suarez-Tangil et al. [90], showing a particular increase in malware targeting mobile devices just in the past few years. The paper also provides an analysis of 20 research efforts that detect and analyze mobile malware. While the focus of these surveys is on malware for diverse mobile platforms, the area of Android security analysis has not been investigated in detail. They do not analyze, among other things, properties of the approaches for detecting and analyzing Android malware, nor the techniques for Android vulnerability detection.

Besides these general, platform-independent malware surveys, we have found quite a number of relevant surveys that describe subareas of Android security, mainly concerned with specific types of security issues in the Android platform. For instance, Chin et al. [26] studied security challenges in the Android inter-application communication, and presented several classes of potential attacks on applications. Another example is the survey of Shabtai et al. [86], [87], which provides a comprehensive assessment of the security mechanisms provided by the Android framework, but does not thoroughly study other research efforts for detection and mitigation of security issues in the Android platform. The survey of Zhou et al. [107] analyzes and characterizes a set of 1,260 Android malware. This collection of malware, called Malware Genome, are then used by many other researchers to evaluate their proposed malware detection techniques.

Each of these surveys overview specific domains, such as analysis of Android inter-app vulnerabilities or families of Android malware. However, none of them provide a comprehensive overview of the existing research in the area of Android security analysis.

4 Research Method

This survey follows the general guidelines for systematic literature review (SLR) process proposed by Kitchenham [59]. We have also taken into account the lessons from Brereton et al. [20] on applying SLR to the software engineering domain. The process includes three main phases: planning, conducting, and reporting the review. Based on the guidelines, we have formulated the following research questions, which serve as the basis for the systematic literature review.

- **RQ1:** How can existing research on Android app security analysis be classified?
- **RQ2:** What is the current state of Android security analysis research with respect to this classification?
- **RQ3:** What patterns, gaps, and challenges could be inferred from the current research efforts that will inform future research?

For RQ1, in order to define a comprehensive taxonomy suitable for classifying Android security analysis research, we first started with a quick "survey of surveys" on related taxonomies. After an initial taxonomy was formulated, we then used the initial paper review process (focusing on abstracts, introduction, contribution, and conclusions sections) to identify new concepts and approaches to augment and refine our taxonomy. The resulting taxonomy is presented in Section 5.

For the second research question (RQ2), we used the validated paper collection and the consolidated taxonomy to conduct a more detailed review of the papers. Each paper was classified using every dimension in the taxonomy, and the results were captured in a research catalog. The catalog, consisting of a set of spreadsheets, allowed us to perform qualitative and quantitative analysis not only in a single dimension, but also across different dimensions in the taxonomy. The analysis and findings are documented in Sec-

tion 6. [3]

To answer the third research question (RQ3), we analyzed the results from RQ2 and attempted to identify the gaps and trends, again using the taxonomy as a critical aid. The possible research directions are henceforth identified and presented in Section 7.

In the planning phase, the search engines and the keywords for the related papers were selected. We used reputable literature search engines and databases in our review protocol with the goal of finding high-quality refereed research papers from respectable venues. The selected search engines consist of IEEE Explore, ACM Digital Library, Springer Digital Library, USENIX Proceedings, and Google Scholar.

Given the scope of our literature review, we focused on selected keywords to perform the search. These keywords were continuously refined and extended during the search process. Examples of the keywords include Android security, Android vulnerability, Android static analysis, and Android dynamic analysis.

Inclusion Criteria. Not all the retrieved papers based on the selected keywords fit within the scope of this paper. As illustrated in Figure 1, the scope of surveyed research in this study falls at the intersection of three domains:

1) *Program Analysis* domain that includes the techniques used for extracting the models of individual Android apps and/or the Android platform.
2) *Security Assessment* domain that covers the analysis methods applied on the the extracted models to identify the potential security issues among them.
3) *Android Platform* domain that takes into account the special features and challenges involved in the Android platform, its architecture, and security model.

Papers that fall at the intersection of these three domains are included in our review.

Exclusion Criteria. We excluded papers that:

1) exclusively developed for platforms other than Android, such as iOS, Windows Mobile and BlackBerry. However, approaches that cover multiple platforms, including Android, fall within the scope of this survey.
2) focused only on techniques for mitigation of security threats [27], [31], [42], but not on any security analysis technique. Note that approaches that consider both security assessment and prevention/mitigation were included in the survey.
3) have not leveraged any program analysis technique, which is one of the three dimensions specifying the scope of this survey. Although a significant portion of the surveyed papers employ techniques that are supplementary to program analysis, such as machine learning or formal analysis, the approaches that purely rely on such supplementary techniques are excluded from our study. Examples include *WHYPER* [71] and *Andromaly* [88] that, unlike the approaches surveyed in this paper, do not leverage any program-analysis technique.
4) have employed low-level monitoring and profiling techniques in identifying anomaly for malware detection [?], [?], [?], [?], [?], [?]. Such approaches perform dynamic analysis at the level of either hardware signals (e.g., power consumption, memory usage, network traffic, etc.) or kernel system call, yet do not leverage any program analysis technique, partially scoping this survey.

Moreover, the analysis tools that are not accompanied by any peer-reviewed paper were excluded, as most of the taxonomy dimensions are not applicable to such tools. *Androguard* [1] and *DroidBox* [5] are two examples that respectively leverage static and dynamic analysis techniques, but lack any peer-reviewed paper, thus were excluded from this survey.

As a result, after excluding the out-of-scope papers, we have included 100 papers published from 2009 to October 2015[4], out of the total of over 200 papers found. Figure 2 shows the number of selected papers by the publication year. Figure 3 shows more detailed

3. The research artifacts, including the survey catalog, are available to the public and can be accessed at https://seal.ics.uci.edu/and-sec-taxonomy

4. The papers published in or after October 2015 are not included in this survey.

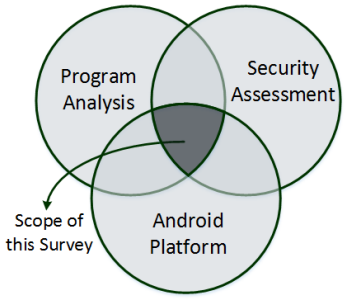

Fig. 1. Scope of this survey.

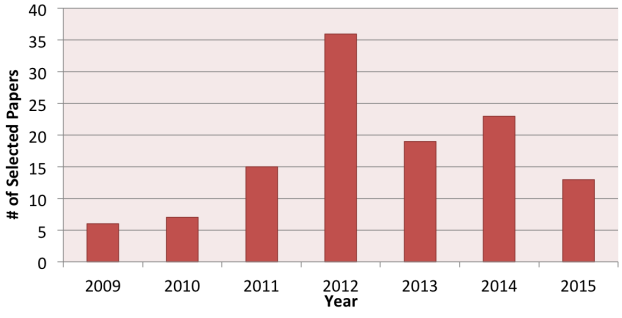

Fig. 2. Number of Surveyed Papers by Publication Year.

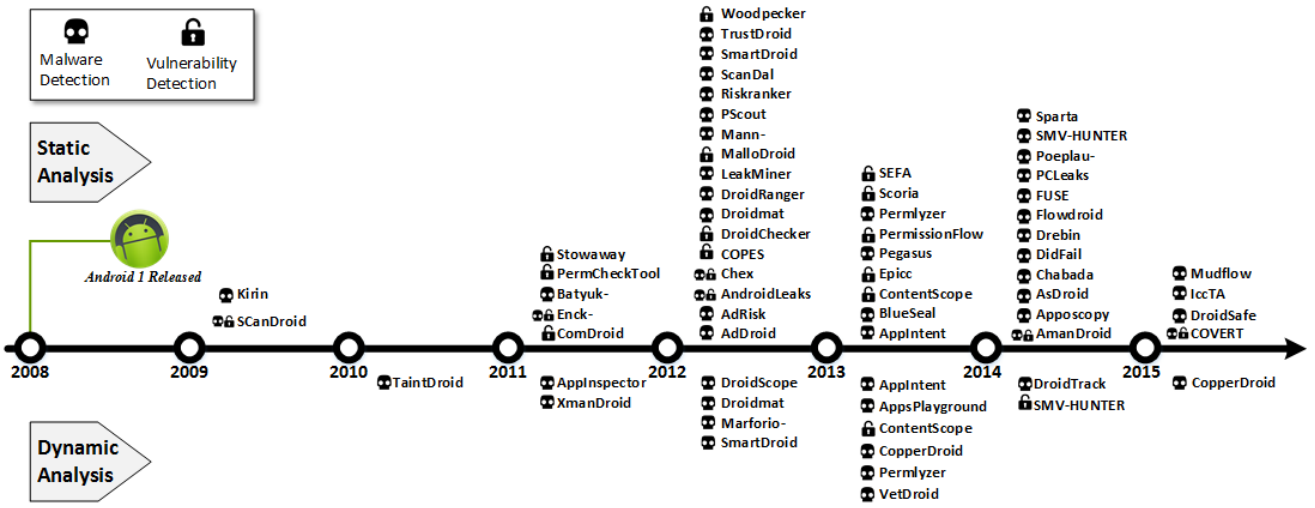

Fig. 3. A timeline of the distinguished research in this survey.

information as a timeline[5], where different surveyed research efforts are presented along with their publication years, objectives, i.e., detection of malicious behaviors, vulnerabilities or both, and the type of program analysis used. Research efforts that employ both static and dynamic analysis techniques appear at both the top and bottom of the timeline.

Threats to Validity. By carefully following the SLR process in conducting this study, we have tried to minimize the threats to the validity of the results and conclusions made in this article. Nevertheless, there are three possible threats that deserve additional discussion.

One important threat is the completeness of this study, that is, whether all of the appropriate papers in the literature were identified and included. This threat could be due to two reasons: (1) some relevant papers were not picked up by the search engines or did not match our keyword search, (2) some relevant papers that were mistakenly omitted, and vice-versa, some irrelevant papers that were mistakenly included. To address these threats, we used multiple search engines, including both scientific and general-purpose search engines. We also adopted an iterative approach for our keyword-list construction. Since different research communities (particularly, software engineering and security) refer to the same concepts using different words, the iterative process allowed us to ensure that a proper list of keywords were used in our search process.

Another threat is the validity of the proposed taxonomy, that is, whether the taxonomy is sufficiently rich to enable proper classification and analysis of the literature in this area. To mitigate this threat, we adopted an iterative content analysis method, whereby the taxonomy was continuously evolved to account for every new concept encountered in the papers. This gives us confidence that the taxonomy provides a good coverage for the variations and concepts that are encountered in this area of research.

Another threat is the objectiveness of the study, which may lead to biased or flawed results. To mitigate this risk, we have tackled the individual reviewer's bias by crosschecking the papers, such that no paper received a single reviewer. We have also tried to base the conclusions on the collective numbers obtained from the classification of papers, rather than individual reviewer's interpretation or general observations, thus minimizing the individual reviewer's bias.

5 Taxonomy

To define an Android security analysis taxonomy for RQ1, we started with selecting suitable dimensions and properties found in existing surveys. The aforementioned studies described in Section 3, though relevant and useful, are not sufficiently specific and systematic enough for classifying the Android security analysis approaches in that they either focus on mobile malware in general, or focus on certain subareas, such as Android inter-application vulnerabilities or families of Android malware software, but not on the Android security analysis as a whole.

We thus have defined our own taxonomy to help classify existing work in this area. Nonetheless, the proposed taxonomy is inspired by existing work surveyed in Section 3. The highest level of the taxonomy hierarchy classifies the surveyed research based on the following three questions:

1) *What* are the problems in the Android security being addressed?
2) *How* and with which techniques the problems are solved?

5. The approaches without name are shown in the form of "*first author's name-*" throughout the paper.

3) How is the validity of the proposed solutions evaluated?

For each question, we derive the sub-dimensions of the taxonomy related to the question, and enumerate the possible values that characterize the studied approaches. The resulting taxonomy hierarchy consists of 15 dimensions and sub-dimensions, which are depicted in Figures 4–6, and explained in the following.

5.1 Approach Positioning (Problem)

The first part of the taxonomy, approach positioning, helps characterize the "WHAT" aspects, that is, the objectives and intent of Android security analysis research. It includes five dimensions, as depicted in Figure 4.

5.1.1 Analysis Objectives (T1.1)

This dimension classifies the approaches with respect to the goal of their analysis. Thwarting malware apps that compromise the security of Android devices is a thriving research area. In addition to detecting malware apps, identifying potential security threats posed by benign Android apps, that legitimately process user's private data (e.g., location information, IMEI, browsing history, installed apps, etc.), has also received a lot of attention in the area of Android security.

5.1.2 Type of Security Threats (T1.2)

This dimension classifies the security threats being addressed in the surveyed research along the Microsoft's threat model, called *STRIDE* [91]:

Spoofing violates the *authentication* security property, where an adversary is illegally accessing and using the information of another authenticated user. An example of this threat in the Android platform is *Intent Spoofing*, where a forged Intent is sent to an exported component, exposing the component to components from other applications (e.g., a malicious application) [26].

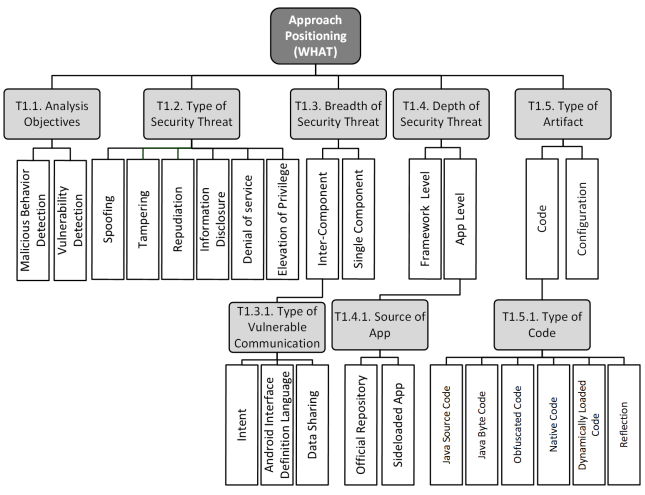

Fig. 4. Proposed Taxonomy of Android Security Analysis, Problem Category.

Tampering affects the *integrity* property and involves a malicious modification of data. *Content Pollution* is an instance of this threat, where an app's internal database is manipulated by other apps [108].

Repudiation is in contrast to *non-repudiation* property, which refers to the situation in which entities deny their role or action in a transaction. An example of this security threat occurs when an application tries to hide its malicious behavior by manipulating log data to mislead a security assessment.

Information Disclosure compromises the *confidentiality* by releasing the protected or confidential data to an untrusted environment. In mobile devices, sensitive or private information such as device ID (IMEI), device location (GPS data), contact list, etc., might, intentionally or unintentionally, be leaked to an untrusted environment, via different channels as SMS, Internet, Bluetooth, etc.

Denial of service (DoS) affects *availability* by denying service to valid users. A common vulnerability in Android apps occurs when a payload of an Intent is used without checking against the null value, resulting in a *null dereference* exception to be thrown, possibly crashing the Android process in which it occurs. This kind of vulnerability has been shown to be readily discoverable by an adversary through reverse engineering of the apps [33], which in turn enables launching a denial of service attack. Unauthorized Intent receipt [26] and battery exhaustion [67] are some other examples of DoS attacks targeted at Android apps.

Elevation of Privilege subverts the *authorization* and happens when an unprivileged user gains privileged access. An example of the privilege escalation, which is shown to be quite common in the apps on the Android markets [51], happens when an application with less permissions (a non-privileged caller) is not restricted from accessing components of a more privileged application (a privileged callee) [30].

5.1.3 Breadth of Security Threats (T1.3)

This dimension classifies the approaches based on the granularity of identifiable security threats. In the basic form, a security issue, either vulnerability or malicious behavior, occurs by the execution of a single (vulnerable and/or malicious) component. However, there exists more complicated scenarios where a security issue may arise from the interaction of multiple components. Moreover, it is possible that interacting components belong to different applications. For example, in an instance of the *app collusion* attack, multiple applications can collude to compromise a security property, such as the user's privacy [22], [30]. Accordingly, security assessment techniques that consider the combination of apps in their analysis are

able to reveal more complicated issues compared to non-compositional approaches.

Type of Vulnerable Communication (T1.3.1) Android platform provides a variety of Inter-Process Communication (*IPC*) mechanisms for app components to communicate among each other, while achieving low levels of coupling. However, due to intrinsic differences with pure Java programming, such communication mechanisms could be easily misimplemented, leading to security issues. From a program analysis perspective, Android communication mechanisms need to be treated carefully, to avoid missing security issues. Our taxonomy showcases three major types of IPC mechanisms that may lead to vulnerable communication:

- As described in Section 2, Intents provide a flexible IPC model for communication among Android components. However, *Intents* are the root of many security vulnerabilities and malicious behaviors.
- *Android Interface Definition Language (AIDL)* is another IPC mechanism in Android that allows client-server RPC-based communication. The implementation of an AIDL interface must be thread-safe to prevent security issues resulting from concurrency problems (e.g., race conditions) [**?**].
- *Data Sharing* is another mechanism that allows app components to communicate with each other. Among the other methods, using Content Providers is the main technique for sharing data between two applications. However, misusage of such components may lead to security issues, such as passive content leaks (i.e., leaking private data), and content pollution (i.e., manipulating critical data) [108].

5.1.4 Depth of Security Threats (T1.4)

The depth of security threats category reflects if the approach addresses a problem at the application level or the framework level. The former aims at solely analyzing the application software. Third party apps, especially those from an unknown or untrustworthy provenance, pose a security challenge. However, there are some issues, such as overarching design flaws, that require system-wide reasoning, and are not easily attainable by simply analyzing individual parts of the system. Approaches at the framework level include research that focuses on modeling and analyzing the Android platform (e.g., for potential system-level design flaws and issues encountered in the underlying framework).

Source of App (T1.4.1) An application's level of security threat varies based on the source from which its installation package (i.e., apk file) is obtained. As a result, it is important to include a sub-dimension representing the source of the app in our taxonomy, which indicates whether the app is obtained from the official Android repository:

- *Official Repository:* Due to the continuous vetting of the official Android repository (i.e., Google Play), apps installed from that repository are safer than third-party apps.
- *Sideloaded App:* Sideloading, which refers to installing apps from sources other than the official Android repository, exposes a new attack surface for malware. Hence, it is critical for security research to expand their analysis beyond the existing apps in Google Play.

5.1.5 Type of Artifact (T1.5)

Android apps are realized by different kinds of software artifacts at different levels of abstraction, from high-level configuration files (e.g., *Manifest*) to low-level Java source code or native libraries implemented with C or C++. From the security perspective, each artifact captures some aspects essential for security analysis. For instance, while permissions are defined in the manifest file, inter-component messages (i.e., *Intents*) are implemented at the source code level. This dimension of the taxonomy indicates the abstraction level(s) of the models extracted for the analysis.

Type of Code (T1.5.1) The approaches that perform the analysis at the code level, are further distinguishable based on the type of code they support, which includes the following:

- *Java Source Code:* Since Android apps are mostly written in the Java language, a basic analysis approach can only rely on the availability of Java source code of Android apps. This assumption, however, limits the applicability of the analysis to either open-source apps or the developers of an app.
- *Java Byte Code:* Techniques that are able to perform their analyses on byte-code[6] widely broaden their applicability compared to the first group. Such techniques can be employed by third-party analysts to assess millions of publicly available apps.
- *Obfuscated Code:* Benign app developers tend to obfuscate their application to protect the source code from being understood and/or reverse engineered by others. Malware app developers also use obfuscation techniques to hide malicious behaviors and avoid detection by antivirus products. Depending on the complexity of obfuscation, which varies from simple renaming to invoking behavior using reflection, security assessment approaches should tackle the challenges in analyzing the obfuscated apps.
- *Native Code:* Beside Java code, Android apps may also consist of native C or C++ code, which is

6. Distinct from Java, Android has its own Dalvik byte-code format called Dex, which is executable by Android virtual machine.

usually used for performance or portability requirements. An analysis designed for Java is not able to support these kinds of apps. To accurately and precisely analyze such apps, they need to be treated differently from non-native apps.

- *Dynamically Loaded Code:* Applications may dynamically load code that is not included in the original application package (i.e., apk file) loaded at installation time. This mechanism allows an app to be updated with new desirable features or fixes. Despite the benefits, this mechanism poses significant challenges to analysis techniques and tools, particularly static approaches, for assessing security threats of Android applications.
- *Reflection:* Using Java reflection allows apps to instantiate new objects and invoke methods by their names. If this mechanism is ignored or not handled carefully, it may cause incomplete and/or unsound static analysis. Supporting reflection is a challenging task for a static analysis tool, as it requires precise string and points-to analysis [**?**].

5.2 Approach Characteristics (Solution)

The second group of the taxonomy dimensions is concerned with classifying the "HOW" aspects of Android security analysis research. It includes three dimensions, as shown in Figure 5.

5.2.1 Type of Program Analysis (T2.1)

This dimension classifies the surveyed research based on the type of program analysis employed for security assessment. The type of program analysis leveraged in security domain could be *static* or *dynamic*. Static analysis examines the program structure to reason about its potential behaviors. Dynamic analysis executes the program to observe its actual behaviors at runtime.

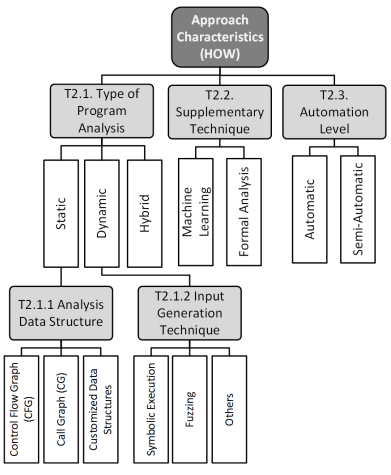

Fig. 5. Proposed Taxonomy of Android Security Analysis, Solution Category.

Each approach has its own strengths and weaknesses. While static analysis is considered to be conservative and sound, dynamic analysis is unsound yet precise [35]. Dynamic analysis requires a set of input data (including events, in event-based systems like Android) to run the application. Since the provided test cases are often likely to be incomplete, parts of the app's code, and thereby its behaviors, are not covered. This could lead to false negatives, i.e., missed vulnerabilities or malicious behaviors in security analysis. Moreover, it has been shown that dynamic approaches could be recognized and deceived by advanced malware, such as what anti-taint tracking techniques do to bypass dynamic taint analyses [84].

On the other hand, by abstracting from the actual behavior of the software, static analysis could derive certain approximations about all possible behaviors of the software. Such an analysis is, however, susceptible to false positives, e.g., a warning that points to a vulnerability in the code which is not executable at runtime.

Analysis Data Structures (T2.1.1) A few well-known data structures that abstract the underlying programs are widely used in various static analysis techniques. The most frequently encountered data structures are as follows:

- *Control Flow Graph (CFG)* is a directed graph that represents program statements by its nodes, and the flow of control among the statements by the graph's edges.
- *Call Graph (CG)* is a directed graph, in which each node represents a method, and an edge indicates the call of (or return from) a method.
- *Inter-procedural Control Flow Graph (ICFG)* is a combination of CFG and CG that connects separated CFGs using call and return edges.

In addition, variation of these canonical data structures are used for special-purpose analyses. The goal of this dimension is to characterize the analysis based on the usage of these data structures.

Input Generation Technique (T2.1.2) The techniques that employ dynamic analysis for security assessment need to run mobile applications in order to perform the analysis. For this purpose, they require test input data and events that trigger the application under experiment. Security testing is, however, a notoriously difficult task. This is in part because unlike functional testing that aims to show a software system complies with its specification, security testing is a form of negative testing, i.e., showing that a certain (often *a priori* unknown) behavior does not exist.

In addition to manually providing the inputs, which is not systematic and scalable, two approaches are often leveraged by the surveyed research: fuzzing and symbolic execution.

- *Fuzz testing or fuzzing* [46] executes the app with random input data. Running apps using inputs

generated by Monkey [2], the state-of-the-practice tool for the Android system testing, is an example of fuzz testing.
- *Symbolic execution* [58] uses symbolic values, rather than actual values, as program inputs. It gathers the constraints on those values along each path of the program and with the help of a solver generates inputs for all reachable paths.

5.2.2 Supplementary Techniques (T2.2)

Besides various program analysis techniques, which are the key elements employed by approaches in the surveyed research, other supplementary techniques have also been leveraged to complement the analysis. Among the surveyed research, *Machine Learning* and *Formal Analysis* are the most widely used techniques. In fact, the program analysis either provides the input for, or consumes the output of, the other supplementary techniques. This dimension of the taxonomy determines the techniques other than program analysis (if any) that are employed in the surveyed research.

5.2.3 Automation Level (T2.3)

The automation level of a security analysis method also directly affects the usability of such techniques. Hence, we characterize the surveyed research with respect to the manual efforts required for applying the proposed techniques. According to this dimension, existing techniques are classified as either automatic or semi-automatic.

5.3 Assessment (Validation)

The third and last section of the taxonomy is about the evaluation of Android security research. Dimensions in this group, depicted in Figure 6, provide the means to assess the quality of research efforts included in the survey.

The first dimension, evaluation method, captures how, i.e., with which evaluation method, a paper validates the effectiveness of the proposed approach, such as empirical experimentation, formal proof, case studies, or other methods.

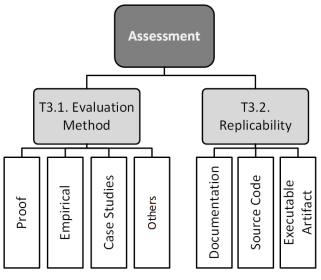

Fig. 6. Proposed Taxonomy of Android Security Analysis, Assessment Category.

The other dimension captures the extent to which surveyed research efforts enable a third party to reproduce the results reported by the authors. This dimension classifies replicability of research approaches by considering the availability of research artifacts. For example, whether the approach's underlying platform, tools and/or case studies are publicly available.

6 Survey Results and Analysis

This section presents the results of our literature review to answer the second research question. By using the proposed taxonomy as a consistent point of reference, many insightful observations surface from the survey results. The number of the research papers surveyed will not allow elaboration on each one of them. Rather, we highlight some of them as examples in the observations and analyses below.

6.1 Approach Positioning (Problem)

Table 1 tabularizes a summary of the problem-specific aspects that are extracted from our collection of papers included in the survey. Note that the classifications are meant to indicate the primary focus of a research paper. For example, if a certain approach is not mentioned in the *Spoofing* column under the *Type of Security Threat*, it does not necessarily indicate that it absolutely cannot mitigate such threat. Rather, it simply means spoofing is not its primary focus. Furthermore, for some taxonomy categories, such as *Depth of Threat*, a paper may have multiple goals and thus listed several times. On the other hand, number of dimensions only applies to specific part of research, e.g., *Test Input Generation* only applies for dynamic or hybrid approaches. As a result, percentages presented in the last column of the table may sum up to more or less than 100%. In the following, we present the main results for each dimension in the problem category.

6.1.1 Analysis Objective

Based on the analysis of the research studies in the literature, it is evident that the majority of Android security approaches have been applied to detection of malicious behaviors, comprising 75% of the overall set of papers collected for this literature review.

Several research efforts on malicious behavior detection target the analysis of advertisement (ad) libraries that are linked and shipped with applications. In fact, a variety of private user data, including a user's call logs, phone numbers, browser bookmarks, and the list of apps installed on a device are collected by ad libraries. Since the required permissions of ad libraries are merged into a hosting app's permissions, it is challenging for users to distinguish, at installation time, the permissions requested by the embedded ad libraries from those actually used by the app [72]. For this reason, *AdRisk* [50] decouples the embedded ad libraries from the host apps and examines the

TABLE 1
Problem Specific Categorization of the Reviewed Research

Dimension			Approaches	Percentage
Analysis Objective	Vulnerability Detection		Amandroid [96], AndroidLeaks [44], Chex [64], ComDroid [26], ContentScope [108], COPES [17], COVERT [16], DroidChecker [23], Enck- [33], Epicc [70], MalloDroid [37], PermCheckTool [95], PermissionFlow [85], SCanDroid [43], Scoria [94], SEFA [98], SMV-HUNTER [52], Stowaway [38], Woodpecker [51]	32%
	Malicious Behavior Detection		AdDroid [72], AdRisk [50], Amandroid [96], AndroidLeaks [44], AppInspector [45], AppIntent [103], Apposcopy [41], AppsPlayground [76], AsDroid [55], Batyuk- [19], BlueSeal [54], Chabada [48], Chex [64], CopperDroid [78], COVERT [16], DidFail [60], Drebin [12], Droidmat [97], DroidRanger [109], DroidSafe [47], DroidScope [100], DroidTrack [82], Enck- [33], Flowdroid [13], FUSE [77], IccTA [61], Kirin [34], LeakMiner [102], Mann- [65], Marforio- [66], Mudflow [15], PCLeaks [62], Pegasus [25], Permlyzer [99], Poeplau- [73], Riskranker [49], ScanDal [57], SCanDroid [43], SmartDroid [106], Sparta [36], TaintDroid [32], TrustDroid [105], VetDroid [104], Xmandroid [21]	75%
Type of Security Threat	Spoofing		Amandroid [96], Chex [64], ComDroid [26], Epicc [70], MalloDroid [37], PCLeaks [62], SMV-HUNTER [52]	12%
	Tampering		ContentScope [108], MalloDroid [37], SMV-HUNTER [52]	5%
	Repudiation			0%
	Information Disclosure		AdRisk [50], Amandroid [96], AndroidLeaks [44], AppInspector [45], AppIntent [103], Apposcopy [41], AppsPlayground [76], AsDroid [55], Batyuk- [19], BlueSeal [54], Chex [64], ComDroid [26], ContentScope [108], CopperDroid [78], CopperDroid2 [92], COVERT [16], DidFail [60], DroidSafe [47], DroidTrack [82], Enck- [33], Epicc [70], Flowdroid [13], IccTA [61], LeakMiner [102], Mann- [65], Mudflow [15], PCLeaks [62], Pegasus [25], ScanDal [57], SEFA [98], Sparta [36], TaintDroid [32], TrustDroid [105]	55%
	Denial of Service		ComDroid [26], Enck- [33]	3%
	Elevation of Privilege		AdDroid [72], AppsPlayground [76], Chex [64], ComDroid [26], CopperDroid [78], COVERT [16], DroidChecker [23], Enck- [33], Epicc [70], FUSE [77], PCLeaks [62], Pegasus [25], PermissionFlow [85], SEFA [98], Woodpecker [51], Xmandroid [21]	27%
Breath of Threat	Single Component		AdRisk [50], Amandroid [96], AndroidLeaks [44], AppInspector [45], AppIntent [103], Apposcopy [41], AppsPlayground [76], AsDroid [55], Batyuk- [19], BlueSeal [54], Chex [64], ComDroid [26], ContentScope [108], CopperDroid [78], CopperDroid2 [92], COVERT [16], DidFail [60], DroidSafe [47], DroidTrack [82], Enck- [33], Flowdroid [13], FUSE [77], IccTA [61], Kirin [34], LeakMiner [102], MalloDroid [37], Mann- [65], Mudflow [15], PCLeaks [62], Pegasus [25], PermissionFlow [85], Poeplau- [73], ScanDal [57], SCanDroid [43], SEFA [98], Sparta [36], TaintDroid [32], TrustDroid [105], Woodpecker [51]	65%
	Inter-Comp.	Intent	Amandroid [96], AppIntent [103], COVERT [16], DidFail [60], Epicc [70], FUSE [77], IccTA [61], PCLeaks [62], Apposcopy [41]	22%
		AIDL	BlueSeal [54], Woodpecker [51]	3%
		SharedData	ContentScope [108]	2%
Depth of Threat	App Level (Installed from Google Play or Sideloaded)		AdRisk [50], Amandroid [96], AndroidLeaks [44], AppInspector [45], AppIntent [103], Apposcopy [41], AppsPlayground [76], AsDroid [55], Batyuk- [19], BlueSeal [54], Chabada [48], Chex [64], ComDroid [26], ContentScope [108], COPES [17], COVERT [16], DidFail [60], Drebin [12], Droidmat [97], DroidChecker [23], DroidRanger [109], DroidSafe [47], DroidTrack [82], Enck- [33], Epicc [70], Flowdroid [13], FUSE [77], IccTA [61], Kirin [34], LeakMiner [102], MalloDroid [37], Mudflow [15], PCLeaks [62], Pegasus [25], PermCheckTool [95], PermissionFlow [85], Permlyzer [99], Poeplau- [73], Riskranker [49], ScanDal [57], SCanDroid [43], Scoria [94], SEFA [98], SmartDroid [106], SMV-HUNTER [52], Sparta [36], TaintDroid [32], TrustDroid [105], VetDroid [104], Woodpecker [51]	83%
	Framework Level		AdDroid [72], AppInspector [45], Bagheri- [?], COPES [17], CopperDroid [78], CopperDroid2 [92], COVERT [16], DroidSafe [47], DroidScope [100], DroidTrack [82], Marforio- [66], PScout [14], ScanDal [57], Scoria [94], SmartDroid [106], Sparta [36], Stowaway [38], TaintDroid [32], VetDroid [104], Xmandroid [21]	32%
Type of Artifact	Configuration		AdRisk [50], Amandroid [96], AndroidLeaks [44], Apposcopy [41], AsDroid [55], Batyuk- [19], BlueSeal [54], Chabada [48], Chex [64], ComDroid [26], ContentScope [108], COPES [17], COVERT [16], DidFail [60], Drebin [12], DroidChecker [23], Droidmat [97], DroidRanger [109], DroidSafe [47], Epicc [70], Flowdroid [13], FUSE [77], IccTA [61], Kirin [34], LeakMiner [102], MalloDroid [37], Mann- [65], Mudflow [15], PCLeaks [62], Pegasus [25], PermCheckTool [95], PermissionFlow [85], Permlyzer [99], Poeplau- [73], PScout [14], Riskranker [49], ScanDal [57], SCanDroid [43], Scoria [94], SEFA [98], SmartDroid [106], SMV-HUNTER [52], Sparta [36], Stowaway [38], TrustDroid [105], Woodpecker [51]	77%
	Code	Source	Enck- [33], Mann- [65], PermCheckTool [95], SCanDroid [43], Sparta [36]	8%
		Byte	AdRisk [50], Amandroid [96], AndroidLeaks [44], Apposcopy [41], AsDroid [55], Batyuk- [19], BlueSeal [54], Chabada [48], Chex [64], ComDroid [26], ContentScope [108], COPES [17], COVERT [16], Drebin [12], Droidmat [97], DidFail [60], DroidChecker [23], DroidRanger [109], DroidSafe [47], Epicc [70], Flowdroid [13], FUSE [77], IccTA [61], LeakMiner [102], MalloDroid [37], Mudflow [15], PCLeaks [62], Pegasus [25], PermissionFlow [85], Permlyzer [99], Poeplau- [73], PScout [14], Riskranker [49], ScanDal [57], Scoria [94], SEFA [98], SmartDroid [106], SMV-HUNTER [52], Stowaway [38], TrustDroid [105], Woodpecker [51]	68%
		Obfuscated	Apposcopy [41]	2%
		Native	CopperDroid [78], CopperDroid2 [92], DroidRanger [109], Flowdroid [13], Poeplau- [73], Riskranker [49]	10%
		Dynamic	AdRisk [50], AppsPlayground [76], DroidRanger [109], Poeplau- [73], Riskranker [49]	8%
		Reflection	AdRisk [50], AppsPlayground [76], DroidSafe [47], FUSE [77], Pegasus [25], Riskranker [49], ScanDal [57], Sparta [36], Stowaway [38], TaintDroid [32], VetDroid [104]	18%

potential unsafe behavior of each library that could result in privacy issues. Beyond this risk assessment of ad libraries, [72] introduces *AdDroid*, an advertising framework with dedicated permissions and APIs that separates privileged advertising functionality from host applications.

Android vulnerability analysis has also received attention from a significant portion of existing research efforts (32% of the studied papers). Since techniques and methods used for one of the above goals are often applicable to other goals, the target of many surveyed research papers falls in both categories. However, there are some approaches that only target vulnerability detection. Among such approaches, *Woodpecker* [51] tries to identify vulnerabilities in the standard configurations of Android smartphones, i.e., pre-loaded apps in such devices, that may lead to *capability leaks*. A capability (or permission) leak is an instance of a privilege-escalation threat, where some privileged functions (e.g., sending of a text message) is

left exposed to apps lacking the required permissions to access those functions.

6.1.2 Type of Security Threat

The Android security approaches studied in this literature review have covered diverse types of security threats. It can be observed from Table 1 that among the *STRIDE* security threats (cf. Section 5.1.2), information disclosure is the most considered threat in Android, comprising 55% of the papers. This is not a surprising result, since mobile devices are particularly vulnerable to data leakage [56]. Elevation of privilege is the second class of threats addressed by 27% of the overall studied papers. Examples of this class of threats, such as *confused deputy vulnerability* [53], are shown to be quite common in the Android apps on the market [30], [38], [40].

Spoofing, tampering, and denial of service issues are also considered in the literature, comprising 12%, 5%, and 3% of the papers, respectively. Spoofing has received substantial attention, particularly because Android's flexible Intent routing model can be abused in multiple ways, resulting in numerous possible attacks, including *Broadcast injection* and *Activity/Service launch* [26]. Among the STRIDE's threats, repudiation is not explicitly studied in the surveyed research. We will revisit this gap in Section 7.

6.1.3 Breadth of Threat

We can observe from Table 1 that the majority of the Android security approaches are intended to detect and mitigate security issues in a single component, comprising 65% of the overall papers studied in this literature review, while a comparatively low number of approaches (27%) have been applied to inter-component analysis.

These compositional approaches take into account inter-component and/or inter-app communication during the analysis to identify a broader range of security threats that cannot be detected by techniques that analyze a single component in isolation. Among others, *IccTA* [61], [63] performs data leak analysis over a bundle of apps. It first merges multiple apps into a single app, which enables context propagation among components in different apps, and thereby facilitates a precise inter-component taint analysis.

The main challenge with such approaches for compositional analysis is the scalability issue. Because as the number of apps increases, the cost of program analysis grows exponentially. To address the scalability issue intrinsic to compositional analysis, some hybrid approaches are more recently proposed that combine program analysis with other reasoning techniques [16], [41]. For example, COVERT [16], [81] combines static analysis with lightweight formal methods. Through static analysis of each individual app, it first extracts relevant security specifications in an analyzable formal specification language (i.e., Alloy). These app specifications are then combined together and checked as a whole with the aid of a SAT solver for inter-app vulnerabilities.

Intent is the main inter-component communication mechanism in Android and thus, it has been studied and focused more than other ICC mechanism (22% compared to 2% and 3%). *Epicc* [70] and its successor *IC3* [**?**], try to precisely infer Intent values, which are necessary information for identifying vulnerable communications. *BlueSeal* [54] and *Woodpecker* [51] briefly discussed AIDL, as another ICC mechanism, and how to incorporate it in control flow graph. Finally, *ContentScope* [108] examines the security threats of using shared data as the third way of achieving ICC.

6.1.4 Depth of Threat

We observe that most approaches perform the analysis at the application-level (83%), but about one third of the approaches consider the underlying Android framework for analysis (32%). The results of analyses carried out at the framework-level are also beneficial in analysis of individual apps, or even revealing the root causes of the vulnerabilities found at the application-level. For example, *PScout* [14] and *Stowaway* [38], through the analysis of the Android framework, captured the permission mappings that specify mappings between Android API calls/Intents and the permissions required to perform those calls. Such permission mappings are then used by many other approaches, among others for detecting *over-privileged* apps that violate the "Principle of Least Privilege" [83].

Apps installed from arbitrary sources pose a higher security risk than apps downloaded from Google Play. However, regardless of the source of the app, it must be installed using the same mechanism for importing the app's code into the Android platform, i.e., by installing APK files. Nevertheless, to measure the effectiveness of a technique for identifying security threats, researchers need to evaluate the proposed technique using both Google Play and sideloaded apps. We discuss, in detail, the sources of apps used to evaluate Android security analysis techniques in Section 6.3.1.

6.1.5 Type of Artifact

As discussed in Section 5, Android apps are composed of several artifacts at different levels of abstraction, such as high-level configuration files and code implementation. Here by "code" we mean either source code or any kind of compiled code, including Java or Dalvik byte-code. We can observe from Table 1 that most of the studied approaches analyze multiple artifacts.

Manifest is an XML configuration file, shipped with all Android apps, and includes some high-level architectural information, such as the apps' components,

their types, permissions they require, etc. Since a large portion of security-related information are encoded in the apps' manifest files (e.g., required or defined permissions), some techniques only focus on the analysis of this file. Kirin [34], for instance, is among the techniques that only performs the analysis on the app manifest files. By extracting the requested permissions defined in the manifest file and comparing their combination against a set of high-level, blacklist security rules, Kirin is able to identify the apps with potential dangerous functionality, such as information leakage. However, the security policies in Kirin, or similar techniques that are limited to the abstract level of configuration files, may increase the rate of false warnings. For instance, a Kirin's security rule, for mitigating mobile bots that send SMS spam, is stated as *"An application must not have SEND_SMS and WRITE_SMS permission labels [34]"*. As a result, an application requesting these two permissions is flagged as malware, even if there are no data-flow between the parts of code corresponding to these two permissions.

In addition to the manifest file, there are some other resources in the Android application package (a.k.a., *apk* file) that also do not require complicated techniques to be analyzed. One example is the layout file that represents the user interface structure of the apps in an xml format. The layout file can be parsed, among other things, to identify the callback methods registered for GUI widget, which in turn improves the precision of generated call graphs. *CHEX* [64] and *BlueSeal* [54], [89] are among the techniques that leverage layout files for this purpose.

Moreover, the layout file contains information that is critical for security analysis. Password fields, which usually contains sensitive data, is an example of security-critical information embedded in layout files [13]. An example of a technique that leverages this information is *AsDroid* [55]. It examines the layout file to detect stealthy malicious behavior through identifying any contradiction between the actual app behavior and the user interface text initiating that behavior (e.g., the name of a button that was clicked), which denotes the user's expectation of program behavior.

In addition to the configuration files, most of the surveyed research perform analysis over apps' code. Different approaches analyze various formats of the Java code, which are broadly distinguishable as source code vs. byte code. The applicability of the former group of approaches, such as *SCanDroid* [43], are confined to apps with available source code.

Most recent approaches, however, support byte-code analysis. Such approaches typically perform a pre-processing step, in which Dalvik byte code, encapsulated in the APK file, is transferred to another type of code or intermediate representation (IR). Figure 7 shows the distribution of the approaches based on the target IR of the analysis.

Jimple [93] and Smali [8] are the most popular intermediate representations, used in 29% and 27% of the studied approaches, respectively. Dexpler [18] is a plugin for the Soot framework that translates Dalvik byte-code to Jimple. According to the diagram, 17% of the approaches, in the pre-processing step, retarget Dalvik byte-code to Java byte-coded JAR files. Examples of publicly available APK-to-JAR libraries widely used by approaches studied in this survey are dex2jar [3] and Dare [69]. An advantage of this approach is the ability to reuse pre-developed, off-the-shelf Java analysis libraries and tools. In exchange, APK-to-JAR decompilers suffer from performance overhead and incomplete code coverage.

Obfuscation challenges security analysis of application code. For this reason, nearly all of the surveyed static analyses cannot handle heavily obfuscated code. An example of a technique that handles certain obfuscations is *Apposcopy* [41]. It is a static approach that defines a high-level language for semantically specifying malware signatures. *Apposcopy* is evaluated against renaming, string encryption, and control-flow obfuscation.

Besides the type of obfuscations that Apposcopy is resilient to, more sophisticated obfuscations include hiding behaviors through native code, reflection, and dynamic class loading. Each of these types of obfuscations have highly limited support among Android security analysis techniques.

Among the static analysis techniques studied in our survey, none are able to perform analysis directly on native code, which is written in languages other than Java, such as C or C++. However, some approaches [49], [73], [109] can only identify the usage of native code, particularly if it is used in an abnormal way. For instance, *RiskRanker* [49] raises red flags if it finds encrypted native code, or if a native library is stored in a non-standardized place.

Few approaches consider dynamically loaded code, which occurs after app installation. Some static ap-

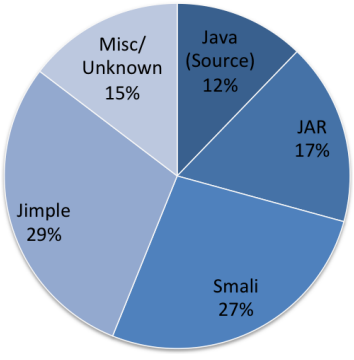

Fig. 7. Distribution of research based on the type of code or intermediate representation (IR) used for analysis.

proaches, such as the tool developed by Poeplau et al. [73], are able to identify the attempts to load external code that might be malicious. Nevertheless, more advanced techniques are required to distinguish the legitimate usages of dynamically loaded code from malicious ones. For example, handling of dynamically loaded code that considers an Android component's life-cycle, where a component can execute from multiple entry points, is not considered. As another example, dynamically loaded code that is additionally encrypted poses another challenge to static or hybrid analyses.

Approaches that consider Java reflection can be classified into two categories. One category, adopts a conservative, black-box approach and simply marks all reflective calls as suspicious. An example of such an approach is *AdRisk* [50]. The other thrust of research attempts to resolve reflection using more advanced analysis. For example, *DroidSafe* [47] employs string and points-to analysis to replace reflective calls with direct calls. As another example, *Pegasus* [25] rewrites an app by injecting dynamic checks when reflective calls are made.

6.2 Approach Characteristics (Solution)

Table 2 presents a summary of the solution-specific aspects that are extracted from the collection of papers included in the literature review. In the following, we summarize the main results for each dimension in the solution category.

6.2.1 Type of Program Analysis

Table 2 separates the approaches with respect to the type of program analysis they leverage. As discussed in Section 5, dynamic analysis is unsound but precise, while static analysis is sound yet imprecise. According to their intrinsic properties, each type of analysis has its own merits and is more appropriate for specific objectives. In particular, for security analysis, soundness is considered to be more important than precision, since it is preferred to not miss any potential security threat, even with the cost of generating false warnings. This explains why the percentage of static analysis techniques (73%) surpasses the percentage of approaches that rely on dynamic analysis techniques (17%).

SCanDroid [43] and TaintDroid [32] are among the first to explore the use of static and dynamic analysis techniques respectively for Android security assessment. SCanDroid employs static analysis to detect data flows that violate the security policies specified within an app's configuration. TaintDroid leverages dynamic taint analysis to track the data leakage from privacy-sensitive sources to possibly malicious sinks.

In addition to pure static or dynamic approaches, there exist few hybrid approaches that benefit from the advantages of both static and dynamic techniques. These methods usually first apply static analysis to detect potential security issues, and then perform dynamic techniques to improve their precision by eliminating the false warnings. For example, SMV-HUNTER [52] first uses static analysis to identify potentially vulnerable apps to SSL/TLS man-in-the-middle attack, and then uses dynamic analysis to confirm the vulnerability by performing automatic UI exploration.

Despite the fact that Android apps are mainly developed in Java, conventional Java program analysis methods do not work properly on Android apps, mainly due to its particular event-driven programming paradigm. Such techniques, thus, need to be adapted to address Android-specific challenges. Here, we briefly discuss these challenges and the way they have been tackled in the surveyed papers.

Event-Driven Structure. Android is an event-driven platform, meaning that an app's behavior is formed around the events caused by wide usage of callback methods that handle user actions, component's life-cycle, and requests from other apps or the underlying platform. If an analysis fails to handle these callback methods correctly, models derived from Android apps are disconnected and unsound. This problem has been discussed and addressed in several prior efforts. Among others, Yang et al. [101] introduced a program representation, called callback control-flow graph (CCFG), that supports capturing a rich variety of Android callbacks, including life-cycle and user interactions methods. To extract CCFG, a context-sensitive analysis traverses the control-flow of the program and identifies callback triggers along the visited paths.

Multiple Entry Points. Another dissimilarity between an Android app and a pure Java program, is the existence of multiple entry points in Android apps. In fact, unlike conventional Java applications with a single *main* method, Android apps comprise several methods that are implicitly called by the Android framework based on the state of the application (e.g., *onResume* to resume a paused app).

The problem of multiple entry points has been considered by a large body of work in this area [13], [54], [61], [64], [89], [102]. For instance, *FlowDroid* [13] models different Android callbacks, including the ones that handle life-cycle, user interface, and system-based events by creating a "dummy" main method that resembles the *main* method of conventional Java applications. Similar to FlowDroid, *IccTA* [61], [63] also generates dummy main methods, but rather than a single method for the whole app, it considers one per component. In addition to handling multiple entry points problem, the way that entry points are discovered is also crucial for a precise analysis. Some approaches [51] [108] simply rely on the domain knowledge, including the Android API documentation, to identify entry points. Some other approaches

TABLE 2
Solution Specific Categorization of the Reviewed Research

Dimension		Approaches	Percentage
Type of Program Analysis	Static	AdDroid [72], AdRisk [50], Amandroid [96], AndroidLeaks [44], Apposcopy [41], AsDroid [55], Batyuk- [19], BlueSeal [54], Chabada [48], Chex [64], ComDroid [26], COPES [17], COVERT [16], DidFail [60],Drebin [12], DroidChecker [23], DroidRanger [109], DroidSafe [47], Enck- [33], Epicc [70], Flowdroid [13], FUSE [77], IccTA [61], Kirin [34], LeakMiner [102], MalloDroid [37], Mann- [65], Mudflow [15], PCLeaks [62], Pegasus [25], PermCheckTool [95], PermissionFlow [85], Poeplau- [73], PScout [14], Riskranker [49], ScanDal [57], SCanDroid [43], Scoria [94], SEFA [98], Sparta [36], Stowaway [38], TrustDroid [105], Woodpecker [51]	73%
	Dynamic	AppInspector [45], AppIntent [103], AppsPlayground [76], CopperDroid [78], CopperDroid2 [92], DroidScope [100], DroidTrack [82], Marforio- [66], TaintDroid [32], VetDroid [104], Xmandroid [21]	17%
	Hybrid	ContentScope [108], Woodpecker [51], Droidmat [97], Permlyzer [99], SmartDroid [106], SMV-HUNTER [52]	10%
Supplementary Techniques	Machine Learning	Chabada [48], Drebin [12], Droidmat [97], Mudflow [15]	7%
	Formal Analysis	Apposcopy [41], AsDroid [55], Bagheri- [?], COVERT [16], Mann- [65], Pegasus [25], ScanDal [57], SCanDroid [43], Scoria [94]	13%
Automation Level	Automatic	AdDroid [72], Amandroid [96], AndroidLeaks [44], AppInspector [45], AppIntent [103], AppsPlayground [76], AsDroid [55], BlueSeal [54], Chabada [48], Chex [64], ComDroid [26], ContentScope [108], COPES [17], CopperDroid [78], CopperDroid2 [92], COVERT [16], DidFail [60], DroidChecker [23], DroidRanger [109], DroidSafe [47], DroidScope [100], DroidTrack [82], Enck- [33], Epicc [70], Flowdroid [13], FUSE [77], IccTA [61], Kirin [34], LeakMiner [102], MalloDroid [37], Marforio- [66], Mudflow [15], PCLeaks [62], Pegasus [25], PermCheckTool [95], PermissionFlow [85], Permlyzer [99], Poeplau- [73], PScout [14], Riskranker [49], ScanDal [57], SCanDroid [43], SEFA [98], SmartDroid [106], SMV-HUNTER [52], Stowaway [38], TaintDroid [32], TrustDroid [105], VetDroid [104], Woodpecker [51], Xmandroid [21], Drebin [12], Droidmat [97]	88%
	Semi-Automatic	AdRisk [50], Apposcopy [41], Batyuk- [19], Chaudhuri- [24], Mann- [65], Scoria [94], Sparta [36]	12%
Analysis Data Structure	Control Flow Graph (CFG)	AsDroid [55], Chex [64], ContentScope [108], DroidChecker [23], Enck- [33], PCLeaks [62], Pegasus [25], ScanDal [57], SCanDroid [43]	15%
	Call Graph (CG)	AdDroid [72], AndroidLeaks [44], AppIntent [103], AsDroid [55], BlueSeal [54], ContentScope [108], COPES [17], COVERT [16], DroidRanger [109], DroidSafe [47], Epicc [70], FUSE [77], LeakMiner [102], Mudflow [15], Pegasus [25], PermCheckTool [95], PermissionFlow [85], PScout [14], Riskranker [49], SCanDroid [43], SEFA [98], SMV-HUNTER [52], TaintDroid [32], TrustDroid [105]	40%
	Customized Data Structures	Amandroid [96], Apposcopy [41], DroidChecker [23], Epicc [70], Flowdroid [13], IccTA [61], Poeplau- [73], Chex [64], COVERT [16], SmartDroid [106], Woodpecker [51]	19%
Input Generation Technique	Fuzzing	AppsPlayground [76], ContentScope [108], CopperDroid [78], CopperDroid2 [92], DroidScope [100], DroidTrack [82], Marforio- [66], Permlyzer [99], SmartDroid [106], SMV-HUNTER [52], TaintDroid [32], VetDroid [104]	20%
	Symbolic Execution	AppInspector [45], AppIntent [103], Woodpecker [51]	5%

employ more systematic methods. For instance, *CHEX* describes a sound method to automatically discover different types of app entry points [64].

Inter-component communication. Android apps are composed of multiple components. The most widely used mechanism provided by Android to facilitate communication between components involves Intent, i.e., a specific type of event message in Android, and Intent Filter. The Android platform then automatically matches an Intent with the proper Intent Filters at runtime, which induce discontinuities in the statically extracted app models. This event-based inter-component communication (ICC) should be treated carefully, otherwise important security issues could be missed. The ICC challenge has received a lot of attention in the surveyed research [26], [33], [60], [61], [70]. *Epicc* [70], among others, is an approach devoted to identify inter-component communications by resolving links between components. It reduces the problem of finding ICCs to an instance of the inter-procedural distributive environment (IDE) problem [79], and then uses an IDE algorithm to solve the ICC resolution problem efficiently.

Modeling the underlying framework. In order to reason about the security properties of an app, the underlying Android platform should be also considered and included in the security analysis. However, analyzing the whole Android framework would result in state explosion and scalability issues. Therefore, a precise, yet scalable model, of the Android framework is crucial for efficient security assessment.

Various methods have been leveraged by the surveyed approaches to include the Android framework in their analysis. *Woodpecker* [51] uses a summary of Android built-in classes, which are pre-processed ahead of an app analysis to reduce the analysis costs associated with each app. To enable a more flexible analysis environment, *CHEX* [64] runs in two modes. In one mode, it includes the entire Android framework code in the analysis, and in the other only a partial model of the Android's external behaviors is used. To automatically classify Android system APIs as sources and sinks, *SuSi* [74] employs machine learning techniques. Such a list of sources and sinks of sensitive data is then used in a number of other surveyed approaches, including, *FlowDroid* [13], *DroidForce* [75], *IccTA* [61], [63], and *DidFail* [60].

6.2.2 Supplementary Techniques

We observe that most approaches (80%) only rely on program analysis techniques to assess the security

of Android software. Only 20% of the approaches employ other complementary techniques in their analysis. Among them, formal analysis and machine learning techniques are the most widely used, comprising 13% and 7% of the overall set of papers collected for this literature review, respectively.

These approaches typically first use some type of program analysis to extract specifications from the Android software that are input to the analysis performed by other supplementary techniques. For example, COVERT, combines formal app models that are extracted through static analysis with a formal specification of the Android framework to check the overall security posture of a system [16].

Machine learning techniques are mainly applied to distinguish between benign and malicious apps. The underlying assumption in this thrust of effort is that abnormal behavior is a good indicator of maliciousness. Examples of this class of research are *CHABADA* [48] and its successor *MUDFLOW* [15], which are both intended to identify abnormal behavior of apps. The focus of CHABADA is to find anomalies between app descriptions and the way APIs are used within the app. MUDFLOW tries to detect the outliers with respect to the sensitive data that flow through the apps.

6.2.3 Automation Level

We observe that most approaches (88%) are designed to perform Android security analysis in a completely automated manner, which is promising as it enables wide-scale evaluation of such automated techniques, discussed more in the following section (§ 6.3).

A number of approaches, however, require some manual effort (12%), for example annotating an app's code with labels representing different security concerns. Once the code is annotated manually, an automatic analysis is run to identify the security breaches or attacks in the source code. For instance, *Sparta* [36] requires app developers to annotate an app's source code with information-flow type qualifiers, which are fine-grained permission tags, such as INTERNET, SMS, GPS, etc. Subsequently, app repository auditors can employ Sparta's type system to check information flows that violate the secure flow policies. Manually applying the annotations affects usability and scalability of such approaches, however, enables a more precise analysis to ensue.

6.2.4 Analysis Data Structures

Data structures that represent apps in an abstract level are commonly used in the program analysis. We observe that call graph (CG) is the most frequently used data structure in the surveyed papers (40%).

Taint information are propagated through call graph, among other things, to determine the reachability of various sinks from specific sources. *LeakMiner* [102], *RiskRanker* [49], *TrustDroid* [105], *ContentScope* [108] and *IPC Inspection* [40] are some examples that traverse the call graph for taint analysis. Among others, *ContentScope* traverses CG to find paths form public content provider interfaces to the database function APIs in order to detect database leakage or pollution.

Moreover, generating and traversing the app's CG is also essential in tracking the message (i.e., Intent) transfer among the app's components. *Epicc* [70] and *AsDroid* [55] are among the approaches that use call graph for this purpose. In addition, *PScout* [14] and *PermissionFlow* [85] perform reachability analysis over the CG to map Android permissions to the corresponding APIs.

Control flow graph (CFG) is also widely used in the surveyed analysis methods (15%). *ContentScope* [108], for example, extracts an app's CFG to obtain the constraints corresponding to potentially dangerous paths. The collected constraints are then fed into a constraint solver to generate inputs corresponding to candidate path executions. Enck et al. [33] have also specified security rules over CFG to enable a control-flow based vulnerability analysis of Android apps.

More advanced and comprehensive program analyses rely on a combination of CFG and CG, a data structure called inter-procedural control flow graph (ICFG) that links the individual CFGs according to how they call each other. *FlowDroid* [13], for example, traverses ICFG to track tainted variables; *Epicc* [70] also performs string analysis over ICFG; *IccTA* [61], [63] detects inter-component data leaks by running data-flow analysis over such a data structure. Since the generated ICFG for the entire application is massive, complicated, and potentially unscalable, a number of approaches leverage a reduced version of ICFG for their analysis. For example, *Woodpecker* [51] locates capability leaks (cf. section 6.1.1) by traversing a reduced permission-specific ICFG, rather than the generic one.

In addition to such canonical, widely-used data structures, a good portion of existing approaches leverage customized data structures for app analysis (19%). One examples is G*, an ICFG-based graph, in which each call site is represented by two nodes, one before the procedure call and the other after returning [70]. *CHEX* [64] introduces two customized data structures of split data-flow summary (SDS) and permutation data-flow summary (PDS) for its data flow analysis. SDS is a kind of CFG that also considers the notion of split, *"a subset of the app code that is reachable from a particular entry point method"*. PDS is also similar to ICFG, and links all possible permutations of SDS sequences.

6.2.5 Input Generation Technique

The Android security assessment approaches that rely on dynamic analysis require test input data and events to drive the execution of apps.

TABLE 3
Assessment Specific Categorization of the Reviewed Research

Dimension		Approaches	Percentage
Evaluation	Case Studies	DroidScope [100], DidFail [60], SCanDroid [43], SmartDroid [106], DroidTrack [82], Scoria [94]	12%
	Empirical	AdDroid [72], AdRisk [50], Amandroid [96], AndroidLeaks [44], AppInspector [45], AppIntent [103], Apposcopy [41], AppsPlayground [76], AsDroid [55], Batyuk- [19], BlueSeal [54], Chabada [48], Chex [64], ComDroid [26], ContentScope [108], COPES [17], CopperDroid [78], CopperDroid2 [92], COVERT [16], DroidChecker [23], DroidRanger [109], DroidSafe [47], Enck- [33], Epicc [70], Flowdroid [13], FUSE [77], IccTA [61], Kirin [34], LeakMiner [102], MalloDroid [37], Marforio- [66], Mudflow [15], PCLeaks [62], Pegasus [25], PermCheckTool [95], PermissionFlow [85], Permlyzer [99], Poeplau- [73], PScout [14], Riskranker [49], ScanDal [57], SEFA [98], SmartDroid [106], SMV-HUNTER [52], Sparta [36], Stowaway [38], TaintDroid [32], VetDroid [104], Woodpecker [51], Xmandroid [21]	86%
	Proof	Apposcopy [41], Chaudhuri- [24], Mann- [65], ScanDal [57], Scoria [94]	10%
Tool	Executable Artifact	Amandroid [96], Chabada [48], ComDroid [26], CopperDroid [78], CopperDroid2 [92], COVERT [16], DidFail [60], DroidSafe [47], DroidScope [100], Enck- [33], Epicc [70], Flowdroid [13], FUSE [77], IccTA [61], Kirin [34], MalloDroid [37], Mudflow [15], PermCheckTool [95], PScout [14], SCanDroid [43], Sparta [36], Stowaway [38], TaintDroid [32]	39%
	Source Code	Amandroid [96], DidFail [60], DroidSafe [47], DroidScope [100], Enck- [33], Flowdroid [13], IccTA [61], Kirin [34], MalloDroid [37], PermCheckTool [95], PScout [14], SCanDroid [43], TaintDroid [14]	22%
	Documentation	Amandroid [96], CopperDroid [78], CopperDroid2 [92], COVERT [16], DidFail [60], DroidSafe [47], DroidScope [100], Enck- [33], Epicc [70], Flowdroid [13], FUSE [77], IccTA [61], Kirin [34], MalloDroid [77]	24%

We can observe from Table 2 that most of such approaches use fuzz testing, comprising 20% of the overall set of papers collected for this literature review. Fuzzing is a form of negative testing that feeds malformed and unexpected input data to a program with the objective of revealing security vulnerabilities. For example, it has been shown that an SMS protocol fuzzer is highly effective in finding severe security vulnerabilities in all three major smartphone platforms [68]. In the case of Android, fuzzing found a security vulnerability triggered by simply receiving a particular type of SMS message, which not only kills the phone's telephony process, but also kicks the target device off the network [68].

Despite the individual success of fuzzing as a general method of identifying vulnerabilities, fuzzing has traditionally been used as a brute-force mechanism. Using fuzzing for testing is generally a time consuming and computationally expensive process, as the space of possible inputs to any real-world program is often unbounded. Existing fuzzing tools, such as Android's Monkey [2], generate purely random test case inputs, and thus are often ineffective in practice.

A comparatively low number of approaches (5%) employ symbolic execution, mainly to improve the effectiveness of generated test inputs. For example, AppInspector [45] applies concolic execution, which is the combination of symbolic and concrete execution. It switches back and forth between symbolic and concrete modes to enable analysis of apps that communicate with remote parties. Scalability is, however, a main concern with symbolic execution techniques. More recently, some approaches try to improve the scalability of symbolic execution. For instance, AppIntent [103] introduces a guided symbolic execution that narrows down the space of execution paths to be explored by considering both the app call graph and the Android execution model. Symbolic execution is also used for feasible path refinement. Among others, Woodpecker [51] models each execution path as a set of dependent program states, and marks a path "feasible" if each program point follows from the preceding ones.

6.3 Assessment (Validation)

We used reputable sites in our review protocol (cf. section 4), which resulted in the discovery of high-quality refereed research papers from respectable venues. To develop better insights into the quality of the research papers surveyed, here we use Evaluation Method (T 3.1) and Replicability (T 3.2), which are the two validation dimensions in the taxonomy.

Table 3 presents a summary of the validation-specific aspects that are extracted from the collection of papers included in the literature review. In the following, we summarize the main results for each dimension in this category.

6.3.1 Evaluation Method

The first row in Table 3 depicts the share of different evaluation methods in assessing the quality of Android security analysis approaches. Most (86%) of the approaches have used empirical techniques to assess the validity of their ideas using a full implementation of their approach (e.g., Chabada [48], Chex [64], Epicc [70], and COVERT [16]). Some research efforts have developed a proof-of-concept prototype to perform limited scale case studies (e.g., SCanDroid [43] and SmartDroid [106]). A limited number (10%) of approaches (e.g., Chaudhuri et al. [24]) have provided mathematical proofs to validate their ideas.

Availability of various Android app repositories, such as the Google Play Store [6], is a key enabling factor for the large-scale empirical evaluation witnessed in the Android security research. Figure 8

shows the distribution of surveyed research based on the number of selected apps that are used in the experiments. We observe that most of the experiments (82%) have been conducted over sets of more than one hundred apps.

Figure 9 depicts the distribution of app repositories used in the evaluations of surveyed research. It can be observed that the Google Play Store, the official and largest repository of Android applications, is the most popular app repository, used by 70% of the papers. There are several other third-party repositories, such as F-Droid open source repository [**?**], used by 20% of the surveyed research. A number of malware repositories (such as [9], [10], [107]) are also widely used in assessing approaches designed for detecting malicious apps (32%). Finally, about 18% of the surveyed research use hand-crafted benchmark suites, such as [4], [7], in their evaluation. A benefit of apps comprising such benchmarks is that the ground-truth for them is known, since they are manually seeded with known vulnerabilities and malicious behavior, allowing researchers to easily assess and compare their techniques in terms of the number of issues that are correctly detected.

6.3.2 *Replicability*

The evaluation of security research is generally known to be difficult. Making the results of experiments reproducible is even more difficult. The second row in Table 3 shows the availability of the executable artifact, as well as the corresponding source code and documentations. According to Table 3, overall only 39% of published research have made their artifacts publicly available. The rest have not made their implementations, prototypes, tools, and experiments available to other researchers.

Having such artifacts publicly available enables, among other things, quantitative comparisons of different approaches. Figure 10 depicts the comparison relationships found in the evaluation of the studied papers. In this graph, X $\rightarrow$ Y means research method X compared itself to method Y. The nodes with higher fan-in (i.e., incoming edges) represent the tools that are widely used in evaluation of other research efforts. For instance, *FlowDroid* [13], with 6 incoming edges, has an active community of developers and discussion group, and is widely used in several research papers surveyed in our study.

6.4 Cross Analysis

In this section, we extend our survey analysis across the different taxonomy dimensions. Given the observations from the reviewing process, we develop the following cross-analysis questions (CQs):

- **CQ1.** What type of program analysis have been used for each security assessment objectives?
- **CQ2.** What type of program analysis have been used for detecting each of the STRIDE's security threats?
- **CQ3.** Is there a relationship between the granularity of security threats and the type of employed program analysis techniques?
- **CQ4.** Is there a relationship between the depth of security threats, i.e., app-level vs. framework-level, and the type of analysis techniques employed in the surveyed research?
- **CQ5.** Which evaluation method(s) is used for different objectives and types of analysis?
- **CQ6.** How reproducible are the surveyed research based on the objectives and types of analysis?

CQ1 Analysis objectives and type of program analysis. As shown in Figure 11ⓐ, static analysis has been widely used for identifying both malicious behavior and vulnerabilities (69%-89%). Pure dynamic analysis, however, are only employed for malware detection, indicating a potential opportunity for further research. Hybrid approaches, though at lower scales, have also been used (9%-11%) for both purposes.

CQ2 STRIDE's security threats and type of program analysis. According to Figure 11ⓑ, static analysis has been widely used for identifying various security threats, except the repudiation that is not considered in the literature at all. Dynamic analysis has also been used for detecting both elevation of

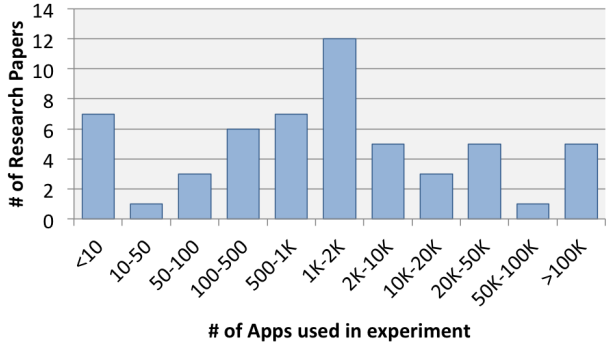

Fig. 8. Distribution of surveyed research based on the number of apps used in their experiments.

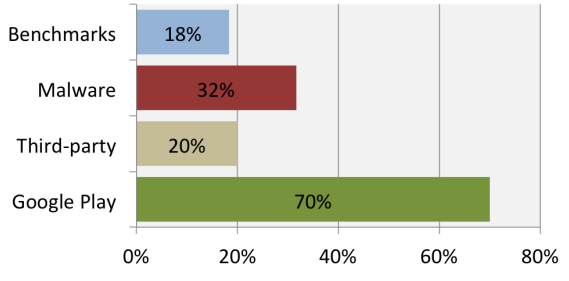

Fig. 9. Distribution of App Repositories used in the Empirical or Case Study-based Evaluations.

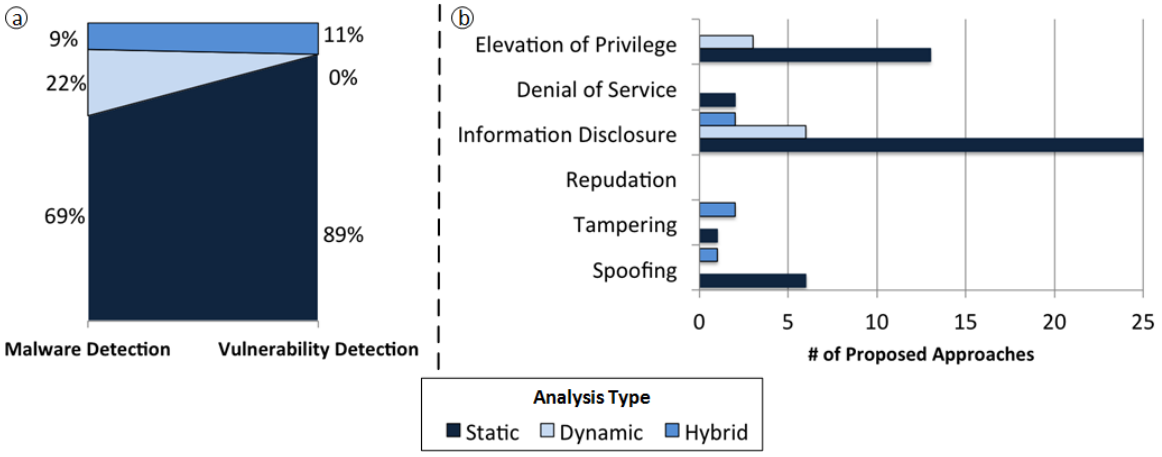

Fig. 11. Types of program analysis have been used for ⓐ detecting each security assessment objectives (i.e. Malware vs. Vulnerability Detection), and ⓑ each type of STRIDE's security threats.

privilege and information disclosure. Finally, hybrid approaches have been employed for detecting spoofing, tampering, and information disclosure.

CQ3 Granularity of security threats and type of analysis techniques. Static analysis techniques are the most common methods (about 80%) used by the state-of-the-art approaches in both single and compositional app analysis (cf., Figure 12ⓐ). However, it can be observed that due to the high complexity of problems that arise in compositional app analysis, a good portion of approaches have used hybrid methods (18%) to identify the security issue may arise from the interaction of multiple components (inter-component).

CQ4 Depth of security threats and type of analysis techniques. The depth of security threats also exhibit a relation with the type of analysis techniques (cf., Figure 12ⓑ). We observe that the dynamic approaches are employed more often for analysis at the framework-level (48%). One reason is that dynamic approaches can employ runtime modules, such as monitors, which are deployed in the Android framework, thereby enabling tracking otherwise implicit relations between system API calls and the Android permissions. Such runtime framework-level activity monitoring is not readily possible using static analysis techniques.

CQ5 Reproducibility vs. the objectives and types of analysis. We observe a similar distribution pattern in use of different evaluation methods across various analysis objectives and types of analysis. Empirical evaluation is the most widely used, followed by the case study method and formal proof (cf., Figure 13).

CQ6 Reproducibility vs. the objectives and types of analysis. As shown in Figure 14, the research artifacts intended to identify security vulnerabilities are more likely to be available in comparison to those designed for malware detection (47% vs. 37%). We also observe that no tool using hybrid (both static and dynamic) program analyses is publicly available, which prevents the other researchers from reproducing, and potentially adopting, achievements in this thrust of research.

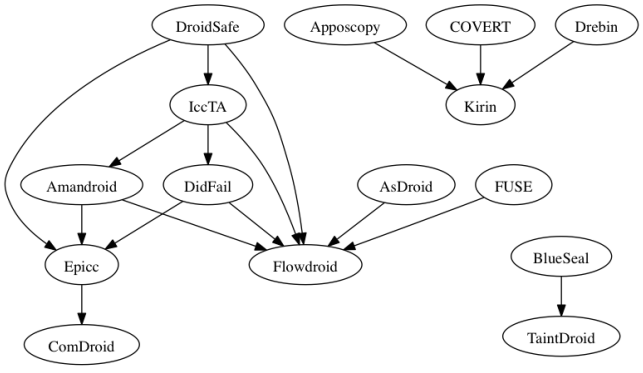

Fig. 10. Comparison Graph: X → Y means research method X has quantitatively compared itself to method Y.

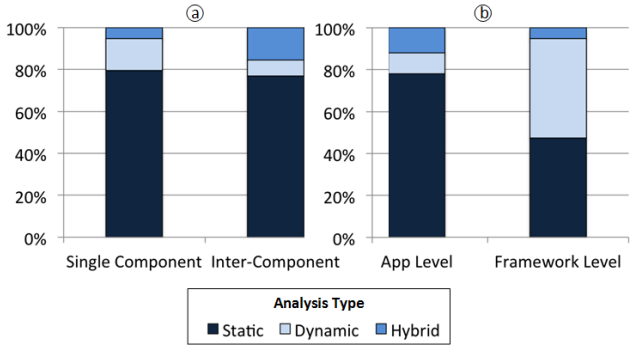

Fig. 12. ⓐ Breadth and ⓑ Depth (Level) of each type of program analysis.

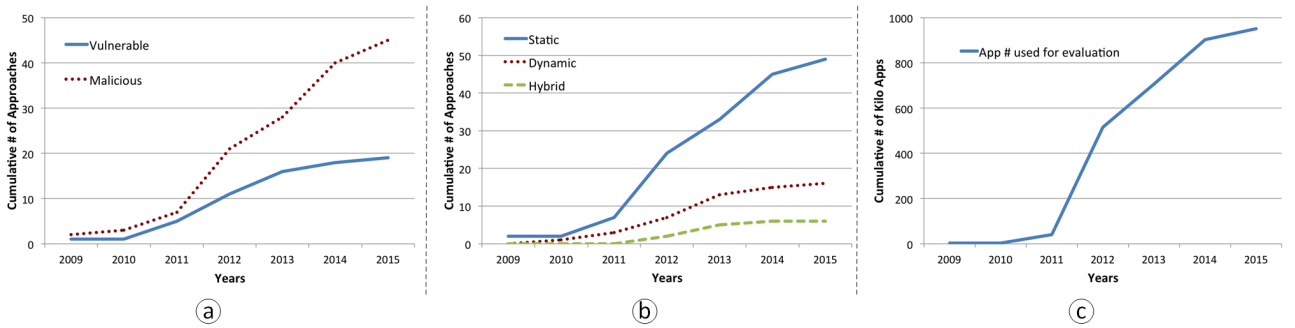

Fig. 15. Observed trends in Android security analysis research with respect to ⓐ objectives of the analysis, ⓑ type of analysis, and ⓒ number of apps used in the evaluation.

7 Discussion and Directions for Future Research

To address the third research question (RQ3), in this section, we first provide a trend analysis of surveyed research, and then discuss the observed gaps in the studied literature that can help to direct future research efforts in this area.

Based on the results of our literature review (cf., Section 6), it is evident that Android security has received a lot of attention in recently published literature, due mainly to the popularity of Android as a platform of choice for mobile devices, as well as increasing reports of its vulnerabilities. We also observe important trends in the past decade, as reflected by the results of the literature review. Figure 15 shows some observed trends in Android security analysis research.

- According to Figure 15ⓐ, malicious behavior detection not only has attracted more attention, compared to vulnerability identification, but also research in malware analysis tends to grow at an accelerated rate.
- As illustrated in Figure 15ⓑ, static analysis techniques dominate security assessment in the Android domain. Dynamic and hybrid analysis techniques are also showing modest growth, as they are increasingly applied to mitigate the limitations of pure static analysis (e.g., to reason about dynamically loaded code, and obfuscated code).
- The more recent approaches reviewed in this survey have used larger collections of apps in their evaluation (cf., Figure 15ⓒ). Such large-scale empirical evaluation in the Android security research is promising, and can be attributed to the meteoric rise of the numbers of apps provisioned on publicly available app markets that in some cases provide free or even open-source apps.

Despite considerable research efforts devoted to mitigating security threats in mobile platforms, we are still witnessing a significant growth in the number of security attacks targeting these platforms [80]. Therefore, our first and foremost recommendation is to increase convergence and collaboration among researchers in this area from software engineering, security, mobility, and other related communities to achieve the common goal of addressing these mobile security threats and attacks.

More specifically, the survey—through its use of our proposed taxonomy—has revealed research gaps in need of further study. To summarize, future research needs to focus on the following to stay ahead of today's advancing security threats:

- *Pursue integrated and hybrid approaches that span not only static and dynamic analyses, but also other supplementary analysis techniques:* Recall from Table 2 that only 20% of approaches leverage supplemen-

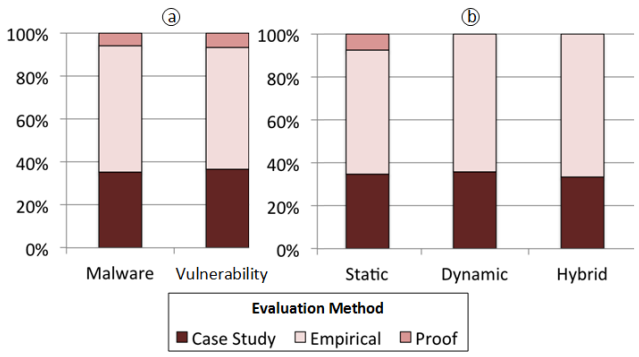

Fig. 13. Approach validation versus ⓐ research objectives and ⓑ types of analysis.

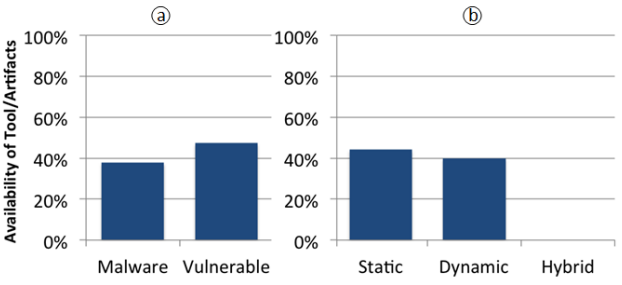

Fig. 14. Availability of tools/artifacts based on the ⓐ objective and, ⓑ type of analysis.

tary techniques, which are shown to be effective in identifying modern malicious behaviors or security vulnerabilities.

- *Move beyond fuzzing for security test input generation:* According to Table 2, only 20% of test input generation techniques use a systematic technique (i.e., symbolic execution), as opposed to brute-force fuzzing. Fuzzing is inherently limited in its abilities to execute vulnerable code. Furthermore, such brute-force approaches may fail to identify malicious behavior that may be hidden behind obfuscated code or code that requires specific conditions to execute.
- *Continue the paradigm shift from basic single app analysis to overall system monitoring, and exploring compositional vulnerabilities:* Recall from Sections 6.1.3 and 6.1.4, and Table 1, that the main body of research is limited to analysis of single apps. However, malware exploiting vulnerabilities of multiple benign apps in tandem on the market are increasing. Furthermore, identifying some security vulnerabilities, such as the case described in [**?**], require a holistic analysis of the Android framework.
- *Construct techniques capable of analyzing ICC beyond Intents:* Only 5% of papers, as shown in Table 1, consider ICCs involving data sharing using Content Providers and AIDL. These mechanisms are, thus, particularly attractive vectors for attackers to utilize, due to the limited analyses available. Consequently, research in that space can help strengthen countermeasures against such threats.
- *Consider dynamically loaded code that is not bundled with installed packages:* Recall from Table 1 that a highly limited amount of research (8%) analyzes the security implications of externally loaded code. This Android capability can be easily misused by malware developers to evade security inspections at installation time.
- *Analyze code of different forms and from different languages:* Besides analyzing Java and its basic constructs, future research should analyze other code constructs and languages used to construct Android apps, such as native C/C++ code or obfuscated code. The usage of complicated obfuscation techniques and/or native libraries for hiding malicious behavior are continually growing. Recall from section 6.1.5 and Table 1 that only 2% and 10% of surveyed approaches consider obfuscated and native codes, respectively.
- *Consider studying Android repudiation:* The SLR process returned no results for Android repudiation, as shown in Table 1. Consequently, there is a large opening for studying such threats, particularly in terms of digital signatures, certificates, and encryption. However, repudiation also has a major legal component [**?**], which may require expertise not held by researchers in software security, software engineering, or computer science. This gap may require inter-disciplinary research to properly fill.
- *Promote collaboration in the research community:* To that end, we recommend making research more reproducible. This goal can be achieved through increased sharing of research artifacts. Recall from Table 3 that less than 40% of reviewed research artifacts are publicly provided. To further aid in achieving reproducibility, researchers can focus on developing common evaluation platforms and benchmarks: Recall from Figure 9 that only 18% of studied approaches considered benchmarks for their evaluation. At the same time, Figure 10 shows that few approaches conduct quantitative comparisons, mainly due to unavailability of prior research artifacts.

8 Conclusion

In parallel with the growth of mobile applications and consequently the rise of security threats in mobile platforms, considerable research efforts have been devoted to assess the security of mobile applications. Android, as the dominant mobile platform and also the primary target of mobile malware threats, has been in the focus of much research. Existing research has made significant progress towards detection and mitigation of Android security.

This article proposed a comprehensive taxonomy to classify and characterize research efforts in this area. We have carefully followed the systematic literature review process, resulting in the most comprehensive and elaborate investigation of the literature in this area of research, comprised of 100 papers published from 2008 to 2015. The research has revealed patterns, trends, and gaps in the existing literature, and underlined key challenges and opportunities that will shape the focus of future research efforts.

In particular, the survey showed the current research should advance from focusing primarily on single app assessment to a more broad and deep analysis that considers combination of multiple apps and Android framework, and also from pure static or dynamic to hybrid analysis techniques. We also identified a gap in the current research with respect to special vulnerable features of the Android platform, such as native or dynamically loaded code. Finally, we encourage researchers to publicly share their developed tools, libraries and other artifacts to enable the community to compare and evaluate their techniques and build on prior advancements. We believe the results of our review will help to advance the much needed research in this area and hope the taxonomy itself will become useful in the development and assessment of new research directions.

REFERENCES

[1] "Androguard." [Online]. Available: https://code.google.com/p/androguard/

[2] "Android monkey." [Online]. Available: http://developer.android.com/guide/developing/tools/monkey.html/

[3] "dex2jar." [Online]. Available: https://code.google.com/p/dex2jar/

[4] "Droidbench." [Online]. Available: http://sseblog.ec-spride.de/tools/droidbench

[5] "Droidbox." [Online]. Available: https://code.google.com/p/droidbox/

[6] "Google play market." [Online]. Available: http://play.google.com/store/apps

[7] "Icc-bench." [Online]. Available: https://github.com/fgwei/ICC-Bench

[8] "smali." [Online]. Available: https://code.google.com/p/smali/

[9] "Virusshare." [Online]. Available: http://virusshare.com/

[10] "Virustotal." [Online]. Available: https://www.virustotal.com/

[11] "Protecting your privacy: App ops, privacy guard, and xprivacy," Mar. 2015. [Online]. Available: http://www.xda-developers.com/protecting-your-privacy-app-ops-privacy-guard-and-xprivacy/

[12] D. Arp, M. Spreitzenbarth, M. Hbner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket," in *Proceedings of the 21st Annual Network & Distributed System Security Symposium*, San Diego, CA, 2014.

[13] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Edinburgh, UK: ACM, 2014, p. 29.

[14] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. Raleigh, NC: ACM, 2012, pp. 217–228.

[15] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," 2015.

[16] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "Covert: Compositional analysis of android inter-app permission leakage," *IEEE Transactions on Software Engineering (TSE)*, 2015.

[17] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Automatically securing permission-based software by reducing the attack surface: An application to android," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. Essen, Germany: ACM, 2012, pp. 274–277.

[18] ——, "Dexpler: converting android dalvik bytecode to jimple for static analysis with soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*. ACM, 2012, pp. 27–38.

[19] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A.-D. Schmidt, and S. Albayrak, "Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications," in *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software*, ser. MALWARE '11. Fajardo, PR: IEEE Computer Society, 2011, pp. 66–72.

[20] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from applying the systematic literature review process within the software engineering domain," *Journal of Systems and Software*, vol. 80, no. 4, pp. 571–583, Apr. 2007.

[21] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi, "Xmandroid: A new android evolution to mitigate privilege escalation attacks," *Technische Universitt Darmstadt, Technical Report TR-2011-04*, 2011.

[22] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on android." in *NDSS*, San Diego, CA, 2012.

[23] P. P. Chan, L. C. Hui, and S. M. Yiu, "DroidChecker: Analyzing android applications for capability leak," in *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '12. Tucson, Arizona: ACM, 2012, pp. 125–136.

[24] A. Chaudhuri, "Language-based security on android," in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, ser. PLAS '09. Dublin, Ireland: ACM, 2009, pp. 1–7.

[25] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. Rinard, and D. X. Song, "Contextual policy enforcement in android applications with permission event graphs." in *NDSS*, San Diego, CA, 2013.

[26] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*. Washington, DC: ACM, 2011, pp. 239–252.

[27] M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich, "Crêpe: A system for enforcing fine-grained context-related policies on android," *Information Forensics and Security, IEEE Transactions on*, vol. 7, no. 5, pp. 1426–1438, 2012.

[28] R. Cozza, I. Durand, and A. Gupta, "Market Share: Ultramobiles by Region, OS and Form Factor, 4Q13 and 2013," *Gartner Market Research Report*, February 2014.

[29] D. Dagon, T. Martin, and T. Starner, "Mobile phones as computing devices: The viruses are coming!" *Pervasive Computing, IEEE*, vol. 3, no. 4, pp. 11–15, 2004.

[30] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *13th International Conference*, ser. ISC'10, M. Burmester, G. Tsudik, S. Magliveras, and I. Ili, Eds. Boca Raton, FL, USA: Springer Berlin Heidelberg, Oct. 2010, pp. 346–360.

[31] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "QUIRE: Lightweight provenance for smart phone operating systems." in *USENIX Security Symposium*, San Francisco, CA, 2011.

[32] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones," pp. 393–407, 2010.

[33] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11. San Francisco, CA: USENIX Association, 2011, pp. 21–21.

[34] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM conference on Computer and communications security*. Chicago, IL: ACM, 2009, pp. 235–245.

[35] M. D. Ernst, "Static and dynamic analysis: synergy and duality," in *Proceedings of the ACM-SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2004, pp. 35–35.

[36] M. D. Ernst *et al.*, "Collaborative verification of information flow for a high-assurance app store," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. Scottsdale, AZ: ACM, 2014, pp. 1092–1104.

[37] S. Fahl, M. Harbach, T. Muders, L. Baumgrtner, B. Freisleben, and M. Smith, "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*, ser. CCS '12. Raleigh, NC: ACM, 2012, pp. 50–61.

[38] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. Chicago, IL: ACM, 2011, pp. 627–638.

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

[40] A. P. Felt, S. Hanna, E. Chin, H. J. Wang, and E. Moshchuk, "Permission re-delegation: Attacks and defenses," in *In 20th Usenix Security Symposium*, San Francisco, CA, 2011.

[41] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware," in *Int'l Symp. on the Foundations of Software Engineering*, Hong Kong, China, Nov. 2014.

[42] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey, "Modeling and enhancing androids permission system," in *17th European Symposium on Research in Computer Security*, ser. Lecture Notes in Computer Science, S. Foresti, M. Yung, and F. Martinelli, Eds. Pisa, Italy: Springer Berlin Heidelberg, Sep. 2012, pp. 1–18.

[43] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, *SCanDroid: Automated Security Certification of Android Applications*, 2009.

[44] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Automatically detecting potential privacy leaks in android applications on a large scale," in *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, ser. TRUST'12. Vienna, Austria: Springer-Verlag, 2012, pp. 291–307.

[45] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung, "Vision: automated security validation of mobile apps at app markets," in *Proceedings of the second international workshop on Mobile cloud computing and services*. ACM, 2011, pp. 21–26.

[46] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[47] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, "Information-Flow Analysis of Android Applications in DroidSafe," in *Proceedings of the 22st Network and Distributed System Security Symposium*, San Diego, CA, 2015.

[48] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 1025–1035.

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

[50] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WISEC '12. Tucson, AZ: ACM, 2012, pp. 101–112.

[51] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones." in *NDSS*, San Diego, CA, 2012.

[52] D. S. J. S. G. Greenwood and Z. L. L. Khan, "SMV-HUNTER: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in android apps," 2014.

[53] N. Hardy, "The confused deputy:(or why capabilities might have been invented)," *ACM SIGOPS Operating Systems Review*, vol. 22, no. 4, pp. 36–38, 1988.

[54] S. Holavanalli, D. Manuel, V. Nanjundaswamy, B. Rosenberg, F. Shen, S. Ko, and L. Ziarek, "Flow permissions for android," in *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, Nov. 2013, pp. 652–657.

[55] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "AsDroid: detecting stealthy behaviors in android applications by user interface and program behavior contradiction." in *ICSE*, Hyderabad, India, 2014, pp. 1036–1046.

[56] P. Institute, "Big data analytics in cyber defense," Feb. 2013. [Online]. Available: http://www.ponemon.org/library/big-data-analytics-in-cyber-defense

[57] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center, "ScanDal: Static analyzer for detecting privacy leaks in android applications," in *MoST 2012: Mobile Security Technologies*, 2012.

[58] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[59] B. Kitchenham, "Procedures for performing systematic reviews," *Keele, UK, Keele University*, vol. 33, p. 2004, 2004.

[60] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. Edinburgh, UK: ACM, 2014, pp. 1–6.

[61] L. Li, A. Bartel, T. Bissyande, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE 2015, Florence, Italy, 2015.

[62] L. Li, A. Bartel, J. Klein, and Y. L. Traon, "Automatically exploiting potential component leaks in android applications," in *Proceedings of the 13th International Conference on Trust, Security and Privacy in Computing and Communications*, Beijing, China, 2014, pp. 388–397.

[63] L. Li, A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "I know what leaked in your pocket: uncovering privacy leaks on android apps with static taint analysis," *arXiv:1404.7431 [cs]*, Apr. 2014, arXiv: 1404.7431. [Online]. Available: http://arxiv.org/abs/1404.7431

[64] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*. Raleigh, NC: ACM, 2012, pp. 229–240.

[65] C. Mann and A. Starostin, "A framework for static detection of privacy leaks in android applications," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, ser. SAC '12. Riva del Garda, Italy: ACM, 2012, pp. 1457–1462.

[66] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun, "Analysis of the communication between colluding applications on modern smartphones," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. Orlando, Florida: ACM, 2012, pp. 51–60.

[67] T. Martin, M. Hsiao, D. S. Ha, and J. Krishnaswami, "Denial-of-service attacks on battery-powered mobile computers," in *Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on*. IEEE, 2004, pp. 309–318.

[68] C. Miller and C. Mulliner, "Fuzzing the phone in your phone," in *Black Hat Technical Security Conference*, 2009.

[69] D. Octeau, S. Jha, and P. McDaniel, "Retargeting android applications to java bytecode," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 6.

[70] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Washington, DC: USENIX Association, 2013, pp. 543–558.

[71] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: Towards automating risk assessment of mobile applications," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Washington, DC: USENIX Association, 2013, pp. 527–542.

[72] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "AdDroid: Privilege separation for applications and advertisers in android," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '12. Seoul, Republic of Korea: ACM, 2012, pp. 71–72.

[73] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in android applications," in *Proceedings of the 20th Annual Network & Distributed System Security Symposium*, San Diego, California, 2014.

[74] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *2014 Network and Distributed System Security Symposium (NDSS)*, 2014.

[75] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden, "Droidforce: Enforcing complex, data-centric, system-wide policies in android," in *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*. IEEE, 2014, pp. 40–49.

[76] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: Automatic security analysis of smartphone applications," in *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '13. San Antonio, TX: ACM, 2013, pp. 209–220.

[77] T. Ravitch, E. R. Creswick, A. Tomb, A. Foltzer, T. Elliott, and L. Casburn, "Multi-app security analysis with FUSE: Statically detecting android app collusion," in *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, ser. PPREW-4. New Orleans, LA: ACM, 2014, pp. 4:1–4:10.

[78] A. Reina, A. Fattori, and L. Cavallaro, "A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors," in *ACM European Work-*

shop on Systems Security (EuroSec), Prague, Czech Republic, 2013.

[79] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1995, pp. 49–61.

[80] S. S. Response, "2015 internet security threat report," 2015. [Online]. Available: http://www.symantec.com

[81] A. Sadeghi, H. Bagheri, and S. Malek, "Analysis of android inter-app security vulnerabilities using covert," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE 2014. Florence, Italy: IEEE, 2015.

[82] S. Sakamoto, K. Okuda, R. Nakatsuka, and T. Yamauchi, "DroidTrack: Tracking and visualizing information diffusion for preventing information leakage on android," *Journal of Internet Services and Information Security (JISIS)*, vol. 4, no. 2, pp. 55–69, 2014.

[83] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[84] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar, "On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices." in *SECRYPT*, 2013, pp. 461–468.

[85] D. Sbirlea, M. Burke, S. Guarnieri, M. Pistoia, and V. Sarkar, "Automatic detection of inter-application permission leaks in android applications," *IBM Journal of Research and Development*, vol. 57, no. 6, pp. 10:1–10:12, Nov. 2013.

[86] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, and S. Dolev, "Google android: A state-of-the-art review of security mechanisms," *arXiv preprint arXiv:0912.5101*, 2009.

[87] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer, "Google android: A comprehensive security assessment," *IEEE security and Privacy*, vol. 8, no. 2, pp. 35–44, 2010.

[88] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "Andromaly: a behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems*, vol. 38, no. 1, pp. 161–190, 2012.

[89] F. Shen, N. Vishnubhotla, C. Todarka, M. Arora, B. Dhandapani, E. J. Lehner, S. Y. Ko, and L. Ziarek, "Information flows as a permission mechanism," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 515–526.

[90] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and A. Ribagorda, "Evolution, detection and analysis of malware for smart devices," *Communications Surveys & Tutorials, IEEE*, vol. 16, no. 2, pp. 961–987, 2014.

[91] F. Swiderski and W. Snyder, *Threat Modeling*. Microsoft Press, 2004.

[92] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic Reconstruction of Android Malware Behaviors," in *Proceedings of the 22st Network and Distributed System Security Symposium*, San Diego, CA, 2015.

[93] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations," 1998.

[94] R. Vanciu and M. Abi-Antoun, "Finding architectural flaws using constraints," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. Silicon Valley, CA: IEEE, 2013, pp. 334–344.

[95] T. Vidas, N. Christin, and L. Cranor, "Curbing android permission creep," in *2011 Web 2.0 Security and Privacy Workshop*, vol. 2, Oakland, CA, 2011.

[96] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. Scottsdale, AZ: ACM, 2014, pp. 1329–1341.

[97] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "Droidmat: Android malware detection through manifest and api calls tracing," in *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*. IEEE, 2012, pp. 62–69.

[98] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The impact of vendor customizations on android security," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '13. Berlin, Germany: ACM, 2013, pp. 623–634.

[99] W. Xu, F. Zhang, and S. Zhu, "Permlyzer: Analyzing permission usage in android applications," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, Nov. 2013, pp. 400–410.

[100] L. K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12. Bellevue, WA: USENIX Association, 2012, pp. 29–29.

[101] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in Android applications," in *International Conference on Software Engineering*, 2015.

[102] Z. Yang and M. Yang, "LeakMiner: Detect information leakage on android with static taint analysis," in *2012 Third World Congress on Software Engineering (WCSE)*, Hong Kong, China, Nov. 2012, pp. 101–104.

[103] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: analyzing sensitive data transmission in android for privacy leakage detection," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '13. Berlin, Germany: ACM, 2013, pp. 1043–1054.

[104] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '13. Berlin, Germany: ACM, 2013, pp. 611–622.

[105] Z. Zhao and F. Osono, "TrustDroid: Preventing the use of SmartPhones for information leaking in corporate networks through the used of static analysis taint tracking," in *2012 7th International Conference on Malicious and Unwanted Software (MALWARE)*, Fajardo, PR, Oct. 2012, pp. 135–143.

[106] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "SmartDroid: An automatic system for revealing UI-based trigger conditions in android applications," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '12. New York, NY, USA: ACM, 2012, pp. 93–104.

[107] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy (SP), 2012 IEEE Symposium on*. San Francisco, CA: IEEE, 2012, pp. 95–109.

[108] ——, "Detecting passive content leaks and pollution in android applications." in *NDSS*, San Diego, CA, 2013.

[109] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets." in *NDSS*, San Diego, CA, 2012.