UNIVERSITY OF CALIFORNIA,
IRVINE

Efficient Permission-Aware Analysis of Android Apps

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

Alireza Sadeghi

Dissertation Committee:
Associate Professor Sam Malek, Chair
Associate Professor James A. Jones
Professor Cristina Videira Lopes

2017

DEDICATION

To my better half, Reyhan—
The best friend, classmate, and colleague.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost, I offer my deepest gratitude and appreciation to my advisor, Professor Sam Malek, for his guidance, support, and inspiration. His guidance during this journey, from short comments on a paper draft to long one-on-one meetings, taught me a life lesson that could be summarized in this well-known quote: "Good, better, best. Never let it rest. Until your good is better and your better is best."

I would like to thank the other members of my committee, Professors James Jones and Cristina Lopes, for their valuable feedback and devoting time. I acknowledge and appreciate the contribution of Dr. Hamid Bagheri in this dissertation. I also enjoyed collaborating with other members of SEAL lab at UCI and GMU, specially Dr. Naeem Esfahani, Dr. Joshua Garcia, and Dr. Nariman Mirzaei.

I thank my parents Maryam Shobeiri and Nasser Sadeghi for their sacrifice, support, and pray. I know it has been a tough time for them to bear my absence at all moments of happiness and sorrow, which I wish I could be with them. I also thank my parents-in-law, Dr. Fariba Khoshzaban, and Dr. Mahmoud Jabbarvand for their kind help and support during all the past years.

Finally, I wish I knew how to truly thank my wife, Reyhaneh Jabbarvand, for her endless sacrifice, inspiration, and love. This dissertation would not have been possible without her support, encouragement, and even her technical contributions. There were many moments of despair during this journey, where she helped me to solve challenging problems with brilliant ideas. It was a blessing and unique opportunity in my life to have Reyhan, as a partner and colleague.

This book has come to end, (but) the story yet remains ...

CURRICULUM VITAE

Alireza Sadeghi

EDUCATION

Doctor of Philosophy in Software Engineering **2017**
University of California, Irvine *Irvine, California*

Masters of Science in Information Technology **2010**
Sharif University of Technology *Tehran, Iran*

Bachelor of Science in Computer (Software) Engineering **2008**
Sharif University of Technology *Tehran, Iran*

RESEARCH EXPERIENCE

Graduate Research Assistant **2015–2017**
University of California, Irvine *Irvine, California*

Graduate Research Assistant **2012–2015**
George Mason University *Fairfax, Virginia*

Graduate Research Assistant **2008–2010**
Sharif University of Technology *Tehran, Iran*

PROFESSIONAL EXPERIENCE

Software Engineering Intern **Summer 2016**
Google Inc. (Gmail app security team) *Mountain View, California*

Software Engineering Intern **Summer 2015**
Google Inc. (Android security team) *Mountain View, California*

JEE Developer/ Software Development Manager **2005–2012**
Raydana Co. *Tehran, Iran*

REFEREED JOURNAL PUBLICATIONS

Ensuring the Consistency of Adaptation through Inter- and Intra-Component Dependency Analysis **2017**
ACM Transactions on Software Engineering and Methodology (TOSEM)

A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Software **2017**
IEEE Transactions on Software Engineering (TSE)

Software Architectural Principles in Contemporary Mobile Software: from Conception to Practice **2016**
Journal of Systems and Software (JSS)

COVERT: Compositional Analysis of Android Inter-App Permission Leakage **2015**
IEEE Transactions on Software Engineering (TSE)

MBTDD: Model based test driven development **2012**
International Journal of Software Engineering and Knowledge Engineering(IJSEKE)

REFEREED CONFERENCE/WORKSHOP PUBLICATIONS

PATDroid: Permission-aware GUI Testing of Android **Sept 2017**
European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)

Mining Mobile App Markets for Prioritization of Security Assessment Effort **Sept 2017**
International Workshop on App Market Analytics (WAMA)

Architecture Modeling and Analysis of Security in Android Systems **Sept 2016**
European Conference on Software Architecture (ECSA)

Energy-aware test-suite minimization of android apps **July 2016**
International Symposum on Software Testing and Analysis (ISSTA)

Practical, formal synthesis and automatic enforcement of security policies for android **June 2016**
International Conference on Dependable Systems and Networks (DSN)

Reducing Combinatorics in GUI Testing of Android Applications **May 2016**
International Conference on Software Engineering (ICSE)

Analysis of Android Inter-App Security Vulnerabilities Using COVERT **May 2015**
International Conference on Software Engineering (ICSE) - Tool Demonstrations

EcoDroid: An Approach for Energy-Based Ranking of Android Apps **May 2015**
International Workshop on Green and Sustainable Software (GREENS)

Automated detection and mitigation of inter-application security vulnerabilities in Android **Nov 2014**
International Workshop on Software Development Life-cycle for Mobile (DeMobile)

Mining the Categorized Software Repositories to Improve the Analysis of Security Vulnerabilities **Apr 2014**
Fundamental Approaches to Software Engineering (FASE)

ABSTRACT OF THE DISSERTATION

Efficient Permission-Aware Analysis of Android Apps

By

Alireza Sadeghi

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2017

Associate Professor Sam Malek, Chair

Permissions are the cornerstone for Android security model, as they enable secure access to sensitive resources of the phone. Consequently, improper use of Android permission model can lead to *permission-induced* issues that disrupt the functional and non-functional behavior of the apps. However, due to the lack of automated tools for detecting such issues, many of those defects are shipped with the final product, which not only dissatisfies end users but also poses security risks to their phones.

This dissertation proposes and describes a set of automated tools, namely Covert, Separ, Terminator, and PATDroid, to detect and prevent permission-induced issues in Android apps, specifically (I) permission-induced security attacks, and (II) permission-induced compatibility defects.

Through combining static analysis with formal methods, Covert and Separ provide compositional analysis and enforcement techniques, respectively, for detection and prevention of permission-induced security attacks, particularly those that occur due to the interaction of multiple apps. However, by ignoring the temporal aspects of an attack, Covert and Separ, as well as the other techniques aimed at protecting the users against permission-induced attacks, are prone to have low-coverage in detection and high-disruption in prevention of such attacks. Terminator addresses this shortcoming by incorporating the notion of time

in both detection and prevention of the attacks. TERMINATOR leverages temporal logic model checking to detect permission-induced threats, and then relies on Android's dynamic permission mechanism to thwart the identified threats by revoking unsafe permissions. However, such countermeasure, i.e., permission revocation, could itself result in other defects, such as crash, if the target app suffers from dynamic-permission-compatibility issue. To identify such permission-induced compatibility defects, developers need to exhaustively re-execute tests for all possible permission combinations, thereby increasing the time and resources required to test apps. PATDROID, the last proposed approach in this dissertation, is intended to help app developers with this challenge. PATDROID can significantly reduce the testing effort by performing a hybrid program analysis that determines which tests should be executed on what permission combinations.

All conducted experiments corroborate the effectiveness and efficiency of COVERT, SEPAR, TERMINATOR, and PATDROID and their ability to identify and eliminate the defects rooted in permission misuse of Android apps.

Chapter 1

Introduction

Android, with well over a million apps and billion users, has become the dominant mobile platforms. Android app markets, such as 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. However, the rapid growth of Android 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 this platform. In fact, Android is the primary target of mobile malware, where many cases of apps infected with malwares and spywares are regularly reported in the news, security statements and reports [481, 482, 483, 346].

Numerous culprits are in play here, but several studies [188, 143] have shown that *permission-induced* attacks, i.e., security breaches enabled by permission misuse, are among the most critical and frequent issues threatening the security of Android devices. Such issues violate the permission model of Android, which is the main security mechanism provided by the framework to protect applications. This security mechanism, which is a popular form of access-control model, prevents an app lacking the proper permissions from accessing both

sensitive system resources (e.g., sensors) as well as other protected applications.

To help app developers properly implement the permission model, Google provides recommendations and best practices [27], and relies on app developers to properly apply them in their products. However, prior research [143, 160, 202] has shown that many app developers fail to follow such principles in practice. Misuse of Android permissions could disrupt the functional (e.g., crashing) and non-functional (e.g., security breach) behavior of apps. Unfortunately, due to the lack of automated tools for detecting such issues, many of those defects are shipped with the final product, which not only dissatisfies end users but also poses security risks to their phones.

In this context, the goal of this dissertation is to provide a set of automated tools for detecting and preventing permission-induced issues in Android applications, namely (I) permission-induced security attacks, and (II) permission-induced compatibility defects. In the remainder of this Chapter, these two problems along with the proposed solutions are briefly discussed.

1.1 Dissertation Overview

While access to phone resources can be controlled by the Android permission system, enforcing permissions is not sufficient to prevent security violations, as permissions may be mismanaged, intentionally or unintentionally. Android's enforcement of permissions is at the level of individual apps, allowing multiple malicious apps to collude and combine their permissions or to trick vulnerable apps to perform actions on their behalf that are beyond their individual privileges. Despite significant progress in Android security research, prior approaches are substantially intended to detect and mitigate vulnerabilities in a single app, but fail to identify vulnerabilities that arise due to the interaction of *multiple* apps.

To address the aforementioned problem, this dissertation proposes an approach called Covert.

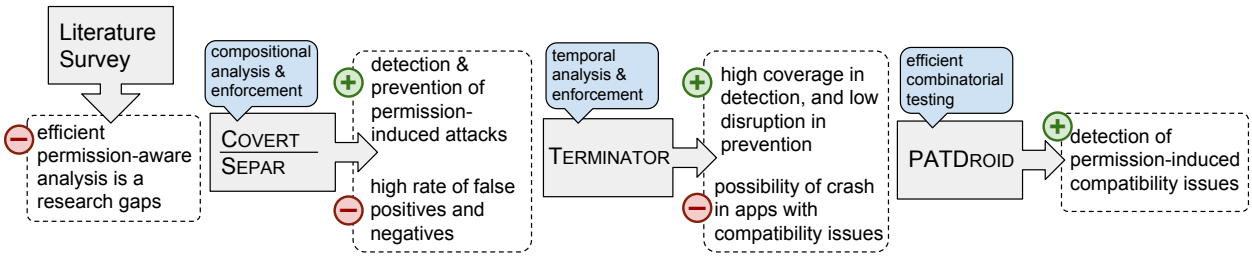


Figure 1.1: Dissertation Roadmap.

COVERT applies combination of static analysis with formal methods: Through static analysis of each app, essential information is extracted in an analyzable formal specification language. The set of extracted models are then checked by a formal analyzer as a whole for vulnerabilities that occur due to the interaction of apps comprising a system. COVERT is intended to *identify* security threats. Hence, as a complementary approach, another tool called SEPAR, is proposed to *thwart* the identified security threats. SEPAR uses a constraint solver to synthesize possible security exploits, from which fine-grained security policies are derived and automatically enforced to protect a given device.

However, due to ignoring the *temporal* aspects of an attack during the analysis and enforcement, COVERT and SEPAR, as well as the other techniques aimed at protecting the users against permission-induced attacks, are prone to have low-coverage in detection and high-disruption in prevention of permission-induced attacks. Moreover, the proposed approaches are mostly realized through modification of either the Android framework or the implementation logic of apps, resulting in all sorts of undesirable side effects, such as app crashes and unexpected behaviors.

The third approach presented in this dissertation, called TERMINATOR, is devised to address the aforementioned shortcomings of COVERT, SEPAR and similar approaches by incorporating the notion of time during the analysis and enforcement. Leveraging temporal logic model checking, TERMINATOR's analyzer identifies permission-induced threats with respect to dynamic permission states of the apps. At runtime, TERMINATOR's enforcer selectively leases

(i.e., temporarily grants) permissions to apps when the system is in a safe state, and revokes the permissions when the system moves to an unsafe state realizing the identified threats.

The countermeasure applied by TERMINATOR, i.e., permission revocation, however, could itself result in other sorts of defects, such as crash, if the target app suffers from dynamic-permission-compatibility issue. To identify such permission-induced compatibility defects, developers should test the app under a wide range of permission combinations, since app's behavior may change depending on the granted permissions. At the state-of-the-art, in the absence of any automated tool support, a developer needs to either manually determine the interaction of tests and app permissions, or exhaustively re-execute tests for all possible permission combinations, thereby increasing the time and resources required to test apps. PATDROID, the last proposed approach in this dissertation, is intended to help app developers with this challenge. PATDROID can significantly reduce the testing effort by performing a hybrid program analysis that determines which tests should be executed on what permission combinations.

The aforementioned problems, which are the subject of this dissertation, are among the research gaps, identified by a comprehensive study conducted through a systematic literature review. To provide an overview of the research, Figure 1.1 shows the roadmap of this dissertation, including the challenges, proposed solutions, and accomplishments of each step.

1.2 Dissertation Structure

The rest of this dissertation is organized as follows. Chapter 2 provides a background on Android, followed by a comprehensive study and literature survey. Chapter 3 describes the problem and specifies the scope of this thesis. Chapter 4 and 5 present COVERT and SEPAR, respectively, which include an analysis and enforcement framework for detection and pre-

Table 1.1: Potential stakeholders for each part of the dissertation

Chapter	Content	Stakeholders
2	Taxonomy and Survey	research community (Software Eng., Security, Mobile)
4	COVERT	app developers, security analysts, researchers
5	SEPAR	app developers, app users, security analysts, researchers
6	TERMINATOR	app developers, app users, security analysts, researchers
7	PATDROID	app developers, app testers, researchers

vention of permission-induced, inter-app security vulnerabilities. Chapter 6 describes TERMINATOR, a temporal permission analysis and enforcement framework for Android. Chapter 7 explains PATDROID, an efficient approach for permission-aware testing that identifies permission-induced compatibility defects. Finally, Chapter 8 concludes this dissertation with the discussion of the contributions and the future work.

To help different readers of this dissertation find their parts of interest more easily, Table 1.1 suggests the potential stakeholders for each part of the dissertation.

The research presented in this dissertation has been published in the following journals and venues:

- A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering (TSE)*, 43(6):492–530, 2017

- A. Sadeghi, H. Bagheri, and S. Malek. Analysis of android inter-app security vulnerabilities using COVERT. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, pages 725–728, 2015

- A. Sadeghi, R. Jabbarvand, and S. Malek. Patdroid: permission-aware GUI testing of android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 220–232, 2017

- A. Sadeghi, R. Jabbarvand, N. Ghorbani, H. Bagheri, and S. Malek. A temporal permission analysis and enforcement framework for android. 2017

- H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek. COVERT: compositional analysis of android inter-app permission leakage. *IEEE Trans. Software Eng. (TSE)*, 41(9), 2015

- H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek. Practical, formal synthesis and automatic enforcement of security policies for android. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*, pages 514–525, 2016

In addition, the following publications are not included in the dissertation but they are along the same line of this research. For instance, the first paper [420] suggests an efficient and safe adaptation technique that could be leveraged by the prevention tools introduced in this dissertation, such as Separ to replace the vulnerable component of an app without any disruption, while the target app is running. Also, the approaches proposed by the second and third papers [418, 421], are able to accelerate the *Static Analysis* components used in Covert, Separ, and Terminator.

- A. Sadeghi, N. Esfahani, and S. Malek. Ensuring the consistency of adaptation through inter- and intra-component dependency analysis. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 26(1):2:1–2:27, 2017

- A. Sadeghi, N. Esfahani, and S. Malek. Mining the categorized software repositories to improve the analysis of security vulnerabilities. In *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 155–169, 2014

- A. Sadeghi, N. Esfahani, and S. Malek. Mining mobile app markets for prioritization of security assessment effort. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on App Market Analytics, WAMA@ESEC/SIGSOFT FSE 2017, Paderborn, Germany, September 5, 2017*, pages 1–7, 2017

Chapter 2

Background and Related Work

Over the past few year, since the inception of Android in 2008, its security has been a thriving subject of research. 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 chapter 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 this dissertation and also the future developments.

We have carefully followed the systematic literature review process, and analyzed the results of 336 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 chapter is organized as follows: Section 2.1 overviews the Android framework to help the reader follow the discussions that ensue. Section 2.2 lists the existing surveys that are directly or indirectly related to the Android security analysis. Section 2.3 presents the research method and the underlying protocol for the systematic literature review. Section 2.4 presents a comprehensive taxonomy for the Android security analysis derived from the existing research literature. Section 2.5 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 2.6 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.

2.1 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 Android application package (APK). Once an APK is installed on an Android device, it runs by using the Android runtime (ART) environment.[1]

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

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

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 run-time 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 [143]. 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 [160]. Moreover, no mechanism exists for preventing an ICC callee from misrepresenting the intentions of its caller to a third party [167].

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. Until recently, no dynamic mechanism was provided by Android for managing permissions after app installation. In the latest release of Android[2], however, Google introduced dynamic permission management that allows users to revoke or grant app permissions at runtime.

Besides required permissions, the app manifest may also include enforced permissions that

[2]Android 6 or Marshmallow

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.

The current permission model of Android suffers from shortcomings widely discussed in the literature [462, 189, 176]. Some examples of such defects include coarse-grained permissions that violate the principle of least privilege [117, 466, 268], enforcing access control policies at the level of individual apps that causes delegation attacks [160, 202, 114, 128], and the lack of permission awareness that leads to uninformed decisions by end users [287, 201, 593, 507].

2.2 Related Surveys

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. [156] 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 2.4. More recently, Felt et al. [200] 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's App Store and Google's Android Market (now called Google Play), for preventing and identifying such malware. Along the same lines, Suarez-Tangil et al. [475] presented a comprehensive survey on the evolution of malware for smart devices and provided an analysis of 20 research efforts that detect and analyze mobile malware. Amamra et al. [57] surveyed malware detection techniques for smartphones and classified them as signature-based or anomaly-based. Haris et al. [246] surveyed the mobile computing research addressing the privacy issues, including 13 privacy leak detection tools and 16 user studies in mobile privacy. Enck [180] reviewed

some of the efforts in smartphone research, including OS protection mechanisms and security analysis techniques. He also discussed the limitations as well as directions for future research.

While the focus of these surveys is mainly on malware for diverse mobile platforms, the area of Android security analysis has not been investigated in detail.

They do not analyze the techniques for Android vulnerability detection. Moreover, they categorize malware detection techniques based only on limited comparison criteria, and several rather important aspects—such as approach positioning, characteristics, and assessment—are ignored. These comparison areas are fully discussed in our proposed taxonomy (see Section 2.4).

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. [143] studied security challenges in Android inter-application communication, and presented several classes of potential attacks on applications. Another example is the survey of Shabtai et al. [450, 451], 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. [586] 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 (e.g., inter-app vulnerabilities [143] or families of Android malware [191, 586]), or certain types of approaches (e.g. techniques relying on dynamic analysis [364], static analysis [438], or machine learning [196] as well as mechanisms targeting the enhancement of the Android security platform [476, 393]). However, none of them provide a comprehensive overview of the existing research in the area of Android secu-

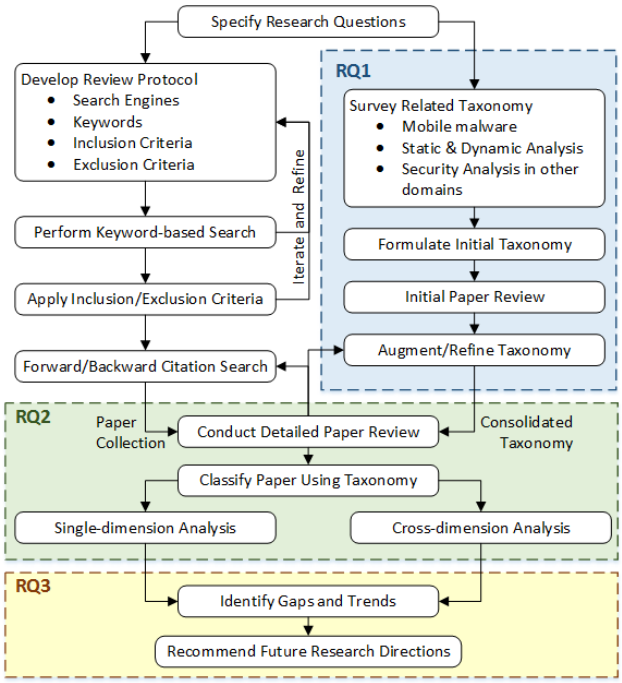


Figure 2.1: Research process flow and tasks.

rity, including but not limited to empirical and case studies, as well as proposed approaches and techniques to identify, analyze, characterize, and mitigate the various security issues in either the Android framework or apps built on top it. Moreover, since a systematic literature review (SLR) is not leveraged, there are always some important approaches missing in the existing surveys. Having compared over 330 related research publications through the proposed taxonomy, this survey, to the best of our knowledge, is the most comprehensive study in this line of research.

2.3 Research Method

Our survey follows the general guidelines for systematic literature review (SLR) process proposed by Kitchenham [293]. We have also taken into account the lessons from Brereton et al. [112] 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?

The remainder of this section describes the details of our review process, including the methodology and tasks that we used to answer the research questions (Section 2.3.1), the detailed SLR protocol including keywords, sources, and selection criteria (Section 2.3.2), statistics on selected papers based on the protocol (Section 2.3.3), and finally a short discussion on the threats to validity of our research approach (Section 2.3.4).

2.3.1 Research Tasks

To answer the three research questions introduced above, we organized our tasks into a process flow tailored to our specific objectives, yet still adhering to the three-phase SLR process including: planning the review, conducting the review, and reporting the review. The overall process flow is outlined in Figure 2.1 and briefly described here.

First, in the planning phase, we defined the review protocol that includes selection of the search engines, the initial selection of the keywords pertaining to Android security analysis, and the selection criteria for the candidate papers. The protocol is described in detail in Section 2.3.2.

The initial keyword-based selection of the papers is an iterative process that involves exporting the candidate papers to a "research catalog" and applying the pre-defined inclusion/exclusion criteria on them. In the process, the keyword search expressions and the inclusion/exclusion criteria themselves may also need to be fine-tuned, which would in turn trigger new searches. Once the review protocol and the resulting paper collection were stabilized, our research team also conducted peer-reviews to validate the selections.

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 abstract, introduction, contribution, and conclusion sections) to identify new concepts and approaches to augment and refine our taxonomy. The resulting taxonomy is presented in Section 2.4.

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 Section 2.5.[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 2.6.

[3]The research artifacts, including the survey catalog, are available to the public and can be accessed at `http://www.ics.uci.edu/~seal/projects/droid-sec-taxonomy`

Table 2.1: Refined search keywords.

Research Domain (D)	Keywords (K)
Program Analysis	Static (Analysis)*, Dynamic (Analysis)*, Control Flow, Data Flow, Taint, Monitoring, Feature Selection
Security Assessment	Security, Vulnerability/Vulnerable, Malware/Malicious, Virus, Privacy
Android Platform	Android, Mobile, Smartphone, App

2.3.2 Literature Review Protocol

This section provides the details of the review protocol, including our search strategy and inclusion/exclusion criteria.

2.3.2.1 Search Method

We used reputable literature search engines and databases in our review protocol with the goal of finding high-quality refereed research papers, including journal articles, conference papers, tool demo papers, as well as short papers from respectable venues. The selected search engines consist of IEEE Explore, ACM Digital Library, Springer Link, and ScienceDirect.

Given the scope of our literature review, we focused on selected keywords to perform the search on the papers' titles, abstracts, and meta-data, such as keywords and tags. Our search query is formed as a conjunction of three research domains, described in Section 2.3.2.2 as inclusion criteria, namely, D_1: *Program Analysis*, D_2:*Security Assessment*, and D_3: *Android Platform*. These research domains appear in the literature under different forms and using synonymous words. To retrieve all related papers, each research domain in our search string is represented as a disjunction of corresponding keywords summarized in Table 2.1. These keywords were continuously refined and extended during the search process. For instance, regarding the security assessment domain, we considered keywords such as, "security", "vulnerability", "malware", "privacy", etc. In summary, our search query is defined as the

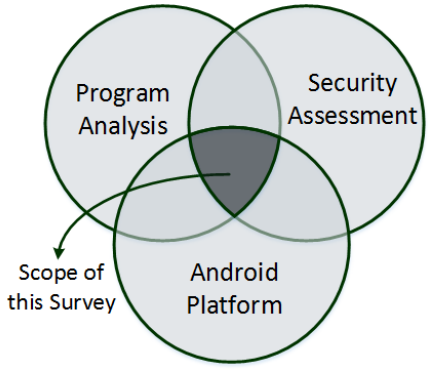


Figure 2.2: Scope of this survey.

following formula:

$$query = \bigwedge_{d \in \{D_1, D_2, D_3\}} (\bigvee_{keyword \in K_d} keyword)$$

Where D_is are the three research domains, and K_d is the set of corresponding keywords specified for domain d in Table 2.1.

Finally, to eliminate irrelevant publications and also make our search process repeatable, we added a time filter to limit the scope of the search for the papers published from 2008[4] to 2016[5].

2.3.2.2 Selection Criteria

Not all the retrieved papers based on the search query fit within the scope of this survey. Therefore, we used the following inclusion and exclusion criteria to further filter the candidate papers.

Inclusion Criteria. As illustrated in Figure 2.2, the scope of surveyed research in this study falls at the intersection of three domains:

[4]The release year of the first version of Android framework.
[5]The papers published after January 2016 are not included in this survey.

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 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. Moreover, we excluded papers that:

1. exclusively developed for platforms other than Android, such as iOS, Windows Mobile, BlackBerry, and Sybmbian (e.g., [175, 52, 110, 441, 324, 141, 323, 517, 113]). 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, but not on any security analysis technique. Such techniques attempt to enhance security mechanisms either at the application-level or the level of the Android platform by means of different approaches, such as isolation and sandboxing [116, 146, 521, 74, 588, 301, 477, 195, 315, 538], enhancing permission management [199, 288, 244, 409, 286, 217], anonymity [458, 296], fine-grained or dynamic policy enforcement [466, 504, 410, 209], anti-repackaging [581, 582, 404, 387, 266], security-enhanced communication [72, 580], database and storage [356, 171, 442], cryptography [90, 163], etc. Approaches that consider both detection and protection (e.g., [379, 341, 114, 478]), are included in the survey.

3. performed the analysis *only* on apps meta-data, such as description [376, 557], category [419], signature [87], ranking and reviews [531, 532], resources [273], apk file's

meta-data [55], or a combination of these attributes [365, 487, 321]. The analyses running on an app's code, but at opcode level [193, 431, 225, 123, 278] are also excluded.

4. focused only on expanding and enhancing Java program analysis techniques, either static [105, 129, 370, 543, 306, 104] or dynamic [66, 242], for the Android framework. In this survey, however, we included general program-analysis research that, at least, provide a case study or experiment related to security analysis (e.g, [67, 307, 429]).

5. focused solely on low-level monitoring and profiling techniques for identifying security-related anomalies or malware. Such research includes intrusion detection, which performs analysis using hardware signals (e.g., CPU utilization [561, 564], power consumption [249, 170], memory usage [58, 289], network traffic [295, 472, 454, 469, 503, 153, 444, 108, 358, 62], or a combination of multiple sensors [452, 537, 235, 342, 103]). These approaches use mechanisms at a lower level than the Android framework, making them out of scope for this survey.

6. elaborated on a particular attack on the Android framework [563, 330, 237, 128, 527] or apps [139, 354, 385, 298, 339, 125, 388, 258, 333], without describing detection techniques to identify the vulnerabilities that lead to the described security breach.

In addition, 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. *Dexter* [9] and *DroidBox* [12] are two examples that respectively leverage static and dynamic analysis techniques, but lack any peer-reviewed paper, thus were excluded from this survey.

2.3.3 Selected papers

Table 2.2 provides statistics on each phase of paper collection, illustrated in Figure 2.1, for each database.

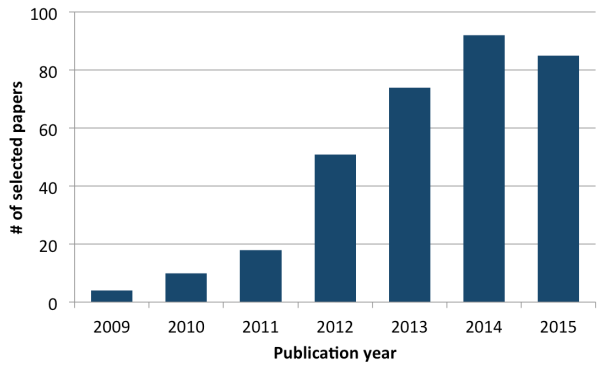
Table 2.2: Number of collected papers at each phase of paper selection.

Selection phase	Database				All
	IEEE	ACM	Springer	ScienceDirect	
Keyword-based search	1,374	938	8,605	2,830	—
Initial filtering	852	721	520	240	—
Merging					2023
Applying criteria					336

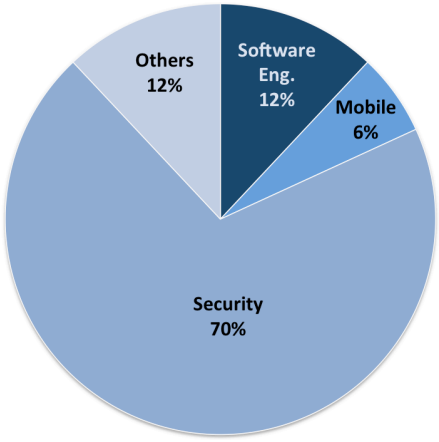
Figure 2.3: Word cloud of the titles of the selected papers.

The first row shows the size of the initial set of papers, selected by keyword-based search over the full paper for each database. Since the search engine of the four databases treat our search query differently, we performed another verification over the initially collected papers, in a consistent manner, based on the same keywords (Row 2). After initial filtering, we merged the search results of all databases into a single repository for further review and filtering (Row 3).

In the fourth row, the number of filtered papers after applying selection criteria is shown. In this stage we applied inclusion and exclusion criteria, enumerated in Section 2.3.2.2, on the



(a) by publication year.



(b) by publication venue.

Figure 2.4: Distribution of surveyed papers.

title, abstract and conclusion of the papers selected in the first phase to remove out-of-scope publications. This process led to the selection of 336 papers for this survey—whose titles are illustrated in the form of a word cloud in Figure 2.3.

Figure 2.4a shows the number of selected papers by publication year. As illustrated in this figure, the number of publications have increased gradually between 2009 and 2011, more than doubled between 2011 and 2012, and hit its peak in 2014.

As shown in Figure 2.2, this study covers multidisciplinary research conducted in various domains, such as software engineering (including programming languages), security, and mobility. Consequently, as depicted in Figure 2.4b, selected papers are also published in different venues related to such domains.

2.3.4 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.

2.4 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 2.2, 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 sub-areas, 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 described in Section 2.2. 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?
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 21 dimensions and sub-dimensions, which are depicted in Figures 2.5–2.7, and explained in the following.

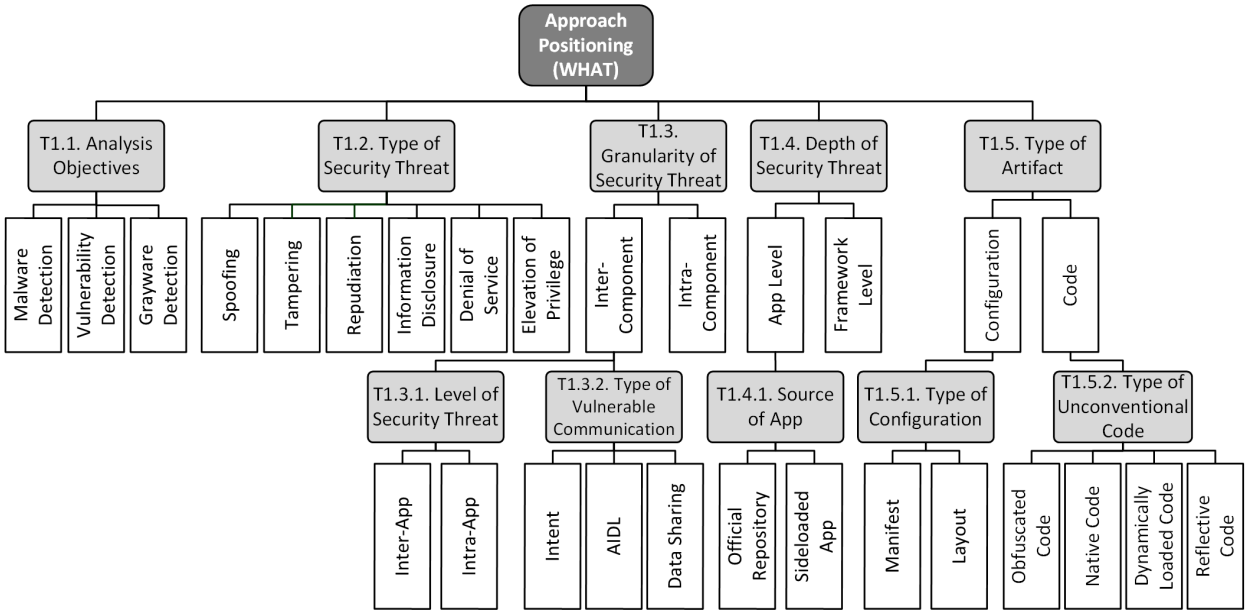


Figure 2.5: Proposed Taxonomy of Android Security Analysis, Problem Category.

2.4.1 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 2.5.

2.4.1.1 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.

Since malware authors exploit the existing vulnerabilities of other apps or the underlying

Android framework to breach system security, malware detection techniques and vulnerability identification methods are complementary to each other. In addition to these two kinds of approaches, there exists a third category of techniques intended to detect and mitigate the risk o *grayware*. Grayware, such as advertisement apps and libraries, are not fully malicious but they could violate users' privacy by collecting sensitive information for dubious purposes [200, 464, 475].

2.4.1.2 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* [480].

Among existing attack models, we selected STRIDE, as it provides a design-centric model that helps us investigate the security properties of Android system, irrespective of known security attacks, thus allowing us to identify gaps in the literature (e.g., security attacks that have not been observed in Android yet, security attacks that have not received much attention in the literature). Moreover, it recognizes a separate category for each type of security property that is widely referred to in the literature.

Spoofing violates the *authentication* security property, where an adversary pretends to be a legitimate entity by properly altering some features that allows it to be recognized as a legitimate entity by the 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)[143].

App Cloning, *Repackaging* or *Piggybacking* are classified under Spoofing, where malware authors attach malicious code to legitimate apps and advertise them as original apps in app markets to infect users. This technique is quite popular among mobile malware developers; it is used by 86% of the Android malware, according to a recent study [586].

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 [587].

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 shown to be readily discoverable by an adversary through reverse engineering of the apps [182], which in turn enables launching a denial of service attack. Unauthorized Intent receipt [143], duplicating content provider authorities and permission names [279], battery exhaustion [343], and ransomware [60, 544], 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 [233], 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) [160].

Over-privileged apps are particularly vulnerable to privilege escalation attack, due to the possibility of an attacker successfully injecting malicious code, exploiting the unnecessary permissions [94, 198]. Therefore, we categorize this type of security threat under elevation of privilege.

2.4.1.3 Granularity 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, more complicated scenarios are possible, where a security issue may arise from the interaction of multiple components. Accordingly, the existing techniques are classified into two categories: intra-component approaches that only consider security issues in a single component, and inter-component approaches that are able to identify security issues in multiple components. We further classify the inter-component class into subclasses based on two sub-dimensions described below.

Level of Security Threat (T1.3.1) It is possible that interacting vulnerable or malicious 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 [160, 115]. Accordingly, security assessment techniques that consider the combination of apps in their analysis (i.e.,inter-app) are able to reveal more complicated issues compared to non-compositional approaches (i.e.,intra-app).

Type of Vulnerable Communication (T1.3.2) 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 mis-

implemented, 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 2.1, 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) [3].
- *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) [587].

2.4.1.4 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.

2.4.1.5 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 extracted models that could lead to identification of a security vulnerability or malicious behavior.

Type of Configuration (T1.5.1) Among different configuration files contributing to the structure of Android app packages (APKs), a few artifacts encode significant security in-

formation, most notably, the *manifest* file that contains high-level information such as app components and permissions, as well as the *layout* file that defines the structure of app's user interfaces.

Type of Unconventional Code (T1.5.2) For different reasons, from legitimate to adversarial, developers may incorporate special types of code in their apps. A security assessment technique needs to tackle several challenges for analyzing such unconventional kinds of code. Thus, we further distinguish the approaches based on the special types of code they support, which includes the following:

- *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 [399, 400, 575, 388, 210, 361].

- *Native Code:* Beside Java code, Android apps may also consist of native C or C++ code, which is 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.

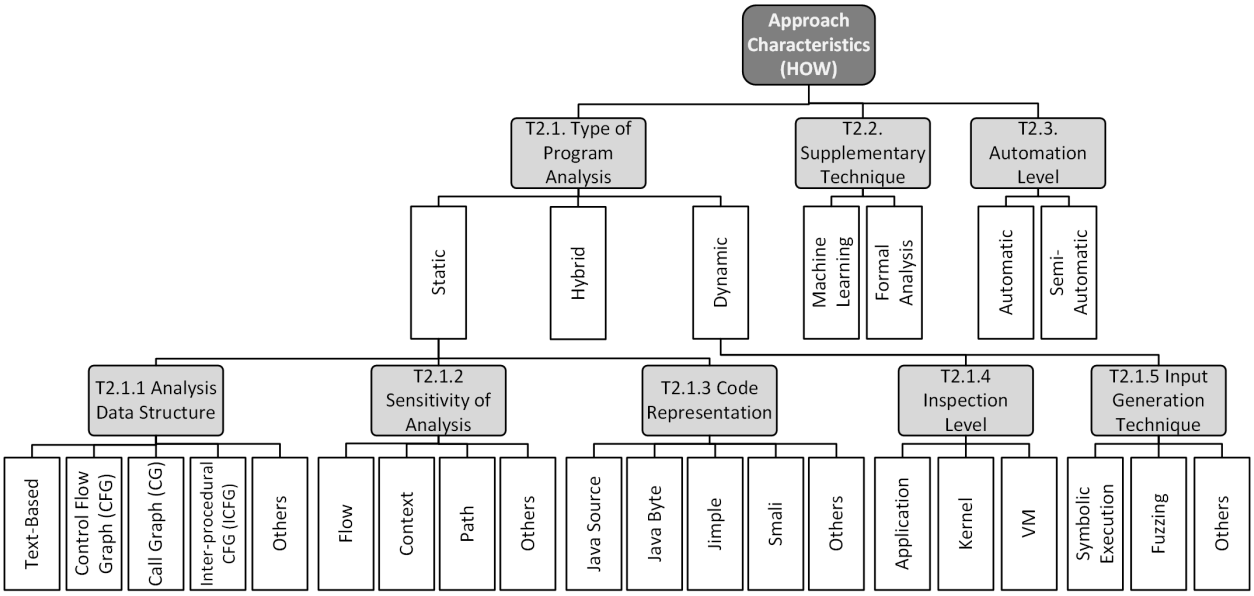


Figure 2.6: Proposed Taxonomy of Android Security Analysis, Solution Category.

- *Reflective Code:* 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 [325].

2.4.2 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 2.6.

2.4.2.1 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.

Each approach has its own strengths and weaknesses. While static analysis is considered to be conservative and sound, dynamic analysis is unsound yet precise [184]. 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 [434, 383, 390, 500, 397, 65, 277, 192].

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.

To better distinguish different approaches with respect to the program analysis techniques they rely on, we suggest sub-dimensions that further classify those two categories (i.e., static and dynamic analyses). Five sub-dimensions are presented below, where the first three (i.e., T2.1.1, T2.1.2, and T2.1.3) classify static analysis techniques and the next two (i.e., T2.1.4, and T2.1.5) are applied to dynamic analyses.

Analysis Data Structures (T2.1.1) In addition to lightweight static analyses that only employ *text-mining* techniques, heavyweight but more accurate static approaches usually leverage a few well-known data structures to abstract the underlying programs. 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.

Sensitivity of Analysis (T2.1.2) The sensitivities of the analyses vary for different algorithms used by a static analysis technique, leading to tradeoffs among analysis precision and scalability. Thus, this dimension classifies the static approaches based on their sensitivity to the following properties.

- *Flow Sensitive* techniques consider the order of statements and compute separate information for each statement.
- *Context Sensitive* approaches keep track of the calling context of a method call and compute separate information for different calls of the same procedure.
- *Path Sensitive* analyses take the execution path into account, and distinguish information obtained from different paths.

There also exist other levels of sensitivity, such as field- and object-sensitivity, which are discussed less often in the surveyed literature.

Code Representation (T2.1.3) Static analysis algorithms and methods are often implemented on top of off-the-shelf frameworks that perform the analysis on their own intermediate

representation (IR) of program code. This dimension classifies the analysis tools based on the used IR (if any), which is translated from apps Dalvik bytecode prior to the analysis.

- *Java Source Code* may be analyzed since Android apps are mostly written in the Java language. This assumption, however, limits the applicability of the analysis to either open-source apps or the developers of an app.
- *Java Bytecode* may be analyzed, which widely broadens the applicability of an approach compared to the first group. Distinct from Java, Android has its own Dalvik bytecode format called Dex, which is executable by the Android virtual machine. As a result, this class of tools needs to retarget Dalvik to Java bytecode prior to the analysis, using APK-to-JAR transformers, such as dex2jar [8], ded [368], and its successor Dare[369].
- *Jimple* is a simplified version of Java bytecode that has a maximum of three components per statement. It is used by the popular static analysis framework Soot [495]. Dexpler[93] is a plugin for the Soot framework that translates Dalvik bytecode to Jimple.
- *Smali* is another intermediate representation, which is used by the popular Android reverse engineering tool, Apktool [5].

Inspection Level (T2.1.4) To capture dynamic behavior of Android apps, analysis techniques monitor the running apps at different levels. This dimension categorizes dynamic analyses based on their inspection level, including:

- *App-level* monitoring approaches trace Java method invocation by weaving the bytecode and injecting log statements inside the original app code or the Android framework. A few approaches achieve this in a more fine-grained manner through instruction-level dynamic analysis, such as data-flow tracking.

- *Kernel-level* monitoring techniques collect system calls, using kernel modules and features such as `strace`, or `ltrace`.

- *Virtual Machine (VM)-level* tools intercept events that occur within emulators. This group of approaches can support several versions of Android. The more recent work in this area supports the interception of Dalvik VM's successor, *Android Runtime* (ART) [149]. However, they are all prone to emulator evasion [364, 383, 357].

Input Generation Technique (T2.1.5) 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 neither systematic nor scalable, two approaches are often leveraged by the surveyed research: fuzzing and symbolic execution.

- *Fuzz testing or fuzzing* [224] executes the app with random input data. Running apps using inputs generated by Monkey[4], the state-of-the-practice tool for the Android system testing, is an example of fuzz testing.

- *Symbolic execution* [292] 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.

2.4.2.2 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.

2.4.2.3 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.

2.4.3 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 2.7, 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, user studies, or other methods. Moreover, we further classify the empirical evaluations according to the source of apps they selected for the experiments, including the official Google Play repository, third-party and local repositories, collections of

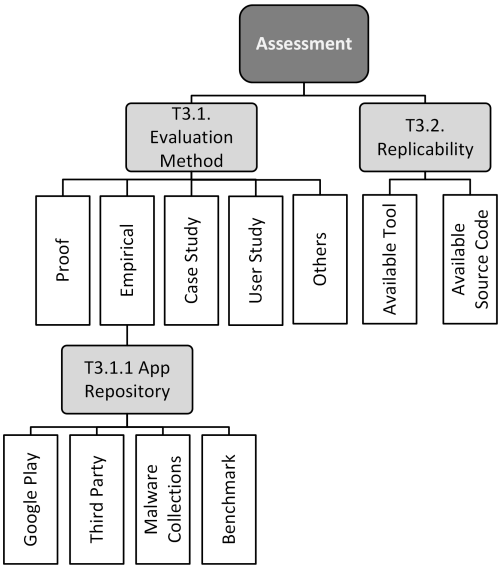


Figure 2.7: Proposed Taxonomy of Android Security Analysis, Assessment Category.

malware, and benchmark apps handcrafted by research groups for the purpose of evaluation.

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.

2.5 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]

[6]Throughout this survey (including tables and figures), the approaches without name are shown in the form of "first author's surname_".

2.5.1 Approach Positioning (Problem)

Tables 2.3 and 2.4 provide 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, several dimensions only apply to a subset of papers surveyed, e.g., *Test Input Generation* only applies to 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.

2.5.1.1 Analysis Objective

Security assessment techniques proposed by a number of previous studies could be directly used or extended for various purposes (e.g., detection of malware, grayware, or vulnerabilities). In this survey, to distinguish the main objective(s) of each approach, we consulted the threat model (or adversary model) and also the evaluation goals and results (if any) described in the surveyed papers.

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 61% of the overall set of papers collected for this literature review. However, sometimes the analysis techniques are not able to determine unequivocally if an application is malicious or benign. Therefore, a number of studied approaches [275, 276, 585, 257, 433, 231] use risk-based analysis to assign each app a level of security risk according to the analysis results

Table 2.3: Problem Specific Categorization of the Reviewed Research, Part 1

Dimension		Approaches	%
Analysis Objective	Vulnerability Detection	ADDICTED [585], Amandroid [514], ApkCombiner [307], App-ray [490], AppAudit [526], AppCaulk [446], AppCracker [121], AppFence [252], AppGuard [75], AppProfiler [411], AppSealer [559], Aquifer [359], ASM [247], AuthDroid [506], Bagheri_ [78], Bartel_ [94], Bartsch_ [95], Bifocals [144], Buhov_ [119], Buzzer [127], CMA [455], CoChecker [155], ComDroid [143], ConDroid [445], ContentScope [587], Cooley_ [148], COPES [92], COVERT [80], COVERT_Tool [417], CredMiner [591], CRePE [147], CryptoLint [174], Desnos_ [165], DexDiff [350], DroidAlarm [579], DroidChecker [133], DroidCIA [140], DroidGuard [81], DroidRay [577], Droidsearch [395], Enck_ [182], Epicc [371], FineDroid [565], Flowdroid [67], Gallo_ [214], Geneiatakis_ [219], Grab'nRun [187], Harehunter [46], HornDroid [122], IccTA [308], IPCInspection [202], IVDroid [190], Juxtapp [241], Kantola_ [283], KLD [453], Lintent [118], Lu_ [329], MalloDroid [186], Matsumoto_ [345], Mutchler_ [355], NoFrak [220], NoInjection [274], Onwuzurike_ [374], PaddyFrog [523], PatchDroid [353], PCLeaks [309], PermCheckTool [501], PermissionFlow [436], Poeplau_ [384], PScout [69], QUIRE [167], Ren_ [405], SADroid [240], SCanDroid [211], Scoria [497], SecUP [533], SEFA [524], Smith_ [467], SMV-HUNTER [470], STAMBA [109], Stowaway [198], SUPOR [259], TongxinLi_ [314], Vecchiato_ [498], VetDroid [566], WeChecker [154], Woodpecker [233], Zuo_ [594]	26%
	Malware Detection	Others *	61%
	Grayware Detection	Achara_ [49], AdDroid [379], AdRisk [232], AndroidLeaks [222], APKLancet [545], AppFence [252], AppProfiler [411], AppsPlayground [398], Han_ [239], LayerCake [408], Leontiadis_ [305], Pedal [322], Seneviratne_ [447], Short_ [465], Wijesekera_ [518]	4%
Type of Security Threat	Spoofing (Cl|Cr)	Amandroid [514], AnDarwin [152]^{Cl}, AppCracker [121]^{Cr}, AppIntegrity [499]^{Cl}, AuthDroid [506]^{Cr}, Bianchi_ [102], Bifocals [144], Buhov_ [119]^{Cr}, Chen_ [136]^{Cl}, ComDroid [143], Compac [512], Cooley_ [148], CredMiner [591]^{Cr}, CryptoLint [174]^{Cl}, DNADroid [151]^{Cl}, DroidCIA [140], DroidKin [226]^{Cl}, DroidMOSS [584]^{Cl}, DroidSim [479]^{cl}, Epicc [371], Gallingani_ [213], HunterDroid [556]^{Cl}, Juxtapp [241]^{Cl}, Kantola_ [283], MalloDroid [186]^{Cr}, MIGDroid [254]^{Cl}, Mutchler_ [355], MysteryChecker [269]^{Cl}, NoFrak [220], NoInjection [274], Onwuzurike_ [374]^{Cr}, PCLeaks [309], PermissionFlow [436], PICARD [169]^{Cl}, Ren_ [405], ResDroid [456]^{Cl}, SCSdroid [318]^{Cl}, SMV-HUNTER [470]^{Cr}, STAMBA [109], WuKong [505]^{Cl}, Zhou_ [583]^{Cl}, Zuo_ [594]^{Cr}	13%
	Tampering	AppCracker [121], AppIntegrity [499], APSET [427], CMA [455], ContentScope [587], Desnos_ [165], DroidCIA [140], DroidFuzzer [549], Harehunter [46], MalloDroid [186], SMV-HUNTER [470], STAMBA [109]	4%
	Repudiation		0%
	Information Disclosure	Achara_ [49], AdDroid [379], AdRisk [232], AdSplit [459], Amandroid [514], AMDetector [569], Ananas [173], AndroidLeaks [222], ApkCombiner [307], APKLancet [545], AppAudit [526], AppCaulk [446], AppFence [252], AppGuard [75], AppInspector [223], AppIntent [548], Apposcopy [203], AppProfiler [411], AppSealer [559], AppsPlayground [398], AsDroid [260], AuDroid [382], Aurasium [535], AutoCog [391], Bagheri_ [78], Bal_ [85], Barbon_ [86], Barros_ [91], Bartsch_ [95], Batyuk_ [96], BayesDroid [493], Berthome_ [99], BlueSeal [251], Brahmastra [101], Brox [335], Capper [560], Cassandra [326], CHEX [328], CMA [455], CoChecker [155], ComDroid [143], ContentScope [587], ConUCON [84], CopperDroid [403], CopperDroid2 [485], COVERT [80], COVERT_Tool [417], DataChest [589], Defensor [375], DexDiff [350], DroidGuard [81], DroidPAD [331], DroidSafe [227], DroidTest [412], DroidTrack [425], Enck_ [182], Epicc [371], Feth_ [205], FineDroid [565], FlaskDroid [117], Flowdroid [67], Graa_ [230], Graa2_ [229], Han_ [239], Harehunter [46], HornDroid [122], ICC_Map [178], IccTA [308], IFT [185], IIF [551], Jia_ [271], KLD [453], Kynoid [443], LazyTainter [516], LeakMiner [547], Lee_ [302], Leontiadis_ [305], Mann_ [341], Matsumoto_ [345], Mobile-Sandbox [471], MobSafe [534], MockDroid [98], MonkeyDroid [334], Morbs [509], MorphDroid [204], MOSES [573], Mudflow [70], Mutchler_ [355], NDroid [389], Nishimoto_ [367], Onwuzurike_ [374], Paupore_ [378], PCLeaks [309], Pegasus [138], Porscha [372], Relda [234], Ren_ [405], SADroid [240], ScanDal [291], SecUP [533], SEFA [524], Seneviratne_ [447], SFG [59], Short_ [465], SmartDroid [574], Smith_ [467], Song_ [468], STAMBA [109], SUPOR [259], TaintDroid [181], TISSA [592], TouchDevelop [528], TrustDroid [570], Uranine [401], VetDroid [566], WeChecker [154], WifiLeaks [48], Wijesekera_ [518], Yaase [413]	35%
	Denial of Service	ASV [257], ComDroid [143], Enck_ [182], Ren_ [405], SecUP [533]	1%
	Elevation of Privilege (O)	ADDICTED [585], AdDroid [379], Ananas [173], AppGuard [75], AppInspector [223], AppsPlayground [398], Aurasium [535], Bagheri_ [78], Bartel_ [94]^{O}, Bartsch_ [95], Bugiel_ [115], CHEX [328], CoChecker [155], ComDroid [143], Compac [512], COPES [92]^{O}, COVERT [80], COVERT_Tool [417], DeepDroid [511], Defensor [375], DroidAlarm [579], DroidBarrier [56], DroidChecker [133], DroidGuard [81], DroidRay [577], DroidTrace [578], Enck_ [182], Epicc [371], FineDroid [565], FlaskDroid [117], FUSE [402], Gallo_ [214], Geneiatakis_ [219], Harehunter [46], IntentFuzzer [542], IPCInspection [202], Johnson_ [280]^{O}, Lee_ [302], Lintent [118], Lu_ [329], MobSafe [534], PaddyFrog [523], PCLeaks [309], Pedal [322], Pegasus [138], PermCheckTool [501]^{O}, PermissionFlow [436], PREC [248], PScout [69]^{O}, QUIRE [167], SADroid [240], SecUP [533], SEFA [524]^{O}, Stowaway [198]^{O}, UID [179], WeChecker [154], Woodpecker [233], Yaase [413]	17%

R: Risk-based, **Cl**: Cloning, Repackaging, or Piggybacking, **Cr**: Cryptography Misuse, **O**: Over-privilege apps
*: Including all other surveyed papers that are not mentioned as Vulnerability or Grayware detection

(Denoted by R in Table 2.3).

4% of efforts in this area are devoted to the analysis of grayware that are less disruptive than malware, but still worrying, particularly from a privacy perspective. Most research efforts on grayware detection target the analysis of advertisement (ad) libraries that are linked and shipped together with the host apps. 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 [379]. For this reason, *AdRisk* [232] decouples the embedded ad libraries from the host apps and examines the potential unsafe behavior of each library that could result in privacy issues. Other techniques, such as *AdDroid* [379], *AFrame* [562], *AdSplit* [459], and *LayerCake* [408], introduce advertising frameworks with dedicated permissions and APIs that separate privileged advertising functionality from host applications. Also, as a more generic solution, Compac [512] provides fine-grained access control to minimize the privilege of all third-party components.

Android vulnerability analysis has also received attention from a significant portion of existing research efforts (26% 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* [233] 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.

2.5.1.2 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 2.3 that among the *STRIDE* security threats (See Section 2.4.1.2), information disclosure is the most considered threat in Android, comprising 35% of the papers. This is not a surprising result, since mobile devices are particularly vulnerable to data leakage [261]. Elevation of privilege (including over-privilege issue marked as *O* in Table 2.3) is the second class of threats addressed by 17% of the overall studied papers. Examples of this class of threats, such as *confused deputy vulnerability* [245], are shown to be quite common in the Android apps on the market [160, 198, 202].

Spoofing has received substantial attention (13%), 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* [143]. Cloning or repackaging, which is a kind of spoofing threat, is a common security issue in Android app markets, and hence is addressed by several techniques, including [152, 499, 151, 584]. Note that these techniques are marked as *Cl* in Table 2.3. Moreover, misusing cryptography techniques, such as failure in the SSL/TLS validation process, might result in *man in the middle* attacks that violate system authentication. Thus, we categorized the techniques attempting to identify cryptography misuse, such as [186, 470], under spoofing. We distinguished these techniques by label *Cr* in Table 2.3.

Tampering and denial of service issues are also considered in the literature, comprising 4% and 1% of the papers, respectively. Among the STRIDE's threats, repudiation is not explicitly studied in the surveyed research. We will revisit this gap in Section 2.6.

Table 2.4: Problem Specific Categorization of the Reviewed Research, Part 2

Dimension			Approaches	%
Granularity of Threat	Intra-Comp.		Others *	79%
	Inter-Comp.	Intent	Amandroid [514], Apex [362], ApkCombiner [307], AppAudit [526], AppCaulk [446], AppContext [546], AppIntent [548], Apposcopy [203], APSET [427], AsDroid [260], Bal_ [85], Barros_ [91], Bartsch_ [95], BlueSeal [251], Brahmastra [101], CoChecker [155], ContentScope [587], ConUCON [84], COVERT [80], COVERT_Tool [417], DataChest [589], DidFail [294], DroidAlarm [579], DroidAPIMiner [45], DroidForce [396], DroidGuard [81], DroidSafe [227], DroidSIFT [558], Epicc [371], Feth_ [205], FineDroid [565], FUSE [402], Gallingani_ [213], Han_ [239], HornDroid [122], ICC_Map [178], IccTA [308], IFT [185], IntentFuzzer [542], IPCInspection [202], IVDroid [190], Jeong_ [270], Kantola_ [283], Lintent [118], Morbs [509], Mutchler_ [355], PaddyFrog [523], PCLeaks [309], PermissionFlow [436], QUIRE [167], Ren_ [405], SADroid [240], Shen_ [461], SmartDroid [574], UID [179], WeChecker [154], Woodpecker [233], Xmandroid [114], Zhou_ [583]	18%
		AIDL	ASV [257], BlueSeal [251], CopperDroid [403], CopperDroid2 [485], DataChest [589], Morbs [509], QUIRE [167], Woodpecker [233]	2%
		SharedData	ContentScope [587], Harehunter [46], KLD [453]	1%
	Inter-App		Amandroid [514], ApkCombiner [307], Bal_ [85], Barros_ [91], Bartsch_ [95], Brahmastra [101], COVERT [80], COVERT_Tool [417], DataChest [589], DidFail [294], DroidForce [396], DroidGuard [81], DroidTrack [425], FineDroid [565], FUSE [402], Harehunter [46], IccTA [308], IntentFuzzer [542], IPCInspection [202], Jia_ [271], Morbs [509], PCLeaks [309], PermissionFlow [436], QUIRE [167], SEFA [524], WeChecker [154], Xmandroid [114]	8%
Depth	App Level		Others **	88%
	Framework Level (K)		ADDICTED [585], AdDroid [379], AntiMalDroid [568], Apex [362], Bagheri_ [78], Bartel_ [94], Bifocals [144], Bugiel_ [115], Buzzer [127], COPES [92], CRePE [147], DataChest [589], DexDiff [350], Dr.Android [268], DroidBarrier [56], DroidRay [577], DroidSafe [227], Feth_ [205], FineDroid [565], Gallo_ [214]^{K}, IFT [185], Jung_ [281], Kantola_ [283], Kynoid [443], Morbs [509], MpDroid [488], Nishimoto_ [367], NoFrak [220], PatchDroid [353], Porscha [372], PScout [69], SecUP [533]^{K}, Shebaro_ [457], Smith_ [467], Stowaway [198], TongxinLi_ [314], Vecchiato_ [498], VetDroid [566], WifiLeaks [48]	12%
Type of Artifact	Config.	Manifest	A5 [502], AdDroid [379], AdRisk [232], Amandroid [514], Ananas [173], Androguard [166], Andrubis [320], ApkCombiner [307], APKLancet [545], App-ray [490], AppAudit [526], AppContext [546], AppGuard [75], Apposcopy [203], AppProfiler [411], APSET [427], Aquifer [359], AsDroid [260], AuthDroid [506], AutoCog [391], AVDTester [256], Bagheri_ [78], Barrera2_ [88], Batyuk_ [96], Bianchi_ [102], BlueSeal [251], Brahmastra [101], Capper [560], Cen_ [130], CoChecker [155], ComDroid [143], Compac [512], ContentScope [587], COPES [92], COVERT [80], COVERT_Tool [417], Dai_ [157], DiCerbo_ [131], DidFail [294], Dr.Android [268], Drebin [64], DroidAlarm [579], DroidAnalytics [576], DroidAPIMiner [45], DroidChecker [133], DroidForce [396], DroidFuzzer [549], DroidGuard [81], DroidMat [522], DroidPermissionMiner [68], DroidRanger [590], DroidRay [577], DroidSafe [227], Droidsearch [395], Duet [253], Epicc [371], Flowdroid [67], FUSE [402], Gates_ [218], Geneiatakis_ [219], Harehunter [46], Huang_ [255], ICC_Map [178], IccTA [308], IFT [185], IVDroid [190], Johnson_ [280], Kantola_ [283], Kate_ [285], Kim_ [290], Kirin [183], LeakMiner [547], Lee_ [302], Leontiadis_ [305], Lintent [118], Lu_ [329], Ma_ [336], Malek_ [340], MalloDroid [186], Mama [430], Manilyzer [197], Mann_ [341], Marvin [319], MassVet [137], MAST [132], Matsumoto_ [345], Mobile-Sandbox [471], Moonsamy_ [351], Moonsamy2_ [352], Mudflow [70], Mutchler_ [355], PaddyFrog [523], PCLeaks [309], Pegasus [138], Peiravian_ [380], PermCheckTool [501], PermissionFlow [436], Permlyzer [536], ProfileDroid [515], PUMA2 [243], Relda [234], ResDroid [456], Riskranker [231], SAAF [250], SADroid [240], Sahs_ [424], Sanz_ [432], Sarma_ [433], Sayfullina_ [435], SCanDroid [211], SEFA [524], Shen_ [461], SherlockDroid [61], Short_ [465], SmartDroid [574], Smith_ [467], SMV-HUNTER [470], StaDynA [571], TMSVM [529], TraceDroid [496], TrustDroid [570], UID [179], Wang_ [510], WeChecker [154], Woodpecker [233], Yerima_ [550], Zhou_ [583], Zuo_ [594]	38%
		Layout	Amandroid [514], AsDroid [260], Bianchi_ [102], BlueSeal [251], Brahmastra [101], DataChest [589], Flowdroid [67], FUSE [402], IccTA [308], MassVet [137], Permlyzer [536], ResDroid [456], SUPOR [259], UIPicker [360], WeChecker [154]	4%
	Code (P)	Obfuscated	AnDarwin [152]^{P}, Apposcopy [203], CredMiner [591], Dendroid [474]^{P}, Desnos_ [165]^{P}, DNADroid [151]^{P}, DroidKin [226]^{P}, DroidSIFT [558], DroidSim [479], Graa_ [230], Graa2_ [229]^{P}, IREA [297], Juxtapp [241]^{P}, MassVet [137], OpSeq [54], Pedal [322], ResDroid [456], Shen_ [461], ViewDroid [555]	6%
		Native	Compac [512]^{P}, CopperDroid [403], CopperDroid2 [485], Dagger [541], DeepDroid [511], DroidRanger [590]^{P}, DroidScope [539], FireDroid [414], Flowdroid [67]^{P}, MAST [132]^{P}, Mobile-Sandbox [471]^{P}, NDroid [389], PatchDroid [353], Poeplau_ [384]^{P}, RetroSkeleton [161]^{P}, Riskranker [231]^{P}, VetDroid [566]	5%
		Dynamic	AdRisk [232]^{P}, AppContext [546]^{P}, AppsPlayground [398], ConDroid [445], DroidAPIMiner [45], DroidRanger [590]^{P}, DroidTrace [578], Grab'nRun [187], Poeplau_ [384]^{P}, Riskranker [231]^{P}, StaDynA [571], Yerima_ [550]	4%
		Reflective	AdRisk [232]^{P}, AppAudit [526]^{E}, AppContext [546]^{P}, AppGuard [75], AppsPlayground [398], Barros_ [91], DroidSafe [227]^{P}, DroidSIFT [558], FUSE [402]^{P}, HornDroid [122]^{P}, IFT [185], IIF [551], IREA [297], Pegasus [138], RetroSkeleton [161], Riskranker [231]^{P}, SAAF [250]^{P}, ScanDal [291]^{P}, StaDynA [571], TaintDroid [181], VetDroid [566], Wognsen_ [519]	7%

*****: Including all other surveyed papers that are not mentioned as Inter-Comp or Inter-App
******: Including all other surveyed papers that are not mentioned as Framework Level
P: Partial coverage (usually adopting conservative approach and marking all instances of special code as dangerous/suspicious.)
K: exclusively at Kernel-level

2.5.1.3 Granularity of Threat

We can observe from Table 2.4 that the majority of the Android security approaches are intended to detect and mitigate security issues in a single component, comprising 79% of the overall papers studied in this literature review, while a comparatively low number of approaches (21%) have been applied to inter-component analysis.

The 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* [308, 310] 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 [203, 83, 80]. For example, COVERT [80, 417] 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 (18% compared to 2% and 1%). *Epicc* [371] and its successor *IC3* [370] try to precisely infer Intent values, which are necessary information for identifying vulnerable communications. *BlueSeal* [251] and *Woodpecker* [233] briefly discuss AIDL, as another ICC mechanism, and how to incorporate it in control flow

graph. Finally, *ContentScope* [587] examines the security threats of using shared data as the third way of achieving ICC.

2.5.1.4 Depth of Threat

We observe that most approaches perform the analysis at the application-level (88%), but about ten percent of the approaches consider the underlying Android framework for analysis (12%). 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* [69] and *Stowaway* [198], through the analysis of the Android framework, obtained permission-API mappings that specify the permissions required to invoke each Android API call. However, due to intrinsic limitations of static and dynamic analyses adopted by PScout and Stowaway, respectively, the generated mappings are incomplete or inaccurate. Addressing this shortcoming, more recent approaches [94, 566] have attempted to enrich the extracted permission mappings. Such permission mappings have then been used by many other approaches, among others, for detecting *over-privileged* apps that violate the "Principle of Least Privilege" [426] (See Section 2.4.1.2).

Among the approaches performing analysis at the framework level, some look into the vulnerabilities of the Android framework that could lead to security breaches of the system, such as design flaws in the permission model [78], security hazards in push-messaging services [314], or security vulnerabilities of the WebView component [274, 144, 220].

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 2.5.3.1.

2.5.1.5 Type of Artifact

As discussed in Section 2.4, Android apps are composed of several artifacts at different levels of abstraction, such as high-level configuration files and code implementation. We can observe from Table 2.4 that most of the studied approaches analyze multiple types of artifacts.

Type of Configuration. 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 [183], 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 [183]"*. 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* [328] and *BlueSeal* [251, 460] 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 contain sensitive data, are an example of security-critical information embedded in layout files [67]. An example of a technique that leverages this information is *AsDroid* [260]. 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. Another example is *MassVet* [137] that captures the user interface of apps by encoding layouts in a graph structure called a view graph and then detects repackaged malware by calculating the similarity of view graphs.

Besides manifest and layout files, a few other types of configuration files are processed by a number of analyses. For instance, string resources (i.e., String.xml) are parsed to capture predefined URL strings [355] and to identify the label of sensitive fields [360], or style definition files, among other resources, are leveraged to detect repackaged malware apps [456].

Type of Unconventional Code. In addition to the configuration files, most of the surveyed research perform analysis on apps' code. However, due to analysis challenges, the majority of those techniques (over 80%) neglect special types of code, such as obfuscated, native, dynamically loaded, or reflective code, existing in many apps, including malware.

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* [203]. 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. These types of obfuscation 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 [590, 384, 231] can only identify the usage of native code, particularly if it is used in an abnormal way. For instance, *RiskRanker* [231] 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 approaches, such as the tool developed by Poeplau et al. [384], 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* [232]. The other thrust of research attempts to resolve reflection using more advanced analysis. For example, *DroidSafe* [227] employs string and points-to analysis to replace reflective calls with direct calls. As another example, *Pegasus* [138] rewrites an app by injecting dynamic checks when reflective calls are made.

As mentioned above, a significant portion of surveyed research that are trying to address special types of code, adopt a conservative approach. That is, instead of analyzing the content of challenging parts of the app code, e.g. called native library or dynamically loaded class, they flag any usage of such code as suspicious. To distinguish those techniques that *partially* analyze native, obfuscated, dynamic, or reflective code, we marked them with P in Table 2.4.

2.5.2 Approach Characteristics (Solution)

Tables 2.5–2.8 present 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.

2.5.2.1 Type of Program Analysis

Table 2.5 separates the approaches with respect to the type of program analysis they leverage. As discussed in Section 2.4, 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 at the cost of generating false warnings. This could explain why the percentage of static analysis techniques (65%) surpasses the percentage of approaches that rely on dynamic analysis techniques (49%).

SCanDroid [211] and TaintDroid [181] 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

Table 2.5: Solution Specific Categorization of the Reviewed Research, Part 1

Dimension		Approaches	%
Type of Program Analysis	Static	A3 [332], A5 [502], AAPL [327], AASandbox [106], Achara_ [49], Adagio [216], AdDroid [379], Adebayo_ [50], AdRisk [232], Amandroid [514], AMDetector [569], Anadroid [317], Ananas [173], AnDarwin [152], Androguard [166], AndroidLeaks [222], Andrubis [320], ApkCombiner [307], APKLancet [545], ApkRiskAnalyzer [142], App-ray [490], Apparecium [489], AppAudit [526], AppCaulk [446], AppContext [546], AppCracker [121], AppIntent [548], Apposcopy [203], AppProfiler [411], AsDroid [260], ASV [257], AuthDroid [506], AutoCog [391], AVDTester [256], Bae_ [76], Bagheri_ [78], Barrera2_ [88], Barros_ [91], Bartel_ [94], Bartsch_ [95], Batyuk_ [96], BayesDroid [493], Bianchi_ [102], Bifocals [144], BlueSeal [251], Brahmastra [101], Brave [377], Brox [335], Buhov_ [119], Canfora3_ [126], Capper [560], Cassandra [326], Cen_ [130], Chabada [228], Chen_ [136], Chen2_ [135], CHEX [328], CMA [455], CoChecker [155], ComDroid [143], ConDroid [445], ContentScope [587], COPES [92], COVERT [80], COVERT_Tool [417], CredMiner [591], Dai_ [157], Dendroid [474], Desnos_ [165], DexDiff [350], DiCerbo_ [131], DidFail [294], DNADroid [151], Dr.Android [268], DRACO [100], Drebin [64], DroidADDMiner [316], DroidAlarm [579], DroidAnalytics [576], DroidAnalyzer [448], DroidAPIMiner [45], DroidChecker [133], DroidCIA [140], DroidForce [396], DroidFuzzer [549], DroidGuard [81], DroidKin [226], DroidMat [522], DroidMiner [540], DroidMOSS [584], DroidPermissionMiner [68], DroidRanger [590], DroidRay [577], DroidRisk [513], DroidSafe [227], Droidsearch [395], DroidSIFT [558], DroidSim [479], Duet [253], Elish_ [177], Enck_ [182], Epicc [371], Fedler_ [194], Fest [567], Flowdroid [67], FUSE [402], Gallingani_ [213], Gallo_ [214], Gates_ [218], Geneiatakis_ [219], Graa_ [230], Graa2_ [229], GroddDroid [47], Han_ [239], Harehunter [46], HornDroid [122], Huang_ [255], HunterDroid [556], ICC_Map [178], IccTA [308], IFT [185], IIF [551], IPCInspection [202], IREA [297], IVDroid [190], Jiao_ [272], Johnson_ [280], Juxtapp [241], Kadir_ [282], Kantola_ [283], Kate_ [285], Kim_ [290], Kirin [183], KLD [453], LeakMiner [547], Li_ [312], Lintent [118], Lu_ [329], Ma_ [336], Mahmood_ [338], Malek_ [340], MalloDroid [186], Mama [430], Manilyzer [197], Mann_ [341], Marvin [319], MassVet [137], MAST [132], Masud_ [344], Matsumoto_ [345], MIGDroid [254], Mobile-Sandbox [471], MobSafe [534], MonkeyDroid [334], Moonsamy_ [351], Moonsamy2_ [352], MorphDroid [204], Mudflow [70], Mutchler_ [355], NoInjection [274], Onwuzurike_ [374], OpSeq [54], PaddyFrog [523], Paupore_ [378], PCLeaks [309], Pedal [322], Pegasus [138], Peiravian_ [380], Peng_ [381], PermCheckTool [501], PermissionFlow [436], Permlyzer [536], Poeplau_ [384], ProfileDroid [515], PScout [69], Quan_ [392], RAMSES [172], Relda [234], ResDroid [456], Riskmon [275], RiskMon2 [276], Riskranker [231], SAAF [250], SADroid [240], Sahs_ [424], Sanz_ [432], Sarma_ [433], Sayfullina_ [435], ScanDal [291], SCanDroid [211], Scoria [497], SecUP [533], SEFA [524], Seneviratne_ [447], Shabtai_ [449], Shen_ [461], SherlockDroid [61], SmartDroid [574], Smith_ [467], SMV-HUNTER [470], StaDynA [571], STAMBA [109], SUPOR [259], TMSVM [529], TouchDevelop [528], TraceDroid [496], TrustDroid [570], UID [179], UIPicker [360], ViewDroid [555], Wang_ [510], WeChecker [154], WifiLeaks [48], Wognsen_ [519], Woodpecker [233], WuKong [505], Yerima_ [550], You_ [552], Zhou_ [583], Zuo_ [594]	65%
	Dynamic (E)	A5 [502], AASandbox [106], Achara_ [49], ADDICTED [585], Afonso_ [51], AMDetector [569], Ananas [173], Andrubis [320], AntiMalDroid [568], Apex [362]^{E}, App-ray [490], AppAudit [526], AppCaulk [446], AppCracker [121], AppFence [252], AppGuard [75], AppInspector [223], AppIntent [548], AppProfiler [411], AppsPlayground [398], APSET [427], Aquifer [359], ASF [73]^{E}, ASM [247]^{E}, ASV [257], AuDroid [382]^{E}, Aurasium [535], AuthDroid [506], AVDTester [256], Bae_ [76], Bal_ [85], Berthome_ [99], Brahmastra [101], Brave [377], Bugiel_ [115], Buzzer [127], Canfora_ [124], Canfora3_ [126], Capper [560]^{E}, CMA [455], Compac [512]^{E}, ConDroid [445], ContentScope [587], ConUCON [84]^{E}, CopperDroid [403], CopperDroid2 [485], CRePE [147], Crowdroid [120], Dagger [541], DataChest [589], DeepDroid [511]^{E}, Defensor [375], Dr.Android [268], DRACO [100], DroidAnalyst [212], DroidAnalytics [576], DroidBarrier [56], DroidDolphin [525], DroidForce [396]^{E}, DroidFuzzer [549], DroidGuard [81]^{E}, DroidLogger [158], DroidPAD [331], DroidRay [577], DroidScope [539], DroidTest [412], DroidTrace [578], DroidTrack [425], EASE-Android [508], Fedler_ [194], Feth_ [205]^{E}, FineDroid [565]^{E}, FireDroid [414], FlaskDroid [117]^{E}, Geneiatakis_ [219], Graa_ [230], Graa2_ [229], GroddDroid [47], Ham_ [236], HunterDroid [556], I-ARM-Droid [162]^{E}, IntentFuzzer [542], IPCInspection [202], Isohara_ [262], Jeon_ [267], Jeong_ [270], Jia_ [271]^{E}, Jiao_ [272], Jung_ [281]^{E}, Kadir_ [282], Kantola_ [283], Karami_ [284]^{E}, Kim_ [290], Kynoid [443], LazyTainter [516], Lee_ [302], Leontiadis_ [305], Li2_ [313], MADAM [168], Mahmood_ [338], Malek_ [340], Manilyzer [197], Marvin [319], Masud_ [344], MeadDroid [304], Mobile-Sandbox [471], MobSafe [534], MockDroid [98]^{E}, Morbs [509], MOSES [573]^{E}, MpDroid [488]^{E}, Mutchler_ [355], NDroid [389], Nishimoto_ [367], Onwuzurike_ [374], Paranoid-Android [386], PatchDroid [353], Patronus [478], Paupore_ [378], pBMDS [530], PeBA [97], Pegasus [138]^{E}, Permlyzer [536], PICARD [169], Porscha [372]^{E}, PREC [248], ProfileDroid [515], PUMA [429], PuppetDroid [221], Quan_ [392], QUIRE [167], RetroSkeleton [161]^{E}, Riskmon [275], RiskMon2 [276], Saint [373]^{E}, Schmidt_ [440], SCSdroid [318], SFG [59], Shebaro_ [457]^{E}, Short_ [465], SmartDroid [574], SMV-HUNTER [470], StaDynA [571], STAMBA [109], Stowaway [198], Su_ [473], TaintDroid [181], Tchakounte_ [486], TISSA [592]^{E}, TMSVM [529], TraceDroid [496], TreeDroid [159]^{E}, UIPicker [360]^{E}, Uranine [401], VetDroid [566], WifiLeaks [48], Wijesekera_ [518], Xmandroid [114]^{E}, Yaase [413]^{E}, Yerima_ [550], You_ [552], Zuo_ [594]	49%
	Hybrid	A5 [502], AASandbox [106], Achara_ [49], AMDetector [569], Ananas [173], Andrubis [320], App-ray [490], AppAudit [526], AppCaulk [446], AppCracker [121], AppIntent [548], AppProfiler [411], ASV [257], AuthDroid [506], AVDTester [256], Bae_ [76], Brahmastra [101], Brave [377], Canfora3_ [126], Capper [560], CMA [455], ConDroid [445], ContentScope [587], Dr.Android [268], DRACO [100], DroidAnalytics [576], DroidForce [396], DroidFuzzer [549], DroidGuard [81], DroidRay [577], Fedler_ [194], Geneiatakis_ [219], Graa_ [230], Graa2_ [229], GroddDroid [47], HunterDroid [556], IPCInspection [202], Jiao_ [272], Kadir_ [282], Kantola_ [283], Kim_ [290], Lee_ [302], Mahmood_ [338], Malek_ [340], Manilyzer [197], Marvin [319], Masud_ [344], Mobile-Sandbox [471], MobSafe [534], Mutchler_ [355], Onwuzurike_ [374], Paupore_ [378], Pegasus [138], Permlyzer [536], ProfileDroid [515], Quan_ [392], Riskmon [275], RiskMon2 [276], SmartDroid [574], SMV-HUNTER [470], StaDynA [571], STAMBA [109], TMSVM [529], TraceDroid [496], UIPicker [360], WifiLeaks [48], Yerima_ [550], You_ [552], Zuo_ [594]	21%

E: Enforcing security policies (providing a level of protection, in addition to dynamic detection)

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 [470] 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. [543] 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 difference between an Android app and a pure Java pro-

gram, 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 [67, 547, 308, 328, 251, 460]. For instance, *FlowDroid* [67] 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* [308, 310] 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 entry points are discovered is also crucial for a precise analysis. Some approaches [233] [587] simply rely on the domain knowledge, including the Android API documentation, to identify entry points. Some other approaches employ more systematic methods. For instance, *CHEX* describes a sound method to automatically discover different types of app entry points [328]. It iterates over all uncalled framework methods overridden by app, and connects those methods to the corresponding call graph node.

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 [143, 182, 294, 308, 371]. *Epicc* [371], 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 [406], 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* [233] 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* [328] 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. *DroidSafe* [227] attempts to achieve a combination of precision and scalability by generating analysis stubs, abstractions of underlying implementation, which are incomplete for runtime, but complete for the analysis. Finally, to automatically classify Android system APIs as sources and sinks, *SuSi* [394] 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* [67], *DroidForce* [396], *IccTA* [308, 310], and *DidFail* [294].

2.5.2.2 Supplementary Techniques

We observe that most approaches (over 70%) only rely on program analysis techniques to assess the security of Android software. Less than 30% of the approaches employ other complementary techniques in their analysis. Among them, machine learning and formal

Table 2.6: Solution Specific Categorization of the Reviewed Research, Part 2

Dimension		Approaches	%
Supplementary Techniques	Machine Learning (P|N)	AAPL [327]^{N}, Adagio [216], Adebayo_ [50]^{P}, Afonso_ [51], AnDarwin [152], AntiMalDroid [568], AppContext [546], AutoCog [391]^{N}, Bae_ [76], Barrera2_ [88], BayesDroid [493]^{P}, Brave [377], Canfora_ [124], Cen_ [130]^{P}, Chabada [228]^{N}, Crowdroid [120], Dai_ [157], Dendroid [474], DiCerbo_ [131], Drebin [64], DroidADDMiner [316], DroidAPIMiner [45], DroidDolphin [525], DroidLegacy [164], DroidMat [522], DroidPAD [331], DroidPermissionMiner [68], DroidSIFT [558], EASEAndroid [508], Fest [567], Gates_ [218]^{P}, Huang_ [255], Jiao_ [272]^{P}, Kate_ [285]^{P}, KLD [453]^{P}, Ma_ [336]^{N}, MADAM [168], Mama [430], Marvin [319], MassVet [137], MAST [132], Milosevic_ [348], MobSafe [534], MonkeyDroid [334]^{N}, Moonsamy_ [351], Moonsamy2_ [352], Mudflow [70], OpSeq [54], pBMDS [530], Pedal [322], Peiravian_ [380], Peng_ [381]^{P}, PICARD [169]^{P}, PUMA2 [243], Quan_ [392], RAMSES [172], ResDroid [456], Riskmon [275], RiskMon2 [276], Sahs_ [424], Sanz_ [432], Sarma_ [433], Sayfullina_ [435]^{P}, Schmidt_ [440], Shabtai_ [449], SherlockDroid [61], Su_ [473], SUPOR [259]^{N}, TMSVM [529], UIPicker [360]^{N}, Wang_ [510], WuKong [505], Yerima_ [550]^{P}, Zhou_ [583]	22%
	Formal Analysis	Apposcopy [203], APSET [427], Armando_ [63], Bagheri_ [78], Barbon_ [86], Cassandra [326], COVERT [80], COVERT_Tool [417], DroidGuard [81], HornDroid [122], Jia_ [271], Lintent [118], Lu_ [329], Mann_ [341], MorphDroid [204], Pegasus [138], SADroid [240], ScanDal [291], Scoria [497], Smith_ [467], Song_ [468], TreeDroid [159]	7%
Auto. Level	Automatic	Others *	93%
	Semi-Automatic	Achara_ [49], AdSplit [459], AndroidLeaks [222], APKLancet [545], Apposcopy [203], AppProfiler [411], Barros_ [91], Batyuk_ [96], Crowdroid [120], Dr.Android [268], DroidForce [396], DroidRay [577], Graa_ [230], Ham_ [236], IFT [185], IREA [297], Isohara_ [262], Mann_ [341], PuppetDroid [221], Scoria [497], Smith_ [467], StaDynA [571], Stowaway [198]	7%

P: Probabilistic approaches, **N**: Natural language processing (NLP) is used
*: Including all other surveyed papers that are not mentioned as Semi-Automatic

analysis techniques are the most widely used, comprising 22% 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 [80].

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* [228] and its successor *MUDFLOW* [70], 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.

Natural language processing (NLP) is another supplementary technique employed by CHABADA

and a few other approaches (e.g., AAPL[327], AutoCog [391], SUPOR [259], UIPicker [360]), mainly to process apps' meta-data, such as app descriptions, which are expressed in natural language form. Moreover, probabilistic approaches are also leveraged by a number of machine learning-based tools (e.g. [493, 381, 130, 435]) to distinguish malware apps from benign ones, according to the observed probability of extracted features. Research using NLP and probabilistic methods are highlighted by *N* and *P*, respectively, in Table 2.6.

2.5.2.3 Automation Level

We observe that most approaches (93%) 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 (§ 2.5.3).

A number of approaches, however, require some manual effort (7%); 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, *IFT* [185] 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 IFT'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.

Table 2.7: Solution Specific Categorization of the Reviewed Research, Part 3

Dimension		Static Approaches	% (%)*
Analysis Data Structure	Text-based	A3 [332], AASandbox [106], Achara_ [49], AdDroid [379], Adebayo_ [50], AMDetector [569], Ananas [173], Andrubis [320], ApkRiskAnalyzer [142], App-ray [490], AppCracker [121], AppProfiler [411], AuthDroid [506], AutoCog [391], Bae_ [76], Bagheri_ [78], Barrera2_ [88], Batyuk_ [96], BayesDroid [493], Bifocals [144], Brave [377], Buhov_ [119], Canfora3_ [126], Cassandra [326], Cen_ [130], Chabada [228], Chen2_ [135], Dai_ [157], DiCerbo_ [131], Dr.Android [268], DRACO [100], Drebin [64], DroidAnalyzer [448], DroidFuzzer [549], DroidKin [226], DroidMat [522], DroidMOSS [584], DroidPermissionMiner [68], DroidRay [577], DroidRisk [513], Droidsearch [395], Duet [253], Fedler_ [194], Fest [567], Gallo_ [214], Gates_ [218], Geneiatakis_ [219], Huang_ [255], IREA [297], Jiao_ [272], Johnson_ [280], Juxtapp [241], Kadir_ [282], Kantola_ [283], Kate_ [285], Kim_ [290], Kirin [183], KLD [453], Lintent [118], Lu_ [329], Ma_ [336], Malek_ [340], MalloDroid [186], Mama [430], Manilyzer [197], Mann_ [341], Marvin [319], MAST [132], Masud_ [344], Matsumoto_ [345], Mobile-Sandbox [471], MobSafe [534], Moonsamy_ [351], Moonsamy2_ [352], Onwuzurike_ [374], OpSeq [54], Paupore_ [378], Pedal [322], Peiravian_ [380], Peng_ [381], PermCheckTool [501], Permlyzer [536], ProfileDroid [515], Quan_ [392], RAMSES [172], ResDroid [456], Riskmon [275], RiskMon2 [276], Sahs_ [424], Sanz_ [432], Sarma_ [433], Sayfullina_ [435], Scoria [497], SecUP [533], Seneviratne_ [447], Shabtai_ [449], SherlockDroid [61], Smith_ [467], SMV-HUNTER [470], STAMBA [109], TraceDroid [496], UIPicker [360], Wang_ [510], WifiLeaks [48], WuKong [505], Yerima_ [550], You_ [552]	49% (32%)
	Control Flow Graph (CFG)	A5 [502], Adagio [216], Amandroid [514], Anadroid [317], Androguard [166], APKLancet [545], Apparecium [489], AppCaulk [446], AppContext [546], Apposcopy [203], AsDroid [260], ASV [257], AVDTester [256], Barros_ [91], Bianchi_ [102], BlueSeal [251], Brahmastra [101], Capper [560], CMA [455], CoChecker [155], ComDroid [143], ContentScope [587], COVERT [80], COVERT_Tool [417], CryptoLint [174], Dendroid [474], Desnos_ [165], DexDiff [350], DroidAlarm [579], DroidChecker [133], DroidCIA [140], DroidForce [396], DroidGuard [81], DroidMiner [540], DroidSIFT [558], DroidSim [479], Elish_ [177], Enck_ [182], Epicc [371], Flowdroid [67], Gallingani_ [213], Graa_ [230], Graa2_ [229], GroddDroid [47], Harehunter [46], IccTA [308], IIF [551], IVDroid [190], LeakMiner [547], MassVet [137], MonkeyDroid [334], MorphDroid [204], NoInjection [274], PaddyFrog [523], PCLeaks [309], PermissionFlow [436], Poeplau_ [384], Riskranker [231], SAAF [250], SADroid [240], Sahs_ [424], ScanDal [291], SEFA [524], TMSVM [529], TouchDevelop [528], UID [179], WeChecker [154], Wognsen_ [519], Woodpecker [233], Zhou_ [583]	31% (21%)
	Call Graph (CG)	Adagio [216], Amandroid [514], Androguard [166], AndroidLeaks [222], APKLancet [545], Apparecium [489], AppAudit [526], AppCaulk [446], AppContext [546], AppIntent [548], Apposcopy [203], AppSealer [559], AsDroid [260], ASV [257], Bartel_ [94], Bianchi_ [102], BlueSeal [251], Brahmastra [101], Brox [335], Capper [560], CMA [455], CoChecker [155], ContentScope [587], COPES [92], COVERT [80], COVERT_Tool [417], CryptoLint [174], DroidAlarm [579], DroidChecker [133], DroidCIA [140], DroidGuard [81], DroidRanger [590], DroidSafe [227], DroidSIFT [558], Epicc [371], Flowdroid [67], FUSE [402], Gallingani_ [213], HunterDroid [556], IccTA [308], IPCInspection [202], LeakMiner [547], Mahmood_ [338], Malek_ [340], MonkeyDroid [334], MorphDroid [204], Mudflow [70], Mutchler_ [355], NoInjection [274], PaddyFrog [523], PCLeaks [309], PermissionFlow [436], Poeplau_ [384], PScout [69], Relda [234], SADroid [240], SEFA [524], StaDynA [571], TouchDevelop [528], TrustDroid [570], UID [179], WeChecker [154], Woodpecker [233], Zuo_ [594]	28% (19%)
	Inter-procedural CFG (ICFG)	Amandroid [514], AppContext [546], Apposcopy [203], Capper [560], COVERT [80], COVERT_Tool [417], CryptoLint [174], DroidChecker [133], DroidGuard [81], Epicc [371], Flowdroid [67], Gallingani_ [213], IccTA [308], LeakMiner [547], MonkeyDroid [334], MorphDroid [204], PCLeaks [309], PermissionFlow [436], Poeplau_ [384], SEFA [524], TouchDevelop [528], UID [179], WeChecker [154], Woodpecker [233]	10% (7%)
Sensitivity of Analysis	Flow	AAPL [327], AdRisk [232], Amandroid [514], Apparecium [489], AppContext [546], Apposcopy [203], AppSealer [559], AsDroid [260], Barros_ [91], Bartsch_ [95], Bianchi_ [102], BlueSeal [251], Brox [335], Capper [560], CHEX [328], CMA [455], ComDroid [143], ContentScope [587], COVERT [80], COVERT_Tool [417], CredMiner [591], CryptoLint [174], DidFail [294], DroidADDMiner [316], DroidAlarm [579], DroidAPIMiner [45], DroidChecker [133], DroidForce [396], DroidGuard [81], DroidSIFT [558], Elish_ [177], Enck_ [182], Epicc [371], Flowdroid [67], Han_ [239], Harehunter [46], HornDroid [122], IccTA [308], IFT [185], LeakMiner [547], MorphDroid [204], NoInjection [274], PCLeaks [309], Pegasus [138], PermissionFlow [436], Poeplau_ [384], Riskranker [231], ScanDal [291], SCanDroid [211], SEFA [524], SUPOR [259], UID [179], WeChecker [154]	23% (16%)
	Context	AAPL [327], Amandroid [514], ApkCombiner [307], AppContext [546], AppIntent [548], Apposcopy [203], AppSealer [559], Brox [335], Capper [560], CHEX [328], COVERT [80], COVERT_Tool [417], DidFail [294], DroidADDMiner [316], DroidForce [396], DroidGuard [81], DroidSafe [227], DroidSIFT [558], Epicc [371], Flowdroid [67], Han_ [239], IccTA [308], IFT [185], NoInjection [274], PCLeaks [309], Pegasus [138], PermissionFlow [436], ScanDal [291], SCanDroid [211], SUPOR [259], UID [179], WeChecker [154]	14% (10%)
	Path	ConDroid [445], ContentScope [587], DroidAnalytics [576], DroidForce [396], Gallingani_ [213], Woodpecker [233]	3% (2%)
Code Representation	Java Source	Bartel_ [94], IFT [185], IVDroid [190], Lu_ [329], Mann_ [341], Matsumoto_ [345], PermCheckTool [501], PScout [69], SCanDroid [211], Smith_ [467], TouchDevelop [528]	5% (3%)
	Java Byte	AnDarwin [152], AndroidLeaks [222], AppAudit [526], AppCracker [121], AppIntent [548], AppProfiler [411], AppSealer [559], AsDroid [260], Bagheri_ [78], Capper [560], Cen_ [130], Chen2_ [135], CoChecker [155], ComDroid [143], Desnos_ [165], DNADroid [151], DroidChecker [133], Duet [253], Enck_ [182], IPCInspection [202], KLD [453], LeakMiner [547], Lee_ [302], Malek_ [340], Onwuzurike_ [374], Pedal [322], Pegasus [138], PermissionFlow [436], Permlyzer [536], Relda [234], UID [179]	13% (9%)
	Jimple	A5 [502], AppContext [546], Apposcopy [203], Bartsch_ [95], BlueSeal [251], Brahmastra [101], COPES [92], COVERT [80], COVERT_Tool [417], DidFail [294], DroidForce [396], DroidGuard [81], DroidSafe [227], Elish_ [177], Epicc [371], Flowdroid [67], Geneiatakis_ [219], Harehunter [46], IccTA [308], MonkeyDroid [334], Mudflow [70], Mutchler_ [355], PCLeaks [309], Shen_ [461], WeChecker [154]	11% (7%)
	Smali	AASandbox [106], AdRisk [232], AdSplit [459], Ananas [173], ApkCombiner [307], APKLancet [545], Apparecium [489], Aurasium [535], AVDTester [256], Batyuk_ [96], Chabada [228], Chen_ [136], CHEX [328], CMA [455], ContentScope [587], CredMiner [591], Dr.Android [268], DroidLegacy [164], DroidMOSS [584], Harehunter [46], IREA [297], Johnson_ [280], Ma_ [336], Mobile-Sandbox [471], MobSafe [534], PaddyFrog [523], ProfileDroid [515], SAAF [250], SADroid [240], Seneviratne_ [447], SmartDroid [574], SMV-HUNTER [470], SUPOR [259], TrustDroid [570], ViewDroid [555], Wognsen_ [519], Woodpecker [233], WuKong [505], Yerima_ [550], Zuo_ [594]	17% (12%)

*: The last column of this table should be read as follows:Percentage among static approaches (Percentage among all approaches)

2.5.2.4 Analysis Data Structures [7]

Almost half of *static* approaches (49%) leverage light-weight analysis that only relies on text-based information retrieval techniques. Such approaches treat app's code and configuration as unstructured texts and try to extract security critical keywords and phrases (e.g., permissions, sensitive APIs) for further analysis using supplementary techniques (See Section 2.4.2.2). For instance, *Drebin* [64] extracts sets of strings, such as permissions, app components, and intent filters by parsing the manifest, and API calls, and network addresses from dex code. It then maps those extracted features to a vector space, which is further used for learning-based malware detection.

On the other hand, many techniques take the structure of code into account when extracting the security model of the apps. For this purpose, various data structures that represent apps at an abstract level are commonly used by those analysis techniques. We observe that call graphs (CGs) and control flow graphs (CFGs) are the most frequently used data structure in the surveyed papers.

Taint information are propagated through call graph, among other things, to determine the reachability of various sinks from specific sources. *LeakMiner* [547], *RiskRanker* [231], *TrustDroid* [570], *ContentScope* [587] and *IPC Inspection* [202] 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* [371] and *AsDroid* [260] are among the approaches that use call graph for this purpose. In addition, *PScout* [69] and

[7]The percentages reported in Sections 2.5.2.4, 2.5.2.5 and 2.5.2.6 are calculated only for the static techniques.

PermissionFlow [436] 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. *ContentScope* [587], 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. [182] 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* [67], for example, traverses ICFG to track tainted variables; *Epicc* [371] also performs string analysis over ICFG; *IccTA* [308, 310] 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* [233] locates capability leaks (See section 2.5.1.1) 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. 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 [371]. *CHEX* [328] 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. Another data structure commonly used by app clone detectors, such as *AnDarwin* [152] and *DNADroid* [151], is

program dependency graph (PDG). By capturing control and data dependencies between code fragments, a PDG is able to compare similarity between app pairs.

2.5.2.5 Sensitivity of Analysis

Apart from lightweight, text-based approaches, other static approaches have adopted a level of sensitivity in their analysis. According to our survey, flow-sensitive approaches that consider the program statements sequence, have the highest frequency (23%) among the *static* approaches. Following that, 14% of static techniques are context-sensitive, that is, they compute the calling context of method calls. Finally, 3% of static analyses are path-sensitive, meaning that only a handful of analysis approaches distinguish information obtained from different execution paths. Generally, approaches with higher sensitivity, i.e.,considering more program properties for the analysis, generate more accurate results, but they are less scalable in practice.

2.5.2.6 Code Representation

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*[211], 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 2.8 shows the distribution of the approaches based on the target IR of the analysis.

According to the diagram, Smali [19] is the most popular intermediate representations, used in 17% of those studied approaches that are performing analysis on a type of IR. Also, 13%

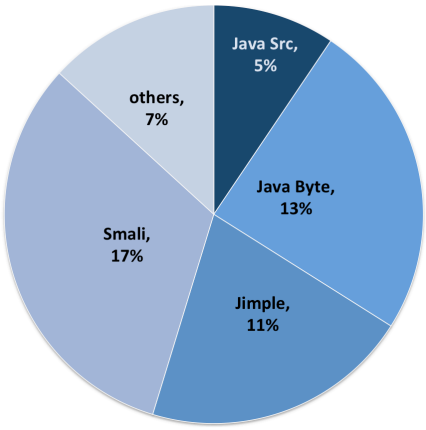


Figure 2.8: Distribution of research based on the type of code or intermediate representation (IR) used for analysis.

of such approaches, in the pre-processing step, retarget Dalvik byte-code to Java byte-coded JAR files. 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.

2.5.2.7 Inspection Level [8]

Dynamic approaches monitor an app's behavior using different techniques. According to our results, about 35% of *dynamic* approaches intercept events that occur within the emulated environments by modifying virtual machines (VMs). VM-based dynamic analyses are further distinguishable by the type of virtual machine they modify: Dalvik VM (e.g., *TaintDroid* [181]) or QEMU VM (e.g., *CopperDroid* [485]). While QEMU-based systems work on a lower level and are able to trace native code, Dalvik-based techniques tend to be more efficient [364]. Therefore, a few tools, such as [320], take advantage of both techniques.

Around 39% of studied dynamic analyses weave monitoring code into Android apps or framework APIs to capture app behaviors. Approaches that monitor the framework are marked with F in Table 2.6. Different libraries are developed by the research community to facili-

[8]The percentages reported in Sections 2.5.2.7 and 2.5.2.8 are calculated only for the dynamic techniques.

Table 2.8: Solution Specific Categorization of the Reviewed Research, Part 4

Dimension		Dynamic Approaches	% (%)*
Inspection Level	Application (F)	Achara_ [49], Ananas [173], Androguard [166], AntiMalDroid [568]^{F}, Apex [362], APKLancet [545], AppCaulk [446], AppGuard [75], AppIntent [548], AppProfiler [411], AppSealer [559], APSET [427], ASM [247], Aurasium [535], AutoCog [391], Bagheri_ [78], Bartsch_ [95], Berthome_ [99], Brahmastra [101], Buzzer [127], Capper [560], CMA [455], ConDroid [445], ContentScope [587], CRePE [147], Dr.Android [268], DroidAnalytics [576], DroidDolphin [525], DroidForce [396], DroidFuzzer [549], DroidGuard [81], DroidLogger [158], DroidRay [577], DroidTrack [425], Feth_ [205]^{F}, FineDroid [565]^{F}, Graa_ [230], GroddDroid [47], HunterDroid [556], I-ARM-Droid [162], ICC_Map [178], IPCInspection [202], Jeon_ [267], Jung_ [281]^{F}, Kantola_ [283], Lee_ [302], Li2_ [313], Lintent [118], Lu_ [329], Mahmood_ [338], Matsumoto_ [345], MobSafe [534], Morbs [509], MpDroid [488]^{F}, Mutchler_ [355], Nishimoto_ [367]^{F}, PaddyFrog [523], Paupore_ [378], Pegasus [138], Permlyzer [536], Porscha [372]^{F}, PUMA [429], PUMA2 [243], QUIRE [167], SADroid [240], SFG [59]^{F}, Shebaro_ [457]^{F}, Short_ [465], SmartDroid [574], Smith_ [467], SMV-HUNTER [470], TruStore [572], Uranine [401], WifiLeaks [48]^{F}, Wijesekera_ [518], Xmandroid [114], You_ [552]	39% (23%)
	Kernel	ADDICTED [585], Afonso_ [51], Ananas [173], Andrubis [320], AppsPlayground [398], Aquifer [359], ASF [73], ASM [247], AVDTester [256], Bugiel_ [115], Canfora_ [124], Canfora3_ [126], Compac [512], CopperDroid [403], CopperDroid2 [485], Crowdroid [120], Dagger [541], DataChest [589], DeepDroid [511], Defensor [375], DroidBarrier [56], DroidScope [539], FineDroid [565], FlaskDroid [117], Ham_ [236], Isohara_ [262], Jeong_ [270], Karami_ [284], MADAM [168], Mobile-Sandbox [471], Paranoid-Android [386], pBMDS [530], PeBA [97], PICARD [169], PREC [248], ProfileDroid [515], Quan_ [392], Schmidt_ [440], SCSdroid [318], Su_ [473], Tchakounte_ [486], TMSVM [529], TraceDroid [496]	27% (13%)
	VM	AASandbox [106], Afonso_ [51], Androguard [166], Andrubis [320], AppAudit [526], AppFence [252], AppInspector [223], AppsPlayground [398], Aquifer [359], ASF [73], ASM [247], ASV [257], AuDroid [382], Aurasium [535], AVDTester [256], Bagheri_ [78], Bal_ [85], Bugiel_ [115], Compac [512], ConUCON [84], CopperDroid [403], CopperDroid2 [485], CRePE [147], DataChest [589], DeepDroid [511], DexDiff [350], Dr.Android [268], DroidPAD [331], DroidRay [577], DroidScope [539], FireDroid [414], FlaskDroid [117], Graa_ [230], IPCInspection [202], Kantola_ [283], Kynoid [443], LazyTainter [516], Lee_ [302], Leontiadis_ [305], MADAM [168], Marvin [319], MeadDroid [304], Mobile-Sandbox [471], Morbs [509], MOSES [573], NDroid [389], Paranoid-Android [386], PatchDroid [353], PeBA [97], Quan_ [392], QUIRE [167], Saint [373], StaDynA [571], TaintDroid [181], TISSA [592], TraceDroid [496], TreeDroid [159], TruStore [572], VetDroid [566], Yaase [413], Zuo_ [594]	35% (18%)
Input Generation	Fuzzing (H)	A5 [502]^{H}, Afonso_ [51], Ananas [173], Andrubis [320], AppsPlayground [398]^{H}, AVDTester [256], Canfora_ [124], ContentScope [587], CopperDroid [403], CopperDroid2 [485]^{H}, Dagger [541], DroidDolphin [525], DroidFuzzer [549], DroidScope [539], DroidTest [412], DroidTrack [425], HunterDroid [556], IntentFuzzer [542], Jiao_ [272], Karami_ [284]^{H}, LazyTainter [516], Mahmood_ [338], Malek_ [340]^{H}, Mobile-Sandbox [471], MobSafe [534], MOSES [573], NDroid [389], Permlyzer [536], PUMA [429]^{H}, RetroSkeleton [161], SmartDroid [574], SMV-HUNTER [470], TaintDroid [181], UIPicker [360], Uranine [401], VetDroid [566], Zuo_ [594]^{H}	23% (11%)
	Symbolic Exec.	AppInspector [223], AppIntent [548], ConDroid [445], DroidAnalytics [576], Malek_ [340], You_ [552]	4% (2%)

H: Heuristics-based Approaches, **F**: Android Framework level
*: The last column should be read as follows: Percentage among dynamic approaches (Percentage among all approaches)

tate app-level monitoring, including: APIMonitor developed and used in *DroidBox* [12], a Soot-based library proposed by [66], and SIF [242], a selective instrumentation framework.

Finally, about 26% of surveyed dynamic techniques capture app behavior through monitoring system calls, using loadable kernel modules (e.g., *ANANAS* [173]) or debugging tools such as `strace` (e.g., *Crowdroid* [120]). Most of the kernel-level techniques are able to trace native code, but they are usually not compatible with multiple versions of Android [364].

To overcome the shortcomings and limitations pertaining to certain monitoring levels, a number of tools leverage a combination of different inspection techniques. According to our survey, around 22% of the studied dynamic approaches perform monitoring at multiple levels. For instance, through monitoring both the Linux kernel and Dalvik VM, *DroidScope* [539],

a dynamic malware analyzer, is able to identify anomalies in app behaviors.

2.5.2.8 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.

We can observe from Table 2.8 that most of such approaches use fuzz testing, comprising 23% of the dynamic approaches studied 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 [347]. 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 [347].

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 [4], generate purely random test case inputs, and thus are often ineffective in practice.

To improve the efficiency of fuzzing techniques, a number of approaches [485, 502, 398] have devised heuristics that guide a fuzzer to cover more segments of app code in an intelligent manner. For instance, by providing meaningful inputs for text boxes by using contextual information, *AppsPlayground* [398] avoids redundant test paths. This in turn enables a more effective exploration of the app code.

A comparatively low number of dynamic approaches (4%) employ symbolic execution, mainly to improve the effectiveness of generated test inputs. For example, AppInspector [223] 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 [548] 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 [233] 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.

2.5.3 Assessment (Validation)

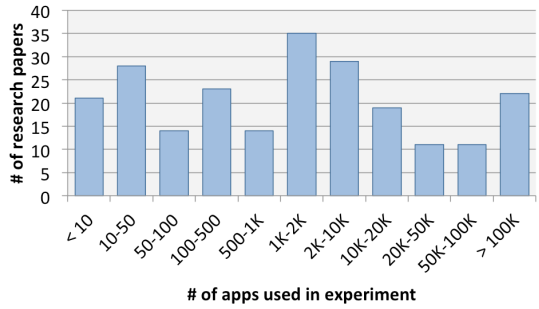
We used reputable sites in our review protocol (See section 2.3), which resulted in the discovery of high-quality refereed research papers from respectable venues. To develop better insights into the quality of the research papers surveyed, here we use Evaluation Method (T 3.1) and Replicability (T 3.2), which are the two validation dimensions in the taxonomy.

Tables 2.9 and 2.10 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.

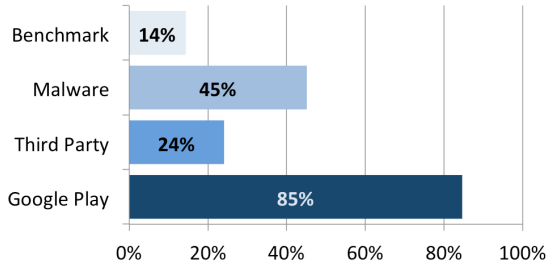
Table 2.9: Assessment Specific Categorization of the Reviewed Research, Part 1

Dimension			Approaches	%
Evaluation	Proof		Apposcopy [203], Bagheri_ [78], Cassandra [326], HornDroid [122], ScanDal [291], SCanDroid [211], Scoria [497], Smith_ [467]	2%
	Empirical	Google Play	AAPL [327], AASandbox [106], Adagio [216], AdDroid [379], Adebayo_ [50], AdRisk [232], Amandroid [514], AMDetector [569], Ananas [173], AnDarwin [152], AndroidLeaks [222], AntiMalDroid [568], ApkCombiner [307], ApkRiskAnalyzer [142], App-ray [490], Apparecium [489], AppAudit [526], AppCaulk [446], AppContext [546], AppCracker [121], AppFence [252], AppGuard [75], AppInspector [223], AppIntegrity [499], AppIntent [548], Apposcopy [203], AppProfiler [411], AppSealer [559], AppsPlayground [398], APSET [427], Aquifer [359], AsDroid [260], AuDroid [382], AutoCog [391], AVDTester [256], Bae_ [76], Bagheri_ [78], Bartsch_ [95], Batyuk_ [96], BayesDroid [493], Bianchi_ [102], Bifocals [144], BlueSeal [251], Capper [560], Cen_ [130], Chabada [228], Chen_ [136], Chen2_ [135], CHEX [328], CMA [455], ComDroid [143], ConDroid [445], ContentScope [587], COPES [92], COVERT [80], CredMiner [591], CryptoLint [174], Dagger [541], Dai_ [157], DataChest [589], DNADroid [151], Dr.Android [268], DRACO [100], DroidADDMiner [316], DroidAnalytics [576], DroidAnalyzer [448], DroidAPIMiner [45], DroidChecker [133], DroidCIA [140], DroidDolphin [525], DroidForce [396], DroidGuard [81], DroidKin [226], DroidMat [522], DroidMiner [540], DroidRanger [590], DroidRay [577], DroidRisk [513], Droidsearch [395], DroidSIFT [558], DroidSim [479], DroidTest [412], DroidTrace [578], Duet [253], Enck_ [182], Epicc [371], Flowdroid [67], FUSE [402], Gallingani_ [213], Gates_ [218], Han_ [239], Harehunter [46], HornDroid [122], HunterDroid [556], I-ARM-Droid [162], ICC_Map [178], IccTA [308], IntentFuzzer [542], IVDroid [190], Jiao_ [272], Johnson_ [280], Juxtapp [241], Kantola_ [283], Kate_ [285], Kirin [183], LeakMiner [547], Leontiadis_ [305], Ma_ [336], MalloDroid [186], Manilyzer [197], Marvin [319], MassVet [137], MAST [132], MeadDroid [304], MockDroid [98], Moonsamy2_ [352], MorphDroid [204], MpDroid [488], Mudflow [70], Mutchler_ [355], NDroid [389], NoFrak [220], NoInjection [274], Onwuzurike_ [374], OpSeq [54], Patronus [478], PCLeaks [309], Pedal [322], Pegasus [138], Peiravian_ [380], Peng_ [381], PermCheckTool [501], PermissionFlow [436], Permlyzer [536], Poeplau_ [384], PREC [248], ProfileDroid [515], PUMA [429], PUMA2 [243], PuppetDroid [221], Quan_ [392], RAMSES [172], Relda [234], Ren_ [405], ResDroid [456], RetroSkeleton [161], Riskmon [275], RiskMon2 [276], Riskranker [231], SAAF [250], Sarma_ [433], ScanDal [291], SEFA [524], Seneviratne_ [447], Shabtai_ [449], Shen_ [461], SherlockDroid [61], Short_ [465], SMV-HUNTER [470], StaDynA [571], Stowaway [198], SUPOR [259], TaintDroid [181], TISSA [592], TMSVM [529], TongxinLi_ [314], UID [179], UIPicker [360], Uranine [401], VetDroid [566], Wang_ [510], WeChecker [154], WifiLeaks [48], Wognsen_ [519], Woodpecker [233], Xmandroid [114], Zhou_ [583], Zuo_ [594]	53%
		Third Party	Adagio [216], Afonso_ [51], AnDarwin [152], AndroidLeaks [222], APKLancet [545], AppCracker [121], AppIntegrity [499], AppProfiler [411], APSET [427], AsDroid [260], Aurasium [535], Bagheri_ [78], Barros_ [91], Chen_ [136], CHEX [328], CoChecker [155], ContentScope [587], COVERT [80], CredMiner [591], DNADroid [151], Drebin [64], DroidAnalytics [576], DroidAnalyzer [448], DroidAPIMiner [45], DroidChecker [133], DroidGuard [81], DroidLegacy [164], DroidMiner [540], DroidMOSS [584], DroidRanger [590], DroidSIFT [558], FUSE [402], HunterDroid [556], Isohara_ [262], Jiao_ [272], Juxtapp [241], Kim_ [290], MassVet [137], Mobile-Sandbox [471], MobSafe [534], Moonsamy_ [351], Moonsamy2_ [352], PaddyFrog [523], Relda [234], Riskranker [231], WuKong [505], Zhou_ [583]	14%
		Malware Collections	A5 [502], Adagio [216], Adebayo_ [50], Afonso_ [51], Amandroid [514], AMDetector [569], Ananas [173], ApkCombiner [307], APKLancet [545], ApkRiskAnalyzer [142], AppAudit [526], AppContext [546], AppIntent [548], Apposcopy [203], AppsPlayground [398], AsDroid [260], Aurasium [535], Bae_ [76], Bianchi_ [102], BlueSeal [251], Brave [377], Brox [335], Cen_ [130], Chabada [228], Chen2_ [135], CopperDroid [403], CopperDroid2 [485], COVERT [80], Dagger [541], Dai_ [157], Dendroid [474], DRACO [100], DroidADDMiner [316], DroidAlarm [579], DroidAnalytics [576], DroidAPIMiner [45], DroidDolphin [525], DroidGuard [81], DroidKin [226], DroidLegacy [164], DroidLogger [158], DroidMat [522], DroidMiner [540], DroidPAD [331], DroidPermissionMiner [68], DroidRisk [513], DroidScope [539], Droidsearch [395], DroidSIFT [558], DroidSim [479], DroidTrace [578], EASEAndroid [508], Elish_ [177], Flowdroid [67], FUSE [402], Gates_ [218], GroddDroid [47], Ham_ [236], Han_ [239], IccTA [308], IIF [551], IREA [297], Jeong_ [270], Jiao_ [272], Juxtapp [241], Kadir_ [282], Karami_ [284], Kate_ [285], Kim_ [290], Lee_ [302], Ma_ [336], Mama [430], Manilyzer [197], Marvin [319], MAST [132], MIGDroid [254], Mobile-Sandbox [471], MpDroid [488], Mudflow [70], OpSeq [54], Patronus [478], Pegasus [138], Peiravian_ [380], Peng_ [381], PREC [248], PUMA2 [243], PuppetDroid [221], Quan_ [392], RAMSES [172], ResDroid [456], SAAF [250], Sanz_ [432], Sarma_ [433], SherlockDroid [61], StaDynA [571], TMSVM [529], UID [179], VetDroid [566], Wang_ [510], Xmandroid [114]	30%
		Benchmark	AAPL [327], Amandroid [514], Anadroid [317], ApkCombiner [307], APKLancet [545], AppAudit [526], BayesDroid [493], CoChecker [155], DidFail [294], DroidBarrier [56], DroidGuard [81], DroidPAD [331], DroidSafe [227], DroidScope [539], Enck_ [182], FireDroid [414], Flowdroid [67], FUSE [402], HornDroid [122], IccTA [308], IFT [185], Kynoid [443], MorphDroid [204], MOSES [573], NDroid [389], QUIRE [167], Smith_ [467], WeChecker [154]	8%
	Case Study		Achara_ [49], AdRisk [232], AdSplit [459], Amandroid [514], Androguard [166], Apex [362], APKLancet [545], AppCaulk [446], AppCracker [121], AppGuard [75], AppIntent [548], AppsPlayground [398], Aquifer [359], AsDroid [260], ASM [247], ASV [257], Aurasium [535], AuthDroid [506], Bagheri_ [78], Bartsch_ [95], Batyuk_ [96], BlueSeal [251], Buhov_ [119], Buzzer [127], Canfora3_ [126], ComDroid [143], Compac [512], ContentScope [587], COVERT [80], COVERT_Tool [417], CredMiner [591], CRePE [147], Dagger [541], Defensor [375], Desnos_ [165], DexDiff [350], DidFail [294], DNADroid [151], DroidAlarm [579], DroidChecker [133], DroidCIA [140], DroidGuard [81], DroidRanger [590], DroidScope [539], DroidTrace [578], Enck_ [182], FineDroid [565], FlaskDroid [117], Flowdroid [67], FUSE [402], Gallo_ [214], Graa_ [230], Graa2_ [229], Harehunter [46], HornDroid [122], IIF [551], Jeong_ [270], Jia_ [271], Juxtapp [241], Kadir_ [282], KLD [453], LayerCake [408], Li2_ [313], Lintent [118], Lu_ [329], Mann_ [341], Morbs [509], NDroid [389], PCLeaks [309], Pegasus [138], PICARD [169], Poeplau_ [384], PuppetDroid [221], Riskmon [275], RiskMon2 [276], Riskranker [231], SCanDroid [211], Scoria [497], SecUP [533], SEFA [524], SFG [59], Shebaro_ [457], Shen_ [461], SmartDroid [574], Smith_ [467], Song_ [468], STAMBA [109], SUPOR [259], TaintDroid [181], Tchakounte_ [486], TreeDroid [159], VetDroid [566], Woodpecker [233]	28%
	User Study (D)		AppProfiler [411], Grab'nRun [187]^{D}, IFT [185]^{D}, MalloDroid [186], Riskmon [275], RiskMon2 [276], WifiLeaks [48], Wijesekera_ [518]	2%

D: App Developer



(a) Distribution of surveyed research based on the number of apps used in their experiments.



(b) Distribution of app repositories used in the empirical evaluations.

Figure 2.9: Distribution of surveyed papers based on the number of source of the apps used for empirical evaluation.

2.5.3.1 Evaluation Method

Table 2.9 depicts the share of different evaluation methods in assessing the quality of Android security analysis approaches. Most of the approaches have used empirical techniques to assess the validity of their ideas using a full implementation of their approach (e.g., Chabada [228], CHEX [328], Epicc [371], and COVERT [80]). Some research efforts (28%) have developed a proof-of-concept prototype to perform limited scale case studies (e.g., SCanDroid [211] and SmartDroid [574]). A limited number (2%) of approaches (e.g., Chaudhuri et al. [134]) have provided mathematical proofs to validate their ideas.

Availability of various Android app repositories, such as the Google Play Store [15], is a key enabling factor for the large-scale empirical evaluation witnessed in the Android security research. Figure 2.9a 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 (72%) have been conducted over sets of more than one hundred apps.

Figure 2.9b depicts the distribution of app repositories used in the evaluations of surveyed research. We observe that the Google Play Store, the official and largest repository of Android applications, is the most popular app repository, used by 85% of the papers with

Table 2.10: Assessment Specific Categorization of the Reviewed Research, Part 2

Dimension		Approaches	%
Replicability	Available Tool	A5 [502], Adagio [216], Amandroid [514], Androguard [166], AndroTotal [337], Andrubis [320], ApkCombiner [307], Apparecium [489], AppContext [546], AppGuard [75], AppProfiler [411], Aquifer [359], ASM [247], Aurasium [535], AutoCog [391], Barros_ [91], Brahmastra [101], Chabada [228], ComDroid [143], CopperDroid [403], CopperDroid2 [485], COVERT [80], COVERT_Tool [417], Dendroid [474], Desnos_ [165], DidFail [294], DroidForce [396], DroidSafe [227], DroidScope [539], Enck_ [182], Epicc [371], FlaskDroid [117], Flowdroid [67], FUSE [402], Geneiatakis_ [219], Grab'nRun [187], IccTA [308], IFT [185], Kirin [183], LayerCake [408], Lintent [118], MalloDroid [186], Marvin [319], Mobile-Sandbox [471], MockDroid [98], Morbs [509], Mudflow [70], NoInjection [274], PermCheckTool [501], PScout [69], SCanDroid [211], StaDynA [571], Stowaway [198], TaintDroid [181], TraceDroid [496], Wognsen_ [519]	17%
	Available Source Code	A5 [502], Adagio [216], Amandroid [514], Androguard [166], ApkCombiner [307], Apparecium [489], AppContext [546], Aquifer [359], ASM [247], Barros_ [91], Desnos_ [165], DidFail [294], DroidSafe [227], DroidScope [539], Enck_ [182], FlaskDroid [117], Flowdroid [67], Geneiatakis_ [219], Grab'nRun [187], IccTA [308], Kirin [183], LayerCake [408], Lintent [118], MalloDroid [186], MockDroid [98], Morbs [509], NoInjection [274], PermCheckTool [501], PScout [69], SCanDroid [211], StaDynA [571], TaintDroid [181], Wognsen_ [519]	10%

an empirical evaluation. There are several other third-party repositories, such as F-Droid open source repository [13], used by 24% of the evaluation methods. A number of malware repositories (such as [586, 21, 20, 64, 576]) are also widely used in assessing approaches designed for detecting malicious apps (45%). Finally, about 14% of the evaluations use hand-crafted benchmark suites, such as [10, 16], 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.

Finally, a few papers (2%) assess their proposed approach by conducting controlled experiments on a set of users, either app developers (e.g., measuring development overhead in [185]), or app consumers (e.g., studying user reactions in [518]).

2.5.3.2 Replicability

The evaluation of security research is generally known to be difficult. Making the results of experiments reproducible is even more difficult. Table 2.10 shows the availability of the executable artifacts, as well as the corresponding source code and documentations in the surveyed papers. According to Table 2.10, overall only 17% of published research have made

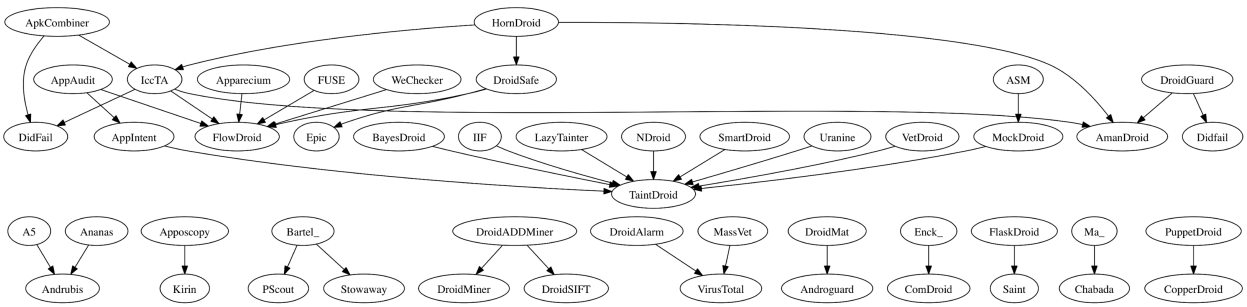


Figure 2.10: Comparison Graph: X $\rightarrow$ Y means research method X has quantitatively compared itself to method Y.

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 2.10 depicts the comparison relationships found in the evaluation of the studied papers. In this graph, 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, Enck et al. [181] provided a stable, well-documented monitoring tool, *TaintDroid*, which is widely used in the literature as the state-of-the-art dynamic analysis for evaluating the effectiveness of the newly proposed techniques.

Similarly, making a research tool available, particularly in the form of source code, enables other researchers to expand the tool and build more advanced techniques on top of it. Figure 2.11 illustrates the dependency relationships found in the implementation of the surveyed papers. In this graph, the nodes with higher fan-in represent the tools that are widely used to realize the other research efforts. For instance, *FlowDroid* [67], with 6 incoming edges, has an active community of developers and a discussion group—and is widely used in several research papers surveyed in our study.

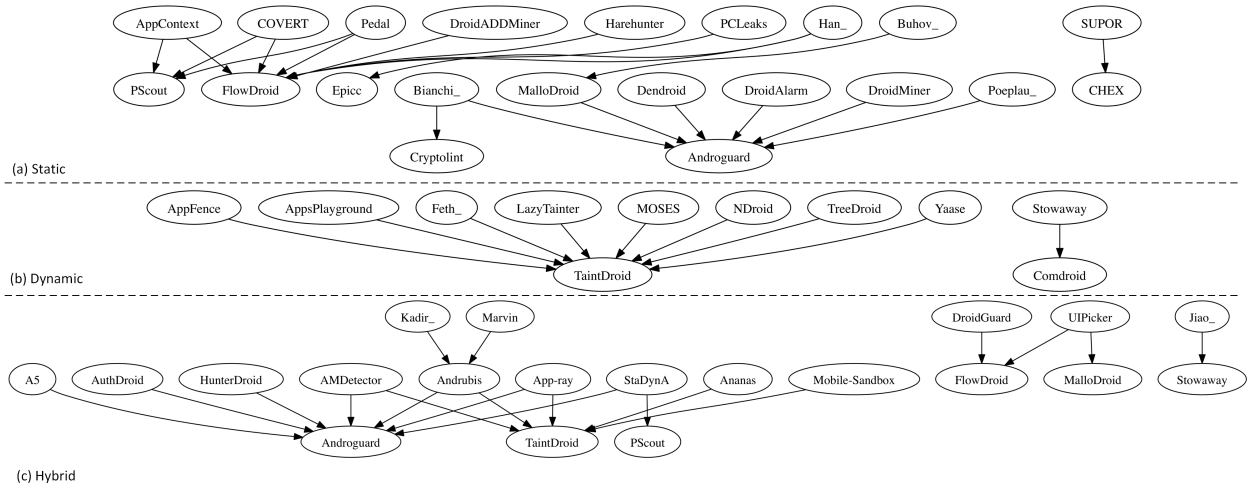


Figure 2.11: Dependency Graph: X $\rightarrow$ Y means research method X is built on top of method Y.

2.5.4 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 types of program analysis have been used for each security assessment objectives?

CQ2. What types 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 methods are used for different objectives and types of analysis?

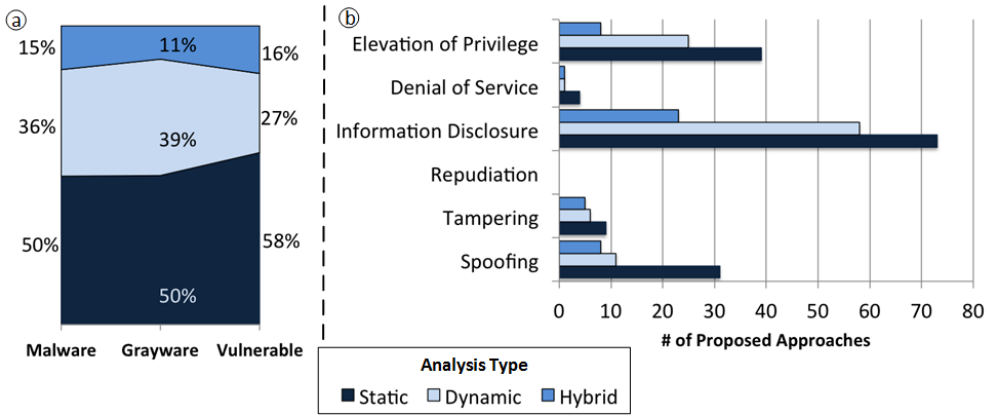


Figure 2.12: Types of program analysis that have been used for detecting ⓐ different security assessment objectives (i.e.,malware vs. vulnerability detection), and ⓑ different security threats.

CQ6. How reproducible are the surveyed research based on the objectives and types of analysis?

CQ7. Is there a relationship between the availability of research artifacts and their respective citation numbers?

CQ1. Analysis objectives and types of program analysis. As shown in Figure 2.12ⓐ, static and dynamic analyses have been used for identifying both malicious behavior and vulnerabilities. However, static approaches are more frequently leveraged for detecting vulnerable apps rather than malware (58% vs. 50%), while dynamic techniques have more application in malware detection compared to vulnerability analysis (36% vs. 27%). Hybrid approaches, though at lower scales, have also been used (15%-16%) for both purposes.

CQ2. STRIDE's security threats and type of program analysis. According to Figure 2.12ⓑ, none of the analysis types (i.e., static, dynamic, hybrid) are intended to identify the repudiation security thread (see discussion and gap analysis in Section 2.6). Moreover, according to this figure, a limited number of research efforts have been devoted to identifying Denial of Service (Dos) attacks. Finally, the cross analysis results show that

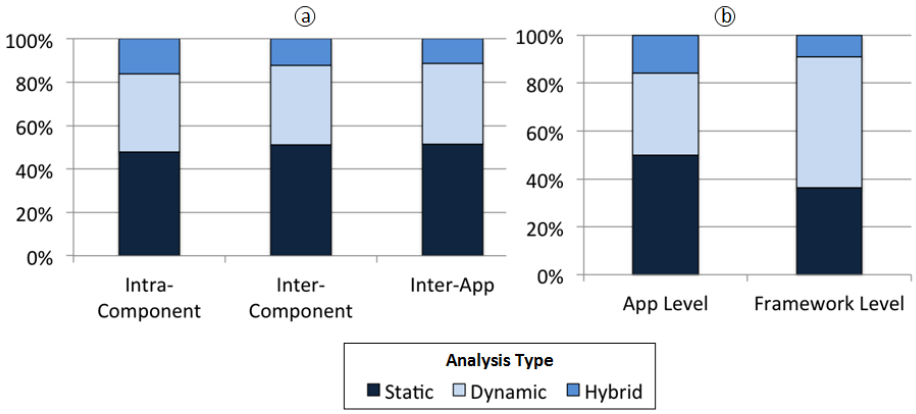


Figure 2.13: ⓐ Granularity and ⓑ Depth (Level) of each type of program analysis.

static approaches, compared to the other types of program analysis, has been widely used for detecting various security threats, particularly spoofing, where the number of static techniques is almost three times higher than the number of dynamic or hybrid approaches. As discussed before, one reason is that for security analysis soundness is usually considered to be more important than precision, since it is preferred to not miss any security threat, even at the cost of generating false warnings.

CQ3. Granularity of security threats and type of analysis techniques. In Figure 2.13ⓐ, we observe a similar distribution pattern in use of different analysis types (i.e.,static, dynamic, hybrid) for capturing security threats at different levels of granularity (i.e, intra-component, inter-component, inter-app). In general, for identifying security threats in a single component, or between multiple component in a single or multiple apps, static analysis techniques are the most common methods (about 50%) used by the state-of-the-art approaches, followed by dynamic analysis (35%), and hybrid techniques (15%).

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 2.13ⓑ). We observe that the dynamic approaches are employed more often for analysis at the framework-level (55%). One reason is that dynamic approaches can employ runtime modules, such as monitors, which are deployed in the Android framework, thereby enabling

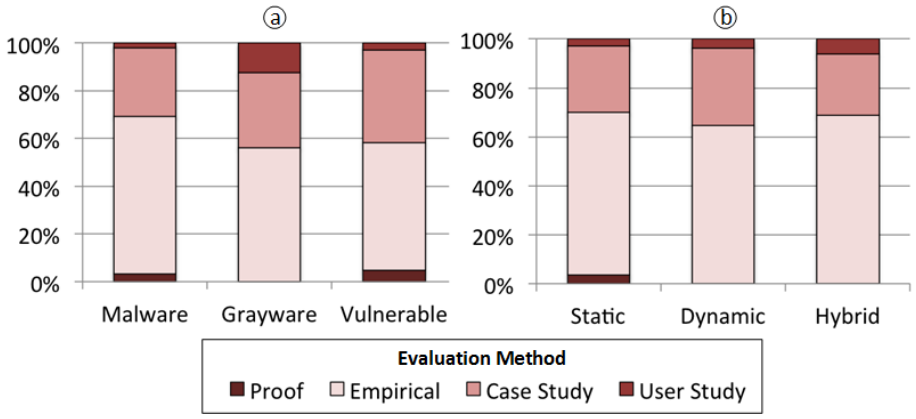


Figure 2.14: Approach validation versus ⓐ research objectives and ⓑ types of analysis.

tracking otherwise implicit relations between system API calls and the Android permissions. Such *runtime* framework-level activity monitoring is not possible using static analysis techniques. Moreover, due to the large size of Android framework (over ten million lines of code), dynamic techniques are more scalable and less-expensive for framework-level monitoring.

CQ5. Evaluation method vs. the objectives and types of analysis. We observe a similar distribution pattern in use of different evaluation methods across various types of analysis and also analysis objectives, except that user study is more popular in grayware analysis, compared to the other objectives. One reason is that the privacy concerns of end users are critical in assessing grayware, such as ad libraries. In general, empirical evaluation is the most widely used, followed by the case study and user study methods and formal proofs (See Figure 2.14).

CQ6. Reproducibility vs. the objectives and types of analysis. As shown in Figure 2.15, the research artifacts intended to identify security vulnerabilities are more likely to be available in comparison to those designed for malware/grayware detection (27% vs. 20%/13%). Moreover, availability ratio of the tools performing different types of analysis (i.e., static, dynamic, and hybrid) are all close and under 20%, which restricts the other researchers from reproducing, and potentially adopting, achievements in this thrust of research.

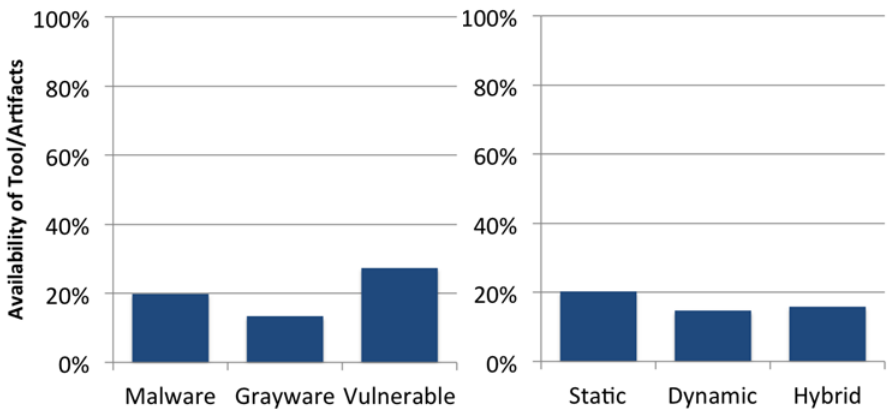


Figure 2.15: Availability of tools/artifacts based on the ⓐ objective and, ⓑ type of analysis.

CQ7. Artifact availability and citation count. To investigate this research question, we ranked the surveyed papers based on their citation counts. Since older papers have a higher chance of getting more citations, we also provided the same ranking for each year, separately, from 2009 to 2015. Afterwards, we checked the artifact availability of highly cited papers. The summary of our findings are provided in Table 2.11, which indicates that papers with publicly available artifacts get more citations. 100% of overall top-5 cited, and 88% of top-cited papers of each year, have available artifacts.

Table 2.11: Artifact availability of highly cited research papers.

Rank	Year	Tool	# of Citations*	Availability
Top cited papers - overall				
1	2010	TaintDroid[181]	1563	✓
2	2011	Stowaway[198]	745	✓
3	2009	Kirin[183]	625	✓
4	2011	Enck_[182]	599	✓
5	2011	ComDroid[143]	502	✓
Top cited papers - yearly				
1	2009	Kirin[183]	625	✓
1	2010	TaintDroid[181]	1563	✓
1	2011	Stowaway[198]	745	✓
1	2012	DroidRanger[590]	429	✗
1	2013	AppsPlayground[398]	129	✓
1	2014	Flowdroid[67]	225	✓
1	2015	IccTA[308]	21	✓

* By the end of 2015

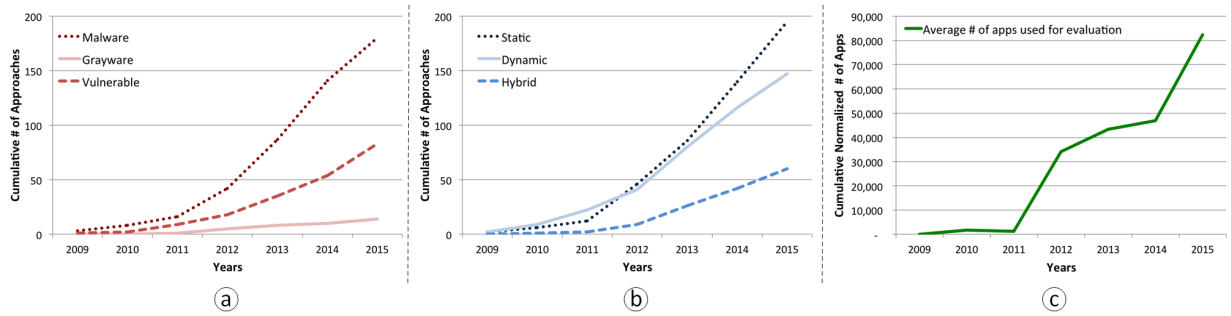


Figure 2.16: Observed trends in Android security analysis research with respect to ⓐ objectives of the analysis, ⓑ type of analysis, and ⓒ number of apps used in the evaluation (normalized by dividing the number of apps to the number of publications in each year).

2.6 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 (See Section 2.5), 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 2.16 shows some observed trends in Android security analysis research.

- According to Figure 2.16ⓐ, 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 2.16ⓑ, 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 (See Figure 2.16©). 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 [407]. 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 (RGs) in need of further study. To summarize, future research needs to focus on the following to stay ahead of today's advancing security threats:

RG1: *Pursue integrated and hybrid approaches that span not only static and dynamic analyses, but also other supplementary analysis techniques:* Recall from Table 2.6 that only 29% of approaches leverage supplementary techniques, which are shown to be effective in identifying modern malicious behaviors or security vulnerabilities.

RG2: *Move beyond fuzzing for security test input generation:* According to Section 2.5.2.8, only 8% of test input generation techniques use a systematic technique (i.e., symbolic execution or heuristic-based fuzzing), 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.

RG3: *Continue the paradigm shift from basic single app analysis to overall system monitoring, and exploring compositional vulnerabilities:* Recall from Sections 2.5.1.3 and 2.5.1.4, and Table 2.4, that the majority of the existing body of research is limited to the analysis of single apps in isolation. However, malware exploiting vulnerabilities of multiple benign apps in tandem on the market are increasing. Furthermore, identifying some security vulnerabilities requires a holistic analysis of the Android framework. For example, consider the analysis of the Android permission protocol to check whether it satisfies the security requirement of preventing unauthorized access [78]. Ensuring that the system achieves such security goals, however, is a challenging task, inasmuch as it can be difficult to predict all the ways in which a malicious application may attempt to misuse the system. Identifying such attacks, indeed, requires system-wide reasoning, and cannot be easily achieved by analysis of individual parts of the system in isolation.

RG4: *Construct techniques capable of analyzing ICC beyond Intents:* Only 3% of papers, as shown in Table 2.4, 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.

RG5: *Consider dynamically loaded code that is not bundled with installed packages:* Recall from Table 2.4 that a highly limited amount of research (4%) analyzes the security implications of externally loaded code. This Android capability can be easily exploited by malware developers to evade security inspections at installation time.

RG6: *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 2.5.1.5 and Table 2.4 that only $5-6\%$ of surveyed approaches consider obfuscated or native code, where most of those approaches do not perform analysis on the content of such code.

RG7: *Improve the precision of analysis:* Recall from Section 2.5.2.5 and Table 2.7 that a low percentage ($3-23\%$) of static analysis techniques use high precision sensitivities, leading to high false positives. Moreover, in parallel to enhancing precision, a practical analysis is also needed to scale up to large and complicated apps.

RG8: *Consider studying Android repudiation:* The SLR process returned no results for Android repudiation, as shown in Table 2.3. Consequently, there is a need for studies that target such threats, particularly in terms of potential weaknesses in the way Android app ecosystem handles digital signatures and certificates. However, repudiation also has a major legal component [206], which may require expertise not held by researchers in software security, software engineering, or computer science. Properly addressing this gap may require inter-disciplinary research.

RG9: *Promote collaboration in the research community:* To that end, we recommend making research results more reproducible. This goal can be achieved through increased sharing of research artifacts. Recall from Table 2.10 that less than 20% of surveyed papers have made their research artifacts available publicly. At the same time, Figure 2.10 shows that few approaches conduct quantitative comparisons, mainly due to unavailability of prior research artifacts. Papers that make their artifacts available publicly are able to make a bigger impact, as measured by the citation count (recall Table 2.11). We hope this will provide another impetus for the research community to publicly share their tools and artifacts. To further aid in achieving reproducibility, we also advocate the development of common evaluation platforms and benchmarks. Recall from Figure 2.9b that only 14% of studied approaches con-

sidered benchmarks for their evaluation. A benchmark of apps with known set of issues allows the research community to compare strengths and weaknesses of their techniques using the same dataset, thus fostering progress in this area of research.

2.7 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 chapter 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 336 papers published from 2008 to the beginning of 2016. Based on the results of our literature review, it is evident that Android security has received much 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 and malicious apps. 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 combinations 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.

Chapter 3

Research Problem

Permissions are the cornerstone for the Android security model, as they enable secure access to sensitive resources as well as cross-application interactions. Google provides recommendations and best practices on the correct use of permission model, and relies on app developers to properly apply them in their products. However, prior research [143, 160, 202] has shown that many app developers fail to follow such principles in practice. Misuse of Android permissions could disrupt the functional (e.g., crashing) and non-functional (e.g., security breach) behavior of apps. Due to the lack of automated tools for detecting such issues, many of those defects are shipped with the final product, which not only dissatisfies end users but also poses security risks to their phones. In this context,

> The goal of my research is to provide a set of automated tools for detection and prevention of permission-induced issues in Android applications, namely (I) permission-induced security attacks, and (II) permission-induced compatibility defects.

These two problems are among the gaps in the current state of the research, identified through a comprehensive study and literature survey (§ Chapter 2).

3.1 Permission-Induced Security Attacks

While access to phone resources can be controlled by the Android permission system, enforcing permissions is not sufficient to prevent security violations, as permissions may be mismanaged, intentionally or unintentionally. Android's enforcement of permissions is at the level of individual apps, allowing multiple malicious apps to collude and combine their permissions or to trick vulnerable apps to perform actions on their behalf that are beyond their individual privileges. Despite significant progress in Android security research, prior approaches are substantially intended to detect and mitigate vulnerabilities in a single app, but fail to identify vulnerabilities that arise due to the interaction of *multiple* apps.

> **Hypothesis 1:** A scalable and formal approach for analysis of Android apps can be developed to identify permission-induced inter-app attacks.

Moreover, by ignoring the *temporal* aspects of an attack during the analysis and enforcement, the state-of-the-art approaches aimed at protecting the users against permission-induced attacks are prone to have low-coverage in detection and high-disruption in prevention of permission-induced attacks. Finally, the proposed approaches are mostly realized through modification of either the Android framework or the implementation logic of apps, resulting in all sorts of undesirable side effects, such as app crashes and unexpected behaviors.

> **Hypothesis 2:** By incorporating the notion of time in both detection and prevention of permission-induced attacks, it is possible to provide an effective, yet non-disruptive defense against permission-induced attacks, which is highly reliable and compatible with the majority of Android apps available on the marketplace.

3.2 Permission-Induced Compatibility Defects

An effective approach that can be used in this research to address the first group of problems, i.e., security breaches, is to leverage dynamic permission mechanism, which is recently introduced in Android and allows revocation of permissions after the installation of an app. This countermeasure, i.e, revoking unsafe permissions, however, could itself result in other sorts of defects, such as crash, if the target app suffers from dynamic-permission-compatibility issues—any unexpected behavior occurs due to improper handling of dynamic permissions, after migrating old apps to Android 6 (or above). To verify the compatibility of an app under dynamic permission model, developers should test it under a wide range of permission combinations, since app's behavior may change depending on the granted permissions. At the state-of-the-art, in the absence of any automated tool support, a developer needs to either manually determine the interaction of tests and app permissions, or exhaustively re-execute tests for all possible permission combinations, thereby increasing the time and resources required to test apps.

> **Hypothesis 3:** An efficient approach for permission-aware testing of Android apps can be developed that achieves comparable code coverage and fault detection ability as the exhaustive testing.

Chapter 4

Compositional Analysis of Permission-Induced Security Vulnerabilities in Android

As discussed in Section 2.6 of Chapter 2, moving beyond single app analysis and exploring compositional vulnerabilities is among the gaps in the current state of the research. This chapter attempts to propose an approach to address this research gap.

4.1 Introduction

Mobile app markets are creating a fundamental paradigm shift in the way software is delivered to the end users. The benefits of this software supply model are plenty, including the ability to rapidly and effectively acquire, introduce, maintain, and enhance software used by the consumers. By providing a medium for reaching a large consumer market at a nominal cost, app markets have leveled the software development industry, allowing small

entrepreneurs to compete with prominent software development companies. Application frameworks are the key enablers of these markets. An application framework, such as the one provided by Android, ensures apps developed by a wide variety of suppliers can interoperate and coexist together in a single system (e.g., a phone) as long as they conform to the rules and constraints imposed by the framework.

This paradigm shift, however, has given rise to a new set of security challenges. In parallel with the emergence of app markets, we are witnessing an increase in the security threats targeted at mobile platforms. This is nowhere more evident than in the Android market (i.e., Google Play), where many cases of apps infected with malwares and spywares have been reported [449]. Numerous culprits are at 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 for those caught provisioning applications with vulnerabilities or malicious capabilities.

In this context, Android's security has been a thriving subject of research in the past few years. Leveraging program analysis techniques, these research efforts have investigated weaknesses from various perspectives, including detection of information leaks [143, 181, 252], analysis of the least-privilege principle [198, 183], and enhancements to Android protection mechanisms [207, 115, 167]. The majority of these approaches, however, are subject to a common limitation: they are intended to detect and mitigate vulnerabilities in a single app, but fail to identify vulnerabilities that arise due to the interaction of multiple apps. Vulnerabilities due to the interaction of multiple apps, such as collusion attacks and privilege escalation chaining [198], cannot be detected by techniques that analyze a single app in isolation. Thus, security analysis techniques in such domains need to become compositional in nature.

This chapter contributes a novel approach, called Covert, for compositional analysis of Android inter-app permission leakage vulnerabilities. Unlike all prior techniques that focus

on assessing the security of an individual app in isolation, our approach has the potential to greatly increase the scope of application analysis by inferring the security properties from individual apps and checking them as a whole by means of formal analysis. This, in turn, enables reasoning about the overall security posture of a system (e.g., a phone device) in terms of the security properties inferred from the individual apps.

Covert combines static analysis with formal methods. At the heart of our approach is a modular static analysis technique for Android apps, designed to enable incremental and automated checking of apps as they are installed, removed, or updated on an Android device. Through static analysis of each app, our approach extracts essential information and captures them in an analyzable formal specification language. These formal specifications are intentionally at the architectural level to ensure the technique remains scalable, yet represent the true behavior of the implemented software, as they are automatically extracted from the installation artifacts. The set of models extracted in this way are then checked as a whole for vulnerabilities that occur due to the interaction of apps comprising a system. Covert uses *Alloy* as a specification language [265], and the *Alloy Analyzer* as the analysis engine. Alloy is a formal specification language based on first order logic, optimized for automated analysis.

Since Covert's analysis is compositional, it provides the analysts with information that is significantly more useful than what is provided by prior techniques. Our experiences with a prototype implementation of the approach and its evaluation against one of the most prominent inter-app vulnerabilities, i.e.,privilege escalation, in the context of hundreds of real-world Android apps collected from variety of repositories have been very positive. The results, among other things, corroborate its ability to find vulnerabilities in bundles of some of the most popular apps on the market.

This chapter makes the following contributions:

- *Formal model of Android framework:* We develop a formal specification representing the behavior of Android apps that is relevant for the detection of inter-app permission leakage vulnerabilities. We construct this formal specification as a reusable Alloy module to which all extracted app models conform.

- *Modular analysis:* We show how to exploit the power of our formal abstractions by building a modular model extractor that uses static analysis techniques to automatically extract formal specifications (models) of apps form their installation artifacts.

- *Implementation:* We develop a prototype implementation on top of our formal framework for compositional security analysis of Android apps.

- *Experiments*: We present results from experiments run on over 500 real-world apps, corroborating COVERT's ability in effective compositional analysis of Android inter-app permission leakage vulnerabilities in the order of minutes.

The remainder of this chapter is organized as follows. Section 4.2 motivates our research through an illustrative example. Section 4.3 provides an overview of COVERT. Sections 4.4 and 4.5 describe the details of model extraction and formal analysis, respectively. Section 4.6 presents the evaluation of the research.

4.2 Motivating Example

To motivate the research and illustrate our approach, we provide an example of a vulnerability pattern having to do with Inter-Process Communication (IPC) among Android apps. Android provides a flexible model of IPC using a type of application-level message known as *Intent* (See Section 2.1). A typical app is comprised of multiple processes (e.g., Activity, Service) that communicate using Intent messages. In addition, under certain circumstances,

```
1  public class CallerActivity extends Activity {
2    public void onCreate (Bundle savedInstanceState) {
3    ...
4    String action;
5    if(selectedMenu == 1)
6      action = "PHONE_CALL";
7    else
8      action = "PHONE_TEXT_MSG";
9    btnOK = (Button) findViewById(R.id.btnOK);
10   btnOK.setOnClickListener(new OnClickListener() {
11     public void onClick(View v) {
12     Intent intent = new Intent (action);
13     intent.setClassName ("com.phoneservice", "com.phoneservice.PhoneActivity");
14     intent.putExtra ("PHONE_NUM", "900-512-1677");
15     startActivity (intent);
16   }
17   }
18 }
```

Figure 4.1: Malicious app: sends an Intent to call a premium-rate phone number.

an app's processes could send Intent messages to another app's processes to perform actions (e.g., take picture, send text message, etc.). As an example, Figure 4.1 shows *CallerActivity* belonging to a malicious app sending an Intent message to *PhoneActivity* (Figure 4.2) belonging to a vulnerable app for placing a call to a premium-rate telephone number.

The vulnerability occurs on line 30 of Figure 4.2, where PhoneActivity initiates a system Intent of type `ACTION_CALL`, resulting in a phone call. This is a reserved Android action that requires special access permissions to the system's telephony service. Although PhoneActivity has that permission, it also needs to ensure that the sender of the original Intent message has the required permission to use the telephony service. An example of such a check is shown in *hasPermission* method of Figure 4.2, but in this particular example it does not get called (line 15 is commented) to illustrate the vulnerability. If *CallerActivity* does not have the permission to make phone calls (i.e., it is not specified in the corresponding app's manifest file), it is able to make *PhoneActivity* perform that action on its behalf. This is a privilege escalation vulnerability and has been shown to be quite common in the apps on the market [143]. It could be exploited by a malware running on the same phone to call premium-rate numbers.

```
1  public class MainActivity extends Activity  {
2    public void onCreate(Bundle savedInstanceState)  {
3    ...
4    Intent intent = new Intent (this, PhoneActivity.class);
5    startActivity (intent);
6    }
7  }
8
9  public class PhoneActivity extends Activity  {
10
11   public void onCreate(Bundle savedInstanceState)  {
12    ...
13    Intent intent = getIntent();
14    String number = intent.getStringExtra("PHONE_NUM");
15    //if (hasPermission())
16       makePhoneCall(number);
17    else
18    ...
19   }
20
21   boolean hasPermission ()  {
22     if(checkCallingPermission("android.permission.CALL_PHONE")==PackageManager.
         PERMISSION_GRANTED)
23     return true;
24   return false;
25   }
26
27   void makePhoneCall (String number) {
28   Intent callIntent = new Intent (Intent.ACTION_CALL);
29   callIntent.setData (Uri.parse(number));
30    startActivity (callIntent); // privilege escalation vulnerability
31  }
32 }
```

Figure 4.2: Vulnerable app: receives an Intent and makes a phone call.

The above example points to one of the most prominent inter-app vulnerabilities, i.e.,privilege escalation, that we take as a running example from a class of vulnerabilities that require compositional analysis to be able to detect effectively.

4.3 Approach Overview

This section overviews our approach to automatically identify such vulnerabilities that occur due to the interaction of apps comprising a system, and determine whether it is safe for a bundle of apps, requiring certain permissions and potentially interacting with each other, to be installed together. As depicted in Figure 4.3, COVERT consists of two parts: (1) *Model Extractor* that uses static code analysis techniques to elicit formal specifications (models) of

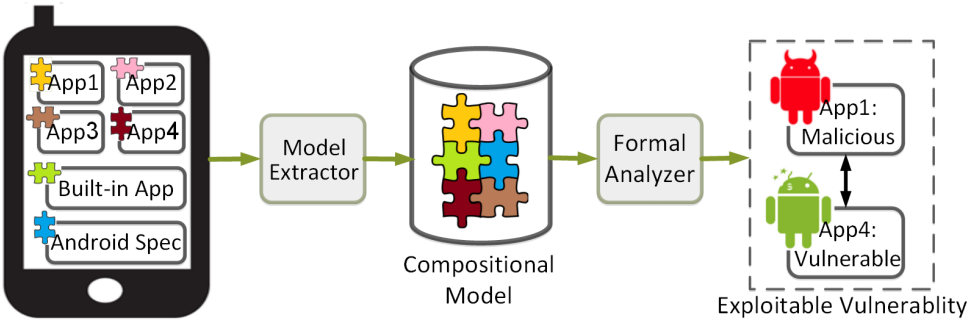


Figure 4.3: Overview of COVERT.

the apps comprising a system as well as the phone configuration; and (2) *Formal Analyzer* that is intended to use lightweight formal analysis techniques to verify certain properties (e.g., known security vulnerability patterns) in the extracted specifications.

COVERT relies on two types of models: 1) *app model* that Model Extractor generates automatically for each Android app; 2) *Android framework spec.* that defines a set of rules to lay the foundation of Android apps, how they behave (e.g., application, component, messages, etc.), and how they interact with each other. The framework specification is constructed once for a given platform (e.g., version of Android) as a reusable model to which all extracted app models must conform. It can be considered as an abstract specification of how a given platform behaves.

Model Extractor takes as input a set of Android application package archives (APK files[1]). To generate the app models, it first examines the application manifest file to determine its architectural information. Besides such high-level, architectural information collected from the manifest file, Model Extractor utilizes static analysis techniques to extract other essential information from the application bytecode. We have built a prototype implementation of the model extractor component on top of *Soot* [495] for static analysis and *Dexpler* [93] for reverse engineering Android APK files. As a result, our prototype implementation of the approach only requires the availability of Android executable files, and not the original

[1]APKs are Java bytecode packages used to distribute and install Android applications.

source code. COVERT, thus, can be used not only by developers, but also by end-users as well as third-party reviewers to assess the trustworthiness of their mobile devices.

The set of app models extracted in this way are then combined together with a formal specification of the application framework, and checked as a whole for vulnerabilities that occur due to the interaction of apps comprising a system. Finally, a report is returned to the user describing the list of detected vulnerabilities. Upon reviewing the report, end-users and third-party reviewers may choose to protect their devices in a variety of ways, e.g., by disallowing the installation of certain combination of apps, or dynamically restricting certain inter-app communications.

In this research work, we rely on *lightweight formal analysis* techniques [520] for modeling and verification purposes. Such lightweight, yet formally-precise methods, bring fully automated analysis techniques to partial models that represent the key aspects of a system [554]. The analysis is accordingly conducted by exhaustive enumeration over a bounded scope of model instances. These methods thus facilitate application of formal analyzers in development of software-intensive systems. In our prototype tool implementation, we use Alloy [265], as the specification language, and the Alloy Analyzer as the analysis engine. Alloy is a formal specification language based on first order logic, optimized for automated analysis.

Our approach can be applied in an offline setting to determine if a particular configuration for a system comprised of several apps harbors security vulnerabilities. Although not the focus of this research, we believe the approach could also be applied at runtime to continuously verify the security properties of an evolving system as new apps are installed, and old ones are updated and removed.

In the following two sections, we describe the details of static analysis used to capture essential application information and formal analysis for verification.

4.4 Model Extractor

In order to automatically analyze vulnerabilities, we first need a model of each application that would allow us to determine the potential inter-process communications and to also reason about the security properties. In our approach, an app model is composed of the information extracted from two sources: manifest file and bytecode. This section first formally defines the model we extract for each app, and then describes the extraction process.

Definition 1. A model for an Android application is a tuple $A =< C, I, F, P, S >$, where

- C is a set of components, where each component $c \in C$ has a set of Intent messages $intents(c) \subseteq I$, a set of Intent filters $ifilters(c) \subseteq F$, a set of permissions $perms(c) \subseteq P$ required to access the component c, and a set of sensitive (i.e., security relevant) paths $paths(c) \subseteq S$. Each component is defined as one of the four Android pre-defined component types: *Activity*, *Service*, *Broadcast Receiver*, and *Content Provider*.

- I is a set of event messages that can be used for both inter- and intra-app communications. Each Intent $i \in I$ has a sender component $sender(i) \in C$, may have a recipient component, and three sets of $action(i)$, $data(i)$ and $categories(i)$, specifying the general action to be performed in the recipient component, additional information about the data to be processed by the action, and the kind of component that should handle i, respectively. If the set $component(i)$ is non-empty, the Intent i is called an *explicit* Intent, as the recipient component is given explicitly.

- F is a set of Intent Filters, where each filter $ifilter \in F$ is attached to a component $c \in C$, and describes an interface (capability) of c in terms of Intents that it can handle. Each $ifilter$ has a non-empty set of $actions(ifilter)$ and two sets of $data(ifilter)$ and $categories(ifilter)$.

- P is a union of required and enforced permissions, $P = P_{Req} \cup P_{Enf}$, where P_{Req} specifies

the permissions to which the application needs to have access to run properly and P_{Enf} specifies the permissions required to access components of the application under consideration. We let the set of permissions *actually* used within a component c as $perm_{Used}(c) \subseteq P_{Req}$.

- S is a finite set of vulnerable paths; each path belongs to a component $c \in C$, and is represented as a tuple $< Entry, Destination >$, where *Entry* and *Destination* represent either permission-required APIs or IPC calls.

Algorithm 4.1: Model Extractor

```
   Input: app: Android App
   Output: A: App's Extracted Model
1  A ←< {}, {}, {}, {}, {} >
2  ICFG ← {}
3  summaries ← {}
   // ▶ Entity Extraction - See Sec. 4.4.1
4  A.C ← extractManifestComponents(app)
5  A.P ← extractManifestPermissions(app)
6  A.F ← extractManifestFilters(app)
7  IFEntities ← {}
8  foreach method ∈ app do
9  |  IFEntities ← identifyIFEntity(method, summaries)
10 end
11 resolveIFEntityAttr(IFEntities)
12 A.I ← getIntents(IFEntities)
13 A.F ← getIntentFilters(IFEntities) ∪ A.F
   // ▶ ICFG Augmentation - cf. Sec. 4.4.2
14 G ← constructICFG(app)
15 E ← extractImplicitCallBacks(app)
16 ICFG ← augmentICFG(G, E)
   // ▶ Vul. Paths Identification - cf. Sec. 4.4.3
17 A.S ← findVulPaths(A.C, ICFG)
```

As shown in Algorithm 4.1, the Model Extractor performs three major steps to obtain a model of Android app: *Entity Extraction and Resolution* (lines 4–13), *Control Flow Augmentation* (lines 14–16), and *Vulnerable Paths Identification*(line 17). In the first step, the entities are extracted from either the manifest file or the bytecode. Second, COVERT builds an inter-procedural control-flow graph augmented to account for implicit invocations. The generated inter-procedural control-flow graph is further annotated with permissions required

to enact Android API calls and Intents. Finally, in the last step, a reachability analysis is performed over the generated graph to determine the exposed components that contain unguarded execution paths reaching permission-required functionalities.

Details of each step, elaborated by Algorithms 4.2 and 4.3, are discussed in the rest of this section. To help explain the approach, Figure 4.4 illustrates the steps of applying our model extraction to the motivating example (See Section 4.2).

4.4.1 Entity Extraction and Resolution

As part of the entity extraction process, the Model Extractor first identifies the entities comprising the app by parsing and examining the app's manifest files. As shown in Algorithm4.1 (lines 4–6), it can readily obtain information such as the app's components (C) and their types, permissions that the app requires (P_{Req}), and the enforced permissions ($Perms_{Enf}$) that the other apps must have in order to interact with the app components. It also identifies some of the public interfaces exposed by each application, which are essentially entry points defined in the manifest file through *Intent Filters* (F) of components. However, not all entry points can be extracted from the manifest file, as discussed further below. Figure 4.4a shows the entities extracted at this stage of analysis corresponding to our running example from Section 4.2. Although the figure depicts the entities extracted for both apps, the reader should note that in practice Covert's program analysis runs separately on each app, the results of which are then transformed into separate formal specification modules, as detailed in Section 4.5.

After collecting these entities through examining the application manifest file, the Model Extractor identifies complementary information latent in the application bytecode. In particular, we also need to extract Intents and Intent Filters, which may be defined programmatically in the bytecode, rather than in the manifest file. Intent Filters for components of

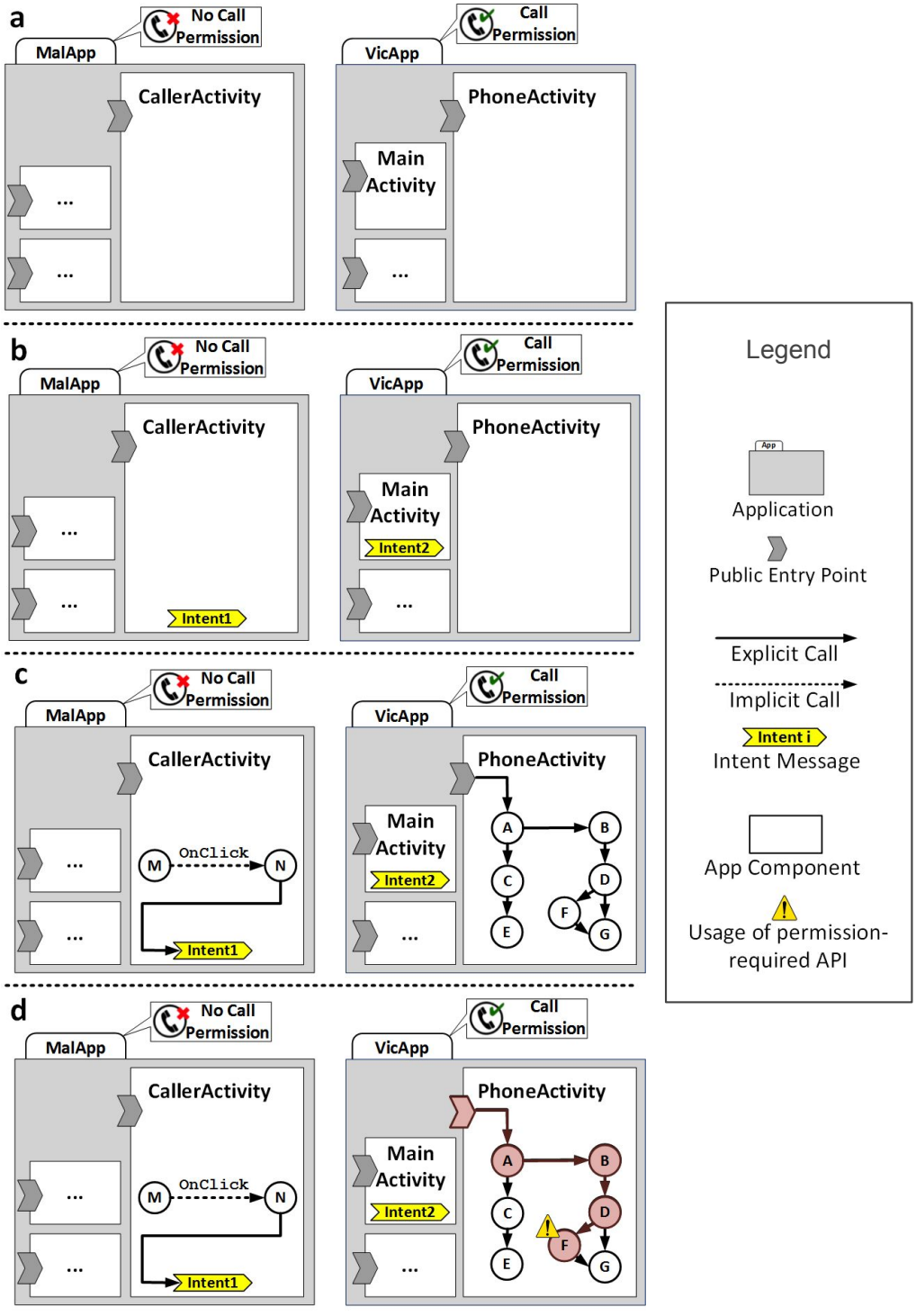


Figure 4.4: Extracted models for the apps described in Figure 4.1 and 4.2 at different steps of analysis.

type Service and Activity must be declared in their manifest, but for Broadcast Receivers, though, either in the manifest or at runtime.

For each method in an app's component, the algorithm detects and extracts Intents and Intent Filters, as shown in lines 8–10. Android API reference documentation [1] is used in this step to associate specific entities to framework-provided APIs defining or manipulating these entities. In the motivating example (Section 4.2), samples of entities are identifiable: an `Intent` entity is created in line 12 of Figure 4.1; the framework API `getIntent` is called in line 13 of Figure 4.2.

Intents and Intent Filters extracted this way need to be further analyzed to obtain additional information about their attributes. To that end, Model Extractor iterates over each method of the app and calls `identifyIFEntity`, which applies a summary-based iterative data-flow analysis [53] to detect entities and their attributes. For each Intent message, for example, it tracks the message's sender, the target component, the type of action it has (if any), data to be processed by the action, and categories of components that should handle the Intent. Note, however, that the values of attributes are resolved through an additional analysis described later in this section.

`identifyIFEntity` computes a method summary for each analyzed method. The method summary describes information about entities that can be inferred from a method. Method summaries make entity resolution inter-procedural, allowing an entire app to be analyzed. Methods are analyzed in reverse topological order with respect to the app's call graph so that a given method's summary is computed before any methods that call it are analyzed. Cycles in the call graph (e.g., from recursion) are handled in the standard manner, by treating the involved methods as one "super method."

The details of `identifyIFEntity` are shown in Algorithm 4.2. `identifyIFEntity` outputs the set *IFEntities*, which contains identified Intents and Intent Filters that are

Algorithm 4.2: identifyIFEntity

```
   Input: method, summaries
   Output: IFEntities
1  IFEntities ← {}
2  gen[entry] ← {entities passed as parameters to method}
3  workList ← {all statements of method}
4  repeat
5  |  stmt ← workList.head
6  |  foreach stmt′ ∈ pred(stmt) do
7  |  |  in[stmt] ← in[stmt] ∪ out[stmt′]
8  |  end
9  |  switch stmt.type do
10 |  |  case Intent or Intent Filter Constructors do
11 |  |  |  entity ← corresponding entity of statement
12 |  |  |  gen[s] ← {entity}
13 |  |  |  kill[s] ← set of reassigned entities
14 |  |  |  IFEntities ← {entity} ∪ IFEntities
15 |  |  end
16 |  |  case Entity Attribute Assignment do
17 |  |  |  entity ← corresponding entity of statement;
18 |  |  |  updateAttr(entity)
19 |  |  end
20 |  |  case Intent Sender or Intent Filter Registration do
21 |  |  |  entity ← corresponding entity of statement
22 |  |  |  kill[s] ← {entity}
23 |  |  end
24 |  |  case Non-Android API Method Call do
25 |  |  |  updateFromSummary(gen, kill, IFEntities, summaries)
26 |  |  end
27 |  end
28 |  prevOut ← out[stmt]
29 |  out[stmt] ← (in[stmt] \ kill[stmt) ∪ gen[stmt]
30 |  if prevOut ≠ out[stmt] then
31 |  |  workList ← workList ∪ succ(stmt)
32 until workList = ∅;
33 summarize(gen, kill, IFEntities, summaries)
```

defined and utilized in the Android app's source code. There are four types of statements that need to be considered to retrieve entity properties: (1) statements that create an entity, (2) statements that set the attributes of an entity, (3) statements that consume an entity, and (4) statements that invoke non-Android API methods.

The first type of statement, handled in lines 10–15 of Algorithm 4.2, correspond to the APIs creating an entity (e.g., through the constructors). In this case, the newly-created entity is added to the *gen* set in order to be used in the other cases; any entities that are reassigned

are added to the *kill* set to prevent further propagation of such entities; and *IFEntities* is updated with the new entity.

The second type of statement, handled by the case of lines 16–19, are the ones that set the attributes of the entity under consideration (i.e., the action, category, data, and target attributes). For example, `Intent.setClassName()` sets the target component for the given Intent.

The third type of statement, handled in lines 20–23 of Algorithm 4.2, correspond to the APIs that consume entities. Entities are consumed in different ways. An Intent, for example, is consumed when it is sent to a component: `startActivity(Intent)` launches a new Activity by sending an Intent that carries the Activity's description. An Intent Filter, however, is consumed when it is used in registering a Broadcast Receiver. Since the attributes of an entity cannot be set after consumption, the consumed entity is added to the *kill* set.

Finally, for method calls that are not part of the Android API, `identifyIFEntity` utilizes the summary of an invoked method to determine the entities and their attributes that are computed in the method (lines 24–26 of Algorithm 4.2). In particular, `identifyIFEntity` utilizes the summary of the method invoked in the program statement *stmt* under analysis to update the *gen*, *kill*, and *IFEntities* sets. For example, in line 16 of Figure 4.2, the non-Android API method `makePhoneCall` is invoked, where a new Intent is created with action and data attributes. `identifyIFEntity` utilizes the method summary for `makePhoneCall` to determine that the invocation of that method results in the creation of a new Intent with action ACTION_CALL and a data attribute. In this case, `updateFromSummary` adds this new Intent to the *gen* and *IFEntities* sets so that the new Intent is recorded and will be propagated by the data-flow analysis. The *kill* set is not modified in this case since the new Intent is not assigned to an already-defined Intent reference.

For aliasing in the case of entities and their attributes, we utilize class hierarchy analysis [53], which produces accurate results for our purposes (as shown in Section 4.6). However, our algorithm can subsitute the class hierarchy analysis for a more precise analysis (e.g., a points-to analysis), possibly trading off efficiency for precision.

The overall algorithm (line 11 of Algorithm 4.1) then calls `resolveEntityAttr` to resolve the values associated with the retrieved entity attributes (e.g., the action, categories, and data types of Intents). To do this, it uses string values obtained from string constant propagation [53], which provides a precise solution since, by convention, Android apps use constant strings to define these values. In cases where a string variable's value cannot be determined statically, we take a conservative approach and assume the value to be any string. Despite this conservative approach, our evaluation results (see Section 4.6) show our technique to be significantly precise, while remaining scalable.

It is also possible that a property is disambiguated to more than one value. For instance, consider our running example, the Intent action could be assigned to two different values at runtime, namely "`PHONE_CALL`" and "`PHONE_TEXT_MSG`" defined on lines 6 and 8 of Figure 4.1, respectively. We take a conservative approach to handle such an issue and generate a separate entity for each of these values, as they contribute different exposure surfaces or event messages in the case of Intent Filters and Intents, respectively.

Figure 4.4b shows the extracted model corresponding to our running example (recall Section 4.2) at this stage of analysis. In this particular example, Intents, as well as their properties (not depicted), are the only additional entities extracted from the bytecode. For clarity of presentation, Figure 4.4b only depicts the Intents relevant to the vulnerability in our example.

4.4.2 Control Flow Augmentation

Subsequent to extracting entities, Model Extractor needs to determine control flow between methods in order to detect vulnerabilities for privilege escalation. To that end, Model Extractor constructs an inter-procedural control-flow graph (ICFG) of the entire application. An ICFG is a collection of control-flow graphs (CFGs) connected to each other at call sites.

However, due to the event-driven structure of the Android platform, the traditional ICFG generation methods do not connect CFGs at call sites corresponding to implicit invocations. To generate an ICFG that takes implicit invocation into account, we need to include call-backs of an app. These are Android-API methods that no other part of the application explicitly invokes.

To connect the CFGs over implicit calls, we traverse the nodes of each CFG in a depth-first manner, and connect all implicit invocation nodes with the corresponding call-back nodes. For example, in lines 11–15 of Figure 4.1, an anonymous inner-class is defined within the `onCreate` method to handle the *Click* events triggered by the `btnOk` button. Thus, an edge is added to the app's ICFG from the `setOnClickListener` invocation to the entry point of `onClick`.

Figure 4.4c shows some parts of ICFGs extracted for each of the apps from Section 4.2. Here, the dashed line between nodes Ⓜ and Ⓝ indicates an implicit invocation.

4.4.3 Vulnerable Paths Identification

The last step is to determine if there is a path from each component's IPC entry point to an invocation of a permission-required functionality that is either inappropriately-guarded or unguarded, which may lead to IPC vulnerabilities. For this purpose, COVERT leverages the reachability analysis described in Algorithm 4.3.

Here, the entry nodes are IPC calls, which represent methods in a component that handle Intents generated by other components or the Android framework itself. Specifically, all app components, including Activities and Services, are required to follow pre-specified life-cycles [2] managed by the framework in an event-driven manner. Each component, thus, registers event handlers that serve as the IPC entry points through which the framework starts or activates the component once handled events occur. An Activity, for example, generates a `StartActivity` event that results in another Activity's `onCreate()` method to be called. Moreover, for each entry node, the corresponding component definition in the manifest file is also examined to ensure the component is public (line 5 of Algorithm 4.3). Recall from Section 2.1, a component is public, if its specification sets the `EXPORTED` flag or declares Intent filter(s).

The destination nodes are defined as permission-required API calls or Intent messages that are not properly checked. As shown in lines 7–11, to determine destination nodes, for each node in ICFG, `tagCheckedPerm` marks it with two tags: (1) Req_{prm} tag denotes that a statement is called at the node under consideration that requires a particular permission of "*prm*"; and (2) $Check_{prm}$ tag shows the node is guarded by permission "*prm*" checking. Thus, a vulnerable destination node is a node tagged with Req_{prm} but not with the corresponding $Check_{prm}$ tag.

To identify Req_{prm} tags, `tagCheckedPerm` uses API permission maps available in the literature, and in particular the PScout permission map [69], one of the most recently updated and comprehensive permission maps available for the Android framework. PScout specifies mappings between Android API calls/Intents and the permissions required to perform those calls. The nodes tagged as permission-required are distinguishable in Figure 4.4d by ⚠ sign. For example, node Ⓕ is a tagged node as it uses *Telephony* API that requires `CALL_PHONE` permission.

Identifying and applying $Check_{prm}$ is trickier, since permission enforcement for a compo-

Algorithm 4.3: findVulPaths

```
   Input: C: set of Components, ICFG
   Output: Vulnerable Paths
 1 Entry ← {}
 2 Dest ← {}
 3 foreach c ∈ C do
 4 |   if isPublic(c) then
 5 |   |   Entry ← Entry ∪ getEntryPoints(c)
 6 end
 7 foreach n ∈ ICFG  do
 8 |   tagCheckedPerm(n)
 9 |   if n.hasTag(Req_prm) ∧ !n.hasTag(Check_prm) then
10 |   |   Dest ← Dest ∪ n
11 end
12 return pathFinder(Entry, Dest, ICFG)
```

nent could be defined at two levels. While the coarse-grained permissions specified in the manifest file are enforced over a whole component, a developer can also add permission checks throughout the code controlling access to particular aspects of a component. The former can be readily checked using the information extracted from the manifest file (recall Section 4.4.1), but the latter requires further program analysis.

To determine permission-check API invocations that act as guards in code, `tagCheckedPerm` leverages a context-sensitive analysis (i.e., it considers the calling context of a method call) that handles the two most common cases. The first case occurs when a permission-check API is called directly. For the second case, `tagCheckedPerm` determines if a statement invokes a method that results in a call to a permission-check API (e.g., the commented permission check on line 15 of Figure 4.2). To handle aliasing in this case, `tagCheckedPerm` utilizes class hierarchy analysis, which has proven sufficiently precise for our purposes.

Once entry and destination nodes are identified, *findVulPaths* determines the paths between them (line 12 of Algorithm 4.3). To achieve high precision in determining paths between entry and destination nodes, our approach is context-sensitive. In the interest of scalability, Covert's analysis, however, is not path-sensitive (i.e., the analysis does not distinguish information obtained from different paths). The results (see Section 4.6) indicate no significant

imprecision caused by path-insensitivity in the context of Android vulnerability analysis.

Components that contain an *entry* → *destination* path, returned by *findVulPaths*, are vulnerable to various inter-app attacks. For instance, in Figure 4.4d the red-colored path of <Ⓐ,Ⓑ,Ⓓ,Ⓕ> is vulnerable, as there is a path from an entry node Ⓐ to an invocation of a permission-required API (i.e., Telephony API). As shown in Figure 4.1, a malicious app can exploit this vulnerability and call the Telephony API without having the proper privilege.

To achieve scalable, yet precise alias analysis for identifying vulnerable paths, we perform an *on-demand alias analysis*[492]. More specifically, instead of applying the analysis to all variables, only the variables utilized at statements invoking source or sink methods are considered for analysis during vulnerable-path identification.

The Model Extractor produces an extended-manifest file for each Android application. This extended-manifest, documented in an XML format, encompasses all information extracted from both the app bytecode as well as the app manifest file. Once an app model is extracted, it can then be reused for analysis within several bundles of apps. Given a set of extended-manifest files corresponding to a bundle of apps, COVERT generates a package of Alloy modules, which in turn enables their compositional analysis. The next section details the structure of generated Alloy models.

4.5 Formal Analyzer

In this section we show that our ideas for compositional, formal, and automated analysis of Android apps can be reduced to practice. Our approach automatically transforms the models derived through static analysis into an analyzable specification language, and verifies them against certain properties using the automated analyzers associated with such languages. As an enabling technology, we use the Alloy language [265], to represent a model of Android

framework, application models, and to-be-analyzed properties.

There are four main reasons that motivate our choice of Alloy for this work. First, its comprehensible, object-oriented-like syntax, backed with logical and relational operators, makes Alloy an appropriate language for declarative specification of both applications and properties to be checked (i.e., assertions). Second, its ability to automatically analyze specifications with no custom programming is useful as an automation mechanism.

Third, and more importantly, its effective module system allows us to split the overall, complicated system model among several tractable modules. A simple module system is not only convenient, but is an important part of our approach, as it enables effective compositional analysis of, among other things, impenetrable scenarios, where for example a malicious app can leverage a chain of vulnerable components to leak sensitive data or to perform actions that are beyond its individual privileges. Android apps and properties to be checked are strictly separated and modularized in different specifications, which further facilitates reusability of such specifications, and this is clearly where much of the power of our work comes from. Specifically, Android framework specification, application specifications, and specifications of vulnerabilities to be analyzed are all reusable, and this research shows the promise of paying a one-time cost to formally specify them to enable compositional analysis of Android vulnerabilities.

Lastly, the extraction approach we take in COVERT to generate bundle specifications is incremental. More specifically, the Model Extractor produces a separate extraction-output file for each Android application, independent of other apps in the bundle. The set of extracted app models are then combined together to check for inter-app vulnerabilities. Hence, once an app model is extracted, it can then be reused for analysis within several bundles of apps. That means to add, update or remove an app from the bundle, we only need to add, update, or remove information for that particular app.

To appreciate COVERT's approach, consider that an alternative approach is to detect the inter-app vulnerabilities by performing the program analysis on a whole set of apps simultaneously. But such an approach suffers from two shortcomings. First, it would face serious scalability issues, as a typical mobile device may have tens or hundreds of apps installed on it, and the analysis space grows exponentially with the number of apps to-be-analyzed. Second, it would require such a complex analysis to be performed every time any of the apps are updated, added, and removed. COVERT does not suffer from the same shortcomings because it analyzes the apps in isolation, and relies on the declarative power of formal specification languages (namely Alloy) to separate the various models needed for the analysis, thereby facilitating reuse of the models as well as the results.

In the rest of this section, we first provide a brief overview of Alloy, and then describe how we use it in modeling and thereby analysis of Android applications.

4.5.1 Alloy Overview

Alloy is a formal modeling language with a comprehensible syntax that stems from notations ubiquitous in object orientation, and semantics based on the first-order relational logic [265]. The Alloy Analyzer is a constraint solver that supports automatic analysis of models written in Alloy. The analysis process is based on a translation of Alloy specifications into a Boolean formula in conjunctive normal form (CNF), which is then analyzed using off-the-shelf SAT solvers.

The analyzer provides two types of analysis: *Simulation*, in which the analyzer demonstrates consistency of model specifications by generating a satisfying model instance; and *Model Checking*, which involves finding a counterexample—a model instance that violates a particular assertion. We use the former to compute model instances, represented as satisfying solutions to the combination of models captured from app implementations. This shows the

validity of such extracted models, confirming that the captured models are self-consistent, mutually compatible and consistent with the Android specifications modeled in a separate module. The latter is used to verify security properties of interest within the models.

The Alloy Analyzer is a bounded checker, so a certain scope of instances needs to be specified. The scope, for example, states the number of app components. The analysis is thus performed through exhaustive search for satisfying instances within the specified scopes. As a result, the analyzer is sound and complete within such scopes. To take advantage of partial models, the latest version of the analyzer uses *KodKod* [491] as its constraint solver so that it can support incremental analysis of models as they are constructed. The generated instances are then visualized in different formats such as graph, tree representation or XML.

The essential constructs of the Alloy modeling language include: *Signatures*, *Facts*, *Predicates*, *Functions* and *Assertions*. Signatures provide the vocabulary of a model by defining the basic types of elements and the relationships between them. Facts are formulas that take no arguments, and define constraints that any instance of a model must satisfy. Predicates are parameterized and reusable constraints that are always evaluated to be either true or false. Functions are parameterized expressions. A function similar to a predicate can be invoked by instantiating its parameter, but what it returns is either a true/false or a relational value instead. An assertion is a formula required to be proved. It can be used to check a certain property of a model.

4.5.2 Formal Model of Android Framework

To carry out the verification analysis, we begin by defining a common Alloy module, *androidDeclaration*, that models the Android application fundamentals (e.g., application, component, intent, etc.) and the constraints that every application must obey. Technically speaking, this module can be considered as a meta-model for Android applications. It is

manually constructed once and does not change, unless there are substantial changes in the way Android operates.

Figure 4.5 partially outlines *androidDeclaration* module, representing Android application fundamentals in Alloy. The essential element types (See Definition 1) are defined as top-level Alloy signatures. As mentioned earlier, a signature introduces a basic element type and a set of its relations, called *fields*, accompanied by the type of each field.

```
1  module androidDeclaration
2
3  abstract sig Application{
4    usesPermissions: set Permission,
5    appPermissions: set Permission
6  }
7  abstract sig Component{
8    app: one Application,
9    intentFilters: set IntentFilter,
10   permissions: set Permission,
11   paths: set Path
12 }
13 abstract sig IntentFilter{
14   actions: some Action,
15   data: set Data,
16   categories: set Category,
17 }
18 fact IntentFilterConstraints{
19   all i:IntentFilter| one i.~intentFilters
20   no i:IntentFilter| i.~intentFilters in Provider
21 }
22 abstract sig Intent{
23   sender: one Component,
24   component: lone Component,
25   action: lone Action,
26   categories: set Category,
27   data: set Data,
28 }
29 abstract sig Path{
30   entry: one Resource,
31   destination: one Resource
32 }
33 abstract sig Permission{}
```

Figure 4.5: Alloy specifications of essential Android application elements.

There are six top-level signatures to model the basic element types: `Application`, `Component`, `IntentFilter`, `Intent`, `Path`, and `Permission`. Note that these signatures are defined as *abstract*, meaning that they cannot have an instance object without explicitly extending them. Containment relations (e.g., between `Applications` and `Permissions`) are defined as Alloy relations.

According to lines 4–5, the `Application` signature contains two fields of `usesPermissions` and `appPermissions` that identify two sets of permissions, representing P_{Req} and P_{Enf}, respectively (See Definition 1).

The `app` field within the `Component` signature (line 8) identifies the parent application that a component belongs to. The keyword ***one*** states that every Component object is mapped to exactly one Application object. Signature declarations of four core component types, namely *Activity*, *Service*, *Receiver* and *Provider*, extend the `Component` signature. In the interest of space, their specifications are omitted from Figure 4.5. A component may have any number of filters, each one describing a different interface of the component. Such filters are captured by the `intentFilters` field (line 9). The `permissions` field represents a set of permissions required to access a component. The `paths` field then indicates vulnerable paths within a component.

The `IntentFilter` signature contains three fields of `actions`, `data` and `categories`. The multiplicity keyword ***some*** in Alloy denotes that the declared `actions` relation contains at least one element, and the keyword ***set*** tells Alloy that `data` and `categories` map each `IntentFilter` object to zero or more `Data` and `Category` objects, respectively.

Properties of the IntentFilter signature are declared as a *fact* paragraph (lines 18–21). The $\sim$ operator denotes the relational inverse operation, forming a new relation by reversing the order of atoms in each tuple of the relation. The statement of line 18, thus, states that each `IntentFilter` belongs to exactly one `Component`. Out of four core component types, three of them can define IntentFilters. To exclude `Content Providers` from having IntentFilters, we add a separate fact constraint specification, represented in line 20.

The `Intent` signature contains five fields of `sender`, `component`, `action`, `data` and `categories`. The first one denotes the component sending the intent. The `component` field identifies the recipient component. The keyword ***lone*** indicates that this element is

optional, and an Intent may have one or no declared recipient component. Recall from Section 4.4, if it maps to a non-empty set, the Intent object is called an *explicit Intent*. The Android intent-resolver delivers explicit Intents to the designated component, without considering other information of the Intent object.

To determine to which component an *implicit Intent*—one that does not specify any recipient component—should be delivered, three elements of `action`, `data`, and `categories` are consulted. The `action` filed names the general action to be performed in the recipient component. The `data` field indicates additional information about the data to be processed by the action, and each Data instance consists of both the URI of the data to be acted on and its MIME media type. Finally, the `categories` field indicates the kind of component that should handle the Intent object. Each of these elements corresponds to a test, in which the Intent's element is matched against that of the IntentFilter. An IntentFilter may have more actions, data, and categories than the Intent, but it cannot contain less.

We define the *entry* and *destination* fields of the `Path` signature based on canonical permission-required resources identified by Holavanalli et al. for Android applications [251]. Examples of entry and destination resources are `NETWORK, IMEI,` and `SDCARD`. Among others, the permission `NETWORK`, for example, allows the app to access the Internet, through either WIFI or cellular network. In addition to permission domains, the `IPC` mechanism augments both entry and destination sets, which allows apps to provide services to one another. Figure 4.6 shows a path identified in *VicApp* with an `IPC` as publicly accessible entry point.

Finally, the last top-level signature is `Permission`. Covert captures both the system-defined permissions—declared within the system's Android Manifest—and application-defined permissions—declared within the application manifest file, and documents them as a separate Alloy model shared between Alloy modules of all apps.

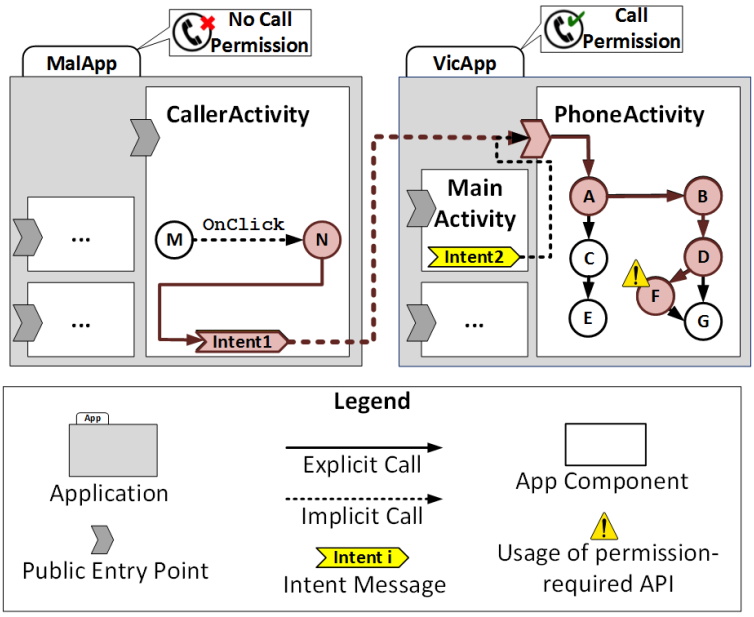


Figure 4.6: A vulnerability identified by COVERT for the apps described in figures 4.1 and 4.2. The red lines and nodes indicate the vulnerable path.

4.5.3 Formal Model of Apps

Three pieces of Alloy specifications are conjoined in the process of modeling various parts of Android apps extracted from their APK files. First, a specification module, called *appDeclaration*, that documents basic element types, such as *Action*, *Category* and *Permission*, shared between Alloy models of all apps. Second, an *app model*, comprising *Components* that constitute the app, *IntentFilters* of each *Component*, as well as required and enforced *Permissions* of the app. This model is represented in a separate Alloy module for each app. Third, an inter-process communication (*IPC*) module that models all Intent messages created within the apps under consideration. All these models rely on the Android framework specification module, presented in the previous Section.

We use snippets of the running example (See Section 4.2) to explain each piece of our formal model. Let us begin with the *appDeclaration* module.

Consider the portion of the *appDeclaration* module, shown in Figure 4.7. At the top, the

```
module appDeclaration

open androidDeclaration

one sig MAIN extends Action{}
one sig CALL_PHONE extends Permission{}
...
```

Figure 4.7: Part of the declaration of basic element types automatically extracted from Android apps.

```
1  module MalApp
2
3  open appDeclaration
4
5  one sig MalApp extends Application{}{
6    no usesPermissions
7    no appPermissions
8  }
9
10 one sig CallerActivity extends Activity{}{
11   app in MalApp
12   intentFilter = IntentFilter1
13   no permissions
14   no paths
15 }
16
17 one sig IntentFilter1 extends IntentFilter{}{
18   actions = MAIN
19   categories = LAUNCHER
20   no data
21 }
```

Figure 4.8: Part of the generated specification for Malicious app shown in Figure 4.1.

specification imports the Alloy module for the Android framework. It then declares `MAIN` to be a singleton subset of `Action`. Typically, one activity in an app is specified as the "main" activity, declaring it as the main entry point to the app, and presented to the user when launching the app. In a signature declaration, the keyword ***one*** specifies the declared signature to contain exactly one atom, thereby restricting the signature to be unique. This naming scheme allows us to reuse the term MAIN when we want to declare the main activity of each application. The next statement represents a permission example declared in a similar way. For the sake of clarity, we use the permissions' shorthand in our Alloy model. For example, here we use `CALL_PHONE` to model the particular permission of `android.permission.CALL_PHONE`.

Figure 4.8 partially delineates the generated specification for the malicious app shown in Figure 4.1. It starts by importing the *appDeclaration* module (line 3), and then the MalApp is declared as an extension of the `Application` signature. This app does not declare any permission neither as required (`usesPermissions`) nor as enforced (`appPermissions`). The MalApp has a Component of type Activity, named `CallerActivity`, which declares an IntentFilter with `MAIN` and `LAUNCHER` settings, marking it as the main activity of the app.

The code snippet of Figure 4.9 represents the generated specification for the Victim app shown in Figure 4.2. The VicApp has access to the `CALL_PHONE` permission (line 6), but declares no permission requirement for other apps to access its own Components (line 7). This app specification then declares the `PhoneActivity` component, exposing a vulnerable path (`path1`) from its entry point to a permission required resource (`PHONECALL`), as represented in Figure 4.6.

```
1  module VicApp
2
3  open appDeclaration
4
5  one sig VicApp extends Application{}{
6    usesPermissions = CALL_PHONE
7    no appPermissions
8  }
9
10 one sig PhoneActivity extends Activity{}{
11   app in VicApp
12   intentFilter = IntentFilter2
13   no permissions
14   paths = path1
15 }
16
17 one sig path1 extends Path{}{
18   entry = IPC
19   destination = PHONECALL
20 }
```

Figure 4.9: Part of the generated specification for Victim app shown in Figure 4.2.

Application interactions in Android occur through Intent messages. We record the interactions among app Components in a separate Alloy module, called *IPC*. The code snippet shown in Figure 4.10 represents part of the generated specification for the IPC module. After

importing modules of the involved apps (lines 3–4), the specification in lines 6–12 models the Intent of Figure 4.1, where the `CallerActivity` Component sends an explicit Intent (i.e., *intent1* as shown in Figure 4.6) to the `PhoneActivity` Component, with specified action to be performed and with extra data.

```
1  module IPC
2
3  open VicApp
4  open MalApp
5
6  one sig intent1 extends Intent{}{
7    sender = CallerActivity
8    component = PhoneActivity
9    action = PHONE_CALL
10   no categories
11   extraData = Yes
12 }
13 ...
```

Figure 4.10: Part of the generated inter-component communication module.

4.5.4 Checking Android Application Models

The previous sections present a formal model of Android framework (Section 4.5.2), developed as a reusable Alloy module to which extracted app models conform (Section 4.5.3). Here, we describe the essence of this work: how one can use the power of proposed formal abstractions to perform the compositional analysis of Android apps.

To that end, we develop assertions that model a set of security properties required to be checked. These assertions express properties that are expected to hold in the extracted specifications. Similar to Android specification, vulnerability assertions are manually constructed once and do not change, unless there are substantial changes in Android that resolve or modify the known types of inter-app vulnerabilities.

Considering the privilege escalation, Davi et al. [160] state it as follows: *"An application with less permissions (a non-privileged caller) is not restricted to access components of a more privileged application (a privileged callee)."* Figure 4.11 formally expresses the privilege

escalation assertion in Alloy. In short, the assertion states that the `dst` component (victim) has access to a permission (`usesPermission`) that is missing in the `src` component (malicious), and that permission is not being enforced in the source code of the victim component, nor by the application embodying the victim component. Recall from Section 4.4 that there are two ways of checking permissions in Android.

```
1 assert  privilegeEscalation{
2   no disj src, dst: Component, i:Intent|
3     (src in i.sender) &&
4     (dst in src.^transitiveIPC) &&
5     (some p: dst.app.usesPermissions |
6       not (p in src.app.usesPermissions) &&
7       not ((p in dst.permissions) ||(p in dst.app.appPermissions)))
8 }
```

Figure 4.11: privilegeEscalation specification in Alloy.

```
1 fun transitiveIPC : Component -> Component {
2   { src, dst : Component | some i:Intent, d:Path|
3       (src in i.sender) &&
4       (dst in intentResolver[i]) && some dst.paths
5   }
6 }
7
8 fun intentResolver(i:Intent): set Component{
9   {c:Component|   some i.component
10     implies {c = i.component}
11     else { some f: IntentFilter|
12         f.~intentFilter in c
13         && i.action in f.actions
14         && i.categories in f.categories
15         && (i.data.uri = f.data.uri
16         && i.data.type = f.data.type) }
17    }
18 }
```

Figure 4.12: IntentResolver and transitiveIPC specifications in Alloy.

To address a situation, in which more than two components are involved in the privilege escalation, the assertion relies on the specification of the `transitiveIPC` function, shown in Figure 4.12. The operator "^" represents transitive closure. The expression `src.^transitiveIPC` thus represents the set of all components reachable from `src` following one or more IPCs. The `transitiveIPC` itself relies on the specification of an `intentResolver` function. The *Component*, *Intent* and *IntentFilter* signatures are specified such that they have all the necessary attributes required for Intent resolution. We thus

describe intent-resolver as a function augmenting the aforementioned *androidDeclaration* module. This function takes as input an Intent and returns a set of Components that may handle the Intent under consideration. Given the Intent is explicit, it should be delivered to the recipient identified by the `component` field of the Intent (line 10). Otherwise, the resolver checks Components' IntentFilters to find those whose elements are matched against the given Intent. Specifically, an implicit Intent must pass a matching test with respect to each of the *action*, *data*, and *categories* elements on the IntentFilters bound to a component (as stated in lines 13–16). Seeing that a component can define multiple IntentFilters, an Intent that does not match one of a component's IntentFilters may match another (lines 11–12).

If an assertion does not hold, the analyzer reports it as a counterexample, along with the information useful in finding the root cause of the violation. Counterexample is a particular model instance that makes the assertion false. Given our running example, the analyzer automatically generates the following counterexample:

```
... // omitted details of model instances
privilegeEscalation_src={MalApp/CallerActivity}
privilegeEscalation_dst={VicApp/PhoneActivity}
privilegeEscalation_i={intent1}
privilegeEscalation_p={appDeclaration/CALL_PHONE}
```

It states that the `VicApp/PhoneActivity` component has access to the `CALL_PHONE` permission, and is resolved by the formal analyzer as the receiver of *intent1* (as shown by a dashed line in Figure 4.6), which is being sent by the `MalApp/CallerActivity` component lacking access to the `CALL_PHONE` permission. The generated counterexample confirms that the composition of Victim and Malicious apps could result in privilege escalation.

4.6 Empirical Evaluation

To assess the effectiveness of our approach in revealing Android inter-app vulnerabilities, we have conducted an evaluation that addresses the following research questions:

RQ1. What is the importance of this research? To what extent are Android apps overprivileged and unsafe due to usage of permission-required APIs?

RQ2. How well does COVERT perform? Does it enable compositional analysis of real-world Android apps? How much manual effort is involved in the analysis process?

RQ3. What is the overall accuracy of COVERT in detecting inter-app vulnerabilities?

RQ4. How does compositional analysis compare to single app analysis?

RQ5. What is the performance of our prototype tool implemented atop SAT solving technologies and static analyzers?

Our experimental subjects are a set of Android apps drawn from four different app repositories. The first sample set consists of a snapshot of the top 100 popular free apps, available on the Google Play [15] in late November 2013. Our second set of test subjects is representative of open source apps, and includes 300 apps collected from the F-Droid open source repository [13]. To cover the apps available in third-party repositories, we also included 50 apps from Bazaar [6], a local app store, as the third set. The fourth one is a collection of 50 malicious apps identified by the MalGenome project [586].

Figure 4.13 illustrates the distribution of apps from Google Play repository that were used in our experiments, showing that they are sufficiently diverse, from different categories, and representative of what one can find installed on a typical device. For brevity, we do not show the distribution of apps from other repositories that have different set of categories, but the apps selected from these other repositories were similarly diverse.

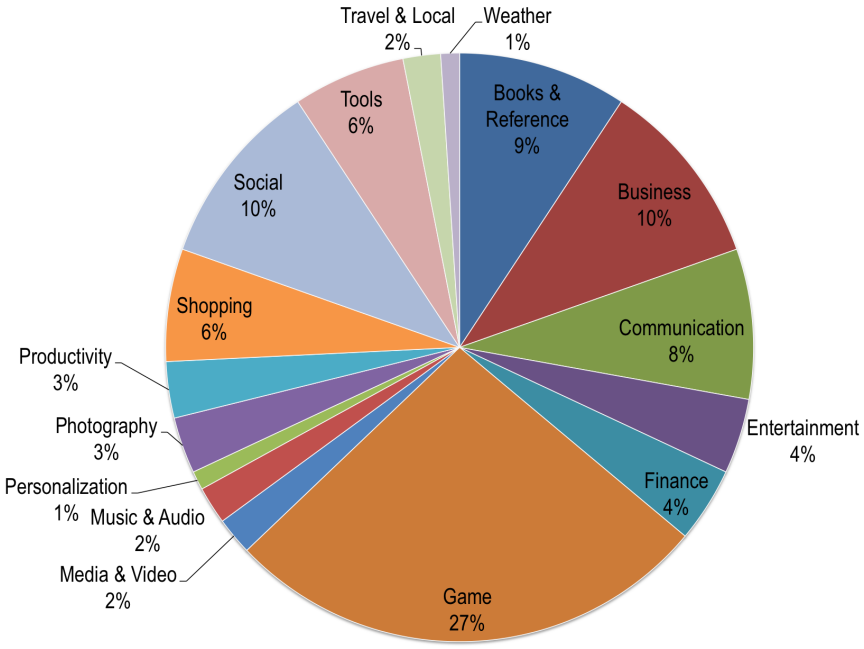


Figure 4.13: Distribution of apps selected from the Google Play repository.

To answer RQ1, we examine all of the aforementioned subject apps, to obtain some evidence as to the likelihood of encountering privilege escalation vulnerability in the apps that are available in such markets (§ 4.6.1).

To address RQ2, we partition the set of apps under study into 10 bundles, each containing 50 apps from three repositories, except the last bundle whose apps are only from the open source repository to enable manual analysis. These bundles simulate collections of apps installed on end-user devices, and we use them to conduct 10 independent experiments. We then report and analyze the experimental results (§ 4.6.2).

To evaluate the accuracy of warnings reported by Covert (RQ3), we randomly select 50 apps from the F-Droid open source apps and run our prototype tool on them. We then manually analyze each warning to detect the rate of tool error, i.e., false positive (§ 4.6.3).

To address RQ4 (single vs. compositional app analysis), we adopt a set of practical security rules, called *Kirin* rules, for Android apps from Enck et al. [183], and formally model each

of these rules in such a way that enables their applications for both "compositional" analysis as well as analysis of each "single" app in isolation. We then analyze all the apps in the Malgenome repository against these rules, and compare the results of single and compositional app analysis (§ 4.6.4).

To address RQ5 (performance benchmarks), we measure the computation time required for both model extraction and formal analysis activities (§ 4.6.5).

We use the COVERT apparatus we developed based on the approach for carrying out the experiments. COVERT is implemented as a publicly available tool[2]. We have built a prototype implementation of the model extractor component on top of the *Soot* [495] static analysis tools. Soot is developed for analyzing Java bytecode [495]. We thus first use the *Dexpler* transformer [93] to translate Android's dalvik bytecode into the Soot's intermediate representation language, called *Jimple*. As a result, our prototype implementation of the approach only requires the availability of Android executable files, and not the original source code. COVERT, thus, can be used not only by developers, but also by end-users as well as third-party reviewers. The translation of captured app models into the Alloy language is implemented using the *FreeMarker* template engine [14].

4.6.1 Significance of Compositional Analysis

Table 4.1 outlines the amount of permissions requested by apps in each repository, along with the fraction of which is actually *used* through API calls, as well as enforced—depicted as *checked* in Table 4.1—by the apps. Based on the permission map provided by Au et al. [69], we analyzed the fraction of permissions actually needed for API calls performed by the apps under consideration (See Section 4.4).The result shows that overall 32% of acquired permissions are necessary for API calls. This confirms previous studies that showed many

[2]Research artifacts and experimental data are available at https://seal.ics.uci.edu/projects/covert

Table 4.1: Summary of statistical information about Permissions in subject systems

Repository	Permissions	
	used	checked
GPlay	1472	
	364 (%24.7.1)	156 (%10.6)
F-Droid	1031	
	505 (%49.0)	77 (%7.5)
MalGenome	499	
	100 (%20.0)	5 (%1.0)
Bazaar	305	
	105 (%34.4)	16 (%5.2)

Android apps on the market are over-privileged [198, 69]. Applications having extraneous permissions violate the least privilege principle. We also analyzed what fraction of the obtained permissions are checked either within the app manifest file or throughout the code. The difference between the set of used and checked permissions are important for privilege escalation. The extraneous permissions that result in overprivilege are not susceptible to privilege escalation, unless they are actually used by the permission holders. On average, each app has about 2 unchecked but used permissions that could lead to exploitable vulnerabilities. Indeed, such an unsafe use of permission-required APIs may lead to an exploitable vulnerability provided that there is a path from the exported interface of the app component to the API use. This analysis is the subject of next section.

4.6.2 Automated Analysis of Applications

The aim of RQ2 is to evaluate the automation level when using COVERT for compositional analysis of real-world Android apps, and how much manual effort is involved in the analysis process. To that end, we evaluate COVERT on bundles of real-world Android apps to determine its ability to detect inter-app vulnerabilities for privilege escalation. Table 4.2 summarizes the statistical results obtained through running COVERT on Android app bun-

dles. The total number of components defined by the apps in each bundle is shown in the second column. Overall, `Activities`, `Services`, `Broadcast receivers`, and `Content providers` account for 73%, 11%, 12% and 2% of components, respectively.

Table 4.2: Summary of experimental results obtained from running COVERT over App bundles.

	Components				Intents		Intent	Exposed		Total
	Activities	Services	Receivers	Providers	explicit	implicit	Filters	Comps	Perms	Warnings
	691				456					
Bundle 1	511 (%73.95)	70 (%10.13)	91 (%13.17)	19 (%2.75)	300 (%65.79)	156 (%34.21)	169	5	10	34
	603				432					
Bundle 2	434 (%71.97)	76 (%12.6)	78 (%12.94)	15 (%2.49)	302 (%69.91)	130 (%30.09)	148	7	2	16
	592				312					
Bundle 3	425 (%71.79)	65 (%10.98)	85 (%14.36)	17 (%2.87)	218 (%69.87)	94 (%30.13)	185	4	3	25
	582				366					
Bundle 4	423 (%72.68)	75 (%12.89)	71 (%12.2)	13 (%2.23)	232 (%63.39)	134 (%36.61)	191	4	9	32
	695				803					
Bundle 5	569 (81.88)%	61 (8.78)%	53 (7.63)%	12 (1.73) %	408 (50.81) %	394 (49.07) %	359	2	15	16
	553				528					
Bundle 6	445 (80.48)%	52 (9.41)%	47 (8.5)%	9 (1.63) %	278 (52.66) %	249 (47.16) %	225	2	10	15
	352				577					
Bundle 7	242 (68.75)%	43 (12.22)%	62 (17.62)%	5 (1.43) %	349 (60.49) %	227 (39.35) %	137	4	16	35
	682				1209					
Bundle 8	556 (81.53)%	57 (8.36)%	62 (9.1)%	7 (1.03) %	728 (60.22) %	480 (39.71) %	175	3	5	22
	511				622					
Bundle 9	358 (70.06)%	71 (13.9)%	75 (14.68)%	7 (1.37) %	251 (40.36) %	370 (59.49) %	231	5	7	22
	660				527					
Bundle 10	496 (%75.15)	67 (%10.15)	68 (%10.3)	29 (%4.39)	347 (%65.84)	180 (%34.16)	132	5	9	30

The *Intents* column delineates the fraction of implicit/explicit Intents out of total Intents in each bundle; on average, about 40% of Intents are implicit, showing that developers, by and large, make inter-component communications explicit. This is promising as there is no guarantee that the implicit Intent will be received by the intended recipient. The next column represents the number of components' interfaces described in terms of Intent filters.

The *Exposed* column shows the number of component surfaces and permissions unsafely exposed to other applications. On average, COVERT detects 5 exposed components in each Bundle. Such components have defined Intent filters that make the components accept

incoming Intents, but do not properly enforce access permission, neither in the manifest file nor in the source code. The last column then presents the total number of warnings generated by COVERT for applications of each bundle, and each one represents a unique combination of source and destination components that can lead to a privilege escalation.

Note that reported warnings are about potential security issues. As with other techniques relying on static analysis, our approach is subject to false positives, which could be due to two types of failures in model extraction:

- Strings are used extensively as identifiers in Android apps. Intent properties such as actions, data types, and permissions are all constructed from strings, as shown in our examples. Such strings could also be altered by stateful operations, such as the `append` method, which makes their accurate value elicitation quite challenging. In case an ambiguous value is encountered, during the entity resolution step (Section 4.4.1), COVERT takes a conservative approach, and considers all possible assignable values.
- COVERT performs reachability analysis (Section 4.4.3) to determine the permissions actually used by each component, thus ignoring permissions that are obtained, but not used. Yet, there is a possibility that at run-time the permission-required API call or System Intent is not actually reached due to some conditional statements, for example.

The conservative approach we take to deal with non-determinism thus may introduce unnecessary false positives. Encouragingly, this automated analysis still results in a substantial reduction in subsequent manual analysis. Specifically, less than 1% of application components (See Table 4.2, exposed components vs. total components) require further analysis by users. Also, the limitations of the static analysis with respect to, among other things, dynamically loaded code could lead to false negatives as well. To facilitate the process of manual analysis, COVERT provides the location of the potential vulnerability (i.e., filename and method) within the source code.

The results also confirm that an approach combining static analysis and model checking is effective in compositional analysis of Android apps. In this particular case, the reported vulnerabilities provide crucial clues to the security analyst tasked with assessing the security properties of a complex system. Such analysis is not possible with state-of-the-practice tools (e.g., Fortify) that analyze the source code of an application in isolation.

In the next section, we interpret the results through manual analysis of a bundle of open-source applications.

4.6.3 Manual Analysis

We selected 50 applications from the F-Droid open source repository, and then manually inspected Covert's warnings for these applications to evaluate how many warnings correspond to real exploitable vulnerabilities. Statistics of the selected app set are provided as `Bundle 10` in the Table 4.2. More details about the apps, including their name and model can be found on the project site[3]. In this section, we present the findings of our manual analysis and discuss three representative examples in detail.

Covert generated 30 warnings for the 50 applications. We manually reviewed all and categorized them according to the classification provided by Chin et al. [143], where each warning is classified as a vulnerability, not a vulnerability, or undetermined. We define a vulnerability to be a component lacking a particular permission getting access to a functionality requiring that permission through an interface exposed by a vulnerable component. In order to detect vulnerabilities, we reviewed the application source code of both sides (sender and destination) for each warning.

Among the 30 reported warnings, we discovered 18 definite vulnerabilities. This represents a 60% true positive rate, which is superior to the prior technique [143], that aimed to identify

[3]https://seal.ics.uci.edu/projects/covert

inter-app vulnerabilities by analyzing the source code of each app in isolation, with a true positive rate of 27.6%. More interestingly, of the 5 application components identified as exposing permissions, all contain at least 1 exploitable vulnerability.

In the rest of this section, we describe a few representative applications and the vulnerabilities we discovered in them.

Case 1: Aard Dictionary → Podax.

The first app is `Aard Dictionary`, a simple dictionary and an offline Wikipedia reader. It defines a *WebViewClient* interface for handling incoming urls, and creates and sends an implicit Intent with the `VIEW` action, should the scheme of the given url matches with one of the specified schemes, such as *http*, *https* and *ftp*.

On the other hand, the app bundle contains the `Podax` app, a podcast downloader and player application. This app accepts Intents with the `VIEW` action, and `http` scheme, which in turn can lead to message passing between the two apps. While the first app that sends the Intent does not have the `INTERNET` permission, the recipient app (`Podax`) has. In addition, the `Podax` app does not check whether the caller has the appropriate permission. This combination, thus, gives rise to a privilege escalation vulnerability.

The sender app here is benign, but if it was malicious it could use the other app's unprotected capability, which may lead to some security risks, such as phishing, by bringing up a web page and enticing the user to enter payment or other private information.

Case 2: Binaural beats therapy → Ermete SMS.

`Ermete SMS` is a free web-based text messaging application that has `WRITE_SMS` permission. An Activity component of this application exposes an unprotected interface that receives Intents with `SEND` action. Upon receiving an Intent, the `ComposeActivity` component extracts the payload of the given Intent, and sends that data via text message to a

number specified in the payload, without checking the permission of Intent sender.

The other app, `Binaural beats therapy`, is designed for relaxation, creativity and many other desirable mental states. This app does not have the `WRITE_SMS` permission, but it sends an Intent with `SEND` action and *text/plain* payload data, which could be received by the first app. This case represents a false positive as the Intent sent by the `Binaural beats therapy` app does not actually contain the fields required by `Ermete SMS` to send a text message, but points to an important security risk, where a malicious app could use the exposed messaging service.

Case 3: PurpleDock → RMaps.

`RMaps` is an on- and off-line navigation tool. In addition to GPS permissions like `ACCESS_FINE_LOCATION`, it has `INTERNET` permissions to work with online maps such as Google and Microsoft maps. This application exposes an activity, which receives `VIEW` Intents with `geo` scheme, a URI scheme for geographic locations. On the other hand, `PurpleDock` is a simple app that automatically turns on when the handset is placed into the car mount, and provides navigation as one of its features.

`RMaps`'s geo Intents are intended for internal use, and other applications, including PurpleDock that sends a geo message via Intent, should not be able to control locations shown by the app interface. However, with the current implementation, as it does not check the permission of Intent senders, the exposed component can be manipulated by a malicious application for GPS spoofing (i.e., display a wrong location).

4.6.4 Compositional vs. Single App Analysis

Enck et al. [183] provide a set of practical security rules, called Kirin rules, to prevent malwares from exploiting Android applications. Each rule represents undesirable security

```
1  // (a) single app analysis
2  assert KirinRule6{
3    no p1,p2: Permission| {some app: Application|
4    (p1 = RECEIVE_SMS) and (p2 = WRITE_SMS) and
5    (p1 in app.usesPermissions) and (p2 in app.usesPermissions)
6    }
7  }
8  ------------------------------
9  // (b) compositional app analysis
10 assert KirinRule6_Compos{
11  no disj p1,p2: Permission| {some app1,app2: Application,
12  c1,c2:Component, intent1:Intent|
13  (p1 in RECEIVE_SMS+WRITE_SMS) and
14  (p2 in RECEIVE_SMS+WRITE_SMS) and
15  (p1 in app1.usesPermissions) and (p2 in app2.usesPermissions)
16  and (isPrivilegeEscalation[c1, c2, intent1, p2]or(app1 = app2))
17  }
18 }
```

Figure 4.14: Specification of a Kirin rule for (a) single and (b) compositional app analysis.

properties in terms of the configuration available in manifest files. Kirin rules, thus, decide whether the security configuration bundled with a single app is safe or not, but they do not consider the case in which malicious apps collude to combine their permissions, allowing them to perform actions beyond their individual privileges.

To analyze these rules using our approach, we formalized them in Alloy. Each rule is modeled as an assertion to be analyzed independently. We also developed a compositional version of each rule, leveraging the privilege escalation predicate. This in turn enabled us to apply the two sets of rules and compare the results of isolated analysis versus compositional analysis.

To make the idea concrete, we illustrate one of these rules along with its formal representations for both compositional and single app analysis. Consider the following Kirin security rule (KSR 6): *"An application must not have `RECEIVE_SMS` and `WRITE_SMS` permission labels [183]."*

Figure 4.14 partially outlines the two Alloy assertions specified to check the rule against either (a) a single app or (b) a combination of apps that may collude to combine their permissions. Assertion (a) states a direct representation of the aforementioned rule in Alloy, while assertion (b) restates the same rule against multiple apps. It uses the `isPrivilegeEscalation`

Table 4.3: Compositional vs. Single App Analysis of Kirin Rules over the Malgenome app repository.

Sec. Rule	Sing. App. Analysis	Compositional Analysis
KSR 1	-	-
KSR 2	-	-
KSR 3	2	2
KSR 4	2	8
KSR 5	2	11
KSR 6	10	14
KSR 7	11	14
KSR 8	3	3
Overall	30	52

predicate (line 16) to check the occurrence of privilege escalation between the two apps with respect to the `p2` permission. The `p1` and `p2` permissions could be either `RECEIVE_SMS` or `WRITE_SMS` (lines 11–12), but they should be distinct as enforced by *disj* keyword (line 11). The predicate takes as input two components `c1` and `c2`, an Intent, and a permission. The `c1` component belongs to the `app1` and `c2` to the `app2`, omitted in Figure 4.14 (b) in the interest of space. The assertion then at the very end of line 16 checks the case in which one app contains both permission labels. Note that in practice developing two different assertions is not necessary as the latter, in effect, covers the former. Here, we developed the former for experimental purposes, and to compare the results of single versus compositional analysis.

We analyzed all the apps in the Malgenome repository against each of these rules. Table 4.3 summarizes the results. Rows represent Kirin security rules that we formally modeled in Alloy to be analyzed using our approach. Columns represent the analysis type, either single app analysis (as performed by the Kirin tool [183]) or compositional analysis. Each cell indicates the number of vulnerabilities detected. As we can see, the compositional rule analysis detects more vulnerabilities, without missing any vulnerability identified by single app analysis. The experimental results indicate the overall improvement of 73% in detecting vulnerabilities using a compositional analysis approach.

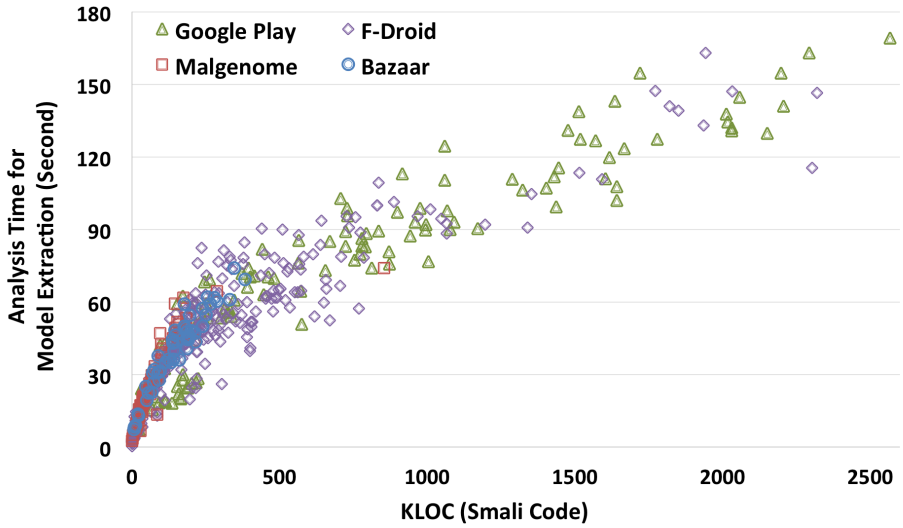


Figure 4.15: Scatter plot representing analysis time for model extraction of Android apps.

4.6.5 Performance and Timing

The final evaluation criteria are the performance benchmarks of model extraction and formal analysis activities. We used a PC with an Intel Core i7 2.4 GHz CPU processor and 8 GB of main memory, and leveraged Sat4J as the SAT solver during the experiments.

Compositional Analysis of Android apps using our approach consists of three steps: (1) The app models are collected and documented as Alloy specifications. (2) The extracted Alloy models are transformed into 3-SAT clauses using the Alloy Analyzer. (3) An off-the-shelf SAT solver explores the space to find counterexamples. We measured the computation time required for each step separately.

The scatter diagram shown in Figure 4.15 plots the time taken to analyze the collected apps for model extraction in seconds. The results show that the analysis time scales almost linearly with the size of apps in all three repositories. However, as the set of most popular apps collected from the Google Play repository—represented by dark blue in the diagram—are typically larger than apps from the other two repositories, their model extraction takes more time. According to the diagram, our approach is able to statically analyze and infer spec-

Table 4.4: Experiments performance statistics.

	Construction Time (Sec)	Analysis Time (Sec)
Bundle 1	412	252
Bundle 2	226	123
Bundle 3	441	65
Bundle 4	158	57
Bundle 5	191	239
Bundle 6	88	85
Bundle 7	120	123
Bundle 8	350	374
Bundle 9	295	299
Bundle 10	204	45

ifications for the largest apps in less than three minutes. As our implementation separates model extraction analysis from Alloy model generation, and each app bytecode is analyzed independently (See Section 4.4), the total static analysis time scales linearly with the total size of apps.

Table 4.4 shows the time involved in compositional verification of Android apps (steps 2 and 3). The first column represents the time spent on transforming Alloy models into 3-SAT clauses, and the second the time spent in SAT solving to find all counterexamples for each app bundle. The timing results show that Covert is able to analyze bundles of apps containing hundreds of components in the order of a few minutes (on an ordinary laptop), confirming that the proposed technology based on a lightweight formal analyzer is feasible.

4.7 Discussion

There is a growing need for technologies that can support the security analysis of complex systems in a compositional manner, whereby the security of a system is reasoned about in terms of the security properties inferred from its constituents. We argue this is the holy grail of software security analysis research. For the security analysis techniques to scale to ever-increasing complex systems, they need to become compositional in nature. Covert

takes an important step towards this overarching objective in the context of Android apps, but we envision the ideas set forth in this research to find a broader application in other computing domains as well.

Note that single app analysis and compositional analysis have their own technical merits. From an application developer's perspective, analyzing each app in isolation may provide sufficient feedback to fix the issues in the code (i.e., remove the vulnerabilities). On the other hand, when the purpose of analysis is to assess the trustworthiness of a system, comprised of multiple proprietary apps that may interact with one another, compositional analysis is needed to detect vulnerabilities that may exist in the system. One can imagine an organization may need to use a tool such as Covert to analyze the security properties of apps deployed on phones assigned to its employees. Such an organization may not be in a position to fix the issues in the apps, as the apps may be proprietary, but it can control the apps that are installed on the devices.

Our analysis indicates that IPC vulnerabilities are ubiquitous, and demonstrates why prior techniques relying only on single app analysis are insufficient for detecting such vulnerabilities. Our experiences with a novel approach for compositional app analysis and its evaluation in the context of hundreds of real-world Android apps collected from variety of repositories have been very positive. The experimental data shows that Covert can effectively detect such inter-app vulnerabilities in the order of few minutes.

Development Effort

The framework specification is not expected to be written by individual users of Covert, rather by the provider of the framework or Covert. The specification for a framework, such as Android, is developed once and can be reused by others. Thus, it poses a one-time cost, and the required effort depends on the level of familiarity with the framework and the

specification language. Using executable specification languages, one can also immediately check the correctness of even partial specifications. In our own experience, Alloy helped us to find errors early in specifying formal semantics. More specifically, during the modeling process, its analyzer performed syntactic checks to expose, for instance, inaccurate use of signatures (such as accessing a nonexistent field of a signature). We also used the analyzer to check the conformance of automatically generated models of apps derived through static analyzer to the framework meta-model.

```
1  assert  appCollusion{
2      no disj cmp1, cmp2: Component|
3          some cmp1.paths && some cmp2.paths &&
4          cmp1.app != cmp2.app &&
5          match[cmp1.paths.destination,cmp2.paths.entry]
6  }
7
8  pred match(pathSink: set Resource+Intent,
9    pathSource: set Resource+IntentFilter){
10     SDCARD in pathSource & pathSink ||
11     LOG in pathSource & pathSink ||
12     (some i:Intent, f: IntentFilter|
13     i in pathSink && f in pathSource && matchIPC[i,f])
14 }
```

Figure 4.16: Specification of the application collusion vulnerability in Alloy.

4.7.1 Other Types of Vulnerabilities

While privilege escalation vulnerability has been the focus of our research, we believe COVERT can be extended, and significant components of it reused, for detecting other types of inter-app vulnerabilities. For instance, an important class of inter-app vulnerabilities are due to information leakage. For these types of vulnerabilities, Covert's program analysis needs to be extended to take information flow into account for Android apps. While not the focus of this research, in an alternative configuration, we augmented Covert's reachability analysis described in Section 5.3 with a taint flow analysis approach (see [67]) to detect possible information leaks between apps.

We illustrate the reuse and extension potential of Covert through an example of the ap-

plication collusion vulnerability. Consider two applications `A` and `B`; `B` reads data from a particular folder in SD card and sends the data out through Internet, and `A` writes data to the folder that `B` reads from. Since `B` does not expose the sending action through its interface (IntentFilter), it cannot be detected by the *privilegeEscalation* check, specified in Figure 4.11.

To extend Covert for supporting the analysis of this scenario, the only thing required is to model it as an assertion, expressing properties to be checked in the extracted specifications. Figure 4.16 expresses such an assertion for the application collusion. The assertion states that there are two components in different applications; each contains a sensitive data flow path, where the sink of one matches the source of the other. Recall from Section 4.5 that the `paths` field denotes information paths between permission domains for each component.

Continuing with our example, the apps `A` and `B` contain the flow permissions: `IMEI` $\rightarrow$ `SDCARD` and `SDCARD` $\rightarrow$ `NETWORK`, respectively. These two paths will set the *match* predicate to be true (line 8), and thus Covert identifies it as an instance of the application collusion. Note that since applications specifications and properties to be checked are strictly separated, arbitrary vulnerabilities can be detected with minimal effort.

4.8 Conclusion

This chapter presents a novel approach for compositional analysis of Android inter-app vulnerabilities. Our approach employs static analysis to automatically recover models that reflect Android apps and interactions among them. It is able to leverage these models to identify vulnerabilities due to interaction of multiple apps that cannot be detected with prior techniques relying on a single app analysis. We formalized the basic elements of our analysis in an analyzable specification language based on relational logic, and developed a prototype

implementation, COVERT, on top of our formal analysis framework. The experimental results of evaluating COVERT against privilege escalation—one of the most prominent inter-app vulnerabilities—in the context of hundreds of real-world Android apps corroborates its ability to find vulnerabilities in bundles of some of the most popular apps on the market.

Chapter 5

Automatic Enforcement of Permission-Based Security Policies for Android

Previous chapter presented COVERT, which is intended to *identify* security threats that occur due to interaction of multiple apps. In this chapter, I present a complementary approach that *thwarts* the identified security threats through enforcement of security policies, allowing end-users to safeguard the apps installed on their device from inter-app vulnerabilities.

5.1 Introduction

The ubiquity of smartphones and our growing reliance on mobile apps are leaving us more vulnerable to cyber-security attacks than ever before. According to the Symantec's Norton report [484], in 2013 the annual financial loss due to cybercrime exceeded $113 billion globally, with every second 12 people become the victim of cybercrime. An equally ominous

report from Gartner [215] predicts 10 percent yearly growth in cybercrime-related financial loss through 2016. This growth is attributed in part to the new security threats targeted at emerging platforms, such as Google Android and Apple iOS, as 38% of mobile users have experienced cybercrime [484]. This is, though, nowhere more evident than in the Android market, where many cases of apps infected with malware and spyware have been reported [449].

In this context, smartphone platforms, and in particular Android, have emerged as a topic *du jour* for security research. These research efforts have investigated weaknesses from various perspectives, including detection of information leaks [341, 222, 181, 252], analysis of the least-privilege principle [198, 183], and enhancements to Android protection mechanisms [207, 115, 167]. Above and beyond such security techniques that are substantially intended to detect vulnerabilities in a single application, researchers have recently investigated techniques tackling security vulnerabilities that arise due to the interaction of multiple applications, such as inter-component data leaks [308, 514, 294] and permission leaks [251] shown to be quite common in the apps on the markets.

While the prior techniques mainly aim to find security weaknesses in existing combination of apps, we are also interested in the dual of this problem, that is *what security attacks are possible given a set of vulnerable apps?* Many Android malware are embedded in supposedly normal apps that aim to leverage vulnerabilities in either the platform or other apps on the market for nefarious purposes [481]. If we could automatically generate security exploits for a given combination of apps, it would allow us to identify possible security attacks before the adversary, and thus protect our systems prior to the realization of such attacks.

In this chapter, we propose a proactive scheme to develop Android security policies for vulnerabilities that occur due to the interaction of apps comprising a system. Our approach aims to automatically find vulnerabilities in a given bundle of apps and generate specifications of possible exploits for them, which then can proactively be applied as preventive measures

to guard against yet unknown malicious behavior.

Specifically, we have developed an automated system for **s**ynthesis and **e**nforcement of security **p**olicies for **A**nd**r**oid, called SEPAR, a Persian word for shield. It combines scalable static analysis with lightweight formal methods. SEPAR leverages static analysis to automatically infer security-relevant facts about software systems.[1] The app specifications are sufficiently abstract—extracted at the architectural level—to be amenable to formal analysis, and to ensure the technique remains scalable to real-world Android apps, yet represent the true behavior of the implemented software, as they are automatically extracted from the app bytecode, and appear sufficiently detailed to express subtle inter-app vulnerabilities.

SEPAR then uses a SAT-based engine to analyze the system model against compositional security properties and generate potential attack scenarios. In fact, it mimics the adversary by leveraging recent advancements in constraint solving techniques to synthesize possible security exploits, from which fine-grained security policies are then derived and enforced for each particular system. The synthesis of system-specific security policies allows the user to proactively deploy preventive measures prior to the discovery of those exploits by the adversaries.

To summarize, this chapter makes the following contributions:

- *Formal Synthesis of Security Policies:* We introduce a novel approach to synthesize specifications of possible exploits for a given combination of apps, from which system-specific security policies are derived. The policy synthesizer relies on a fully analyzable formal model of Android framework and a scalable static analysis technique extracting formal specifications of Android apps.

- *Runtime Enforcement of Security Policies:* We develop a new technology to automati-

[1]By a software system, we mean a set of independently developed apps jointly deployed on top of a common computing platform, e.g. Android framework, that interact with each other, and collectively result in a number of software solutions or services.

cally apply and dynamically enforce the synthesized, fine-grained policies (at the level of event messaging), specifically generated for a particular collection of apps installed on the end-user device.

- *Experiments*: We present results from experiments run on 4,000 real-world apps as well as DroidBench2.0 test suite [11], corroborating SEPAR's ability in (1) effective compositional analysis of Android inter-application vulnerabilities and generation of preventive security policies, that many of those vulnerabilities cannot be even detected by state-of-the-art security analysis frameworks; (2) outperforming other compositional analysis tools also in terms of scalability; and (3) finding multiple crucial security problems in the apps on the markets that were never reported before.

The remainder of chapter is organized as follows. Section 5.2 motivates our research through an illustrative example. Section 5.3 provides an overview of SEPAR. Sections 5.4, 5.5 and 5.6 describe the details of static model extraction, formal synthesis and dynamic enforcement of policies, respectively. Section 5.7 present implementation and evaluation of the research.

5.2 Motivating Example

To motivate the research and illustrate our approach, we provide an example of a vulnerability pattern having to do with inter-component communication (ICC) among Android apps. Android provides a flexible model of component communication using a type of application-level message known as *Intent*. A typical app is comprised of multiple components (e.g., Activity, Service) that communicate using Intent messages. In addition, under certain circumstances, an app's component could send Intent messages to another app's components to perform actions (e.g., take picture, send text message, etc.). Figure 5.3 partially shows a bundle of two benign, yet vulnerable apps, installed together on a device.

```
1  public class LocationFinder extends Service {
2    public void onStartCommand(Intent intent, int flags, int startId){
3     LocationManager lm = getSystemService(Context.LOCATION_SERVICE);
4     Location lastKnownLocation =
5     lm.getLastKnownLocation(LocationManager.GPS_PROVIDER);
6     Intent intent = new Intent();
7     intent.setAction("showLoc");
8     intent.putExtra("locationInfo", lastKnownLocation.toString());
9     startService(intent);
10    ...
11   }
12 }
```

Figure 5.1: LocationFinder sends the retrieved location data to another component of the same app via implicit Intent messaging.

```
1  public class MessageSender extends Service  {
2   public void onStartCommand(Intent intent, int flags, int startId) {
3    String number = intent.getStringExtra("PHONE_NUM");
4    String message = intent.getStringExtra("TEXT_MSG");
5    //if (hasPermission())
6    sendTextMessage(number, message);
7    ...
8   }
9   void sendTextMessage (String num, String msg) {
10   SmsManager mngr = SmsManager.getDefault();
11   mngr.sendTextMessage(num,null,msg,null,null);
12  }
13  boolean hasPermission () {
14   if(checkCallingPermission("android.permission.SEND_SMS")==PackageManager.
       PERMISSION_GRANTED)
15     return true;
16   return false;
17  }
18 }
```

Figure 5.2: MessageSender receives an Intent and sends a text message.

The first application is a navigation app that obtains the device location (GPS data) in one of its components and sends it to another component of the app via Intra-app Intent messaging. The Intent involving the location data (Figure 5.1, lines 3–9), instead of explicitly specifying the receiver component, i.e., *RouteFinder* service, implicitly specifies it through declaring a certain `action` to be performed in that component. This represents a common practice among developers, yet an anti-pattern that may lead to unauthorized Intent receipt [143], as any component, even if it belongs to a different app, that matches the *action* could receive an implicit Intent sent this way.

On the other hand, the vulnerability of the second application, a messenger app, occurs

on line 11 of Figure 5.2, where *MessageSender*, specified as a public component in the app manifest file, uses system-level API `SmsManager`, resulting in a message sent to the phone number previously retrieved from the Intent. This is a reserved Android API that requires special access permissions to the system's telephony service. Although MessageSender has that permission, it also needs to ensure that the sender of the original Intent message has the required permission to use the SMS service. An example of such a check is shown in `hasPermission` method of Figure 5.2, but in this particular example it does not get called (line 6 is commented) to illustrate the vulnerability.

Given these vulnerabilities, a malicious app can send the device location data to the desirable phone number via text message, without the need for any permission. As shown in Figure 5.3, the malicious app first hijacks the Intents containing the device location info from the first app. Then, it sends a fake Intent to the second app, containing the GPS data and adversary phone number as the payload. While the example of Figure 5.3 shows exploitation of vulnerabilities in components from two apps, in general, a similar attack may occur by exploiting the vulnerabilities in components of either single app or multiple apps. Moreover, since the malicious app does not require any security sensitive permission, it is easily concealed as a benign app that only sends and receives Intents. This makes the detection of such malicious apps a challenging task for individual security inspectors or anti-virus tools.

The above example points to one of the most challenging issues in Android security, i.e., detection and enforcement of compositional security policies to prevent such possible exploits. What is required is a system-level analysis capability that not only identifies the vulnerabilities and capabilities in individual apps, but also determines how those individual vulnerabilities and capabilities could affect one another when the corresponding apps are installed together. In the next sections, we first provide an overview of Separ and then delve into more details about its approach to address these issues.

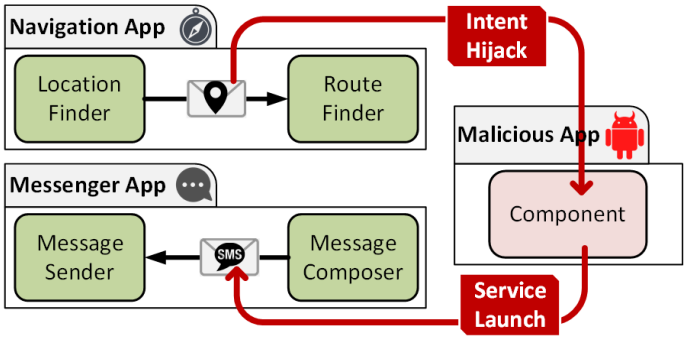


Figure 5.3: A potential malicious application— its signature automatically generated by SEPAR—leverages vulnerabilities in other already installed benign applications to perform actions (like sending device location through text messages) that are beyond its individual privileges. As the Android access control model is per app, it cannot check security posture of the entire system. SEPAR generates and enforces compositional policies that prevent such an exploit.

5.3 Approach Overview

This section overviews our approach to automatically synthesize and enforce system-specific security policies for such vulnerabilities that occur due to the interaction of apps comprising a system. As depicted in Figure 5.4, SEPAR consists of three main components: (1) The *Android model extractor (AME)* that uses static analysis techniques to automatically elicit formal specifications of the apps comprising a system; (2) The *analysis and synthesis engine (ASE)* that uses lightweight formal analysis techniques [7] to find vulnerabilities in the extracted app models, and generates specifications of possible exploits, and in turn, policies for preventing their manifestation; (3) The *Android policy enforcer (APE)* that enforces automatically generated, system-wide policies on Android applications.

The AME component takes as input a set of Android application package archives, called APK files. APKs are dalvik bytecode packages used to distribute and install Android applications. To generate the app specifications, AME first examines the application manifest file to determine its architectural information. It then utilizes different static analysis techniques, i.e., control flow and data flow analyses, to extract other essential information from

the application bytecode into an analyzable specification language.

The ASE component, in addition to extracted app specifications, relies on two other kinds of specifications: a formal foundation of the application framework and the axiomatized inter-app vulnerability signatures. The Android framework specification represents the foundation of Android apps. Our formalization of these concepts includes a set of rules to lay this foundation (e.g., application, component, messages, etc.), how they behave, and how they interact with each other. It can be considered as an abstract, yet precise, specification of how the framework behaves. We regard vulnerability signatures as predicates that model Android inter-app vulnerabilities in relational logic, representing their essential characteristics as exhibited when the vulnerability is exploited. All the specifications are uniformly captured in the Alloy language [7]. Alloy is a formal specification language based on relational logic, amenable to fully automated yet bounded analysis.

Separ is designed as a plugin-based software that provides extension points for analyzing apps against different types of vulnerabilities. In order to analyze each app, we distill each known inter-app vulnerability into a corresponding formally-specified signature to capture its essential characteristics, as manifested when the vulnerability is exploited. Our current Separ prototype supports inter-component vulnerabilities, such as Activity/Service launch, Intent hijack, privilege escalation, and information leakage [115, 143, 233]. Its plugin-based architecture supports the necessary extensions that can be provided by users at anytime to enrich the environment.

Given these specifications, the ASE component analyzes them as a whole for instances of vulnerabilities in the extracted app specifications, and using formally-precise scenario-generating tools, such as Alloy Analyzer [7] and Aluminum [363], it attempts to generate possible security exploit scenarios for a given combination of apps. Specifically, we go beyond the detection of vulnerabilities by asking: *what security attacks are possible given a set of vulnerable apps?*

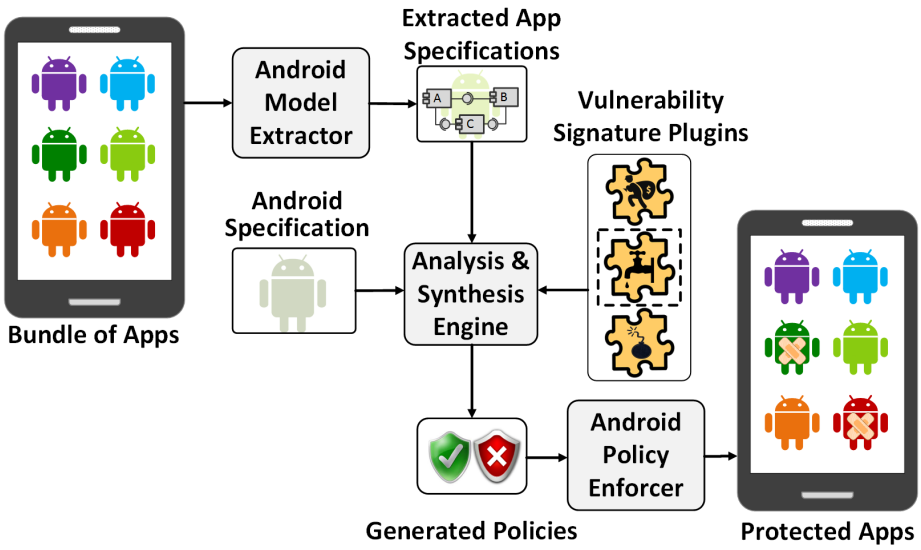


Figure 5.4: Approach Overview of SEPAR.

Having computed system-wide policies to prevent the postulated attacks, SEPAR parses and transforms them from models generated in relational logic to a set of configurations directly amenable to efficient policy enforcement. Our policy enforcer (APE) then monitors each vulnerable app at runtime to dynamically intercept event messages, check them against generated policies, and possibly inhibits their executions if violating any such policies. As such, to the best of our knowledge, SEPAR is the first approach capable of detecting and protecting Android systems against zero-day inter-app attacks.

In the following three sections, we describe the details of each component in turn.

5.4 AME: Android Model Extractor

The AME module, that individually analyzes each app to extract a model of its behavior, is built upon state-of-the-art static analysis techniques for the Android framework. This section describes the extraction process, with an emphasis on the important improvements on prior work.

Architecture Extraction. To obtain an app model, AME first examines the app manifest file to capture the high-level architectural information, including the components comprising the app, permissions that the app requires, and the enforced permissions that the other apps must have in order to interact with the app components. AME also identifies public interfaces exposed by each application, which are essentially entry points defined in the manifest file through Intent Filters of components.

Intent Extraction. The next step of model extraction involves an inter-procedural data flow analysis [107], to track the Intents and Intent Filters that are declared in code, rather than the manifest file, as well as their properties. Each Intent belongs to one particular component that sends it, may have one recipient component and may include an action, data and a set of categories. The action field specifies the general action to be performed in the recipient component; the data field represents additional information about the data to be processed by the action; and the categories filed specifies the kind of component that should handle the Intent. An Intent can also include extra data. Similar to Intents, each Intent Filter has a non-empty set of actions and two sets of data and categories. Note that Intent Filters for components of type Service and Activity must be declared in their manifest; for Broadcast Receivers, though, either in the manifest or at runtime.

To resolve the values associated with the retrieved attributes (e.g., the Intent action) AME uses string constant propagation [145], which provides a suitable solution since, by convention, Android apps use constant strings to define these values. In case a property is disambiguated to more than one value (e.g., due to a conditional assignment), AME generates a separate entity for each of these values, as they contribute different exposure surfaces or event messages in case of Intent Filters and Intents, respectively. AME handles aliasing through performing *on-demand alias analysis*[492]. More specifically, for each attribute that is assigned to a heap variable, the backward analysis finds its aliases and updates the set of its captured values accordingly.

Algorithm 5.1: Update Passive Intent Target

```
Input: Intents: Set of all identified Intents
Output: Target components for passive Intents
for p in Intents do
    if p.isPassiveIntent then
        for i in Intents do
            if i.hasRequestResult & i.target = p.sender then
                p.addTarget(i.sender)
            end
        end
    end
end
```

There are some special cases in implicit invocations of inter-component entry points, where the caller method triggers a two-way communication between components. Examples include `bindService` and `startActivityForResult`. A component, for instance, can use `startActivityForResult` to start another component, which itself implicitly calls the first component with a new Intent embodying the results once finishes running. However, the returning implicit Intent, which we call *passive Intent*, includes no information (e.g., action and category) specifying its target component, making it difficult for static analyzers to identify the receiver in this second implicit invocation. Algorithm 5.1 outlines identifying target components for passive Intents. The logic of the algorithm is as follows. For each passive Intent, p, look up Intents that both request for results and their target components match senders of p. Insert the senders of such Intents into the target set of p.

Path Extraction. AME analyzes the app using a static taint analysis to track sensitive data flow tuples $< Source, Sink >$, where *Source* represents a sensitive data (e.g., the device ID) and *Sink* represents a method that may leak data, such as sending text messages. To achieve a high precision in data flow analysis, our approach is flow-, field-, and context-sensitive [67], meaning that our analysis distinguishes a variable's values between different program points, distinguishes between different fields of a heap object, and that in analysis of method calls is sensitive to their calling contexts, respectively. In the interest of scalability, SEPAR's analysis,

however, is not path-sensitive. The results (See Section 5.7) though indicate no significant imprecision caused by path-insensitivity in the context of Android vulnerability analysis.

AME uses a set of most frequently used source and sink Android API methods from the literature [394], identified through the use of machine-learning techniques. To further detect those paths traversing through different components, we adapted this set by identifying source and sink methods corresponding to inter-component communication. The identified sensitive data flows paths are later used in the ASE module to detect data leaks vulnerabilities, and thereby to generate respective policies preventing their potential exploits.

Permission Extraction. To ensure the permission policies are preserved during an inter-component communication, one should compare the granted permissions of the caller component against the enforced permissions at the callee component side. Therefore, the permissions actually used by each component should be determined. While we already identified the coarse-grained permissions specified in the manifest file, AME analyzes permission checks throughout the code to identify those controlling access to particular aspects of a component (e.g., recall *hasPermission* method of Figure 5.2). In doing so, it relies on API permission maps available in the literature, and in particular the PScout permission map [69], one of the most recently updated and comprehensive permission maps available for the Android framework. API permission maps specify mappings between Android API calls/Intents and the permissions required to perform those calls.

A node could be directly tagged as *permission-required* node, or transitively tagged by tracking the call chains. To find the transitive permission tag, AME performs backward reachability analysis starting from the permission-required node. The tagged permission are propagated from all children to their parent nodes, until reaching to the root nodes. In case an entry-point node of a component is tagged by a permission, it will be added to the list of exposed permissions of that component.

5.5 ASE: Analysis and Synthesis Engine

We now show that our ideas for automated synthesis of exploit specifications can be reduced to practice. The insight that enabled such synthesis was that we could interpret the synthesis problem as the dual of formal verification. Given a system specification S, a model M, and a property P, formal verification asserts whether M satisfies the property P under S. Whereas the synthesis challenge is given a system specification S and a property P, generate a model M satisfying property P under system S. M is an instance model of S that satisfies P.

This observation enables leveraging verification techniques to solve synthesis problems. As shown in Figure 5.5, we can view the bundle of app specifications, S_a, and the framework specification, S_f, collectively as system S and a compositional security issue as property P, and model them as a set of constraints. The problem then becomes to generate a candidate set of violation scenarios, M, that satisfies the space of constraints: $M \models S_f \wedge S_a \wedge P$. Our approach is thus based on a reduction of the synthesis problem into a constraint-solving problem represented in relational logic (i.e., Alloy). Alloy is a formal modeling language optimized for automated analysis, with a comprehensible syntax that stems from notations ubiquitous in object orientation, and semantics based on the first-order relational logic [7].

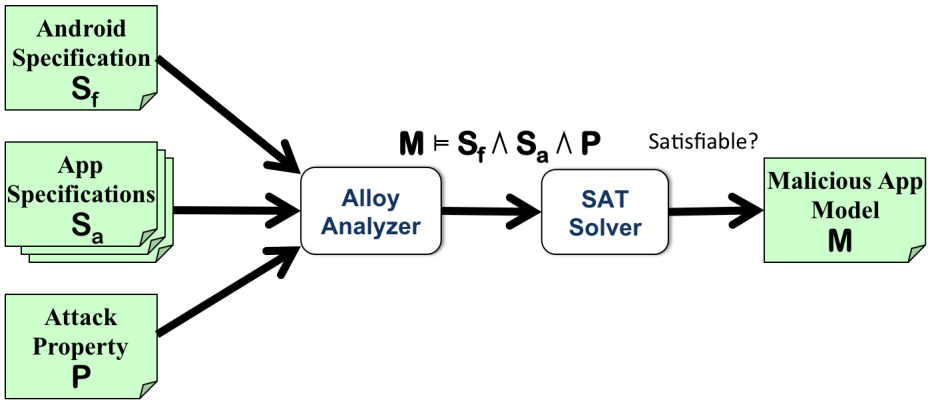


Figure 5.5: Automated synthesis of possible exploit specifications.

The formulation of the synthesis problem in Alloy consists of three parts: (1) a fixed set of signatures and facts describing the Android application fundamentals (e.g., application, component, Intent, etc.) and the constraints that every application must obey. Technically speaking, this module can be considered as a meta-model for Android applications; (2) a separate Alloy module for each app modeling various parts of an Android app extracted from its APK file. The automatically extracted model for each app relies on the Android framework specification module (the first item above); and (3) a set of signatures used to reify inter-component vulnerabilities in Android, such as privilege escalation.

Alloy is an appropriate language for our modeling and synthesis purposes for several reasons: (1) its simple set theoretic language, backed with logical and relational operators, was sufficiently expressive for formal declarative specification of both applications and properties to be checked; (2) its ability to automatically analyze specifications is useful as an automation mechanism, enabling automatic synthesis of violation scenarios as satisfying solutions; finally, (3) the formal analyzers available for Alloy (e.g., [363]) translate our high-level model specifications into a SAT formula that can be solved by off-the-shelf SAT solvers, and thereby enable utilizing state-of-the-art constraint solvers for our model synthesis. The rest of this section first provides a brief overview of Alloy, and then details different parts of implementing the synthesis problem.

Alloy Overview. Alloy is a declarative language based on the first-order relational logic with transitive closure [7]. The inclusion of transitive closure extends its expressiveness beyond first-order logic. Essential data types, that collectively define the vocabulary of a system, are specified in Alloy by their type signatures (sig). Signatures represent basic types of elements, and the relationships between them are captured by the the declarations of *field*s within the definition of each signature. Consider the following Alloy model. It defines two Alloy signatures: `Application` and `Component`. The `cmps` relation is defined over these two signatures.

```
sig Application{
  cmps: Component
}
sig Component{}
```

Analysis of specifications written in Alloy is completely automated, based on transformation of Alloy's relational logic into a satisfiability problem. Off-the-shelf SAT solvers are then used to exhaustively search for either satisfying models or counterexamples to assertions. To make the state space finite, certain scopes need to be specified that limit the number of instances of each type signature. The following specification asks for instances that contain at least one `Component`, and specifies a scope that bounds the search for instances with at most two objects for each top-level type (`Application` and `Component` in this example).

```
pred modelInstance{ some Component }
run modelInstance for 2
```

When executed, the Alloy Analyzer produces model instances, two of which are shown in Fig. 5.6. The model instance of Fig. 5.6a includes one application and two components, one of them belongs to no application. Fig. 5.6b shows another model instance with two applications, each one having one component.

Facts (fact) are formulas that take no arguments, and define constraints that every instance of a model must satisfy, thus restricting the instance space of the model. The following fact paragraph, for example, states that each `Component` should belong to exactly one `Application`. Re-executing the Alloy Analyzer produces a new set of model instances, where while Fig. 5.6b is still a valid instance, model of Fig. 5.6a is eliminated.

```
fact {
  all c: Component| one c.~cmps
}
```

The other essential constructs of the Alloy language include: *Predicates*, *Functions* and *Assertions*. Predicates (pred) are named logical formulas used in defining parameterized and reusable constraints that are always evaluated to be either true or false. Functions (fun) are

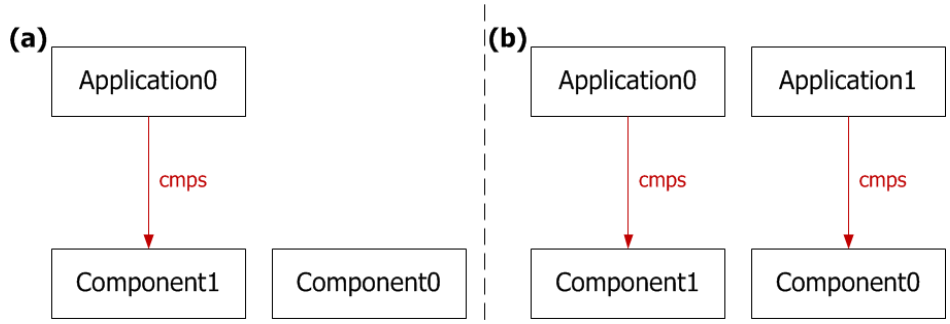


Figure 5.6: Two model instances of the above Alloy specification.

parameterized expressions. A function similar to a predicate can be invoked by instantiating its parameter, but what it returns is either a true/false or a relational value instead. An assertion (assert) is a formula required to be proved. It can be used to check a certain property of a model.

The Alloy language comes with a set of logical and relational operators. The dot (.) and tilde (∼) operators denote a relational join of two relations and the transpose operation over a binary relation, respectively. The transitive closure (ˆ) of a relation is the smallest enclosing relation that is transitive. The reflexive-transitive closure (*) of a relation is the smallest enclosing relation that is both transitive and reflexive.

We will introduce additional details of the Alloy language as necessary to present our policy synthesis approach. For further information about Alloy, we refer the interested reader to [7].

Formal Model of Android Framework. Formal modeling of the Android framework was the subject of earlier work [80]. To make this chapter self-contained, this subsection briefly reviews the model. Figure 5.7 shows (part of) the Alloy code describing the meta-model for Android application models. Our model is based on the official Android documentation [26]. Android is a large and complex operating system, and modeling it in its entirety would be infeasible. Thus, we focused on the parts of Android that are relevant to the inter-component communication and their potential security challenges. For example, note the signatures `Component` and `Intent`. Signatures defined as abstract represent types of elements that

```
1  abstract sig Component{
2    app: one Application,
3    intentFilters: set IntentFilter,
4    permissions: set Permission,
5    paths: set DetailedPath
6  }
7  abstract sig IntentFilter{
8    actions: some Action,
9    dataType: set DataType,
10   dataScheme: set DataScheme,
11   categories: set Category
12 }
13 fact IFandComponent{
14   all i:IntentFilter| one i.~intentFilters  }
15 fact NoIFforProviders{
16   no i:IntentFilter| i.~intentFilters in Provider  }
17 abstract sig Intent{
18   sender: one Component,
19   receiver: lone Component,
20   action: lone Action,
21   categories: set Category,
22   dataType: lone DataType,
23   dataScheme: lone DataScheme,
24   extra: set Resource
25 }
```

Figure 5.7: Excerpts from the meta-model for Android application models in Alloy.

cannot have an instance object without explicitly extending them. A component belongs to exactly one application, and may have any number of `IntentFilters`—each one describing a different interface (capability) of the component—and a set of permissions required to access the component. The `paths` field then indicates information flows between permission domains in the context of this component. We define the source and destination of a path based on canonical permission-required resources identified by Holavanalli et al. for Android applications [251]. Examples of such resources are NETWORK, IMEI, and SDCARD. Thirteen permission-required resources are identified as source, and five resources as destination, of a sensitive data flow path. The ICC mechanism augments both source and destination sets. Note that to eliminate private components from inter-app analysis, Separ considers the component's exported attribute. In fact, a component can receive Intents from other applications, or is public, if its exported attribute is set or contains at least one Intent filter. Such elimination of private components from inter-app analysis also contributes to the scalability of the approach (i.e., less components to be analyzed).

```
1  \\(a) App1 model
2  open androidDeclaration
3  ...
4  one sig LocationFinder extends Service{}{
5    app in App1
6    no intentFilters
7    paths = pathLocationFinder1
8    permissions = ACCESS_FINE_LOCATION
9  }
10 one sig pathLocationFinder1 extends Path{}{
11   source = LOCATION
12   sink = ICC
13 }
14 one sig Intent1 extends Intent{}{
15   sender = LocationFinder
16   no receiver
17   action=showLoc
18   categories= DEFAULT
19   no dataType
20   no dataScheme
21   extra= LOCATION
22 }
23 \\(b) App2 model
24 one sig MessageSender extends Service{}{
25   app in App2
26   intentFilter = IntentFilter1
27   paths = pathMessageSender1
28   no permissions
29 }
30 one sig pathMessageSender1 extends Path{}{
31   source = ICC
32   sink = SMS
33 }
```

Figure 5.8: Excerpts from generated specifications for (a) App1 (Figure 5.1) and (b) App2 (Figure 5.2).

The fact `IFandComponent` specifies that each IntentFilter belongs to exactly one `Component`, and the fact `NoIFforProviders` specifies that out of four core component types, only three of them can define IntentFilters; no IntentFilter can be defined for `Content Provider` components.

An Intent belongs to one particular component sending it, and may have one recipient component. Each Intent may also include an `action`, `data` (`type` and `scheme`) and a set of `categories`.[2] These elements are used to determine to which component an *implicit Intent*—one that does not specify any recipient component—should be delivered. Each of

[2]The multiplicity keyword ***some*** in Alloy denotes that the declared `IntentFilter.actions` relation contains at least one element; the keyword ***set*** tells Alloy that `categories` map each `IntentFilter` object to zero or more `Category` objects, and the keyword ***lone*** indicates that this `Intent.component` is optional, and an Intent may have one or no declared recipient component.

these elements corresponds to a test, in which the Intent's element is matched against that of the IntentFilter. An IntentFilter may have more actions, data, and categories than the Intent, but it cannot contain less. The `extra` field indicates the types of resources carried by the Intent.

Formal Model of Apps. Figure 5.8 partially shows the Alloy specifications for the apps shown in Figures 5.1 and 5.2. As already mentioned (See Section 5.4), these app specifications are automatically extracted by the AME component from each Android application. Each app specification starts by importing the *androidDeclaration* module (See Figure 5.7). Among other things, the `LocationFinder` component contains a sensitive path (`pathLocationFinder1`), that represents a data-flow from where the sensitive GPS data is retrieved, to an Intent event message. The `extra` field of the Intent in the generated Alloy model (line 21) is accordingly set. The `path` field of the *MessageSender* in the generated Alloy model (lines 27, 30–33) reflects another data-flow path, started from an IntentFilter and reaches to a node, which uses the data in the body of a text message. Note that this component does not enforce any access permission neither in the manifest file nor in the code (line 28).

Formal Model of Vulnerabilities. To provide a basis for precise analysis of app bundles against inter-app vulnerabilities and further to automatically generate possible scenarios of their occurrence given particular conditions of each bundle, we designed specific Alloy signatures. Specifically, each vulnerability model captures a specific type of inter-component communication security threat, according to those identified by Chin et al. [143] and Bugiel et al. [115]. The security property check is then formulated as a problem of finding a valid trace that satisfies the vulnerability signature specifications. If the Alloy Analyzer finds a solution to this problem, the property is violated; the returned solution encodes an exact scenario (states of all elements, such as components and Intents) leading to the violation. As a concrete example, we illustrate the semantics of one of these vulnerabilities in the following.

```
1  sig GeneratedServiceLaunch{
2    disj launchedCmp,malCmp: one Component,
3    malIntent: Intent  }{
4    malIntent.sender = malCmp
5    launchedCmp in setExplicitIntent[malIntent]
6    no launchedCmp.app & malCmp.app
7    launchedCmp.app in device.apps
8    not (malCmp.app in device.apps)
9    some launchedCmp.paths && launchedCmp.paths.source = ICC
10   some malIntent.extra
11   malCmp in Activity
12 }
```

Figure 5.9: Alloy specifications of Service Launch vulnerability in Android.

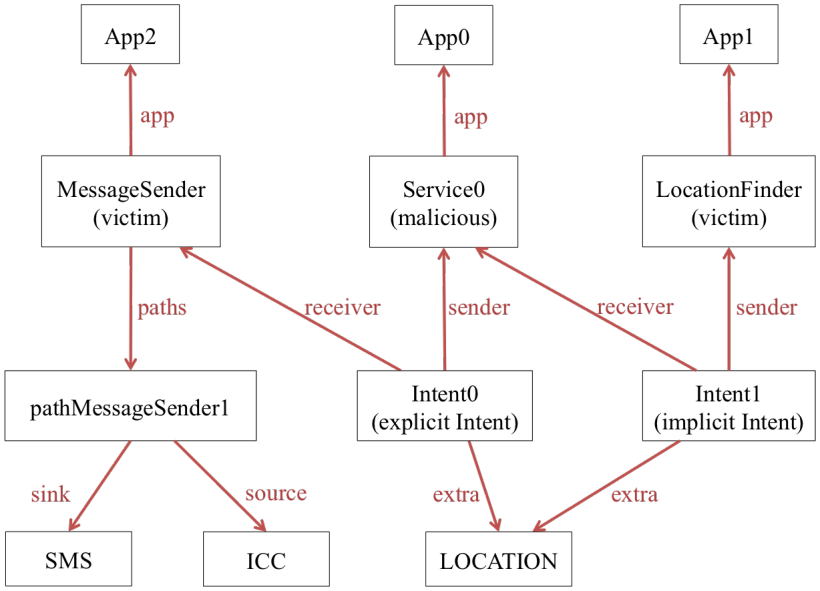
The others are evaluated similarly.

Figure 5.9 presents the `GeneratedServiceLaunch` signature along with its *signature fact* that specifies the elements involved in, and the semantics of, a service launch exploit, respectively. In short, a malicious component (`malCmp`) can launch a component by sending an Intent (`malIntent`) to an exported component (`launchedCmp`) that is not expecting Intents from that component. According to line 9, the *launchedCmp* component has a path from the exported interface to a permission-required resource. It, thus, may leak information or perform unauthorized tasks, depending on the functionalities exposed by the victim component.

Generating possible exploit scenarios. We run the modules defined above with a command that tries to satisfy the vulnerabilities signature facts. Note that Alloy analysis must be done within a given scope, which specifies an upper bound for, or an exact, number of instances per element signature. In our case, the exact scope of each element, such as Application and Activity, required to instantiate each vulnerability is automatically derived from the specification.

If an instance is found, SEPAR reports it along with the information useful in finding the root cause of the violation, from which fine-grained security policies are then derived for the given system. Given our running example, the analyzer automatically generates the

following scenario, among others:



The diagram is accurate for the result that the analyzer computed, but we have edited it to omit some details for readability. It essentially states the scenario represented in Figure 5.3, in which a postulated malicious component, here the generated `App0/Service0` component, can send the device location data captured from a vulnerable Intent, `Intent1` (See Figure 5.8, lines 14–22), to the desirable phone number via an explicit Intent, `Service0/Intent0`, sent to the `App2/MessageSender` component that is vulnerable to service launch. Here the analysis has found that it is possible to devise a malicious capability that can leverage the vulnerabilities in the apps installed on the device for nefarious purposes. Given this, SEPAR formulates a policy, as described next, that prevents certain Intent-based interactions from occurring to prevent the exploitation of vulnerabilities, thereby achieving proactive defense if such a malicious capability were to be installed on the device.

The next section describes how we can prevent occurrence of such vulnerability exploits through generation and enforcement of respective policies.

5.6 APE: Android Policy Enforcer

In the implementation of APE, we faced three possible alternatives: (1) modify the Android OS to enforce the policies, (2) modify an app through injection of policy enforcement logic into the app's implementation by instrumenting the APK file, and (3) dynamic memory instrumentation of the app's process. We chose the third approach, as it allows SEPAR to be used on an unmodified version of Android, thereby making it widely applicable and practical for use by many.

Similar to a conventional access control model [428], our approach is comprised of two elements: *policy decision point (PDP)*—the entity which evaluates access requests against a policy, and *policy enforcement point (PEP)*—the entity which intercepts the request to a resource, makes a decision request to the PDP, and acts on the received decision. The protected resources in our research are mainly Android APIs that can result in ICC calls.

Our Android policy enforcer relies on the Xposed [44] framework for modifying the behavior of Android apps at runtime, without making any changes in the apps' APK files. It provides mechanisms to "hook" method calls. A hook is a method that is called before or after a certain method, making it possible to control pre/post method call activities, by modifying a method's parameters, its return values, or even entirely skipping the call to the method.

The PDP is realized as an independent Android app that stores the synthesized policies for preventing or allowing ICC access. Our policies are in the form of event-condition-action (ECA) rules. The PEP in our case corresponds to an Xposed module to dynamically intercept event messages. More specifically, each ICC method in an app's APK file (e.g., `startService(Intent)`) is hooked, such that whenever it is invoked, it is first assessed to see whether the operation should proceed (e.g., `Intent` to be delivered to its destination) by calling the PDP. The major advantages of using run-time process instrumentation over modifying individual apps are scalability and framework generalization. Additionally, in-

strumentation of APK files changes the signature of apps, which might prevent their proper execution.

PEP hooks these operations and uses PDP to check whether they are allowed to run or not. Whenever an application is about to run a sensitive operation, it is checked against the synthesized policies. The respective application is then allowed to perform the given operation as long as it conforms to such policies. Otherwise, the PDP prompts the user for consent along with the information that would help the user in making a decision, including the description of security threat as well as the name and parameters of the intercepted event. Should the user refuse, the application skips the given operation and continues with running the subsequent one. As ICC mechanisms in Android are essentially performed by asynchronous API calls, inhibiting them implies that no response for the event is ever received, without causing unexpected crashes. Of course, preventing ICC calls would naturally force the app to operate in a degraded mode.

Continuing with our running example, SEPAR generates the following policy, where the conditions in the generated ECA rule correspond to the properties of the malicious Intent in the synthesized vulnerability model instance.

```
{ event : ICC_received,
  condition : [{ Intent.extra: LOCATION},
               { Intent.receiver: MessageSender}],
  action : user_prompt
}
```

It states that every attempt of sending device LOCATION data through the MessageSender component must be manually approved by the user. Observe that each app, such as App2 can, and in this case would, be guarded against more than one policy at the same time. Indeed, App1 and App2 would also be guarded with policies generated regarding Intent hijacking and Service Launch, respectively.

5.7 Evaluation

This section presents the experimental evaluation of Separ. We have implemented Separ's static analysis capability on top of the Soot [495] framework. We used Flowdroid for intra-component taint analysis [67], and extended it to improve precision of analysis especially to support complicated ICC methods (See Section 5.4). The prototype implementation of Separ only requires the APK files—not the original source code—which is important, of course, for running it over non-open source apps. The translation of captured app models into the Alloy language is implemented using FreeMarker template engine [14]. The core components of our analysis and synthesis model are embedded in a relational logic language, i.e., Alloy [7]. As a back-end analysis engine, Separ relies on Aluminum [363], a recently developed principled scenario explorer that generates only minimal scenarios for specifications axiomatized in Alloy. Lastly, our policy enforcer (See APE module) leverages the Xposed framework [44] for preventing event messages violating synthesized policies.

We used the Separ apparatus for carrying out the experiments. Our evaluation addresses the following research questions:

RQ1. What is the overall accuracy of Separ in detecting ICC (i.e., both inter-component and inter-application) vulnerabilities compared to other state-of-the-art techniques?

RQ2. How well does Separ perform in practice? Can it find security exploits and synthesize their corresponding protection policies in real-world applications?

RQ3. What is the performance of Separ's analysis realized atop static analyzers and SAT solving technologies?

RQ4. What is the performance of Separ's policy enforcement?

5.7.1 Results for RQ1 (Accuracy)

To evaluate the effectiveness and accuracy of our analysis technique and compare it against the other static analysis tools, we used the DroidBench [11] and ICC-Bench [17] suites of benchmarks, two sets of Android applications containing ICC based privacy leaks for which all vulnerabilities are known in advance—establishing a ground truth. These test cases comprise the most frequently used ICC methods found in Google Play apps. The benchmark apps also include unreachable, yet vulnerable components; reported vulnerabilities that involve such unreachable components are thus considered as false warnings. Using the apps in this benchmark, which is developed by other research groups, we have attempted to eliminate internal threats to the validity of our results. Further, using the same benchmark apps as prior research allows us to compare our results against them.

We compared Separ with existing tools targeted at ICC vulnerability detection, namely DidFail [294] and AmanDroid [514]. We also tried to run IccTA [308], another tool intended to identify inter-app vulnerabilities, but faced technical difficulties. The tool terminated with error while capturing ICC links. This issue has also been reported by others [18]. Though we have been in contact with the authors, we have not been unable to fix it so far.

Table 5.1 summarizes the results of our experiments for evaluating the accuracy of Separ in detecting ICC vulnerabilities compared to other state-of-the-art techniques. Separ succeeds in detecting all 23 known vulnerabilities in DroidBench benchmarks, and 7 vulnerabilities out of 9 in ICC-Bench suite. It correctly finds both cases of privacy leak in *bindService4* and *startActivityForResults4*. It also correctly ignores two cases where there are no leaks, since the code harboring those vulnerabilities is not reachable, i.e., *startActivity*{*4,5*}. The only missed vulnerabilities are the ones that are caused by dynamic registration of Broadcast Receivers, which is not handled by Separ's model extractor.

In addition to missing the vulnerabilities in the bound services, AmanDroid is unable to

Table 5.1: Comparison between SEPAR, DidFail, and AmanDroid. TP, FP and FN are represented by symbols ☑, ☒, ☐, respectively. (X#) indicates the number # of detected instances for the corresponding symbol X.

	Test Case	DidFail	AmanDroid	SEPAR
DroidBench2	ICC_bindService1	☒☐	☐	☑
	ICC_bindService2	☐	☐	☑
	ICC_bindService3	☐	☐	☑
	ICC_bindService4	☒(☐2)	(☐2)	(☑2)
	ICC_sendBroadcast1	☑	☑	☑
	ICC_startActivity1	☐	☑	☑
	ICC_startActivity2	☐	☑	☑
	ICC_startActivity3	☐	☑	☑
	ICC_startActivity4	☒		
	ICC_startActivity5	(☒2)		
	ICC_startActivityForResult1	☐	☑	☑
	ICC_startActivityForResult2	☐	☐	☑
	ICC_startActivityForResult3	☐	☐☒	☑
	ICC_startActivityForResult4	(☐2)	☑☒☐	(☑2)
	ICC_startService1	☒☐	☑	☑
	ICC_startService2	☒☐	☐	☑
	ICC_delete1	☐	☐	☑
	ICC_insert1	☐	☐	☑
	ICC_query1	☐	☐	☑
	ICC_update1	☐	☐	☑
	IAC_startActivity1	☑☒	☐	☑
	IAC_startService1	☑	☐	☑
	IAC_sendBroadcast1	☑	☐	☑
ICC-Bench	Explicit_Src_Sink	☐	☑	☑
	Implicit_Action	☑	☑	☑
	Implicit_Category	☑	☑	☑
	Implicit_Data1	☑	☑	☑
	Implicit_Data2	☑	☑	☑
	Implicit_Mix1	☑	☑	☑
	Implicit_Mix2	☑	☑	☑
	DynRegisteredReceiver1	☐	☑	☐
	DynRegisteredReceiver2	☐	☐	☐
Precision		55%	86%	100%
Recall		37%	48%	97%
F-measure		44%	63%	98%

examine Content Providers for security analysis. DidFail does even worse. Based on the results, DidFail found only the vulnerabilities caused by implicit Intents, missing the vulnerabilities that are due to explicit Intents, such as information leak. The results show that SEPAR outperforms the other two tools in terms of both precision and recall.

5.7.2 Results for RQ2 (Separ and Real-World Apps)

To evaluate the implications of our tool in practice, we collected 4,000 apps from the following four different sources:

(1) Google Play [15]: This repository serves as the official Android app store. Our Google play collection consists of 600 randomly selected and 1,000 most popular free apps in the market. **(2) F-Droid** [13]: This is a software repository that contains free and open source Android apps. Our collection includes 1,100 apps from this Android market. **(3) Malgenome** [586]: This repository contains malware samples that cover the majority of existing Android malware families. Our collection includes all (about 1,200) apps in this repository. **(4) Bazaar** [6]: This website is a third-party Android market. We collected 100 popular apps from this repository, distinguished from apps downloaded from Google Play and F-Droid.

We partitioned the subject systems into 80 non-overlapping bundles, each comprised of 50 apps, simulating a collection of apps installed on an end-user device. The bundles enabled us to perform several independent experiments. Out of 4,000 apps, Separ identified 97 apps vulnerable to Intent hijack, 124 apps to Activity/Service launch, 128 apps to inter-component sensitive information leakage, and 36 apps to privilege escalation. We then manually inspected the Separ's results to assess its utility in practice. In the following, we describe some of our findings. To avoid leaking previously unknown vulnerabilities, we only disclose a subset of those that we have had the opportunity to bring to the app developers' attention.

Activity/Service Launch. *Barcoder* is a barcode scanner app that scans bills using the phone's camera, and enables users to pay them through an SMS service. It also stores the user's bank account information, later used in paying the bills. Given details of a bill as payload of an input Intent, the *InquiryActivity* component of this app pays it through SMS service. This component exposes an unprotected Intent Filter that can be exploited by a

malicious app for making an unauthorized payment.

Intent Hijack. *Hesabdar* is an accounting app for personal use and money transaction that, among other things, manages account transactions and provides a temporal report of the transaction history. One of its components handles user account information and sends the information as payload of an implicit Intent to another component. When a component sends an implicit Intent, there is no guarantee that it will be received by the intended recipient. A malicious application can intercept an implicit Intent simply by declaring an Intent Filter with all of the actions, data, and categories listed in the Intent, thus stealing sensitive account information by retrieving the data from the Intent.

Information Leakage. *OwnCloud* provides cloud-based file synchronization services to the user. By creating an account on the back-end server, user can sync selected files on the device and access synced files to browse, manage, and share. Our study indicates that OwnCloud app is vulnerable to leak sensitive information to other apps. One of its components obtains the account information and through a chain of Intent message passing, eventually logs the account information in an unprotected area of the memory card, which can be read by any other app on the device.

Privilege Escalation. *Ermete SMS* is a text messaging app with *WRITE_SMS* permission. Upon receiving an Intent, its *ComposeActivity* component extracts the payload of the given Intent, and sends it via text message to a number also specified in the payload, without checking the permission of the sender. This vulnerable component, thus, provides the *WRITE_SMS* permission to all other apps that may not have it.

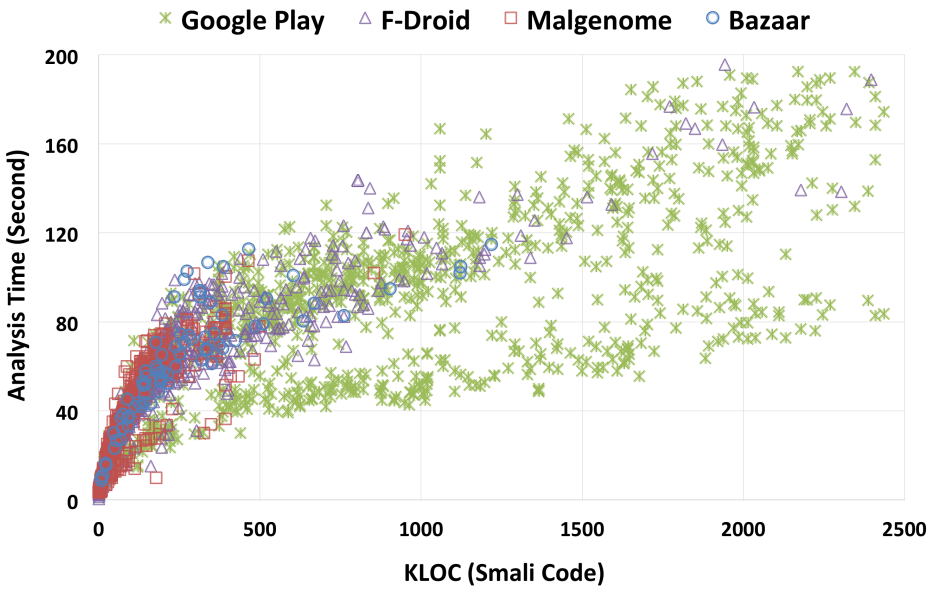


Figure 5.10: Scatter plot representing analysis time for model extraction of Android apps.

5.7.3 Results for RQ3 (Performance and Timing)

The next evaluation criteria are the performance benchmarks of static model extraction and formal analysis and synthesis activities. We used a PC with an Intel Core i7 2.4 GHz CPU processor and 4 GB of main memory, and leveraged Sat4J as the SAT solver during the experiments.

Figure 5.10 presents the time taken by Separ to extract app specifications for 4,000 real-world apps. This measurement is done on the data-sets collected from 4 repositories: Google Play, F-Droid, Malgenome, and Bazaar. The scatter plot shows both the analysis time and the app size. According to the results, our approach statically analyzes 95% of apps in less than two minutes. As our approach for model extraction analyzes each app independently, the total static analysis time scales linearly with the size of the apps.

Table 5.2: Experiments performance statistics.

Components	Intents	Intent Filters	Time (sec)	
			Construction	Analysis
313	322	148	260	57

Table 5.2 shows the average time involved in compositional analysis and synthesis of policies for a set of apps. The first three columns represent the average number of Components, Intents, and Intent filters within each analyzed bundle. The next two columns represent the time spent on transforming the Alloy models into 3-SAT clauses, and in SAT solving to find the space of solutions for each bundle. The timing results show that on average SEPAR is able to analyze bundles of apps containing hundreds of components in the order of a few minutes (on an ordinary laptop), confirming that the proposed technology based on a lightweight formal analyzer is feasible.

5.7.4 Results for RQ4 (Policy Enforcement)

The last evaluation criterion is the performance benchmark of SEPAR's policy enforcement. To measure the runtime overhead required for APE (i.e., policy enforcement), we have tested a set of benchmark applications. Our benchmark applications repeatedly perform several ICC operations, such as the *startService* method. We have handled uncontrollable factors in our experiments by repeating the experiments 33 times, the minimum number of repetitions needed to accurately measure the average execution time overhead at 95% confidence level. Overall, the execution time overhead incurred by APE for policy enforcement is $11.80\% \pm 1.76\%$, making the effect on user experience negligible. Note that using the runtime process instrumentation (See Section 5.6), our infrastructure only introduces overhead with the ICC calls, and does not have any overhead in terms of the non-ICC calls. Thus, in practice, the overhead introduced by our approach is significantly less than 11.80%.

5.8 Conclusion

This chapter presents a novel approach for automatic synthesis and enforcement of security policies, allowing the end-users to safeguard the apps installed on their device from inter-app vulnerabilities. The approach, realized in a tool, called SEPAR, combines static program analysis with lightweight formal methods to automatically infer security-relevant properties from a bundle of apps. It then uses a constraint solver to synthesize possible security exploits, from which fine-grain security policies are derived and automatically enforced to protect a given device. The results from experiments in the context of thousands of real-world apps corroborates SEPAR's ability in finding previously unknown vulnerable apps as well as preventing their exploitation.

Chapter 6

Incorporating Time in Permission Analysis and Enforcement of Android

The detection and prevention approaches described in Chapters 4 and 5, namely Covert and Separ, ignore temporal aspects of an attack during the analysis and enforcement. This chapter presents an improvement on those techniques by incorporating the notion of time.

6.1 Introduction

Popular mobile operating systems, such as Android, apply a permission-based model to patrol resources that each application is allowed to access. In this model, critical system and application resources are protected by an explicit permission, which then must be obtained by any application that would like to access the resources. Yet, in the past few years since the inception of Android, a number of flaws have been identified in its permission mechanism that can lead to serious security and privacy breaches [188]. A large body of research, thus, has been devoted to address detection and prevention of permission-induced attacks in Android

(§ Chapter 2).

The state-of-the-art approaches, however, fail to consider the temporal aspects of permission-induced attacks during the analysis and enforcement, thereby suffer from shortcomings that aggravate their effectiveness. Detection of several permission-induced attacks, such as those exploiting the *TOCTOU* (Time of Check to Time of Use) vulnerability in Android [463, 208], requires careful consideration of the *order* of events. Hence, existing *detection* techniques, which ignore the element of time in their analysis, are prone to miss important security breaches. Additionally, due to the highly dynamic state of an Android system, the identified security vulnerabilities may only be exploitable at specific time intervals, e.g., when some specific permissions are granted. Hence, the existing conservative *prevention* techniques, which regardless of the system state enforce security rules *permanently*, tend to produce plenty of false alarms. As a result, users can be unnecessarily disrupted, even in the absence of material security threats, and prevented from taking full advantage of the apps on their device.

Finally, the proposed approaches are mostly realized through modification of either the Android framework [114, 238, 379, 511, 252] or the implemenation logic of apps [75, 396, 535, 138]. But, such modifications are not necessarily expected, nor properly tested by the application developers, resulting in all sorts of undesirable side effects, such as app crashes and unexpected behaviors. To address this state of affairs, a pragmatic approach for detection and prevention should explicitly consider the temporal aspects of attack during analysis and enforcement. Moreover, the realization of the approach should be naturally compatible with the implementation practices in Android.

This chapter contributes a novel approach and accompanying tool suite, called TERMINATOR, short for **T**emporal P**ermi**ssio**n** **A**nalysis and enforcemen**t** framew**o**rk for And**r**oid. Unlike all prior techniques, TERMINATOR incorporates the notion of time as a first class entity in both detection and prevention of permission-induced attacks. Our approach has the

potential to greatly improve our ability to thwart permission-induced attacks by introducing the concept of *temporal permissions*, i.e., the temporary granting of permissions to apps. Specifically, constructed atop *temporal logic*, TERMINATOR leverages temporal permissions to (1) formulate dynamic aspects of the system over time and reason about the security properties thereof as the system transitions from one state to another (risk detection), and (2) regulate app permissions at runtime based on the current state of the system (risk prevention).

TERMINATOR provides a safe, reliable, yet non-disruptive approach to protect mobile users against permission misuses. Upon receiving a permission request from an app, TERMINATOR evaluates the security posture of the system with respect to the current state of the granted-permission configuration as well as potential threats conservatively identified via the state-of-the-art static analysis tools. If granting the requested permission does not lead to a real security threat given the current state of the system, TERMINATOR *leases* (i.e., temporarily grants) that permission to the requester. The leased permission is then automatically revoked as soon as a change in the system status is observed that may lead to realization of an identified security threat. TERMINATOR uses TLA+ model checker (TLC) [553] as an analysis engine for temporal permissions. To prevent permission-induced attacks, TERMINATOR relies on the Android's dynamic permission mechanism without needing to make any modification to the Android framework or the implementation logic of apps.

Our experiments indicate that TERMINATOR is up to 68% more successful in preventing permission-induced attacks, while issuing significantly less (56-100%) false alarms. It also causes less disruption in the availability of permission-protected app functionality due to restrictive permission configurations.

To summarize, this chapter makes the following contributions:

- *Theory*: To the best of our knowledge, this is the first attempt at leveraging temporal logic

and incorporating the notion of time in modeling and analyzing the security properties of Android;

- *Tool*: A fully automated framework, TERMINATOR, that realizes the idea of temporal permissions for Android, which we have made publicly available [42];
- *Experiments*: Empirical evaluation of the approach on real-world Android apps demonstrating its efficacy.

The remainder of this chapter is organized as follows. Section 6.2 motivates our research through various examples of permission-induced security attacks. Section 6.3 formally specifies those attacks and introduces our approach to effectively thwart them. Section 6.4 provides details of our approach and its implementation. Section 6.5 presents the experimental evaluation of the research. The chapter concludes with an outline of the related research and future work.

6.2 Permission-Induced Attacks

To motivate the research and demonstrate the need for temporal permissions, we describe four types of permission-induced security attacks in Android, identified in prior research [188]. Permission-induced attacks are security breaches enabled by Android permissions misuse. This section elaborates on the attack scenarios summarized in Figure 6.1. We will later show how temporal permissions help thwart these attack scenarios with minimum disruption.

6.2.1 Privilege Escalation

Privilege escalation occurs when an application with less privilege is not restricted from accessing components of a more privileged application [115]. In the case of the particular

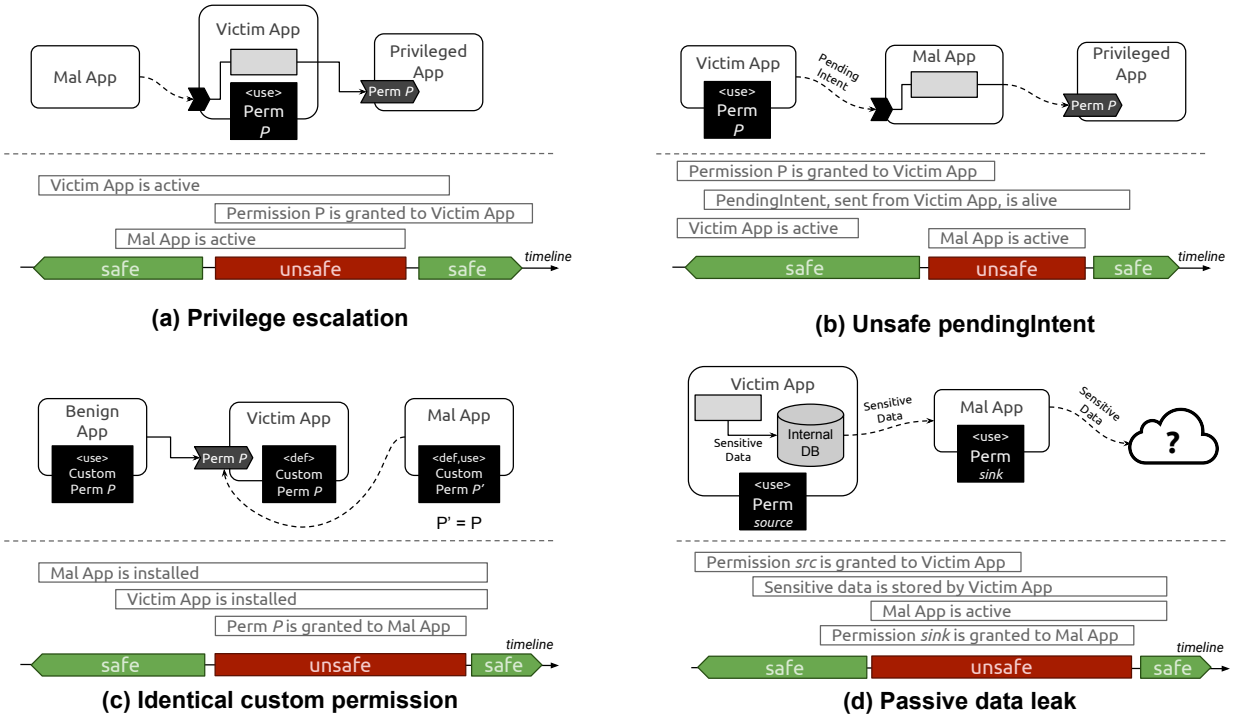


Figure 6.1: Examples of permission-induced security attacks that could be thwarted using temporal permission. In each example, the top of the figure shows the elements involved in the attack, and the bottom shows a possible attack scenario over time. The permission can be leased, i.e., temporarily granted, during the "safe" time slots. In all of the scenarios, the malicious communications are distinguished by dashed lines.

example shown in Figure 6.1(a), Mal App can indirectly reach the permission-protected interface of the Privileged App, by exploiting the vulnerability of the Victim App — that is, an unprotected exposed interface, shown to be quite common in the app markets [143]. The *collusion attack* [437], carried out by multiple malicious apps through combining a set of permissions to perform unauthorized actions, is also categorized under this group of attacks.

The state-of-the-art techniques for preventing inter-app security attacks (§ Chapter 2) conservatively assume that this vulnerability is exploitable, as soon as the apps are installed on the device. However, a more careful look at the timeline of the attack scenario, shown in Figure 6.1(a), would reveal that the presented security vulnerability is only exploitable during the "unsafe" time slot, where the following two conditions hold at the same time: (1) the malware and victim apps are both active, i.e., running in foreground or background, and

(2) permission P is granted to *Victim App*. If those applications are installed but not active, the vulnerability cannot be exploited. On the other hand, if permission P is not granted to the victim app, the permission-protected interface of the other app is not accessible.

6.2.2 Unsafe PendingIntent

In Android, *PendingIntent* is a wrapper around Intent that enables performing the Intent's action in future, even if the original app that sent the Intent is not active anymore. For this purpose, Android transfers the permission and identity (UID) of the sender app to the target app that receives the PendingIntent. As such, careless use of PendingIntent can lead to severe security consequences. Examples include the privilege leakage vulnerability in the Android Settings application (CVE-2014-8609) [22].

For this reason, Android's developer guidelines strongly discourage using blank base PendingIntents: *"the base Intent you supply should have the component name explicitly set to one of your own components, to ensure it is ultimately sent there and nowhere else* [35]." Despite that, many app developers fail to follow such security principle in action.

Figure 6.1(b) shows an example of using unsafe PendingIntent, exploited by *Mal App* to illegally access permission-protected interface provided by *Privileged App*. This example is similar to the privilege escalation attack, illustrated in Figure 6.1(a), except that the conditions for exploitability are more relaxed in two ways: first, the *Victim App* does not need to be necessarily active, and second, its permission P may be revoked prior to malware executing the wrapped Intent.

6.2.3 Identical Custom Permission

Besides the predefined built-in permissions, such as `SMS`, `LOCATION`, etc., Android apps can define their own custom permissions and request those permissions from other apps. However, the custom permission model suffers from a security vulnerability rooted in a design flaw: *"If two apps define the same custom permission, whichever app is installed first is the one whose definition is used"* [78].

A malicious app can exploit the custom permission vulnerability to illegally access the interface of another app, protected by that custom permission. A sample attack scenario is shown in Figure 6.1(c). In this example, *Victim* and *Mal* apps have both defined the same custom permission, i.e., the names of permissions P and P', defined by the `<permission>` element in the manifest are identical. Since the malicious app is installed prior to the victim app, permission P', defined in the manifest of the *Mal App* at the Normal level, is the one recognized by the Android framework. Consequently, *Mal App* can access the interface defined by the victim app, which is intended to be only accessible to those requesting the custom permission P, such as *Benign App*. The custom permission breach can happen even though the permissions with the same name have different protection levels.[1] Essentially, the malware can define a permission with `normal` protection level, rather than `dangerous` or `signature` protection level, to evade user attention and interaction.[2]

6.2.4 Passive Data Leak

A passive data leak occurs when an app does not properly protect its internal database that contains sensitive data [587]. A malware can exploit this vulnerability by retrieving the stored data, without having the permission needed for directly accessing such sensitive information.

[1] The protection level indicates the trustworthiness of an application that may be granted this permission.

[2] The system automatically grants *normal* permission to a requesting application at installation, without asking for the user's explicit approval.

Thereafter, the malicious app can transfer the sensitive data to an untrustworthy location.

Figure 6.1(d) depicts an example of passive data leakage. In this attack scenario, *Victim App* with an access to *Sensitive Data* due to obtaining *source* permission (e.g., permission for accessing phone identifier), stores this information in its internal database, which is not properly protected. As a result, *Mal App* can retrieve the sensitive data and send it to an untrustworthy location, if it has been granted with a *sink* permission (e.g., SMS permission).

From the attack scenarios shown in Figure 6.1, we can see that the notion of time is critical in the precise description of all attacks. In other words, a precise analysis should keep track of the security posture of the system as it moves from one state to another over time. Hence, to formally describe the attack scenarios we need to formulate the system properties in terms of time. For this purpose, we leverage temporal logic, as described in the next section.

6.3 Temporal Permission

In this section, we describe a formal model of the Android system with a focus on its security properties such as permission status. Using this model, we then define a set of safety formulas corresponding to the permission-induced security attacks, described in the previous section. Finally, we demonstrate that control of the permissions granted to apps is sufficient for effectively thwarting all such attacks.

6.3.1 Modeling the Android System

We model the Android system as a *Kripke Structure*, a variation of *Transition System* that mathematically models dynamic systems [150]. Nodes represent the reachable states of the system, and edges represent state transitions. Each node is also labeled with a set of

```
/** Determines whether app is installed on the device */
Installed(app)
/** Determines whether app is running, either in the foreground or background */
Active(app)
/** Determines whether permission is declared by app */
Declared(app, permission)
/** Determines whether permission is requested by app */
Requested(app, permission)
/** Determines whether permission is (requested by and) granted to app at runtime */
Granted(app, permission)
/** Determines whether app defines an exposed interface reaches to permission-required capability
    */
Exposed(app, permission)
/** Determines whether app sends a pendingIntent with blank base Intent (w/o explicit target)
    containing permission-required data payload */
BlankPI(app, permission)
/** Determines whether app retrieves permission-protected data (e.g., IMEI, location, etc)*/
Retrieve(app, permission, data)
/** Determines whether app sends data through permission-protected channels (e.g., SMS, Internet,
    etc.) */
Send(app, permission, data)
/** Determines whether data is stored by the app in an unprotected database*/
StoreUnprotected(app, data)
```

Figure 6.2: Atomic Propositions (AP) defined for modeling the security properties of Android system.

properties that hold in the corresponding state.

More formally, we model the system as a 4-tuple $M = (S, I, R, L)$, where S is a set of states, $I \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is a total transition relation, and L is the labeling function that assigns to each state the subset of properties that are valid in the state. To define the labeling function, we first need to define *atomic propositions*, or AP, which is a set of boolean expressions that specify the properties of the system S. For instance, $Granted(app_a, perm_p)$ states that the permission p is granted to the app a. We use atomic propositions to define the labeling function as follows: $L : S \times AP \rightarrow \{true, false\}$. In other words, for each system state $s \in S$, the labeling function, L, determines whether the atomic proposition $ap \in AP$ holds at that state or not.

Modeling the Android operating system in its entirety with all its compound structures would be infeasible. We thus concentrate on the parts that are particularly relevant to the permission mechanism—how permissions are granted and maintained, and how they constrain the behavior of an application. Figure 6.2 provides the set of atomic propositions, defined as

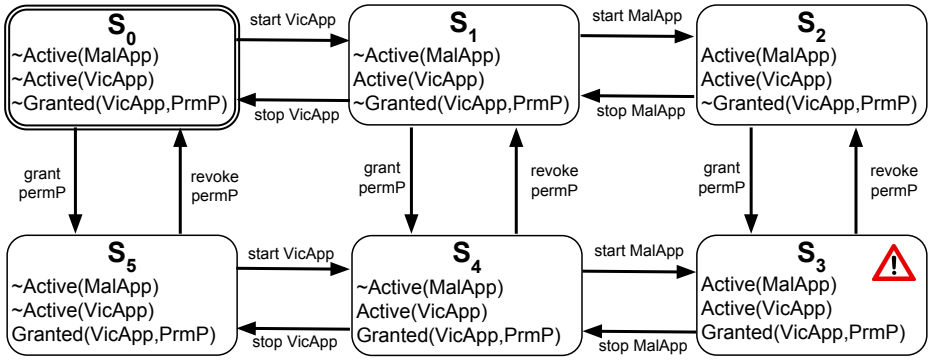


Figure 6.3: A subset of Kripke structure for a hypothetical Android system

parameterized predicates. For instance, $Granted(app, permission)$ has two arguments, the first one is an *app* and the second one is a *permission* requested by that app.

As a concrete example, Figure 6.3 shows a small subset of the Kripke structure of a hypothetical Android system with six states: the initial state $s_0 \in I$ along with five other states, $s_{1-5} \in S$. Also, two kinds of actions triggering state transitions $r \in R$, are shown here, namely *start/stop* actions that alter the system configuration apropos of the *Active* proposition and *grant/revoke* actions that alter the system configuration apropos of the *Granted* proposition.

Interacting with the user and other environmental actors, Android system moves from one state to another in the Kripke structure over time. As a result, under specific sequence of actions, the system can move to an *unsafe* state—a state that violates the security of the system. For instance, in Figure 6.3, S_3 represents an unsafe state, corresponding to the red time slot of the privilege escalation attack scenario shown in Figure 6.1(a). S_3 is unsafe since a privilege escalation attack is possible when the system moves to this state.

6.3.2 Formulating Safety Rules

We specify *safety rules* in terms of conditions that need to hold throughout the states of the system. For this purpose, we leverage *linear-time temporal logic (LTL)*. Temporal logics

enable specifying a system's behavior as it evolves over time. Indeed, in temporal logics, the truth of a statement is not fixed in the semantics, rather relies on the point in time when it is considered. Temporal logics, thus, besides the usual logical operators, such as *and*, *or*, *not*, and *implies*, also contain temporal operators, such as *eventually*, *always*, and *until*. For instance, "SMS permission can *eventually* be granted to the Messenger app", or "SMS permission should not be granted to the Messenger app *until* the Malware app is terminated", are two examples of such statements that can be expressed using temporal logic. In LTL, time is represented by a sequence of discrete time steps.

Privilege Escalation: The first formula, $SafetyRule_{PE}$, specifies the conditions needed to hold in order to prevent the privilege escalation attack (Recall Section 6.2.1).

$$\begin{aligned} &SafetyRule_{PE} \models \\ &\forall\ app_{vic}, app_{mal} \in Apps,\ p \in Permissions : \\ &Vul_{PE}(app_{vic}, app_{mal}, p) \Rightarrow \\ &\Box\neg(Granted(app_{vic}, p) \wedge Active(app_{mal})) \end{aligned} \tag{6.1}$$

The precondition of rule 6.1 checks for the privilege escalation vulnerability (Vul_{PE}), which is formulated as follows:

$$\begin{aligned} &Vul_{PE}(app1, app2, perm) := \\ &Requested(app1, perm)\ \wedge \neg Requested(app2, perm) \\ &\wedge Exposed(app1, perm) \end{aligned}$$

According to the above expression, for the given two apps, $app1$ and $app2$, and the Android permission $perm$, the hosting Android device is vulnerable to the privilege escalation attack, if $app1$, granted permission $perm$, exposes an unprotected interface to a capability protected by $perm$, while $perm$ is not requested by the other app.

$SafetyRule_{PE}$ states that the system is safe against the privilege escalation attack, if none of the system's apps expose the aforementioned vulnerability, or otherwise the unsafe permission of the vulnerable app should remain as "not granted" as long as malicious app is active. Note the usage of temporal operator $\Box$, read *henceforth*[3], in the safety rule specified in formula 6.1, which states that the conditional consequent should hold in all future states.

Unsafe PendingIntent: The second formula, $SafetyRule_{UPI}$, specifies the conditions needed to hold in order to prevent the attacks exploiting an unsafe PendingIntent (Recall Section 6.2.2).

$$\begin{aligned} &SafetyRule_{UPI} \models \\ &\forall\ app_{vic}, app_{mal} \in Apps,\ p \in Permissions : \\ &Vul_{UPI}(app_{vic}, app_{mal}, p) \wedge Granted(app_{vic}, p) \Rightarrow \\ &\neg\Diamond Active(app_{mal}) \end{aligned} \tag{6.2}$$

The precondition of rule 6.2 checks for the unsafe PendingIntent vulnerability (Vul_{UPI}), which is formulated as follows:

$$\begin{aligned} &Vul_{UPI}(app1, app2, perm) := \\ &Requested(app1, perm)\ \wedge \neg Requested(app2, perm) \\ &\wedge\ BlankPI(app1, perm) \end{aligned}$$

Unlike the privilege escalation exploits, PendingIntent exploits do not require the breached permission to be granted to the vulnerable app prior to the attack. This is essentially because the required permission is already transferred to the mal app through the PendingIntent. Hence, the temporal operator $\Diamond$, read *eventually*[4], is used in the conditional consequent. According to formula 6.2, the system is safe against exploiting unsafe pendingIntent, if there is no such vulnerability or the vulnerable app is not granted with the breached permission.

[3] $\Box\ \Phi$ means Φ is true at all future states

[4] $\Diamond\ \Phi$ means Φ is true in some future state

Otherwise, the system is unsafe as soon as the mal app is activated.

Identical Custom Permission: The third formula, $SafetyRule_{ICP}$, specifies the conditions that need to hold to prevent the attacks exploiting the identical custom permission vulnerability (Recall Section 6.2.3).

$$
\begin{aligned}
&SafetyRule_{ICP} \models \\
&\forall app_{vic}, app_{mal} \in Apps,\ p \in Permissions : \\
&Vul_{ICP}(app_{vic}, app_{mal}, p) \wedge Installed(app_{mal}) \Rightarrow \\
&\Box\neg(Granted(app_{mal}, p) \wedge Installed(app_{vic}))
\end{aligned}
\tag{6.3}
$$

The precondition of rule 6.3 checks for the unsafe identical custom permission vulnerability (Vul_{ICP}), which is formulated as follows:

$$
\begin{aligned}
&Vul_{ICP}(app1, app2, perm) := \\
&Declared(app1, perm) \wedge Declared(app2, perm)
\end{aligned}
$$

Recall from Section 6.2.3 that the order of installation matters in the case of identical custom permission. To formulate this chronological order, *henceforth* temporal operator ($\Box$) is used. According to rule 6.3, if (a potentially malicious) application with a declared custom permission p has been already installed on the device, no other app declaring the same permission is allowed to be installed, as long as that permission is granted to the first app.

Passive Data Leak: The last formula, $SafetyRule_{PDL}$ specifies the conditions needed to hold in order to prevent the leakage of sensitive data stored in an unprotected app database

(Recall Section 6.2.4).

$$
\begin{aligned}
&SafetyRule_{PDL} \models \\
&\forall\ app_{vic}, app_{mal} \in Apps, p_1, p_2 \in Permissions : \\
&Vul_{PDL}(app_{vic}, app_{mal}, p_1, p_2) \wedge Granted(app_{vic}, p_1) \Rightarrow \\
&\neg\Diamond Granted(app_{mal}, p_2)
\end{aligned}
\tag{6.4}
$$

Passive data leak vulnerability, formally defined below (Vul_{PDL}), occurs when a sensitive (i.e.,permission-protected) data is sent out of the device by another app, via a (typically) permission-protected channel:

$$
\begin{aligned}
&Vul_{PDL}(app_{src}, app_{snk}, p_{src}, p_{snk}) := \exists\ data \in PhoneData : \\
&Requested(app_{src}, p_{src})\ \wedge Retrieve(app_{src}, p_{src}, data) \\
&\wedge StoreUnprotected(app_{src}, data) \\
&\wedge Requested(app_{snk}, p_{snk}) \wedge Send(app_{snk}, p_{snk}, data)
\end{aligned}
$$

According to rule 6.4, the system is safe against the passive data leak, if either there is not such a vulnerability or the vulnerable app has never been granted the permission to access sensitive data. Otherwise, the system is unsafe as soon as the malicious app is granted the permission, allowing the app to send data out of the device.

6.3.3 Leasing Temporal Permissions

To keep the Android device safe against the attack scenarios described in Section 6.2, one should guarantee that the corresponding safety rules hold at all times. A careful revisit of the safety rules (Rules 6.1–6.4) reveals that the ¬*Granted(app, permission)* proposition

is incorporated in all formulas.[5] Therefore, permanently revoking specific permissions can guarantee the safety of the system. This approach, however, is too conservative as it revokes app permission even when the other criteria needed for exploitation of security vulnerability is not satisfied. In other words, since $\neg Granted$ proposition is qualified in terms of time, it is not necessary to satisfy it over all system states. Instead, the app permission should only be revoked during specific unsafe states, and can be granted in the rest of system states.

Based on this intuition, we propose a defense mechanism against permission-induced attacks, called TERMINATOR. Upon receiving a permission request from an app, TERMINATOR *leases* (i.e., temporarily grants) that permission to the requester, only if granting the requested permission does not violate any safety rule. The leased permission is automatically revoked as soon as a change in the system status could lead to the violation of the safety rules.

To appreciate the advantage of temporal permissions, consider the *Victim App* in Figure 6.1(a) that requires the permission P to accomplish its main functionality (e.g., `Location` permission in a navigator app). Permanently revoking of the permission P by the existing approaches makes this app practically useless. However, a careful investigation of the attack scenario makes it clear that the permission P should only be revoked during the "unsafe" time slot. In other words, leasing permission P during the "safe" time slots cannot pose a security risk, yet enables the user to take the full advantage of this app. As a result, an analysis and enforcement approach based on temporal permissions, provides **less disruption** in the normal execution flow of apps.

Another significant advantage of TERMINATOR, attributed to its *permission-based* approach, is the **high coverage** of permission-induced attacks that it can thwart. The existing enforcement techniques only consider certain types of breaches, thereby fail to protect those attacks carried out differently. For instance, according to our survey (§ Chapter 3), the ma-

[5] In the safety rule 6.2, $Vul_{UPI}() \wedge Granted() \Rightarrow \neg\Diamond Active()$ is logically equivalent to $\neg Vul_{UPI}() \vee \neg Granted() \vee \neg\Diamond Active()$.

jority of Android security research approaches only consider Intent-based communications to identify inter-component security vulnerabilities, while there are other potentially vulnerable communication methods, such as data-sharing or remote procedure call, which could be exploited by malicious apps. Through meticulous regulation of the common element in all such permission-induced attacks, i.e, permissions, TERMINATOR is able to effectively thwart all of them, regardless of the specific channels exploited by the attackers.

The third distinguishing characteristic of TERMINATOR is its **reliability**. By leveraging the dynamic permissions in Android, our approach avoids any unintended side effects, as it is naturally compatible with the development constraints imposed by the latest versions of Android. Specifically, with the introduction of dynamic permission mechanisms in the latest versions of Android, an app should continue to work properly even if the user does not grant some of the permissions requested by the app [43]. The app in such a case, of course, performs in a downgraded mode, i.e., with some functionalities disabled. Here, we leverage the same feature to revoke an unsafe permission without risking app failure.

6.4 TERMINATOR

In the previous section, we introduced the idea of using temporal permissions to provide an effective, yet non-disruptive, defense against permission-induced attacks. This section describes how we realized this idea using Android's dynamic permission mechanism.

6.4.1 Approach Overview

Figure 6.4 depicts a high-level overview of TERMINATOR, comprised of two phases: *Analysis*, and *Enforcement*. The analysis phase runs once for a set of apps and identifies the potential security risks threatening the Android System (risk detection). The enforcement phase runs

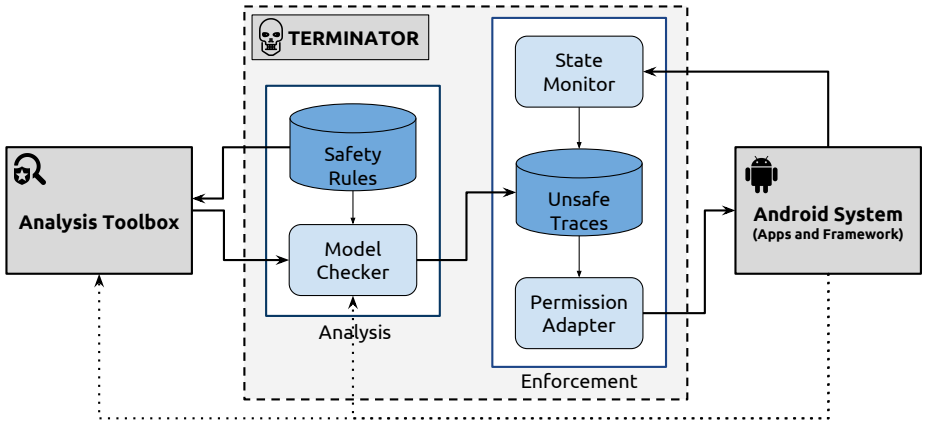


Figure 6.4: Overview of TERMINATOR framework

continuously and prevents the security threats to occur at run-time (risk prevention). The enforcement components are deployed as an Android app embedded in the device, while the analysis components are deployed externally.

To identify the potential security threats, TERMINATOR relies on the state-of-the-art static analysis tools [311], represented as *Analysis Toolbox* in Figure 6.4. Recall from Section 6.3.3 that the safety rule formulas include the specification of security vulnerabilities. In TERMINATOR, Analysis Toolbox is responsible for analysis of the installed apps and detecting any instance of the atomic propositions (listed in Figure 6.2) constituting the security vulnerabilities of the given $SafetyRules$.

As discussed in the beginning of this Chapter, it is overly conservative to assume that identified security risks could be realized in all states of the Android system. To accurately identify the exact conditions under which the identified risks can be realized, TERMINATOR relies on a temporal *Model Checker*, and tracks down any counterexample violating the safety rules with respect to the state transitions of the system. The analysis results are then stored in the *Unsafe Traces* database.

Two components are involved in the enforcement of safety rules: *State Monitor* and *Permission Adapter*. State Monitor keeps track of the system states, particularly those affecting

the security properties of the system, and attempts to match the current state of the system against the Unsafe Traces provided by Model Checker. Upon detecting the possibility of a security attack, *Permission Adapter* adopts appropriate countermeasures to prevent the attack from occurring. For this purpose, Permission Adapter refrains to lease the requested permissions enabling the attack, or revokes the previously leased permissions. Once the system moves to a safe state, the adapter re-grants the previously revoked permissions.

In the next two sections, we provide a more detailed description of the components involved in the analysis and enforcement phases.

6.4.2 Analysis

The set of security risks identified by the Analysis Toolbox are only realizable in specific system states. The analysis phase of TERMINATOR involves identifying those state transitions of the Android system that lead to realization of the security risks.

Recall from Section 6.3 that we modeled an Android system as a transition system $M = (S, I, R, L)$ and formulated the safety conditions of the system as a set of temporal rules, i.e., $SafetyRules$. Given an Android system M, and a temporal safety rule $r \in SafetyRules$, TERMINATOR is intended to ensure $M \models r$. For this purpose, the *Model Checker* attempts to find any violations, in terms of counterexamples, of the temporal rules.

To realize the Model Checker component, TERMINATOR uses TLC [553], which is intended to check the specifications written in TLA+. TLA, or Temporal Logic of Actions, and its extension TLA+, are originally designed to provide a simple and practical language for high-level specifications of concurrent and distributed systems [299]. TLA specifies the behavior of a system as a sequence of states, where each state is an assignment of values to variables. To model the state transitions, TLA defines next-state relation describing how variables are

changed in each step. For this purpose, it uses the primed variable to represent the value of the variable in the next state.

For instance, consider the following TLA+ formula, defined in TERMINATOR to model the state transition of the Android system that occurs due to granting the permission *perm* to an *app*:

$$
\begin{aligned}
&Grant(app, perm) \triangleq \\
&\land perm \in RequestedPerms[app] \\
&\land permStat[\langle app, perm \rangle] = \text{“}Revoked\text{”} \\
&\land permStat' = [permStat\ EXCEPT\ ![\langle app, perm \rangle] = \text{“}Granted\text{”} \\
&\land \text{UNCHANGED}\ appStat
\end{aligned}
\tag{6.5}
$$

In this TLA+ formula, two variables are used to model the state of the system: *permStat* representing the current state of the permission configuration for each app, and *appStat* specifying the state of each individual app. Types of those two variables are formally defined via the following type invariant assertion:

$$
\begin{aligned}
&TypeInvariant \triangleq \\
&\land appStat \in [Apps \rightarrow \{\text{“}Running(Active)\text{”}, \text{“}Terminated\text{”}\}] \\
&\land permStat \in [(Apps \times Perms) \rightarrow \{\text{“}Granted\text{”}, \text{“}Revoked\text{”}\}
\end{aligned}
\tag{6.6}
$$

Formula 6.5 defines the *Grant* action as an operator with two arguments, *app* and *perm*. The first two lines of the formula specify the state conditions *enabling* the Grant action: (1) *perm* should be among the requested permissions of the *app*, and (2) *perm* should not be previously granted to the *app*. If both conditions are satisfied in a state, the system can move from that state to the next state described in the third line of the formula.

Using the action formulas, such as formula 6.5, the *Next* state of the system is defined as

the disjunction of all possible next-state actions:

$$
\begin{aligned}
&Next(app, perm) \triangleq \\
&\exists\ app \in Apps : Run(app) \vee Terminate(app) \vee \\
&\exists\ perm \in Perms : Grant(app, perm) \vee Revoke(app, perm) \vee \ldots
\end{aligned}
\tag{6.7}
$$

Note that in this formula only four next-state actions, including *Grant* action defined in formula 6.5, are shown. To see the full list of TLA+ next-state actions defined for TERMINATOR, refer to our online documentation [42].

Finally, the specification of Android system is formulated as follows:

$$Spec \triangleq Init \wedge \Box[Next]_{\langle appStat, permStat \rangle} \tag{6.8}$$

In this TLA+ formula, *Init* represents the initial state of the system (not shown here), where all apps are inactive and all permissions are revoked.

According to the specification formula, there are two possibilities for the next state of a system state: (1) either one of the actions incorporated in the *Next* formula (Formula 6.7) will take place, or (2) the state variables, namely *appStat* and *permStat*, will leave unchanged.[6]

In addition to modeling the behavior of an Android system using the above formulas, Model Checker should reason about the security properties of the system. More specifically, Model Checker should identify any sequence of actions (i.e., state transitions) that leads to the violation of safety rules. For this purpose, the following theorem is defined as the system invariant needed to be checked:

$$\text{THEOREM } Spec \Rightarrow TypeInvariant \wedge \bigwedge_{rule \in SafetyRules} \tag{6.9}$$

According to theorem 6.9, given the specification of an Android system (*Spec*), all system

[6] In TLA+, those steps leaving state variables unchanged are called *stuttering* steps

behaviors should satisfy the safety rules, such as Rules 6.1–6.4 defined in Section 6.3.

In this formula, $TypeInvariant$ is also added to the theorem to ensure that the model checker only explores the valid states of the system, as specified by definition 6.6.

Having the specification of Android system and safety rules in TLA+, TLC model checkers verifies formula 6.9. For this purpose, TLC explores reachable states, looking for any unsatisfying safety rules. In case of finding a violation, it reports the minimal-length trace from an initial state that leads to an unsafe state. The unsafe traces are then stored in the *Unsafe Traces* database in the Android device.

6.4.3 Enforcement

State Monitor keeps track of the Android system states, looking for violating traces that match any of the traces stored in the database. The monitor component is realized as an Android app that uses Xposed module for collection of runtime data [44]. The Xposed module instruments the root process of Android, without making any changes in the apps' APK files. The implemented module intercepts those events corresponding to TLA+ action operators defined in Section 6.4.2. Examples of monitored events include but not limited to: granting/revoking of a permission via a permission request dialog, granting/revoking of a permission via system settings, and running or terminating an app.

If the State Monitor finds a match, it marks the matching unsafe trace in the database, which triggers the *Permission Adapter* component. The goal of this component is to regulate the permission configuration of apps such that the system remains in a safe state. Since the identified security risks are all permission-induced, it is sufficient to revoke the corresponding risk-enabling permissions to thwart the attack.

In its effort to thwart an attack, *Permission Adapter* may encounter a situation in which

there are multiple candidate permissions for revocation. For instance, consider the inter-app data leak, where a sensitive data protected by the *source* permission in App1 is leaked through a sensitive channel protected by the *sink* permission in App2. In this case, permission Adapter has two choices, since revoking either of the source or sink permissions would prevent the leak form happening. To provide an *effective* yet *non-disruptive* defense against permission-induced attacks, Permission Adapter applies the following method to select the best permission.

It first calculates two scores for each candidate permission:

Risk score that reflects the number of attacks enabled by granting the permission. A permission with high involvement in the identified security threats would have a higher Risk Score. The risk score is calculated based on the analysis results of Model Checker.

Usage score that indicates the usage frequency of the app requesting the permission. If the permission is requested by an app that is highly used by the user, that permission would receive a high usage score. Unlike the risk score, the usage score is based on user behavior and calculated by the State Monitor component using the Android's `USAGE_STATS_SERVICE` APIs.

Afterwards, Permission Adapter selects the permission with the highest *Revoke Score*, which is calculated as a function parameterized by both the Risk Score and the reverse of the Usage Score, i.e., $\mathcal{F}(RiskScore, UsageScore^{-1})$. In other words, to prevent a security risk, Permission Adapter revokes permissions with higher security risks that are requested by less-frequently-used apps.

Since Terminator is not aware of the user's context, in certain situations the user may disagree with the way in which it prioritizes permissions for revocation. This might happen, for example, when the user anticipates using a rarely used app. Our implementation allows the user to override the Terminator's decision by adding exception rules. Such rules exclude specific app permissions from being revoked, even if they violate the safety rules.

6.5 Evaluation

Our evaluation of TERMINATOR addresses the following research questions:

RQ1. *Coverage*: How does TERMINATOR compare against alternative approaches in preventing the variety of permission-induced attacks?

RQ2. *Disruption*: How effective is TERMINATOR in reducing the unnecessary disruptions due to unavailability of permission-protected app functionality?

RQ3. *Applicability & Reliability*: What percentage of Android apps are compatible with TERMINATOR? Does the temporal enforcement of TERMINATOR cause any unexpected behaviors?

RQ4. *Performance*: What are the performance characteristics for each phase of TERMINATOR?

6.5.1 RQ1: Coverage

For a thorough evaluation, we compared the coverage of TERMINATOR with the other state-of-the-art approaches, enumerated in Table 6.1 under the "*Alternative Approaches*" column. We considered two criteria in selecting other approaches for our comparative analysis. First, the approach should support both detection and prevention of security attacks.[7] Second, the approach should provide a publicly available tool suite. In accordance with the above criteria, we selected three alternative approaches intended to prevent permission-induced security attacks, namely SEPAR [83], SEALANT [303], and DELDroid [238]. SEPAR enforces fine-grained security policies, synthesized by a SAT-based constraint solver, to prevent capability

[7]Approaches such as IccTA [308], JITANA [494], COVERT [80] DialDroid [111], etc. are excluded in our study as they only perform detection, not prevention.

Table 6.1: Ability of TERMINATOR in preventing permission-induced attacks in comparison with alternative approaches.

#	Permission-Induced Attack Type (Subtype)	Data Set	Involved Permissions	TERMI--NATOR	Alternative Approaches [83]	[238]	[303]
1	Custom Permission	DD	STORAGE, SMS	☒	☐	☐	☐
2	Privilege Escalation	DB	LOCATION, STORAGE	☒	☒	☒	☒
3	Passive Content Leak (CP)	DD	STORAGE, SMS	☒	☐	☐	☐
4	Passive Content Leak (CP)	DD	SMS	☒	☐	☐	☐
5	Passive Content Leak (CP)	DD	STORAGE, SMS	☒	☐	☐	☐
6	Privilege Escalation	DB	READ_PHONE_STATE, STORAGE	☒	☒	☒	☒
7	Custom Permission	DD	STORAGE, SMS	☒	☐	☐	☐
8	Privilege Escalation (AH)	DD	STORAGE, SMS	☒	☐	☒	⊙
9	Privilege Escalation (DCL)	DD	STORAGE, SMS	☒	☐	☒	⊙
10	Custom Permission	DD	SMS	☒	☐	☐	☐
11	Passive Content Leak (CP)	DD	STORAGE, SMS	☒	☐	☐	☐
12	Privilege Escalation (PR)	SP	WAKE_LOCK	☒	☐	☒	☐
13	Privilege Escalation (PR)	DD	SET_WALLPAPER	☒	☐	☒	☐
14	Custom Permission	DD	STORAGE, SMS	☒	☐	☐	☐
15	Privilege Escalation	SL	SMS	☒	☒	☒	☒
16	Passive Content Leak (CP)	DD	STORAGE, SMS	☒	☐	☐	☐
17	Privilege Escalation (BT)	SP	WAKE_LOCK	☒	☐	☒	☐
18	Privilege Escalation	SL	LOCATION	☒	☐	☒	☒
19	Privilege Escalation (SH)	DD	SMS	☒	☐	☒	☒
20	Privilege Escalation (PR)	DD	LOCATION, SMS	☒	☒	☒	⊙
21	Privilege Escalation	SL	CONTACTS	☒	☐	☒	☒
22	Custom Permission	DD	STORAGE, SMS	☒	☐	☐	☒
23	Privilege Escalation (PR)	DD	LOCATION, SMS	☒	☒	☒	☒
24	Privilege Escalation (MAL)	DD	SMS	☒	☒	☒	☐
25	Privilege Escalation	DB	LOCATION, STORAGE	☒	☒	☒	☒
26	Privilege Escalation	SL	LOCATION	☒	☐	☒	☐
27	Privilege Escalation (PR)	DD	SMS	☒	☐	☒	☒
28	Privilege Escalation	DB	READ_PHONE_STATE, STORAGE	☒	☒	☒	☒
29	Passive Content Leak (CP)	DD	SMS	☒	☐	☐	☐
30	Privilege Escalation	DB	READ_PHONE_STATE, STORAGE	☒	☒	☒	☒
31	Privilege Escalation (PR)	DD	SMS	☒	☒	☒	☐
32	Privilege Escalation (PR)	DD	LOCATION	☒	☐	☒	☒
33	Privilege Escalation (DCL)	DD	STORAGE, LOCATION	☒	☐	☒	☐
34	Passive Content Leak (CP)	DD	STORAGE, SMS	☒	☐	☐	☐
35	Privilege Escalation (AH)	DD	STORAGE	☒	☐	☒	☐
36	Custom Permission	DD	STORAGE, SMS	☒	☐	☐	☐
37	Privilege Escalation	DB	READ_PHONE_STATE, SMS	☒	☒	☒	☒
38	Privilege Escalation (AH)	DD	STORAGE	☒	☐	☒	☐
39	Privilege Escalation	DB	ACCESS_FINE_LOCATION	☒	☒	☒	☒
40	Privilege Escalation	DB	LOCATION, STORAGE	☒	☒	☒	☒
41	Custom Permission	DD	SMS	☒	☐	☐	☐
Total thwarted attacks				41	13	27	16
Coverage (true-positive rate)				100%	31.7%	65.9%	39.0%

☒(☐): attack scenario is (not) prevented by the approach, ⊙: the approach crashed during the analysis
Attack Subtypes: PR: Permission Re-Delegation, AH: Activity Hijack, SH: Service Hijack, BT: Broadcast Theft, MAL: Malicious Activity Lunch, DCL: Dynamic Class Loading, Content Provider (CP)
Data-sets: DD: DELDroid [238], SL:SEALANT[303], SP:SEPAR[83], DB: DroidBench [67]

leaks. SEALANT extends Android framework to provide an interceptor that blocks potentially malicious intents. Finally, DELDroid uses a multiple-domain matrix to eliminate the security vulnerabilities violating the least-privilege property of the system.

To eliminate bias in favor of TERMINATOR, we built a collection of subject apps consisting of the apps used in the evaluation of the three mentioned prior approaches as well as a reputable benchmark collection, namely DroidBench [67]. The resulting dataset consisted of a collection of 255 subject apps with known security issues. Out of this collection of apps,

we selected those that target Android 6.0 (API level 23) or newer versions. Older versions of the Android framework provide just a static permission model and do not allow users to dynamically grant or revoke permissions at run-time. We then ended up with a total of 69 apps suitable for our experiments.

To evaluate the extent TERMINATOR can prevent security attacks, we executed the attack scenarios from our dataset on an Android phone running TERMINATOR. Recall from Section 6.4, TERMINATOR relies on static analysis tools to identify the potential security threats. In our experiments, we used a combination of two static analysis tools, namely FlowDroid [67] and IC3 [370], that have also been used in the construction of three prior approaches to which we compare.

Table 6.1 shows the result of assessing the effectiveness of TERMINATOR compared to the state-of-the-art techniques. The first three columns of Table 6.1 show the attack scenarios, their source dataset, and the permissions involved in the attack scenarios, respectively. The other columns indicate whether each of the four approaches assessed was successful in preventing the attack (☒) or not (☐, ⊡).

According to the results, TERMINATOR is able to prevent all the attack scenarios with no false negatives. The success rate of the other techniques in preventing the permission-induced attacks ranges from 31.7% to 65.9%. A detailed look at Table 6.1 indicates that most of the missing attacks are those whose detection requires temporal analysis. For instance, consider the attack scenario #36, where a malicious application has defined a custom permission identical to the permission defined by a vulnerable app to protect its internal database. As a result, the malware can illegally access sensitive information stored in the vulnerable app. This vulnerability, however, is only exploitable if the malware is installed *before* the victim app. Thereby, all prior non-temporal approaches fail to detect such attacks.

To tackle this issue, a conservative approach might prevent the aforementioned attack by

permanently revoking database access of the victim app. This approach, however, would cause unnecessary disruptions, particularity when the vulnerability is not exploitable, i.e., the victim app is installed before malware in this case. In the next research question (RQ2), we investigate the consequences of permanently revoking the permissions of vulnerable apps through additional experiments.

6.5.2 RQ2: Disruption

For this research question, we focus on alternative permission-based enforcement techniques. Generally speaking, permission-based security enforcement can be applied at install-time or run-time [188]. An install-time approach prevents the installation of vulnerable apps, while a run-time approach revokes the permissions upon identification of an attack scenario. Run-time approaches can further be either permanent, whereby the permission decisions are final, or temporal, as in the case of Terminator, whereby the permission decisions are adjusted over time. Since the prior tools implementing the competing techniques are either not available, as is the case with AppFence [252] and AppGuard [75], or outdated and inapplicable, as is the case with Kirin [183], we implemented both install-time and permanent-run-time enforcement approaches described in the prior work to compare against Terminator's enforcement strategy.

To evaluate the level of disruption due to unavailability of permission-protected app functionality, we needed access to legitimate use-cases for apps in our dataset. Attack scenarios used in the evaluation of RQ1 are not representative of the apps' functional use-cases; thereby, they are not suitable for evaluating the level of disruption caused by the revocation of app permissions. To that end, we followed a semi-systematic approach to extract functional use-cases for the vulnerable apps in our dataset. We first downloaded the description of subject apps from the app markets (Google Play or F-Droid). We then asked a group of graduate

Table 6.2: Efficacy of permission-based techniques in reducing the unnecessary disruptions.

Vulnerable Apps	#Use-cases	#Allowed Scenarios		
		Run-time		Install-time
		Temporal	Permanent	
de.*.geobookmark	3	3	0	0
com.*.multismssender	8	8	4	0
com.*.calendar	14	14	6	0
com.*.smsscheduler ↯	9	9	5	0
org.*.trackbook	9	9	2	0
com.*.simpledeadline	13	13	9	0
com.getback_gps	10	10	6	0
com.*.camera	12	12	0	0
com.*.gallery	4	4	3	0
com.*.manager	11	11	5	0
com.*.anki	18	18	0	0
com.*.screennotification	3	3	2	0
com.*.notes	2	2	1	0
cz.*.forcastie	7	7	4	0
com.*.loginexample	2	2	1	0
com.*.sms	3	3	1	0
com.*.opps_wrong_tab	8	8	7	0
code.*.sendsmstest	3	3	1	0
org.*.myexpenses	24	24	8	0
com.*.ukweather	4	4	0	0
com.*.client	8	8	7	0
fr.*.ommons	11	11	9	0
Total allowance		186	81	0
Disruption (false-positive rate)		0%	56.45%	100%

students to construct, if possible, functional use-cases for each sentence or bullet in the app description. Additionally, we used available system tests for open-source subject apps as another source for identifying the legitimate use-cases. In total, we were able to derive 186 legitimate use-cases for subject apps in this research question. The full set of use-cases and subject apps are publicly available on the project website [42].

To measure the disruptions caused by the two run-time approaches, we first executed the attack scenarios from Table 6.1 to instigate an enforcement decision, i.e., force the approach to adjust the permission configuration. Subsequently, we ran the legitimate use-cases involving the apps in the attack to determine if the use-cases can be executed successfully or not. Table 6.2 summarizes the results of comparing different enforcement strategies for permission-based approaches. The first column shows the subject apps. The second column shows the number of legitimate use-cases for the subject apps. The last three columns show the number of use-cases allowed by each approach. The results from this analysis confirm

that the run-time-temporal approach adopted in TERMINATOR outperforms other enforcement techniques in terms of unnecessary disruption, i.e., false-positive rate. The install-time enforcement approach performs worst (100% false positive), as it does not allow the installation of a vulnerable app. The run-time-permanent approach (with 56% false positive) on the other hand, allows installation, yet revokes unsafe permissions permanently. Therefore, some of the legitimate permission-protected use-cases can never execute after revocation, even in the absence of a security threat.

For example, the security analysis performed by TERMINATOR identified *GetbackGPS* app (com.getback_gps in Table 6.2) as being vulnerable to privilege escalation attack—attack scenario #20 in Table 6.1, whereby its sensitive location information can be leaked. This vulnerability is only exploitable if two conditions are satisfied simultaneously: (1) a malware app with access to a sink channel (e.g., SMS) is installed and *running* on the phone, and (2) the malware has been *granted* the sink permission. Since the app is vulnerable, the install-time approach simply does not allow its installation to avoid any chance of leaking user's location information. The run-time-permanent approach on the other hand, allows the installation of GetbackGPS, yet permanently revokes its `LOCATION` permission to remove the vulnerability. Our run-time-temporal enforcement approach, however, leases `Location` permission to GetbackGPS, as long as the above conditions are not satisfied, during which all of the legitimate use-cases of the app are available.

6.5.3 RQ3: Applicability & Reliability

6.5.3.1 Applicability

Recall from Section 6.4.3, TERMINATOR relies on the dynamic permission mechanism, supported by Android 6 and newer versions of the framework, to regulate app permissions at run-time. However, not all the apps available on the Android marketplace are compati-

Table 6.3: Percentage of Android-6-compatible apps in Google Play

Randomly Selected by Category															Average	Top 100
Art	Books	Finance	Food	Health	Maps	Music	News	Photo	Shopping	Social	Tools	Video	Weather	Game		
100.0%	78.9%	86.6%	96.1%	85.9%	92.2%	89.1%	90.4%	89.9%	94.6%	90.9%	86.5%	86.1%	88.2%	91.6%	89.8%	100.0%

ble with the new versions of Android. To investigate the extent to which TERMINATOR is applicable to Android apps, we measured percentage of the apps on the official Android marketplace, i.e., Google Play, that target API level 23 (Android 6) and above.

To that end, we randomly collected 48,795 apps from different app categories, and distinguished Android-6-compatible apps by examining the `targetSdkVersion` tag specified in their *manifest* file. To avoid any bias in the results, we did not use any particular criteria, such as high popularity or high ranking, in selection of the apps to be analyzed. Table 6.3 demonstrates percentage of the apps targeting API level 23 and above among the apps collected from 15 different app categories of the Google Play repository[8]. According to the results, on average 89.8% of the Google Play apps support dynamic permissions.

To further investigate the support for dynamic permissions among popular apps, we also collected top 100 popular apps on Google Play. As shown in the last column of Table 6.3, all of the top 100 apps on Google Play support dynamic permissions, thereby are compatible with TERMINATOR. These results indicate that a large majority of the apps on the Android official marketplace can benefit from TERMINATOR for run-time security enforcement.

6.5.3.2 Reliability

Although the majority of collected apps support Android 6 and above, it is possible that they do not properly handle dynamic permissions. Failing to adjust the functionality of an app to dynamic permissions can lead to unexpected behaviors, e.g., app crashes if the user

[8]Due to space constraint, we merged similar categories into a representative one shown in Table 6.3

decides to revoke a permission. Hence, we also need to investigate the reliability of adopting an approach like TERMINATOR, which revokes permissions at run-time.

To investigate reliability of TERMINATOR, we recorded Logcat outputs during the execution of both the attack scenarios and canonical use-cases for subject apps discussed in RQ1 and RQ2. We later explored collected logs, searching for any crash messages due to improper handling of dynamic permissions.[9] Out of the 69 subject apps in our dataset, we found one app, *SMS Scheduler* (marked with ↯ in Table 6.2), that crashes due to the permission revocation.

From this data—low percentage (around 1.5%) of apps crashing when revoking their permissions and high percentage (around 89.8%) of app compatibility with recent versions of Android—we conclude that TERMINATOR can reliably be applied to a large majority of Android apps available on the market.

The permission-aware testing approach, presented in the next Chapter (Chapter 7), help identify those apps that crash when revoking their permission.

6.5.4 RQ4: Performance

To examine the performance characteristics of TERMINATOR, we measured the execution time taken for each phase of TERMINATOR, i.e., *analysis* and *enforcement* . We performed our experiments on a PC with an Intel Core i7 2.4 GHz CPU processor and 16 GB of main memory for the analysis phase, and a Nexus 5x phone operated by the Android framework version 6 for the enforcement phase.

TLC is configurable in two operating modes, *simulation* and *model-checking*. In the simulation mode, TLC verifies the system behavior up to a fixed number of system states. In the

[9] In case of improper handling of dynamic permissions, a `SecurityException` is thrown by Android framework, including the information of missing permission.

model-checking mode, on the other hand, there is no limit for the number of states to be explored. Applying an upper bound over the state exploration may lead to the possibility of missing attacks concealed within states not explored. We configured TLC to operate in the simulation mode to guarantee the termination of the analysis phase. This guarantee is required for TERMINATOR given that the reachable states of our model for the Android system is infinite. In our experiments, TERMINATOR was able to identify all of the attack scenarios (see Table 6.1). For these attack scenarios, TLC took at most 7 seconds to find the attack through the exploration of over 707,000 states.

To determine the performance of the enforcement phase, we calculated the overhead of running Monitor and Adapter components of TERMINATOR during the execution of 227 (41 attack scenarios and 186 canonical use-cases) scenarios exercised in RQ1 and RQ2. We repeated the execution of each scenario 5 times to ensure 95% confidence interval for the reported values. According to our experiments, the run-time overhead of TERMINATOR enforcement phase is 714±33 milliseconds on average for each use-case. Given that the average execution time for each use-case is 12 seconds, this overhead is negligble, as it is less than the threshold users can perceive slowness in an app, according to offical Android documentation [32]. Note that the analysis phase is performed once per system configuration, while the enforcement component runs continuously as the user interacts with the apps.

6.6 Conclusion

In this chapter, I presented a permission analysis and enforcement framework that, in contrast to the prior work, considers the temporal aspects of permission-induced attacks for their detection and prevention. The framework, called TERMINATOR, is realized in two phases. In the analysis phase, it uses a temporal logic model checker to identify the security risks with respect to dynamic states of the system. In the enforcement phase, it relies on Android's

dynamic permission mechanism to prevent the identified security threats from materializing by regulating the permission configuration of the system.

The evaluation results indicate that TERMINATOR is able to provide an effective, yet non-disruptive, defense against permission-induced attacks. The results also show that our approach, which is implemented without modification of Android framework or implementation logic of apps, is highly reliable and compatible with the great majority of Android apps available on the marketplace.

Chapter 7

Permission-Aware Testing of Android

As discussed in Section 6.5 of Chapter 6, revoking permissions, as a countermeasure for thwarting permission-induced attacks, could result in other sorts of defects, if the target app suffers from dynamic-permission compatibility issue. To identify this sort of permission-induced defects that occur either by the end-users via permission manager user interface, or by automatic enforcement tools such as TERMINATOR, an efficient permission-aware testing approach for Android apps is presented in this chapter.

7.1 Introduction

Access control is one of the key pillars of software security [300]. Many access control models exist for selectively restricting access to a software system's security-sensitive resources and capabilities. Among such models, *permission-based* access control has gained prominenace in recent years, partly due to its wide adoption in several popular platforms [89], including Android.

In Andriod, permissions are granted to apps. The Android runtime environment prevents

an app lacking the proper permissions from accessing both sensitive system resources (e.g., sensors) as well as other protected applications. Initially, Android employed a *static* permission system, meaning that the users were prompted to consent to all the permissions requested by an app prior to its installation, and the granted permissions could not be revoked afterwards. To provide the users more control over their device, in 2015, starting with API level 23, Android switched to a *dynamic* permission system, allowing users to change the permissions granted to an app at run-time [27].

The introduction of a dynamic permission system, however, poses an important challenge for testing Android apps. A test executed on an app may pass under one combination of granted permissions, yet fail under a different combination. As recommended by Android's best practices: *"Beginning with Android 6.0 (API level 23), users grant and revoke app permissions at run-time, instead of doing so when they install the app. As a result, you'll have to test your app under a wider range of conditions."* [27].

At the state-of-the-art, properly testing an Android app with respect to its permission-protected behavior entails re-execution of each test on all possible combination of permissions requested by an app, as there are no tools available to assist the developers with determining the interplay between tests and permissions. Such an *exhaustive* approach is time consuming, and often impractical, particularly in the case of regression testing, where the execution of an entire test suite needs to be repeated for an exponential number of permission combinations.

To mitigate this challenge, we have developed PATDroid, short for **P**ermission-**A**ware GUI **T**esting of An**Droid**. The insight guiding our research is that a given test may not interact with all the permissions requested by an app, meaning that some permissions, regardless of whether they are granted or revoked, may not affect the app's behavior under a particular test. By excluding the permissions that do not interact with tests, we can achieve a significant reduction in testing effort, yet achieve a comparable coverage and fault detection capability as exhaustive testing.

PATDroid leverages a hybrid program analysis approach to determine the interactions between an app's GUI tests and its permissions. It first dynamically pinpoints the entry-points of the app exercised by each test case. It then statically examines the parts of code that are reachable from the identified entry points to find the permission-protected code fragments. Afterwards, it statically determines the app inputs (i.e., GUI widgets) that control the execution of permission-protected code fragments. Finally, it statically identifies usages of the app inputs in the test scripts. Employing a sufficiently precise, yet scalable technique, PATDroid is able to effectively determine which tests should be executed under what permission combinations for an app.

Our experiments indicate that PATDroid is able to reduce both number of tests and their execution time by 71% on average, while maintaining a similar coverage as exhaustive execution of tests on all permission combinations. In addition, using PATDroid, we were able to identify several defects in real-world apps, as confirmed by their developers, that can only be exposed under certain permission settings, further demonstrating the usefulness of PATDroid in practice.

The chapter makes the following contributions:

- *Theory*: To the best of our knowledge, the first approach that considers the dependencies between a program, its test suite, and access control model for the reduction of testing effort;
- *Tool*: A fully automated environment that realizes the approach for Android programs, and made available publicly [34];
- *Experiments*: Empirical evaluation of the approach on a large number of real-world android apps demonstrating its efficacy.

The remainder of this chapter is organized as follows. Section 7.2 introduces an illustra-

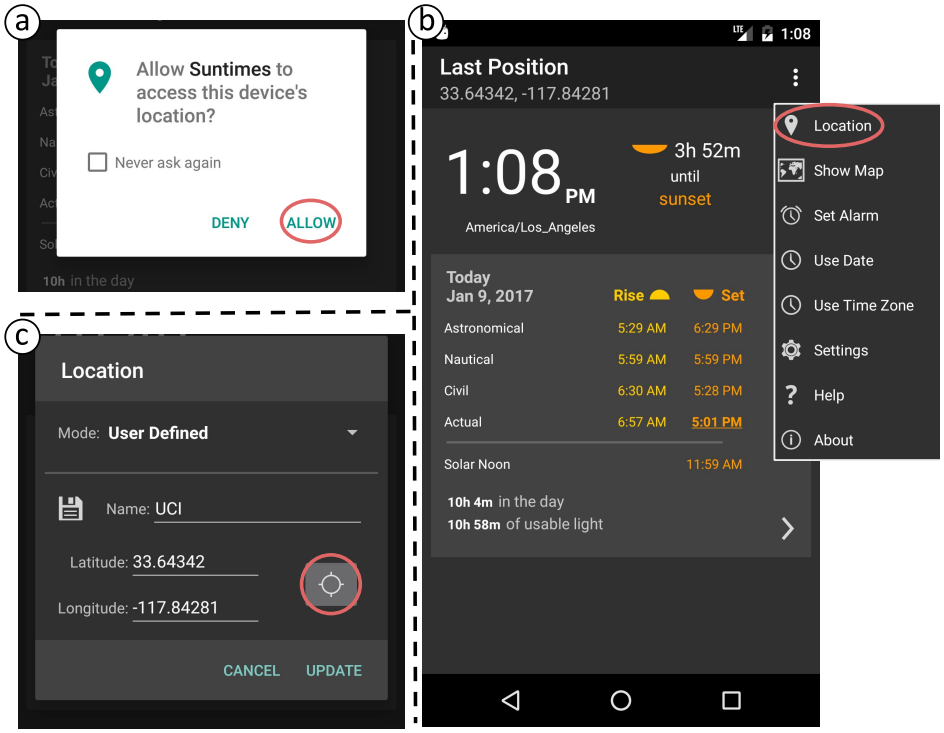


Figure 7.1: Screenshots of Suntimes app (a) Initially, asking user for the "Location" permission; (b) Main activity with available menu options, where the first option, i.e., location setting, is selected by the user; (c) Adding a new location to the app using GPS data

tive example to motivate the research. Section 7.3 provides an overview of PATDroid, while more details are presented in Sections 7.4-7.7. Section 7.8 provides the implementation information associated with the tool realizes our approach. Section 7.9 presents the experimental evaluation of the research.

7.2 Illustrative Example

We use a simplified version of an Android app, called *Suntimes*, to motivate the research and illustrate our approach. Suntimes calculates and displays sunrise, sunset, and twilight times for a particular location. It is developed to target Android version 6. Sample screen shots of this app are captured in Figure 7.1.

```
@Test //Test#1
public void testSunTimesNavigation() {
 onView(withId(R.id.info_note_flipper)).perform(click());
 onView(withId(R.id.info_note_flipper)).perform(click());
 onView(withId(R.id.info_time_nextbtn)).perform(click());
 onView(withId(R.id.info_time_prevbtn)).perform(click());
 // Check the navigation between suntimes is correct ...
}
@Test //Test#2
public void testSettingLocationToUserDefined() {
 onView(withId(R.id.action_location_add)).perform(click());
 onView(withId(R.id.appwidget_location_edit)).perform(click());
 onView(withId(R.id.appwidget_location_getfix)).perform(click())
 onView(withId(R.id.appwidget_location_name)).perform(replaceText("My Location"));
 onView(withId(R.id.appwidget_location_save)).perform(click());
 onView(withId(android.R.id.button1)).perform(click());
 // Check the newly added location is shown properly ...
}
@Test //Test#3
public void testExportLocations() {
 openContextualActionModeOverflowMenu();
 onView(withId(R.id.action_settings)).perform(click());
 onData(withKey(configLabel_places)).perform(click());
 onData(withKey(configLabel_places_export)).perform(click());
 // Check the locations are saved correctly ...
}
```

Figure 7.2: A subset of Espresso [29] tests embedded in the THA to verify the behavior of Suntimes app. The test assertions are not shown here

Since the app requires access to GPS data, it asks for *Location* permission once launched for the first time (Figure 7.1a). If a user grants the *Location* permission, the app periodically calculates and updates sunrise, sunset, and twilight times based on the current user location. Alternatively, the user can update her current location on demand from the option menu (Figure 7.1b), either by manually providing specific latitude and longitude, or using GPS to obtain location data (Figure 7.1c). However, *Suntimes* crashes when a user, who has previously denied the requested location permission, tries to update the current location using GPS, as the app at that point is neither granted the required permission (i.e., *Location*) to accomplish this task, nor it asks for it again.

To validate its behavior, *Suntimes* comes with a GUI test suite, a subset of which is shown in Figure 7.2. In contrast to unit tests, these tests run on a hardware device or emulator and commonly referred to as *instrumented tests* [28]. Regardless of the testing framework (e.g., Espresso [29], Robotium [38]), instrumented tests are compiled and packed as a separate *apk*

file and installed together with the apk of the main app. To distinguish these two software artifacts throughout this chapter, we call the apk containing the test suit and testing libraries as *Test Harness App (THA)*, and the apk of the main app as *App Under Test (AUT)*.

In the test cases shown in Figure 7.2, `testSunTimesNavigation` (Test #1) verifies the smooth navigation between different suntimes and dates, `testSettingLocationToUserDefined` (Test #2) validates adding a new user-defined location based on GPS data, and `testExportLocations` (Test #3) ensures the correctness of exporting retrieved location information to storage. Since Android version 6, it is recommended to test an app with various combinations of granted and revoked permissions to ensure correct behavior of the app under different conditions [27]. For example, `testSettingLocationToUserDefined` can reveal the aforementioned crash only when the developer has revoked the *Location* permission before running the test.

As another example, consider *Test #3* of Figure 7.2, which requires *Location* and *Storage* permissions to save user's location. Depending on the permissions granted to *Suntimes*, *Test #3* can exhibit different behaviors:

1) Both required permissions are already granted and *Suntimes* is able to successfully save the user's location on the external storage.

2) Only the *Location* permission is already granted. Hence, *Suntimes* asks for the *Storage* permission. In case of denial, *Suntimes* saves the location information in the app's internal storage, which does not require *Storage* permission.

3) Only the *Storage* permission is already granted. Hence, *Suntimes* asks for the *Location* permission. In case of denial, the app takes no action.

4) Neither of the required permissions have been previously granted. Hence, *Suntimes* asks for both of them. In case of denial, the app takes no action.

In any case, if the user denies any of the requested permissions, *Suntimes* should not crash.

The problem of testing an app's behavior under different permission settings becomes more complicated as the number of permissions defined in the app configuration file, a.k.a. *Manifest*, increases. One approach is to randomly grant and revoke permissions and run the test suite. Though simple, this approach fails to thoroughly test the app's behavior and is prone to miss important defects. Alternatively, a developer could manually review the test scripts and source code of an app to determine which tests should be executed under what app permissions. Such an approach, however, is quite cumbersome, especially considering that every time the app's source code changes, the developer needs to manually establish the relationships between the app's tests and its permissions.

Another approach is to exhaustively run the test suite under all possible combinations of requested permissions. In this approach, if an application requires p permissions, each test should be executed 2^p times, since each permission takes two values of $\{granted, revoked\}$.[1] For instance, *Suntimes* requests four permissions in its manifest file (i.e., Location, Storage, Alarm, and Internet). Considering the three tests in Figure 7.2, exhaustive approach runs each test $2^4 = 16$ times. For only the 3 test shown in Figure 7.2, we would need a total of $3 \times 16 = 48$ test runs. Clearly, such an approach does not scale as the number of requested permissions and the size of test suite increase.

The insight guiding our research is that exhaustive execution of tests for all permission combinations is overly conservative. For instance, we found that *Test #2* requires only *Location* permission, as the code executed by this test does not require access to capabilities guarded by other permissions. As a result, this test can only be executed twice—with and without the *Location* permission—rather than the 16 times required under the exhaustive

[1]In Android, only the dangerous permissions are configurable at run-time, while normal permissions are automatically granted at installation time. Without loss of generality, we consider all the permissions can be granted/revoked at run-time. for the evaluation however, we distinguish between dangerous and normal permissions.

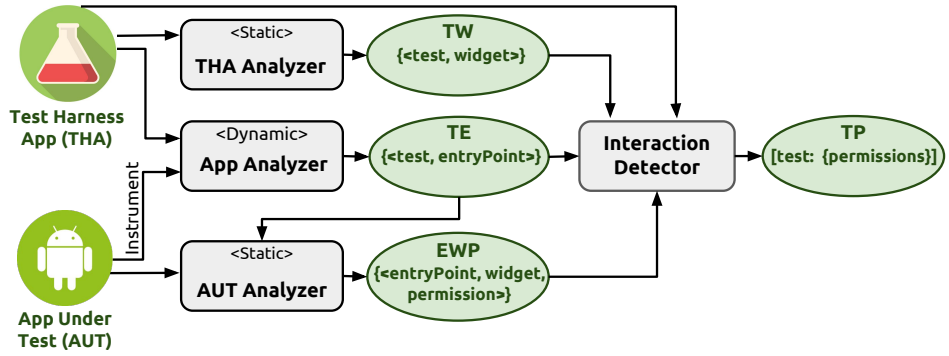


Figure 7.3: Overview of the approach

scenario.

7.3 Approach Overview

As mentioned in the previous section, in all popular Android testing frameworks (e.g., [29], [38]), a test suite is compiled and packed to produce the Test Harness App (THA), which is installed together with the App Under Test (AUT). Given a pair of THA and AUT, PATDroid identifies the minimum number of permission combinations for AUT that should be tested for each of the test cases embedded in THA. Figure 7.3 depicts an overview of PATDroid, consisting of four major components.

PATDroid first identifies those parts of AUT that could be exercised by the test cases embedded in THA. However, this is a challenging task as the test suite and test subject are realized in the form of two separate software artifacts (apk files). Moreover, THA is composed of *instrumented* test cases that require more involved analysis compared to, for example, *unit* tests. In contrast to unit tests that have no Android framework dependencies and directly invoke AUT's methods, instrumented tests run on a hardware device or emulator, and indirectly trigger a sequence of actions via GUI events.[2] The triggered GUI events

[2]Instrumented tests can also trigger other events, such as sending *Intents*. Those events, however, are outside the scope of this research, which focuses on GUI testing.

```
1  @Override
2  public boolean onOptionsItemSelected(MenuItem item){
3    switch (item.getItemId()){
4      case R.id.action_alarm:
5        scheduleAlarm();
6        return true;
7      case R.id.action_settings:
8        showSettings();
9        return true;
10     case R.id.action_location_add:
11       configLocation();
12       return true;
13     case R.id.action_location_refresh:
14       refreshLocation();
15       return false;
16     case R.id.action_timezone:
17       configTimeZone();
18       return true;
19     // other options
20     default:
21       return super.onOptionsItemSelected(item);
22     }
23 }
```

Figure 7.4: An entry-point of Suntimes app that handles the event corresponding to the selection of menu items shown in Figure 7.1b (A subset of options are shown here)

are handled initially by the testing framework, then Android run-time environment, and eventually delegated to certain methods in AUT, called *entry-points*. Due to such implicit dependency, static analysis cannot resolve the parts of the AUT executed by THA.

To mitigate the difficulties of resolving the relationships between AUT and THA statically, PATDroid leverages a hybrid (static and dynamic) approach that traces the dependencies between AUT and THA at two levels of granularity. First, at the method level, dynamic analysis identifies the entry-point methods of AUT that are exercised as a result of running the tests embedded in THA (represented as the set *TE* in Figure 7.3). Second, at the sub-method level, static analysis components narrow the entry-points discovered by dynamic analysis down to the blocks executable by a particular test case. The selected blocks of the entry-point methods are the targets for further static analysis.

To appreciate the need for restricting the scope of analysis, recall Suntimes app and the test suite shown in Figure 7.2. The second test (`testSettingLocationToUserDefined`) triggers an event by selecting the Location option from the main menu (line 11), which is

eventually handled by an entry-point method shown in Figure 7.4. This method is among the entry-point methods identified by dynamic analyzer for Test #2. However, inspecting `onOptionsItemSelected` method more carefully, it is clear that only the third case of the *switch* statement (i.e., lines 10-12 in Figure 7.4) is executable by Test #2, since other cases are intended to handle the other options never triggered by this test. Including the entire method, instead of focusing on lines 10-12, in the search for relevant permissions would increase the false-positive rate of our analysis.

The above example demonstrates that the execution flow of the GUI event handlers is controlled by the widgets triggering those events. Hence, a precise analysis should also take the GUI widgets affecting the control-flow of the app into account, otherwise it would over-approximate the code segments that could be exercised by each test. To that end, *THA Analyzer* determines the widgets used in each test case (represented as the set TW in Figure 7.3), While *AUT Analyzer* determines the permissions needed for executing each block of code in AUT, if any, along with the widgets affecting the reachability of those blocks (represented as the set EWP in Figure 7.3).

Finally, *Interaction Detector* integrates the outputs of the static and dynamic components and generates the relevant permissions for each test case (represented as the map TP in Figure 7.3). The following sections describe the four components of PATDroid in more details.

7.4 Dynamic Analysis

Unlike the conventional Java program with a single `main` method, Android apps comprise several methods that are implicitly called by the framework, usually referred to as *entry-points*. Entry-points are responsible for handling various events, including GUI events (e.g.,

```
{
<test:testSettingLocationToUserDefined,
 entryPoint:"SunActivity:boolean onOptionsItemSelected(Menu)">,
<test:testSettingLocationToUserDefined,
 entryPoint:"LocationConfigDialog$3$1:void onClick(View)">,
<test:testSettingLocationToUserDefined,
 entryPoint:"LocationConfigDialog:void onResume()">,
}
```

Figure 7.5: A subset of Suntimes app's entry-points exercised by Test #2 of Figure 7.2

`onOptionsItemSelected` shown in Figure 7.4 that handles the selection of a menu option), as well as changing the status of the application, a.k.a. life-cycle events (e.g., `onResume` to activate a paused app).

As a result of running a test, an app's entry-points are invoked by the Android framework. These are identified by the *Dynamic App Analyzer* component. For this purpose, PATDroid first automatically instruments the given AUT and injects *loggers* at the beginning of every possible entry-point of the app, which are distinguishable by the virtue of implementing specific interfaces of the Android framework (e.g., `onOptionsItemSelected`, `onResume`, etc.). For a comprehensive list of Android's entry-point interfaces, we have relied on the results of prior research [67, 308, 328, 251].

PATDroid subsequently runs the entire test suite on the instrumented app with an arbitrary permission setting. Since the invocation of entry-points are independent of the permission settings, our approach effectively finds the THA-dependent entry-points in the AUT. Unlike the test script, the code covered inside the entry-points depends on the permission settings during the test execution. Thus, we use static analysis technique, described in Section 7.6, to further explore the logic inside the entry-point methods.

Finally, the log obtained through the instrumentation of app's entry-points is processed to capture the executed entry-points for each test case. The generated output of this phase, called *TE*, is a set of tuples $\langle test, entryPoint \rangle$, where the first element is the test identifier and the second element is an exposed entry-point during the test execution. Figure 7.5

provides a subset of generated output for Test #2 of Figure 7.2.

7.5 Static Analysis of Test Harness App

As briefly discussed in Section 7.3, PATDroid traces the dependencies between AUT and THA at two levels of granularity. At a high-level of granularity, the dependencies at the method-level are identified by dynamic analysis, as described in the previous section. At a low-level of granularity, within the entry-point methods, the dependencies are refined through static analysis.

To statically trace the dependencies between AUT and THA, PATDroid resolves the app inputs, namely GUI widgets, that are the target of actions performed by test scripts. In the running example, `action_location_add` is a widget identifier used in both THA and AUT artifacts (lines 11 and 10 in Figures 7.2 and 7.4, respectively). For this purpose, PATDroid's static analysis component extracts the widget information from both AUT and THA. The extracted information should uniquely identify the widget throughout the entire app's implementation, and thus, usually includes a widget *identifier* or a *key*. While this section focuses on extracting widgets from THA, Section 7.6.2 describes how our approach applies to AUT.

Each Android testing framework (e.g., Espresso [29], Robotium [38], etc.) encodes the widget interactions in its own unique way, based on the framework's APIs and patterns. To generalize the problem of finding the used widgets and make our approach test-framework-agnostic, we define this problem as a general *data-flow analysis*. Accordingly, our goal is to find the flow of data within the test programs, from certain *source*s to *sink*s. For this purpose, data sources are defined as the set of testing framework APIs for retrieving a widget by a specific property, e.g., finding widgets based on ID using `ViewMatcher.withId(int)`

```
{
<test:testSettingLocationToUserDefined,
 widget:action_location_add(2131624168)>,
<test:testSettingLocationToUserDefined,
 widget:appwidget_location_getfix(2131624120)>
}
```

Figure 7.6: A subset of widgets extracted from Test #2 of Figure 7.2

and `Solo.findViewById(String)` APIs in Espresso and Robotium frameworks, respectively. Similarly, data sinks are defined as the set of testing framework APIs for performing an action on the widgets, e.g., a click action defined by `ViewActions.click()` and `Solo.clickOnButton()` APIs.

Defining the problem in this way allows us to perform the static analysis independent of the testing framework. To support a new testing framework, it is only needed to provide the list of framework's APIs for retrieving and performing actions on widgets. A slightly faster, yet less precise approach to find the widgets is to only look for widget retrieval APIs (i.e., source set only) and simply return the extracted information. This approach, however, can increase the false-positive rate, since some widgets might be retrieved for purposes other than performing an action (e.g., making an assertion). For this reason, we opted for a precise analysis.

To solve the data-flow problem, we employed an Android-compatible data-flow analysis framework, *FlowDroid* [67], but with a significant modification that allows us to perform the analysis on a THA. By default, FlowDroid is intended to analyze apps that comply with the conventional structure expected by the Android framework, e.g., to be composed of Android components. In contrast to AUT, THA does not follow such conventional structure and thus, is not supported by FlowDroid. Therefore, we replaced FlowDroid's default entry-point creator with a customized creator specifically tailored for THA analysis. For each THA, PATDroid creates a dummy `main` method, which is responsible for preparing the test environment encoded in `@Before` methods, and then invoking the `@Test` methods embedded in THA. Recall the use of such annotations in the test script example shown in Figure 7.2.

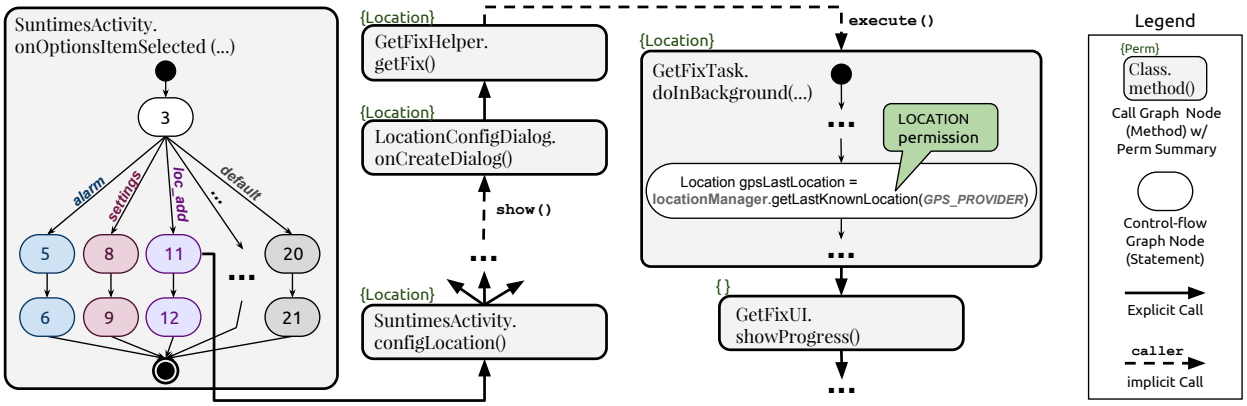


Figure 7.7: A sub-graph of inter-procedural control-flow graph for Suntimes app. The collapsed parts of the sub-graph are denoted by "...". The method call in node (=line) 11 of the control-flow-graph for `SuntimesActivity.onOptionsItemSelected()` method eventually leads to calling an Android framework API that requires LOCATION permission (i.e.,`getLastKnownLocation`). Since this permission is used under the branch with widget id `location_add`, it is inferred that LOCATION is a relevant permission for a GUI test that exercises this entry-point method (`onOptionsItemSelected`) by performing an action on `location_add` widget

Solving the data-flow problem, *THA Analyzer* generates the output, *TW*, which is a set of tuples $\langle test, widget \rangle$, where the first element is the test identifier and the second element is a widget that is the target of an action performed by the test. Figure 7.6 provides a subset of the analysis output generated for Test #2 of Figure 7.2.

7.6 Static Analysis of App Under Test

Running under an arbitrary permission settings, *Dynamic App Analyzer* partially explores the AUT code executable by each test. Subsequently, PATDROID leverages *AUT Analyzer* to statically examine all parts of the code that could be exercised by each test.

As depicted in Figure 7.3, the *AUT Analyzer* receives the AUT and *TE* as input and generates *EWP* as output. The generated output is a set of tuples, each containing three elements $\langle entryPoint, widget, permission \rangle$, indicating an entry-point method invoked during the execution of a test, a widget that can affect the reachability of permission-protected code

within that entry-point, and the corresponding permission. *AUT Analyzer*'s main procedure is summarized in Algorithm 7.1.

Algorithm 7.1: AUT Analysis

```
   Input: AUT: App under test, TE: Tests to entry-points set
   Output: EWP: {⟨entryPoint, widget, perm⟩}
1  EWP ← ∅
   // ▶ Permission Analysis - see Section 7.6.1
2  permSummaries ← PermissionAnalysis(AUT, TE)
   // ▶ Widget Analysis - see Section 7.6.2
3  widgetSummaries ← WidgetAnalysis(AUT)
4  foreach entryPoint ∈ TE do
5      foreach stmt ∈ entryPoint.statements do
6          if stmt.type is METHOD INVOCATION then
7              perms ← permSummaries[stmt.targetMethod]
8              foreach perm ∈ perms do
9                  widgets ← widgetSummaries[stmt]
10                 if widget = ∅ then
11                     EWP ← EWP ∪ ⟨entryPoint, ∅, perm⟩
12                 else
13                     foreach widget ∈ widgets do
14                         if ⟨entryPoint, widget, perm⟩ ∉ EWP then
15                             EWP ← EWP ∪ ⟨entryPoint, widget, perm⟩
16                     end
17             end
18     end
19 end
```

The analysis procedure performs several steps to generate the output. Initially, *PermissionAnalysis* sub-procedure (line 2) identifies the required permissions for executing each statement, if any, for all of the app's entry-point methods exercised by the test suite. The details of this sub-procedure are described in Section 7.6.1. Subsequently, *WidgetAnalysis* procedure is invoked in line 3 to determine the statements that are controlled by each widget, the details of which are described in Section 7.6.2.

For each entry-point method (line 4) and each statement within it (line 5), the algorithm determines whether it is a method invocation statement (line 6) that is permission protected (lines 7–8). These could be either Android API calls or user-defined methods. For each permission-protected method invocation statement, all the widgets that control the execution

of this statement are retrieved (line 9). Finally, the algorithm adds tuples consisting of method, widget, and permission information to set EWP, unless they already exist in this set (lines 14–15). If a permission-protected statement is not controlled by any widget, the widget element is set to NULL in the corresponding generated tuple (lines 10–11).

7.6.1 Permission Analysis

For each method defined in a given AUT, *Permission Analysis* procedure captures all permissions required for executing that method, called *Permission Summaries (PS)*, through performing an *inter-procedural fixed-point* analysis, summarized in Algorithm 7.2.

In the first step, *Permission Analysis* constructs a call graph (CG) of the entire application (line 2). However, due to the event-driven structure of the Android platform, the traditional CG generation methods do not connect the call sites corresponding to implicit invocations. The challenges of generating call graph for Android apps are widely discussed in the prior research and several techniques are suggested for this purpose [67, 370], which are employed by PATDROID. Figure 7.7 depicts a subset of the call graph for the Sun-times app. In this graph, the implicit calls are denoted by dashed lines. For instance, the method `GetFixHelper.getFix()` starts an `AsyncTask`, namely `GetFixTask`, by invoking the `execute()` interface. Consequently, the method `doInBackground()` of the task class is invoked indirectly by the Android framework.

Permission Analysis iterates over all Android framework APIs that are called throughout the given app (lines 3–7) and adds the required permission for the API to the permission summaries (*PS*) of the methods where that API is called.[3] We have relied on permission-API mappings produced in the prior work [69, 71] to determine the required permission for Android APIs.

[3]In addition to the Android framework APIs, certain *Intents* and queris on *Content Providers* need specific permissions. For brevity, however, only the iteration over the APIs is shown in Algorithm 7.2.

Algorithm 7.2: Permission Analysis

Input: AUT: App under test, TE: Tests to entry-points set
Output: PS: Permission Summaries

```
1  PS ← ∅
   // PS is a map with method signature as its key and the corresponding set
      of required permissions as its value
2  CG ← constructCG(AUT)
3  foreach API ∈ AUT.AndroidAPICalls do
4  |  method ← caller(API)
5  |  perm ← perm(API)
6  |  PS[method] ← perm
7  end
8  repeat
9  |  foreach method ∈ BFS.next(CG, TE) do
10 |  |  callerMethods ← G.edgesTo(method)
11 |  |  foreach callerMethod ∈ callerMethods do
12 |  |  |  perms ← PS[method]
13 |  |  |  PS[callerMethod] ← PS[callerMethod] ∪ perms
14 |  |  end
15 |  end
16 until PS reaches a fixed-point;
```

Finally, Permission Analysis traverses the constructed call graph (CG) using breadth-first search (BFS) method (lines 9–15). Starting from the given entry-point methods (EE), it propagates the permission in the graph. In each iteration, the algorithm updates the permission summaries (PS) of all methods calling the current method, by augmenting their PS with the PS of the callee method (line 13). This procedure is repeated until a fixed point is reached for the permission summaries (line 16), meaning that PS does not change in further iterations. In Figure 7.7, the permission summaries are shown at the top-left corner of each call-graph node.

7.6.2 Widget Analysis

Recall the entry-point method presented in Figure 7.4. To handle a selected menu option, this method (`onOptionsItemSelected`) invokes several other methods, each one under a `case` corresponding to the menu option. For instance, *Set Alarm* (third option in Figure 7.1b) is

handled by the first `case` statement shown in Figure 7.4, where `scheduleAlarm()` method is called consequently (line 5). Therefore, if a GUI test only clicks on Set Alarm option, it does not execute the methods called in other cases, and thereby the summarized permissions for other methods (e.g., `showSetting()`, `configLocation()`, etc.) are irrelevant to this test. To exclude the irrelevant permissions, we need to determine which widgets affect the control-flow of which program statements, particularly the statements that invoke methods with non-empty permission summaries. *Widget Analysis* procedure, summarized in Algorithm 7.3, provides this capability.

Algorithm 7.3: Widget Analysis

Input: AUT: App under test
Output: WS: Widget Summaries

```
1  WS ← ∅
2  gen, in, out ← ∅
   // WS, gen, in, and out are maps with program statement as its key and set
      of related widgets as its value.
3  ICFG^T ← constructTrimmedICFG(AUT)
4  foreach stmt ∈ AUT.methods.statements do
5      if stmt.type is IF & stmt.condition.type is WIDGET then
6          gen[stmt.target] ← stmt.condition
7      else if stmt.type is SWITCH & stmt.condition.type is WIDGET then
8          foreach case ∈ stmt.cases do
9              gen[case.target] ← case.condition
10         end
11 end
12 repeat
13     foreach stmt ∈ BFS.next(ICFG^T) do
14         foreach stmt' ∈ pred(stmt) do
15             in[stmt] ← in[stmt] ∪ out[stmt']
16         end
17         foreach stmt' ∈ succ(stmt) do
18             out[stmt'] ← in[stmt] ∪ gen[stmt]
19         end
20     end
21     WS ← out
22 until WS reaches a fixed-point;
```

For a given method, *Widget Analysis* procedure performs a *branch-sensitive, partial, inter-procedural* data-flow analysis and generates the *Widget Summaries (WS)* as the output. For

```
{
<entryPoint:"SuntimesActivity:boolean onOptionsItemSelected(MenuItem)",
 widget: action_location_add(2131624168),
 permission: LOCATION>
}
```

Figure 7.8: A subset of *EWP* generated for Suntimes app

this purpose, a trimmed version of inter-procedural control-flow graph (ICFG) is constructed first (line 3). An ICFG is a collection of control-flow graphs connected to each other at all call sites. Our analysis, however, targets app widgets exclusively and thus, only the call sites that pass a widget object are included in the trimmed ICFG, denoted as $ICFG^T$. Performing the analysis over $ICFG^T$, instead of ICFG, significantly improves the scalability of our approach, yet keeps the precision acceptable.

Afterwards, the *gen* set is populated through iterating over every statement of each method (lines 4–11). We are only interested in the conditional statements that affect the control-flow of the program, namely `IF` (lines 5–6) and `SWITCH` (lines 7–10) statements, with the widget as the condition. For instance, the `switch` statement in Figure 7.4 could be a target of our analysis, as it is (1) a conditional statement controlling the flow of the program, and (2) a widget, i.e., *MenuItem*, is used as the statement's condition.

Finally, the algorithm traverses the $ICFG^T$ in a breadth-first search manner and propagates the widget information through the graph. By doing this, at each statement we have the information of all widgets that can affect the control-flow of the program from the beginning to that statement. For example, as highlighted in the control-flow graph of `onOptionsItemSelected` method depicted in Figure 7.7, with `location_add` as the selected menu option, the control-flow of the program will reach to lines 11 and 12. Hence, `location_add` is added to the widget summaries of the statements at nodes 11 and 12. The widget analysis terminates upon reaching a fixed point for the widget summaries (WS).

It is essential to note the difference between the precision and scope of two sub-procedures

described in Sections 7.6.1 and 7.6.2, namely *Permission Analysis* and *Widget Analysis*. Due to flow and branch sensitivity, Widget Analysis is more costly than Permission Analysis. On the other hand, while Permission Analysis is performed on every method in the app through traversing its call graph, the scope of Widget Analysis is limited to a few entry-point methods exercised by running the tests. This distinction lets PATDroid keep the app analysis precise and yet, scalable.

Combining the outputs of *Permission Analysis* and *Widget Analysis* sub-procedures, the main procedure (Algorithm 7.1) generates the final output of *AUT Analyzer* component, i.e., *EWP*. A subset of generated EWP for Suntimes app is provided in Figure 7.8.

7.7 Building Permission Combinations

As shown in Figure 7.3, *Interaction Detector* generates the final output, TP, which is a map from tests to the set of relevant permissions. It does so by correlating the outputs of the other components, namely *TE*, *TW*, and *EWP*, as follows.

Interaction Detector procedure, summarized in Algorithm 7.4, iterates over the three input sets (*TE*, *TW*, *EWP*), and matches the tuple members of these sets based on the shared elements, i.e., *entryPoint*, *test*, and *widget*.[4] The only exception occurs when no widget is found for an *EWP* (i.e., no widget is used to control access to permission-protected code in an entry-point), in which case it is conservatively assumed that the entire entry-point method could be executed by a single test and hence, the algorithm does not attempt to match *EWP.widget* and *TW.widget* (line 5). Based on the matched tuples, relevant permissions for a test are added to the output, TP (line 9). Finally, an empty set is assigned to those tests that have no relevant permissions (lines 14–17). The generated output, TP, for the test set of Suntimes app is provided in Figure 7.9.

[4]Matching elements are distinguished by the same colors in Figures 7.5, 7.6, 7.8, and 7.9.

Algorithm 7.4: Interaction Detector

Input: TE= $\{\langle test, entryPoint\rangle\}$, TW= $\{\langle test, widget\rangle\}$,
EWP= $\{\langle entryPoint, widget, permission\rangle\}$, THA:Test harness app
Output: TP, A map with tests as the key and the set of relevant permissions as the value.

```
1  TP ← ∅
2  testWithPerm ← ∅
3  foreach ewp ∈ EWP do
4  |  foreach tw ∈ TW do
5  |  |  if ewp.widget = ∅ Or ewp.widget = tw.widget then
6  |  |  |  foreach te ∈ TE do
7  |  |  |  |  if ewp.entryPoint = te.entryPoint then
8  |  |  |  |  |  if te.test = tw.test then
9  |  |  |  |  |  |  TP[te.test] ← TP[te.test] ∪ ewp.perm
10 |  |  |  |  |  |  testWithPerm ← testWithPerm ∪ te.test
11 |  |  |  end
12 |  end
13 end
14 foreach test ∈ THA.tests do
15 |  if test ∉ testWithPerm then
16 |  |  TP[test] ← ∅
17 end
```

The output of this algorithm enables efficient permission-aware testing of the given app. In total, for an app consisting of T tests and P permissions, the number of test-runs by PATDroid are calculated as follows:

$$\sum_{t=1}^{T} 2^{|TP[t].perms|}$$

where $TP[t].perms$ denotes the relevant permissions for test t identified by PATDroid. As our experiments will show, this number turns out to be significantly smaller than $|T| \times 2^{|P|}$ tests required for execution under the exhaustive approach.

```
[
testSunTimesNavigation:{},
testSettingLocationToUserDefined:{LOCATION},
testExportLocations:{LOCATION, STORAGE}
]
```

Figure 7.9: Relevant permissions for a subset of the tests listed in Figure 7.2

7.8 Implementation

PATDroid is realized with over 2,500 lines of Java code and 800 lines of Python script. It also relies on a few third-party libraries, most notably *Soot* [495] for static analysis of Android apps, *IC3* [370] to resolve ICC communications, and *Xposed* [44] for run-time instrumentation of the root Android process.

PATDroid runs in two modes: (1) Developers mode, and (2) Testers mode. The first mode is applicable when the source code of subject apps (AUT) and their GUI tests (THA) are available. The second mode can be used when only apk files (AUT) are available. PATDroid currently supports the major Android's GUI test frameworks, namely Espresso, Robotium, and Monkey.

The artifacts associated with PATDroid, including the executable tool and its user manual are available for download from PATDroid's web page [34].

7.9 Evaluation

Our evaluation of PATDroid addresses the following questions:

RQ1. *Efficiency*: How does PATDroid compare against alternative approaches with respect to test-run size and test-execution time?

RQ2. *Coverage*: How does PATDroid compare against alternative approaches with respect to code coverage?

RQ3. *Effectiveness*: Is PATDroid able to reveal defects in real-world apps, particularly those that are only exposed under certain permission settings?

RQ4. *Performance*: How does PATDroid scale in relation to the size of app?

Table 7.1: A subset (those with available source code) of subject apps.

App	Size (KLOC)	# of permissions		test-suite size
		all	dangerous	
A2DP Volume [24]	9.1	17	9	17
AlwaysOn [25]	15.9	18	6	16
Budget Watch [30]	8.0	3	2	12
Dumbphone Assistant [31]	1.4	3	3	7
Notes [33]	5.7	3	2	29
RadioBeacon [36]	31.4	10	6	17
Riot [37]	55.2	15	6	20
SMS Scheduler [39]	1.5	4	2	6
Suntimes [40]	22.4	4	3	13
SysLog [41]	12.1	4	2	13

7.9.1 Experiment Setup

To evaluate our approach on realistic subjects, we crawled Google Play and GitHub repositories and searched for Android apps with the following criteria:

(i) Should target Android API level ≥ 23; otherwise, the app does not support run-time permission modification, and thereby does not suffer from the problems that are the focus of our work.

(ii) Should define at least two *dangerous* permissions in the manifest file, because other types of permissions are not adjustable at run-time and solving the problem with less than two adjustable permissions is trivial.

In accordance with the above criteria, we collected 110 apps: (1) 100 popular apps from Google Play, and (2) 10 open-source apps from Github (listed in Table 7.1), since investigating RQ2, i.e., measuring code coverage, requires the availability of source code.

For the open-source apps, we manually created or extended the existing GUI tests using Espresso [29] or Robotium [38] frameworks to achieve at least appriximately 50% statement coverage. For Google Play apps we used Monkey [4] to generate black-box GUI tests.

We have compared PATDroid against three alternative strategies, as follows:

Exhaustive—exhaustively includes all permission combinations.

Pairwise—generated according to pairwise technique [366]; that is, for any two permissions, all possible pairs of permission settings (i.e., granted, revoked) should be in the output set.

All-and-None—includes two combinations, one with all permissions granted, the other with all permissions revoked.

None of the existing test suite reduction tools support Android framework, nor consider its access control model, therefore, are not included in our evaluation.

7.9.2 Efficiency

To answer RQ1, we compare the test-run size and test-execution time of PATDroid with *exhaustive*, *pairwise* and *all-and-none*, as shown in Tables 7.2 and 7.3. *Test-run size* indicates the cumulative number of tests required to run for each technique. This number is calculated by the formulas shown in Table 7.2, under the corresponding columns. In addition, the table shows the percentage of decrease or increase for each reported metric in comparison to PATDroid.

The results in Table 7.2 and 7.3 confirm that PATDroid can significantly reduce the number of test-runs and test-execution time. On average, PATDroid requires 71.35% and 41.78% fewer test executions than *exhaustive* and *pairwise*, respectively. Similarly, on average, PATDroid takes 71.07% and 39.07% less execution time than *exhaustive* and *pairwise*, respectively. In comparison to *all-and-none*, however, the results are mixed, where in some cases PATDroid achieves a higher reduction (e.g., *Budget Watch*), while in other cases PATDroid achieves a lower reduction (e.g., *RadioBeacon*). Although *all-and-none* achieves a higher reduction in some cases, the next section shows that it does not maintain the same coverage as other approaches.

Table 7.2: Test *size* reduction achieved by PATDroid compared to other approaches. +, - indicate that the reduction achieved by the the alternative approach is greater, or less than PATDroid, respectively.

App	Test-run size (% of difference compared to PATDroid)			
	PATDroid $\sum_{t=1}^{\|T\|} 2^{\|TP[t].perms\|}$	Exhaustive $\|T\| \times 2^{\|P\|}$	Pairwise	All&None $2 \times \|T\|$
A2DP Volume	59	8,704(-99.32%)	136(-56.62%)	34(+73.53%)
Always On	29	1,024(-97.17%)	96(-69.79%)	32(-9.38%)
Budget Watch	15	48(-68.75%)	48(-68.75%)	24(-37.50%)
Dumbphone Assist	56	56(0%)	28(+100.00%)	14(+300.00%)
Notes	35	116(-69.83%)	116(-69.83%)	58(-39.66%)
RadioBeacon	66	1,088(-93.93%)	102(-35.29%)	34(+94.12%)
Riot	48	1,280(-96.25%)	120(-60.00%)	40(+20.00%)
SMS Scheduler	7	24(-70.83%)	24(-70.83%)	12(-41.67%)
Suntimes	32	104(-69.23%)	52(-38.46%)	26(+23.08%)
SysLog	27	52(-48.08%)	52(-48.08%)	26(+3.85%)

In the presented formulas used for calculating the size of the test-runs, T is the set of tests, P is the set of app's permission, and $TP[t].perms$ is the set of relevant permissions for the test t generated by PATDroid.

Table 7.3: Test *time* reduction achieved by PATDroid compared to other approaches. +, - indicate that the reduction achieved by the the alternative approach is greater, or less than PATDroid, respectively.

App	Testing time in sec. (% of difference compared to PATDroid)			
	PATDroid	Exhaustive	Pairwise	All&None
A2DP Volume	314	39,591(-99.21%)	619(-49.32%)	155(+102.74%)
Always On	208	7,133(-97.09%)	669(-68.92%)	223(-6.75%)
Budget Watch	67	221(-69.47%)	221(-69.47%)	110(-38.94%)
Dumbphone Assist	447	447(0%)	224(+100.00%)	112(+300.00%)
Notes	162	531(-69.50%)	531(-69.50%)	266(-38.99%)
RadioBeacon	462	6,200(-92.55%)	581(-20.50%)	194(+138.50%)
Riot	398	10,379(-96.16%)	973(-59.05%)	324(+22.84%)
SMS Scheduler	34	110(-69.11%)	110(-69.11%)	55(-38.23%)
Suntimes	317	931(-65.92%)	465(-31.84%)	233(+36.32%)
SysLog	144	299(-51.77%)	299(-51.77%)	149(-3.54%)

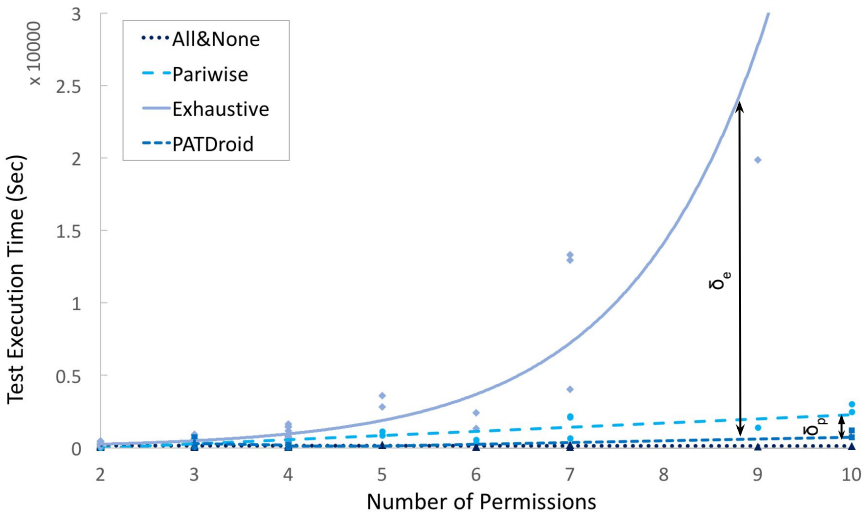


Figure 7.10: Test execution time based on the number of permissions. δ_e and δ_p represent the reduction in test execution time, achieved by PATDroid, compared to exhaustive and pairwise approaches, respectively

Figure 7.10 plots the test-execution time for all of the 110 subject apps. As illustrated in the figure, test-execution time grows exponentially with respect to the number of permissions in exhaustive approach. Therefore, the reduction rates compared to *exhaustive* approach are higher in apps with more permissions. For example, the reduction in the case of *A2DP Vol* app with 9 permissions is above 99%, while the reduction in the case of *Budget Watch* app with 2 permissions is close to 70%.

7.9.3 Coverage

To answer RQ2, we compare the statement and branch coverage achieved by PATDroid against that of achieved by the alternative techniques. As shown in Tables 7.4 and 7.5, PATDroid achieves the same exact coverage as *exhaustive* in all subject apps. The fact that PATDroid achieves the same coverage as *exhaustive* is particularly important, as it shows that PATDroid does not produce many false negatives, i.e., failing to execute a test with a relevant permission combination for an app.[5]

[5]In our experiments, PATDroid did not produce any false negatives, but in principle it could, due to limitations of static analysis upon which PATDroid relies.

Table 7.4: *Statement* coverage achieved by PATDroid compared to other approaches. +, - indicate that the coverage of the alternative approach is greater, or less than PATDroid, respectively.

App	Statement Coverage (% of difference compared to PATDroid)			
	PATDroid	Exhaustive	Pairwise	All&None
A2DP Volume	49.55%	49.55%(0%)	49.55%(0%)	47.18%(-5%)
Always On	45.31%	45.31%(0%)	10.54%(-77%)	45.31%(0%)
Budget Watch	72.24%	72.24%(0%)	72.24%(0%)	56.52%(-22%)
Dumbphone Assist	64.90%	64.90%(0%)	7.56%(-88%)	64.90%(0%)
Notes	77.89%	77.89%(0%)	77.89%(0%)	62.24%(-20%)
RadioBeacon	49.22%	49.22%(0%)	49.22%(0%)	43.24%(-12%)
Riot	50.40%	50.40%(0%)	46.49%(-8%)	46.92%(-7%)
SMS Scheduler	65.25%	65.25%(0%)	65.25%(0%)	65.25%(0%)
Suntimes	50.23%	50.23%(0%)	50.23%(0%)	44.14%(-12%)
SysLog	71.33%	71.33%(0%)	71.33%(0%)	65.66%(-8%)

Table 7.5: *Branch* coverage achieved by PATDroid compared to other approaches. +, - indicate that the coverage of the alternative approach is greater, or less than PATDroid, respectively.

App	Branch Coverage (% of difference compared to PATDroid)			
	PATDroid	Exhaustive	Pairwise	All&None
A2DP Volume	23.87%	23.87%(0%)	23.87%(0%)	21.61%(-9%)
Always On	25.58%	25.58%(0%)	1.69%(-93%)	25.58%(0%)
Budget Watch	51.04%	51.04%(0%)	51.04%(0%)	37.35%(-27%)
Dumbphone Assist	43.10%	43.10%(0%)	11.21%(-74%)	43.10%(0%)
Notes	61.30%	61.30%(0%)	61.30%(0%)	48.54%(-21%)
RadioBeacon	25.76%	25.76%(0%)	25.76%(0%)	22.41%(-13%)
Riot	42.28%	42.28%(0%)	40.24%(-5%)	39.81%(-6%)
SMS Scheduler	45.32%	45.32%(0%)	45.32%(0%)	45.32%(0%)
Suntimes	32.95%	32.95%(0%)	32.95%(0%)	27.23%(-17%)
SysLog	48.75%	48.75%(0%)	48.75%(0%)	42.19%(-13%)

Moreover, on average, PATDroid achieves 14% and 10% higher coverage than *pairwise* and *all-and-none* techniques, respectively. It is worth noting that while in 7 apps *pairwise* achieves the same coverage as PATDroid, in 3 apps it achieves significantly lower coverage. A closer look at the apps where PATDroid outperformed *pairwise* showed that these situations occur when certain capabilities provided by an app depend on more than two permissions. For instance, *AlwaysOn* app asks for four permissions, and if any of those permissions are not granted, the app's functionally is significantly downgraded. Since the *pairwise* technique does not include a combination with all four permissions granted, it achieves 77% lower statement coverage and 93% lower branch coverage than PATDroid.

In summary, the results of RQ1 and RQ2 confirm that PATDroid is able to significantly reduce the number of tests without trading-off code coverage.

7.9.4 Effectiveness

To answer RQ3, we investigate the power of our approach in identifying permission-related defects in real-world apps. To that end, we carefully analyzed Android log, and the output of the tests executed under the permission combinations generated by PATDroid. Particularly, we were interested in crashes or unexpected behaviors that could only be verified by running the tests under certain permission combinations.

Running PATDroid on the set of 110 apps, we found 14 apps (i.e., 13%) with defects that are due to inappropriate handling of dynamic permissions. We reported the identified defects for the open-source apps to their developers through GitHub issue tracker, along with information to reproduce the faults and suggestions for fixing the defects. Table 7.6 provides a summary of the reported defects and the current status of each issue for the apps that provide a public issue tracker. As of the date of writing this dissertation, most of the defects are verified and fixed by the app developers.

Table 7.6: A subset (those with public issue tracker) of defects in real-world Android apps identified for the first time by PATDroid.

App	Reported issue link	Defect Type	Status
Open Food	https://goo.gl/4eIm3E	Crash	Fixed
Budget Watch	https://goo.gl/8XBvkf	Unexpected Behavior	Fixed
A2DP Volume	https://goo.gl/9sfS09	Unexpected Behavior	Fixed
RadioBeacon	https://goo.gl/80Mb5j	Crash	Verified
Riot	https://goo.gl/MNEdkx	Unexpected Behavior	Fixed
OpenNoteScanner	https://goo.gl/yKNiRZ	Crash	Reported

Note that *exhaustive* and *pairwise* approaches are also able to identify the reported defects, except they take significantly longer time to execute as shown in Section 7.9.2. *all-and-none* on the other hand, is not able to reveal these issues. For instance, in *Open Note Scanner* app, which asks required permissions initially, revoking the Storage permission while granting Camera permission would make the application crash. Such behavior is not reproducible using *all-and-none* technique. Furthermore, *exhaustive* approach was not able to find a defect that PATDroid missed, further demonstrating the efficacy of PATDroid in revealing permission-related defects.

7.9.5 Performance

To answer RQ4, we measured the performance of running PATDroid over the subject apps. The experiments are run on a PC with an Intel Core i7 2.4 GHz CPU processor and 16 GB of main memory. According to the experimental results, the average time spent on identifying the relevant permissions is 356 seconds, which is negligible compared to the time saved due to reducing the test-run size (See Section 7.9.2).

Figure 7.11 shows the performance measurements of running PATDroid. The analysis times for each phase of PATDroid, i.e., static and dynamic analyses, are plotted separately in the figure. On average, static and dynamic analyses take 97 and 259 seconds, respectively.

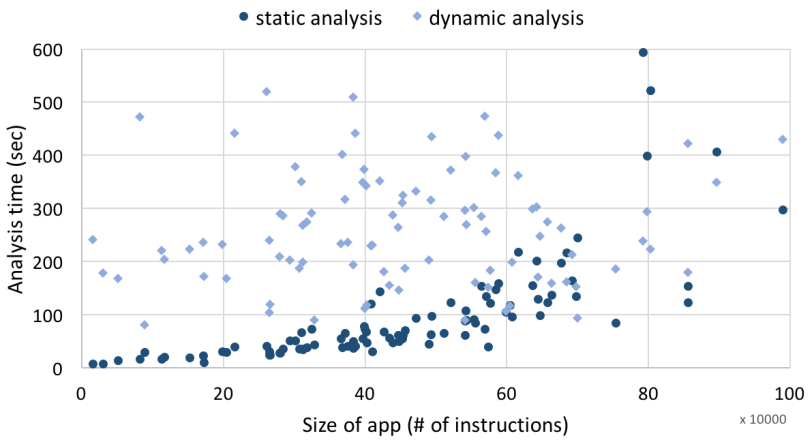


Figure 7.11: Performance measurements of PATDroid

According to the figure, the static analysis time increases as the app size increases, while there is no correlation between the dynamic analysis time and the app size. Dynamic analysis time depends on the logic and workload of the subject app. For instance, the size of the data that an app downloads from the Internet can affect the execution time of the app's system tests.

7.10 Conclusion

Recent introduction of a dynamic permission system in Android has made it necessary to test the behavior of Android apps under a variety of permission settings. Without an automated solution to reason about which tests should be executed under what permission combinations, the developers have to either manually make such determinations or exhaustively re-run each test under an exponential number of permission combinations. Both approaches are impractical, time-consuming, and cumbersome.

To overcome this problem and help developers efficiently test Android apps under various permission settings, this chapter presented PATDroid. Through a hybrid program analysis of Android app and its test suite, PATDroid is able to identify relevant permissions for

each test case. By excluding the irrelevant permissions, PATDroid is able to significantly reduce the number of test runs and execution time of tests without trading-off coverage and fault detection ability of tests. The experimental results show that PATDroid can achieve 71% reduction in execution time of tests compared to the exhaustive approach, without any degradation in code coverage. Moreover, using PATDroid, we were able to identify several previously unknown permission-related defects in real-world apps.

Chapter 8

Conclusion

In this dissertation I presented a set of permission-aware analysis techniques for efficient detection and prevention of permission-induced issues—the defects enabled by Android permission misuse. More specifically, I focused on two types of permission-induced issues, those that breach the security of the system, and those that disrupt the functional behavior of the apps, particularly under dynamic permission mechanism. To that end, I designed, implemented, and evaluated four approaches, namely Covert, Separ, Terminator, and PATDroid.

Covert and Separ use combination of program analysis and formal methods to detect and prevent security vulnerabilities, particularly those that occur due to unsafe interaction of multiple apps. The third proposed technique, Terminator, extends these two approaches by incorporating the notion of time during the analysis and enforcement, which leads to significant improvement on detection and prevention of security breaches. Finally, PATDroid provides an efficient permission-aware testing technique for Android apps, which can be used, among the other applications, to identify potential compatibility issues in the apps, whose permissions are revoked by Terminator. All conducted experiments corroborate the

effectiveness and efficiency of COVERT, SEPAR, TERMINATOR, and PATDROID and their ability to identify and eliminate the defects rooted in permission misuse of Android apps.

In the remainder of this chapter I conclude my dissertation by enumerating the contributions of my work and avenues for future work.

8.1 Research Contributions

The following is the concrete list of contributions in this research:

- **A taxonomy and qualitative comparison of Android security research:** I proposed a taxonomy for the papers published in 2008–2016 on Android security and, using that taxonomy, provided a qualitative comparison of the research. Comparing over 330 research papers, the conducted survey is the most comprehensive study in this line of research that revealed patterns, trends, and gaps in the existing literature, and underlined key challenges and opportunities that will shape the focus of future research efforts.

- **A formal model of Android framework:** I constructed a formal specification, as a reusable Alloy module, representing the behavior of Android framework and the apps. Also, by incorporating the notion of time, I extended the proposed model to reason about Android system's behavior as it evolves over time. The temporal model is developed as a TLA+ module that supports temporal logic analysis.

- **A formal model of permission-induced attacks:** I provided a formal definition (in both Alloy and TLA+ languages) of permission-induced attacks, collected through an extensive study of various sources, including but not limited to research papers, security reports, and Android documentations. Such attack model is not only useful

in precise identification of security vulnerabilities implemented in this research, e.g., COVERT and TERMINATOR, but also can be extended and re-used by the future research.

- **A model extractor for Android apps:** Several static analysis algorithms and methods are designed and developed in this research to extract different aspect of Android apps. In the development of these algorithms, I tackled the challenges of implementing static analysis techniques for Android apps, as discussed in Chapter 2. In addition to using the provided algorithms in this dissertation, they also contributed to the other researches I involved in, such as architecture recovery [77, 439], energy profiling [263, 264] and testing of Android apps [349].

- **A policy enforcement mechanism for Android framework:** I developed two policy enforcement mechanisms that apply the security rules at two levels: (1) event messaging (SEPAR approach), and (2) permission enforcement (TERMINATOR approach). Both approaches are realized through instrumentation of the root process of Android, i.e., without making any changes to the apps' APK or Android framework.

- **Fundamental contribution to combinatorial testing:** I proposed a novel approach that considers the dependencies between a program, its test suite, and access control model for the reduction of testing efforts. Although this approach is applied for testing Android apps (PATDROID approach), but its theoretical contribution is applicable to any software with a permission-based access-control model.

- **Large-scale empirical evaluation:** I extensively evaluated the effectiveness and efficiency of the proposed approaches by large-scale empirical evaluations over thousands of real-world Android apps, from various app repositories. Through conducting the experiments, I revealed several functional and non-functional (particularly security) defects in real-world apps and helped their developers fix those issues.

- **Tool implementation:** Finally, I developed working implementations of COVERT,

SEPAR, TERMINATOR, and PATDROID. To help other researchers re-use and expand the proposed approaches and build more advanced techniques on top of them, I made all research artifacts and tools publicly available, via the following web addresses:

- COVERT and SEPAR: `https://seal.ics.uci.edu/projects/covert`

  The reseach has a tool demo paper [416] and a demo video [23].

- PATDROID: `https://seal.ics.uci.edu/projects/patdroid`

  This research has received ACM artifact badge.

- TERMINATOR: `https://sites.google.com/view/terminator18`

8.2 Future Work

In this dissertation, I attempted to cover a number of research gaps (RGs) identified through the systematic literature review described in Chapter 2, including the development of hybrid approaches that leverage combination of program analysis with formal methods (RG1), for compositional analysis of inter-app security vulnerabilities (RG2) that go beyond Intents (RG4). As an avenue for the future research, I propose investigating the other research gaps, which are not addressed yet.

In addition, in the remainder of this chapter, I propose some other areas for the future research that are highlighted in Figure 8.1. This figure extends the research roadmap, presented in Figure 1.1 (Chapter 1), by adding the areas for the future research.

Working on *automatic exploit generation* for permission-induced vulnerabilities could be an extension of my work on identification of such vulnerabilities. Employing static analysis techniques, all detection approaches presented in this dissertation, and similar approaches, are prone to generating false alarms. Hence, investigating the exploitability of identified vulnerabilities can reduce false positive and help security analysis focus on real, exploitable

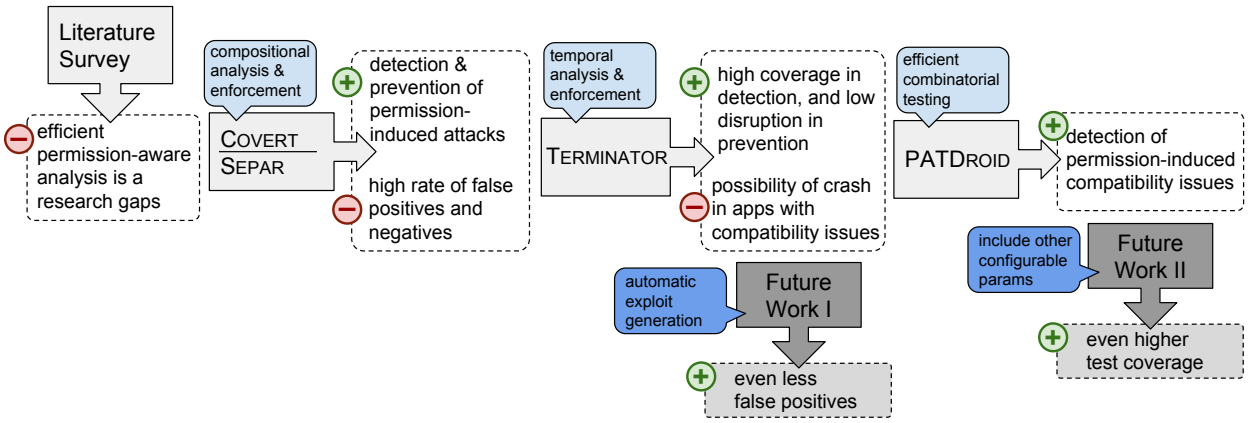


Figure 8.1: Dissertation Roadmap, including the avenues for future work.

security issues.

Android app testing is another theme of this dissertation, which is used for identification of permission-induced compatibility issues. As another area for the future research, I suggest extending the proposed combinatorial testing technique to include other configurable parameters in Android that can affect the behavior of programs, such as the settings for network and battery usage.

Finally, since the theoretical contribution of permission-aware analysis of Android is applicable to any software with a permission-based access-control model, one can investigate the applicability of the approaches, presented in this dissertation, to other platforms that use permission-based security model.

Bibliography

[1] Android api reference document. http://developer.android.com/reference.

[2] Android developers guide. http://developer.android.com/guide/topics/fundamentals.html.

[3] Android interface definition language. http://developer.android.com /guide/components/aidl.html. Online; accessed 2016.

[4] Android monkey. http://developer.android.com /guide/developing/tools/monkey.html/. Online; accessed 2016.

[5] Apktool. http://ibotpeaches.github.io/Apktool/. Online; accessed 2016.

[6] Cafe bazaar. https://cafebazaar.ir//.

[7] *D. Jackson, Software Abstractions, 2nd ed. MIT Press, 2012.*

[8] dex2jar. https://code.google.com/p/dex2jar/. Online; accessed 2016.

[9] Dexter. http://dexter.dexlabs.org/. Online; accessed 2016.

[10] Droidbench. http://sseblog.ec-spride.de/tools/droidbench. Online; accessed 2016.

[11] Droidbench2.0. http://github.com/secure-software-engineering/DroidBench/tree/iccta/apk.

[12] Droidbox. https://code.google.com/p/droidbox/. Online; accessed 2016.

[13] F-droid: Free and open source android app repository. https://f-droid.org/. Online; accessed 2016.

[14] Freemarker java template engine. http://freemarker.org/.

[15] Google play market. http://play.google.com/store/apps. Online; accessed 2016.

[16] Icc-bench. https://github.com/fgwei/ICC-Bench. Online; accessed 2016.

[17] Iccbench. `https://github.com/fgwei/ICC-Bench/tree/master/apks`.

[18] Iccta tool on github, reported issues. `https://github.com/lilicoding/soot-infoflow-android-iccta/issues/7`.

[19] smali. https://code.google.com/p/smali/. Online; accessed 2016.

[20] Virusshare. http://virusshare.com/. Online; accessed 2016.

[21] Virustotal. https://www.virustotal.com/. Online; accessed 2016.

[22] Nvd:cve-2014-8609. https://nvd.nist.gov/vuln/detail/CVE-2014-8609, 2014.

[23] Covert video demo on youtube. http://youtu.be/bMKk7OW7dGg, 2015.

[24] A2dp volume. https://github.com/jroal/a2dpvolume, 2017.

[25] Always on. https://github.com/rosenpin/AlwaysOnDisplayAmoled, 2017.

[26] Android api reference document. http://developer.android.com/reference, 2017.

[27] Android permissions best practices. https://developer.android.com/training/permissions/best-practices.html, 2017.

[28] Android studio user guide: Test your app. https://developer.android.com/studio/test/index.html, 2017.

[29] Android testing support library : Espresso. https://google.github.io/android-testing-support-library/docs/espresso/, 2017.

[30] Budget watch. https://github.com/brarcher/budget-watch, 2017.

[31] Dumbphone assistant. https://github.com/yeriomin/DumbphoneAssistant, 2017.

[32] Keeping your app responsive. https://developer.android.com/training/articles/perf-anr.html, 2017.

[33] Notes. https://github.com/SecUSo/privacy-friendly-notes, 2017.

[34] Patdroid: Permission-aware testing for android. http://www.ics.uci.edu/ seal/project-s/patdroid/, 2017.

[35] Pendingintent. https://developer.android.com/reference/android/app/PendingIntent.html, 2017.

[36] Radio beacon. https://github.com/openbmap/radiocells-scanner-android, 2017.

[37] Riot. https://github.com/vector-im/riot-android, 2017.

[38] Robotium. http://robotium.com/pages/about-us, 2017.

[39] Sms scheduler. https://github.com/yeriomin/SmsScheduler, 2017.

[40] Suntimes. https://github.com/forrestguice/SuntimesWidget, 2017.

[41] Syslog. https://github.com/Tortel/SysLog, 2017.

[42] Terminator web page. https://sites.google.com/view/terminator18, 2017.

[43] Working with system permissions. https://developer.android.com/training/permissions, 2017.

[44] Xposed framework. http://repo.xposed.info/, 2017.

[45] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *Security and Privacy in Communication Networks - 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers*, volume 127 of *SecureComm'13*, pages 86–103. Springer, 2013.

[46] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X.-y. Zhou, W. Du, and M. Grace. Hare Hunting in the Wild Android: A Study on the Threat of Hanging Attribute References. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, CCS'15, pages 1248–1259. ACM, 2015.

[47] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande, and V. V. T. Tong. GroddDroid: a gorilla for triggering malicious behaviors. In *10th International Conference on Malicious and Unwanted Software, MALWARE 2015, Fajardo, PR, USA, October 20-22, 2015*, MALWARE'15, pages 119–127. IEEE, 2015.

[48] J. P. Achara, M. Cunche, V. Roca, and A. Francillon. WifiLeaks: underestimated privacy implications of the access_wifi_state android permission. In *7th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WiSec'14, Oxford, United Kingdom, July 23-25, 2014*, WISEC'14, pages 231–236. ACM, 2014.

[49] J. P. Achara, J.-D. Lefruit, V. Roca, and C. Castelluccia. Detecting privacy leaks in the RATP App: how we proceeded and what we found. *J. Computer Virology and Hacking Techniques*, 10(4):229–238, 2014.

[50] O. S. Adebayo and N. AbdulAziz. Android malware classification using static code analysis and Apriori algorithm improved with particle swarm optimization. In *Information and Communication Technologies (WICT), 2014 Fourth World Congress on*, WICT'14, pages 123–128. IEEE, 2014.

[51] V. M. Afonso, M. F. d. Amorim, A. R. A. Grégio, G. B. Junquera, and P. L. d. Geus. Identifying Android malware using dynamically obtained features. *J. Computer Virology and Hacking Techniques*, 11(1):9–17, 2015.

[52] Y. Agarwal and M. Hall. ProtectMyPrivacy: detecting and mitigating privacy leaks on iOS devices using crowdsourcing. In *The 11th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'13, Taipei, Taiwan, June 25-28, 2013*, MobiSys'13, pages 97–110. ACM, 2013.

[53] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[54] A. Ali-Gombe, I. Ahmed, G. G. Richard III, and V. Roussev. OpSeq: Android Malware Fingerprinting. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop, PPREW@ACSAC, Los Angeles, CA, USA, December 8, 2015*, SSPREW'15, page 7. ACM, 2015.

[55] K. Allix, Q. Jérome, T. F. Bissyandé, J. Klein, R. State, and Y. L. Traon. A Forensic Analysis of Android Malware - How is Malware Written and How it Could Be Detected? In *IEEE 38th Annual Computer Software and Applications Conference, COMPSAC 2014, Vasteras, Sweden, July 21-25, 2014*, COMPSAC'14, pages 384–393. IEEE, 2014.

[56] H. M. J. Almohri, D. D. Yao, and D. G. Kafura. DroidBarrier: know what is executing on your android. In *Fourth ACM Conference on Data and Application Security and Privacy, CODASPY'14, San Antonio, TX, USA - March 03 - 05, 2014*, CODASPY'14, pages 257–264. ACM, 2014.

[57] A. Amamra, C. Talhi, and J.-M. Robert. Smartphone malware detection: From a survey towards taxonomy. In *7th International Conference on Malicious and Unwanted Software, MALWARE 2012, Fajardo, PR, USA, October 16-18, 2012*, pages 79–86. IEEE Computer Society, 2012.

[58] B. Amos, H. A. Turner, and J. White. Applying machine learning classifiers to dynamic Android malware detection at scale. In *2013 9th International Wireless Communications and Mobile Computing Conference, IWCMC 2013, Sardinia, Italy, July 1-5, 2013*, pages 1666–1671. IEEE, 2013.

[59] R. Andriatsimandefitra and V. V. T. Tong. Capturing Android Malware Behaviour Using System Flow Graph. In *Network and System Security - 8th International Conference, NSS 2014, Xi'an, China, October 15-17, 2014, Proceedings*, volume 8792 of *NSS'14*, pages 534–541. Springer, 2014.

[60] N. Andronio, S. Zanero, and F. Maggi. HelDroid: Dissecting and Detecting Mobile Ransomware. In *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings*, volume 9404 of *RAID'15*, pages 382–404. Springer, 2015.

[61] L. Apvrille and A. Apvrille. Identifying Unknown Android Malware with Feature Extractions and Classification Techniques. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*, TrustCom'15, pages 182–189. IEEE, 2015.

[62] M. Aresu, D. Ariu, M. Ahmadi, D. Maiorca, and G. Giacinto. Clustering android malware families by http traffic. In *10th International Conference on Malicious and Unwanted Software, MALWARE 2015, Fajardo, PR, USA, October 20-22, 2015*, MALWARE'15, pages 128–135. IEEE, 2015.

[63] A. Armando, G. Costa, and A. Merlo. Formal Modeling and Reasoning about the Android Security Framework. In *Trustworthy Global Computing - 7th International Symposium, TGC 2012, Newcastle upon Tyne, UK, September 7-8, 2012, Revised Selected Papers*, volume 8191 of *TGC'12*, pages 64–81. Springer, 2012.

[64] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, San Diego, CA, 2014.

[65] S. Arzt, K. Falzon, A. Follner, S. Rasthofer, E. Bodden, and V. Stolz. How Useful Are Existing Monitoring Languages for Securing Android Apps? In *Software Engineering 2013 - Workshopband (inkl. Doktorandensymposium), Fachtagung des GI-Fachbereichs Softwaretechnik, 26. Februar - 1. März 2013 in Aachen*, volume 215, pages 107–122. GI, 2013.

[66] S. Arzt, S. Rasthofer, and E. Bodden. Instrumenting Android and Java Applications as Easy as abc. In *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, volume 8174 of *RV'13*, pages 364–381. Springer, 2013.

[67] 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, PLDI'14, page 29, Edinburgh, UK, 2014. ACM.

[68] A. M. Aswini and P. Vinod. Droid permission miner: Mining prominent permissions for Android malware analysis. In *Applications of Digital Information and Web Technologies (ICADIWT), 2014 Fifth International Conference on the*, ICADIWT'14, pages 81–86. IEEE, 2014.

[69] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android Permission Specification. In *the {ACM} Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, CCS'12, pages 217–228, Raleigh, NC, 2012. ACM.

[70] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining Apps for Abnormal Usage of Sensitive Data. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, ICSE'15, pages 426–436. IEEE, 2015.

[71] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber. On demystifying the android application framework: Re-visiting android permission specification analysis. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1101–1118.

[72] M. Backes, S. Bugiel, and S. Gerling. Scippa: system-centric IPC provenance on Android. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, ACSAC'14, pages 36–45. ACM, 2014.

[73] M. Backes, S. Bugiel, S. Gerling, and P. v. Styp-Rekowsky. Android security framework: extensible multi-layered access control on Android. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, ACSAC'14, pages 46–55. ACM, 2014.

[74] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. v. Styp-Rekowsky. Boxify: Full-fledged App Sandboxing for Stock Android. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 691–706. USENIX Association, 2015.

[75] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. v. Styp-Rekowsky. AppGuard - Enforcing User Requirements on Android Apps. In *Tools and Algorithms for the Construction and Analysis of Systems- 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *TACAS'13*, pages 543–548. Springer, 2013.

[76] C. Bae, J. Jung, J. Nam, and S. Shin. A Collaborative Approach on Behavior-Based Android Malware Detection. In *Security and Privacy in Communication Networks - 11th International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Revised Selected Papers*, volume 164 of *SecureComm'15*, pages 594–597. Springer, 2015.

[77] H. Bagheri, J. Garcia, A. Sadeghi, S. Malek, and N. Medvidovic. Software architectural principles in contemporary mobile software: from conception to practice. *Journal of Systems and Software*, 119:31–44, 2016.

[78] H. Bagheri, E. Kang, S. Malek, and D. Jackson. Detection of Design Flaws in the Android Permission Protocol Through Bounded Verification. In *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, pages 73–89. Springer, 2015.

[79] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek. COVERT: compositional analysis of android inter-app permission leakage. *IEEE Trans. Software Eng. (TSE)*, 41(9), 2015.

[80] 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.

[81] H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek. Automated dynamic enforcement of synthesized security policies in android. Technical report, Department of Computer Science, George Mason University, 2015.

[82] H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek. Practical, formal synthesis and automatic enforcement of security policies for android. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*, pages 514–525, 2016.

[83] H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek. Practical, formal synthesis and automatic enforcement of security policies for android. In *Proceedings of the 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2016)*, pages 514–525, 2016.

[84] G. Bai, L. Gu, T. Feng, Y. Guo, and X. Chen. Context-Aware Usage Control for Android. In *Security and Privacy in Communication Networks - 6th Iternational ICST Conference, SecureComm 2010, Singapore, September 7-9, 2010. Proceedings*, volume 50 of *SecureComm'10*, pages 326–343. Springer, 2010.

[85] G. Bal. Revealing privacy-impacting behavior patterns of smartphone applications. In *IEEE Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy, MoST 2012, San Francisco, California, USA, May 24, 2012*, MOST'12, 2012.

[86] G. Barbon, A. Cortesi, P. Ferrara, M. Pistoia, and O. Tripp. Privacy Analysis of Android Apps: Implicit Flows and Quantitative Analysis. In *Computer Information Systems and Industrial Management - 14th IFIP TC 8 International Conference, CISIM 2015, Warsaw, Poland, September 24-26, 2015. Proceedings*, volume 9339 of *CISIM'15*, pages 3–23. Springer, 2015.

[87] D. Barrera, J. Clark, D. McCarney, and P. C. v. Oorschot. Understanding and improving app installation security mechanisms through empirical analysis of android. In *SPSM'12, Proceedings of the Workshop on Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2012, October 19, 2012, Raleigh, NC, USA*, SPSM'12, pages 81–92. ACM, 2012.

[88] D. Barrera, H. G. Kayacik, P. C. v. Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, CCS'10, pages 73–84. ACM, 2010.

[89] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 73–84.

[90] D. Barrera, D. McCarney, J. Clark, and P. C. van Oorschot. Baton: certificate agility for android's decentralized signing infrastructure. In *7th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WiSec'14, Oxford, United Kingdom, July 23-25, 2014*, WISEC'14, pages 1–12. ACM, 2014.

[91] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst. Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, ASE'15, pages 669–679. IEEE, 2015.

[92] 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 *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, ASE 2012, pages 274–277, Essen, Germany, 2012. ACM.

[93] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Dexpler: converting Android Dalvik bytecode to Jimple for static analysis with Soot*, pages 27–38. ACM, 2012.

[94] A. Bartel, J. Klein, M. Monperrus, and Y. L. Traon. Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges and Solutions for Analyzing Android. *IEEE Trans. Software Eng.*, 40(6):617–632, 2014.

[95] S. Bartsch, B. J. Berger, M. Bunke, and K. Sohr. The Transitivity-of-Trust Problem in Android Application Interaction. In *2013 International Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013*, ARES'13, pages 291–296. IEEE Computer Society, 2013.

[96] 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 *6th International Conference on Malicious and Unwanted Software, MALWARE 2011, Fajardo, Puerto Rico, USA, October 18-19, 2011*, MALWARE '11, pages 66–72, Fajardo, PR, 2011. IEEE Computer Society.

[97] I. Bente, G. Dreo, B. Hellmann, S. Heuser, J. Vieweg, J. v. Helden, and J. Westhuis. Towards Permission-Based Attestation for the Android Platform. In *Trust and Trustworthy Computing - 4th International Conference, TRUST 2011, Pittsburgh, PA, USA, June 22-24, 2011. Proceedings*, volume 6740 of *TRUST'11*, pages 108–115. Springer, 2011.

[98] A. R. Beresford, A. C. Rice, N. Skehin, and R. Sohan. MockDroid: trading privacy for application functionality on smartphones. In *12th Workshop on Mobile Computing Systems and Applications, HotMobile '11, Phoenix, AZ, USA, March 1-3, 2011*, HotMobile'11, pages 49–54. ACM, 2011.

[99] P. Berthomé, T. Fécherolle, N. Guilloteau, and J.-F. Lalande. Repackaging Android Applications for Auditing Access to Private Data. In *Seventh International Conference on Availability, Reliability and Security, Prague, ARES 2012, Czech Republic, August 20-24, 2012*, ARES'15, pages 388–396. IEEE Computer Society, 2012.

[100] S. Bhandari, R. Gupta, V. Laxmi, M. S. Gaur, A. Zemmari, and M. Anikeev. DRACO: DRoid analyst combo an android malware analysis framework. In *Proceedings of the 8th International Conference on Security of Information and Networks, SIN 2015, Sochi, Russian Federation, September 8-10, 2015*, SIN'15, pages 283–289. ACM, 2015.

[101] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall. Brahmastra: Driving Apps to Test the Security of Third-Party Components. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, SEC'15, pages 1021–1036. USENIX Association, 2014.

[102] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the App is That? Deception and Countermeasures in the Android User Interface. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, S&P'15, pages 931–948. IEEE Computer Society, 2015.

[103] M. Bierma, E. Gustafson, J. Erickson, D. Fritz, and Y. R. Choe. Andlantis: Large-scale Android Dynamic Analysis. *arXiv:1410.7751 [cs]*, Oct. 2014.

[104] S. Blackshear, B.-Y. E. Chang, and M. Sridharan. Thresher: precise refutations for heap reachability. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, PLDI'13, pages 275–286. ACM, 2013.

[105] S. Blackshear, A. Gendreau, and B.-Y. E. Chang. Droidel: a general approach to Android framework modeling. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2015, Portland, OR, USA, June 15 - 17, 2015*, SOAP'15, pages 19–25. ACM, 2015.

[106] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Çamtepe, and S. Albayrak. An Android Application Sandbox system for suspicious software detection. In *5th International Conference on Malicious and Unwanted Software, MALWARE 2010, Nancy, France, October 19-20, 2010*, MALWARE'10, pages 55–62. IEEE, 2010.

[107] E. Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 3–8. ACM, 2012.

[108] N. Boggs, W. Wang, S. Mathur, B. Coskun, and C. Pincock. Discovery of emergent malicious campaigns in cellular networks. In *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, ACSAC'13, pages 29–38. ACM, 2013.

[109] S. Bojjagani and V. N. Sastry. STAMBA: Security Testing for Android Mobile Banking Apps. In *Advances in Signal Processing and Intelligent Recognition Systems - Proceedings of Second International Symposium on Signal Processing and Intelligent Recognition Systems (SIRS-2015) December 16-19, 2015, Trivandrum, India*, volume 425 of *SIRS'15*, pages 671–683. Springer, 2015.

[110] A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral detection of malware on mobile handsets. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services (MobiSys 2008), Breckenridge, CO, USA, June 17-20, 2008*, pages 225–238. ACM, 2008.

[111] A. Bosu, F. Liu, D. D. Yao, and G. Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, pages 71–85, 2017.

[112] 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*, 80(4):571–583, Apr. 2007.

[113] M. Bucicoiu, L. Davi, R. Deaconescu, and A.-R. Sadeghi. XiOS: Extended Application Sandboxing on iOS. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, ASIACCS'15, pages 43–54. ACM, 2015.

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

[115] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, NDSS'12. The Internet Society, 2012.

[116] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and lightweight domain isolation on Android. In *SPSM'11, Proceedings of the 1st ACM Workshop Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2011, October 17, 2011, Chicago, IL, USA*, pages 51–62. ACM, 2011.

[117] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, SEC'13, pages 131–146. USENIX Association, 2013.

[118] M. Bugliesi, S. Calzavara, and A. Spanò. Lintent: Towards Security Type-Checking of Android Applications. In *Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOODS/FORTE 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings*, volume 7892 of *DisCoTec'13*, pages 289–304. Springer, 2013.

[119] D. Buhov, M. Huber, G. Merzdovnik, E. R. Weippl, and V. Dimitrova. Network Security Challenges in Android Applications. In *10th International Conference on*

Availability, Reliability and Security, ARES 2015, Toulouse, France, August 24-27, 2015, ARES'15, pages 327–332. IEEE, 2015.

[120] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for Android. In *SPSM'11, Proceedings of the 1st ACM Workshop Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2011, October 17, 2011, Chicago, IL, USA*, SPSM'11, pages 15–26. ACM, 2011.

[121] F. Cai, H. Chen, Y. Wu, and Y. Zhang. AppCracker: Widespread Vulnerabilities in User and Session Authentication in Mobile Apps. Technical Report SHTech/SIST-2014-1, 2014.

[122] S. Calzavara, I. Grishchenko, and M. Maffei. HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving.

[123] G. Canfora, A. D. Lorenzo, E. Medvet, F. Mercaldo, and C. A. Visaggio. Effectiveness of Opcode ngrams for Detection of Multi Family Android Malware. In *10th International Conference on Availability, Reliability and Security, ARES 2015, Toulouse, France, August 24-27, 2015*, ARES'15, pages 333–340. IEEE, 2015.

[124] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio. Detecting Android malware using sequences of system calls. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile, DeMobile 2015, Bergamo, Italy, August 31 - September 4, 2015*, DeMobile'15, pages 13–20. ACM, 2015.

[125] G. Canfora, F. Mercaldo, G. Moriano, and C. A. Visaggio. Composition-Malware: Building Android Malware at Run Time. In *10th International Conference on Availability, Reliability and Security, ARES 2015, Toulouse, France, August 24-27, 2015*, ARES'15, pages 318–326. IEEE, 2015.

[126] G. Canfora, F. Mercaldo, and C. A. Visaggio. A Classifier of Malicious Android Applications. In *2013 International Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013*, ARES'13, pages 607–614. IEEE Computer Society, 2013.

[127] C. Cao, N. Gao, P. Liu, and J. Xiang. Towards Analyzing the Input Validation Vulnerabilities associated with Android System Services. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, ACSAC'15, pages 361–370. ACM, 2015.

[128] C. Cao, Y. Zhang, Q. Liu, and K. Wang. Function Escalation Attack. In *International Conference on Security and Privacy in Communication Networks - 10th International ICST Conference, SecureComm 2014, Beijing, China, September 24-26, 2014, Revised Selected Papers, Part I*, volume 152 of *SecureComm'14*, pages 481–497. Springer, 2014.

[129] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the

Android Framework. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*, San Diego, CA, 2015.

[130] L. Cen, C. S. Gates, L. Si, and N. Li. A Probabilistic Discriminative Model for Android Malware Detection with Decompiled Source Code. *IEEE Trans. Dependable Sec. Comput.*, 12(4):400–412, 2015.

[131] F. D. Cerbo, A. Girardello, F. Michahelles, and S. Voronkova. Detection of Malicious Applications on Android OS. In *Computational Forensics - 4th International Workshop, IWCF 2010, Tokyo, Japan, November 11-12, 2010, Revised Selected Papers*, volume 6540 of *IWCF'10*, pages 138–149. Springer, 2010.

[132] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. MAST: triage for market-scale mobile malware analysis. In *Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks, WISEC'13, Budapest, Hungary, April 17-19, 2013*, WISEC'13, pages 13–24. ACM, 2013.

[133] 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, WISEC 2012, Tucson, AZ, USA, April 16-18, 2012*, WiSec '12, pages 125–136, Tucson, Arizona, 2012. ACM.

[134] A. Chaudhuri. Language-based security on android. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*, PLAS '09, pages 1–7, Dublin, Ireland, 2009. ACM.

[135] C.-M. Chen, J.-M. Lin, and G. H. Lai. Detecting Mobile Application Malicious Behaviors Based on Data Flow of Source Code. In *2014 International Conference on Trustworthy Systems and their Applications, TSA 2014, Taichung, Taiwan, June 9-10, 2014*, TSA'14, pages 1–6. IEEE, 2014.

[136] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, ICSE'14, pages 175–186. ACM, 2014.

[137] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, SEC'15, pages 659–674. USENIX Association, 2015.

[138] 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 *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, San Diego, CA, 2013.

[139] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, SEC'14, pages 1037–1052. USENIX Association, 2014.

[140] Y.-L. Chen, H.-M. Lee, A. B. Jeng, and T.-E. Wei. DroidCIA: A Novel Detection Method of Code Injection Attacks on HTML5-Based Mobile Apps. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*, TrustCom'15, pages 1014–1021. IEEE, 2015.

[141] J. Cheng, S. H. Y. Wong, H. Yang, and S. Lu. SmartSiren: virus detection and alert for smartphones. In *Proceedings of the 5th International Conference on Mobile Systems, Applications, and Services (MobiSys 2007), San Juan, Puerto Rico, June 11-13, 2007*, pages 258–271. ACM, 2007.

[142] S. Cheng, S. Luo, Z. Li, W. Wang, Y. Wu, and F. Jiang. Static Detection of Dangerous Behaviors in Android Apps. In *Cyberspace Safety and Security - 5th International Symposium, CSS 2013, Zhangjiajie, China, November 13-15, 2013, Proceedings*, volume 8300 of *CSS'13*, pages 363–376. Springer, 2013.

[143] 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 (MobiSys 2011), Bethesda, MD, USA, June 28 - July 01, 2011*, pages 239–252, Washington, DC, 2011. ACM.

[144] E. Chin and D. Wagner. Bifocals: Analyzing WebView Vulnerabilities in Android Applications. In *Information Security Applications - 14th International Workshop, WISA 2013, Jeju Island, Korea, August 19-21, 2013, Revised Selected Papers*, volume 8267 of *Lecture Notes in Computer Science*, pages 138–159. Springer, 2013.

[145] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proc. of SAS*, 2003.

[146] M. Conti, E. Fernandes, J. Paupore, A. Prakash, and D. Simionato. OASIS: Operational Access Sandboxes for Information Security. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, SPSM@CCS 2014, Scottsdale, AZ, USA, November 03 - 07, 2014*, pages 105–110. ACM, 2014.

[147] M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-Related Policy Enforcement for Android. In *Information Security - 13th International Conference, ISC 2010, Boca Raton, FL, USA, October 25-28, 2010, Revised Selected Papers*, volume 6531 of *ISC'10*, pages 331–345. Springer, 2010.

[148] B. Cooley, H. Wang, and A. Stavrou. Activity Spoofing and Its Defense in Android Smartphones. In *Applied Cryptography and Network Security - 12th International Conference, ACNS 2014, Lausanne, Switzerland, June 10-13, 2014. Proceedings*, volume 8479 of *ACNS'14*, pages 494–512. Springer, 2014.

[149] V. Costamagna and C. Zheng. ARTDroid: A Virtual-Method Hooking Framework on Android ART Runtime. In *Proceedings of the 1st International Workshop on Innovations in Mobile Privacy and Security, IMPS 2016, co-located with the International Symposium on Engineering Secure Software and Systems (ESSoS 2016), London, UK, April 6, 2016*, volume 1575 of *IMPS'16*, pages 20–28. CEUR-WS.org, 2016.

[150] M. J. Cresswell and G. E. Hughes. *A new introduction to modal logic*. Routledge, 2012.

[151] J. Crussell, C. Gibler, and H. Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, volume 7459 of *ESORICS'12*, pages 37–54. Springer, 2012.

[152] J. Crussell, C. Gibler, and H. Chen. AnDarwin: Scalable Detection of Semantically Similar Android Applications. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, volume 8134 of *ESORICS'13*, pages 182–199. Springer, 2013.

[153] J. Crussell, R. Stevens, and H. Chen. MAdFraud: investigating ad fraud in android applications. In *The 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'14, Bretton Woods, NH, USA, June 16-19, 2014*, MobiSys'14, pages 123–134. ACM, 2014.

[154] X. Cui, J. Wang, L. C. K. Hui, Z. Xie, T. Zeng, and S.-M. Yiu. WeChecker: efficient and precise detection of privilege escalation vulnerabilities in Android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, New York, NY, USA, June 22-26, 2015*, WISEC'15, pages 25:1–25:12. ACM, 2015.

[155] X. Cui, D. Yu, P. P. F. Chan, L. C. K. Hui, S.-M. Yiu, and S. Qing. CoChecker: Detecting Capability and Sensitive Data Leaks from Component Chains in Android. In *Information Security and Privacy - 19th Australasian Conference, ACISP 2014, Wollongong, NSW, Australia, July 7-9, 2014. Proceedings*, volume 8544 of *ACISP'14*, pages 446–453. Springer, 2014.

[156] D. Dagon, T. Martin, and T. Starner. Mobile phones as computing devices: The viruses are coming! *Pervasive Computing, IEEE*, 3(4):11–15, 2004.

[157] G. Dai, J. Ge, M. Cai, D. Xu, and W. Li. SVM-based malware detection for Android applications. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, New York, NY, USA, June 22-26, 2015*, WISEC'15, pages 33:1–33:2. ACM, 2015.

[158] S. Dai, T. Wei, and W. Zou. DroidLogger: Reveal suspicious behavior of Android applications via instrumentation. In *Computing and Convergence Technology (ICCCT), 2012 7th International Conference on*, ICCCT'12, pages 550–555. IEEE, 2012.

[159] M. Dam, G. L. Guernic, and A. Lundblad. TreeDroid: a tree automaton based approach to enforcing data processing policies. In *the ACM Conference on Computer and*

Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012, CCS'12, pages 894–905. ACM, 2012.

[160] L. Davi, A. Dmitrienko, A. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Information Security - 13th International Conference, ISC 2010, Boca Raton, FL, USA, October 25-28, 2010, Revised Selected Papers*, pages 346–360, 2010.

[161] B. Davis and H. Chen. Retroskeleton: Retrofitting android apps. In *The 11th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'13, Taipei, Taiwan, June 25-28, 2013*, MobiSys '13, pages 181–192, Taipei, Taiwan, 2013. ACM.

[162] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. In *IEEE Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy, MoST 2012, San Francisco, California, USA, May 24, 2012*, MoST'12, San Francisco, CA, 2012.

[163] H. Deng, Q. Wu, B. Qin, W. Susilo, J. K. Liu, and W. Shi. Asymmetric Cross-cryptosystem Re-encryption Applicable to Efficient and Secure Mobile Access to Outsourced Data. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, ASIACCS'15, pages 393–404. ACM, 2015.

[164] L. Deshotels, V. Notani, and A. Lakhotia. DroidLegacy: Automated Familial Classification of Android Malware. In *Proceedings of the 3rd ACM SIGPLAN Program Protection and Reverse Engineering Workshop 2014, PPREW 2014, January 25, 2014, San Diego, CA*, PPREW'14, pages 3:1–3:12. ACM, 2014.

[165] A. Desnos. Android: Static Analysis Using Similarity Distance. In *45th Hawaii International International Conference on Systems Science (HICSS-45 2012), Proceedings, 4-7 January 2012, Grand Wailea, Maui, HI, USA*, HICSS'12, pages 5394–5403. IEEE Computer Society, 2012.

[166] A. Desnos and G. Gueguen. Android: From reversing to decompilation. In *Black Hat, 2011, Abu Dhabi*, BlackHat'11, pages 77–101, 2011.

[167] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*, SEC'11. USENIX Association, 2011.

[168] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra. MADAM: A Multi-level Anomaly Detector for Android Malware. In *Computer Network Security - 6th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security, MMM-ACNS 2012, St. Petersburg, Russia, October 17-19, 2012. Proceedings*, volume 7531 of *MMM-ACNS'12*, pages 240–253. Springer, 2012.

[169] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra. Probabilistic Contract Compliance for Mobile Applications. In *2013 International Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013*, ARES'15, pages 599–606. IEEE Computer Society, 2013.

[170] B. Dixon, Y. Jiang, A. Jaiantilal, and S. Mishra. Spsm'11, proceedings of the 1st ACM workshop security and privacy in smartphones and mobile devices, co-located with CCS 2011, october 17, 2011, chicago, il, USA. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 27–32, Chicago, IL, 2011. ACM.

[171] Q. Do, B. Martini, and K.-K. R. Choo. Enforcing File System Permissions on Android External Storage: Android File System Permissions (AFP) Prototype and ownCloud. In *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2014, Beijing, China, September 24-26, 2014*, TrustCom'14, pages 949–954. IEEE Computer Society, 2014.

[172] L. Dolberg, Q. Jérôme, J. François, R. State, and T. Engel. RAMSES: Revealing Android Malware Through String Extraction and Selection. In *International Conference on Security and Privacy in Communication Networks - 10th International ICST Conference, SecureComm 2014, Beijing, China, September 24-26, 2014, Revised Selected Papers, Part I*, volume 152 of *SecureComm'14*, pages 498–506. Springer, 2014.

[173] T. Eder, M. Rodler, D. Vymazal, and M. Zeilinger. ANANAS - A Framework for Analyzing Android Applications. In *2013 International Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013*, ARES'13, pages 711–719. IEEE Computer Society, 2013.

[174] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 73–84. ACM, 2013.

[175] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, NDSS'11. The Internet Society, 2011.

[176] A. Egners, U. Meyer, and B. Marschollek. Messing with Android's permission model. In *11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2012, Liverpool, United Kingdom, June 25-27, 2012*, TrustCom'12, pages 505–514. IEEE, 2012.

[177] K. O. Elish, D. Yao, and B. G. Ryder. User-centric dependence analysis for identifying malicious mobile apps. In *IEEE Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy, MoST 2012, San Francisco, California, USA, May 24, 2012*, MoST'12, San Francisco, CA, 2012.

[178] K. O. Elish, D. Yao, and B. G. Ryder. On the need of precise inter-app ICC classification for detecting Android malware collusions. In *IEEE Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy, MoST 2015, San Jose, California, USA, May 21, 2015*, MoST'15, San Jose, CA, 2015.

[179] K. O. Elish, D. D. Yao, B. G. Ryder, and X. Jiang. A static assurance analysis of android applications. Technical Report TR-13-03, Department of Computer Science, Virginia Polytechnic Institute & State University, 2013.

[180] W. Enck. Defending Users against Smartphone Apps: Techniques and Future Directions. In *Information Systems Security - 7th International Conference, ICISS 2011, Kolkata, India, December 15-19, 2011, Procedings*, volume 7093 of *Lecture Notes in Computer Science*, pages 49–70. Springer, 2011.

[181] 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. pages 393–407, 2010.

[182] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*, SEC'11, pages 21–21, San Francisco, CA, 2011. USENIX Association.

[183] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 235–245, Chicago, IL, 2009. ACM.

[184] M. D. Ernst. Invited talk static and dynamic analysis: synergy and duality. In *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'04, Washington, DC, USA, June 7-8, 2004*, page 35, 2004.

[185] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1092–1104, Scottsdale, AZ, 2014. ACM.

[186] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android SSL (in)security. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, CCS '12, pages 50–61, Raleigh, NC, 2012. ACM.

[187] L. Falsina, Y. Fratantonio, S. Zanero, C. Kruegel, G. Vigna, and F. Maggi. Grab 'n Run: Secure and Practical Dynamic Code Loading for Android Applications. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, ACSAC'15, pages 201–210. ACM, 2015.

[188] Z. Fang, W. Han, and Y. Li. Permission based android security: Issues and countermeasures. *Computers & Security*, 43:205–218, 2014.

[189] Z. Fang, W. Han, and Y. Li. Permission based Android security: Issues and countermeasures. *Computers & Security*, 43:205–218, 2014.

[190] Z. Fang, Q. Liu, Y. Zhang, K. Wang, and Z. Wang. IVDroid: Static Detection for Input Validation Vulnerability in Android Inter-component Communication. In *Information Security Practice and Experience - 11th International Conference, ISPEC 2015, Beijing, China, May 5-8, 2015. Proceedings*, volume 9065 of *ISPEC'15*, pages 378–392. Springer, 2015.

[191] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan. Android Security: A Survey of Issues, Malware Penetration, and Defenses. *IEEE Communications Surveys and Tutorials*, 17(2):998–1022, 2015.

[192] P. Faruki, A. Bharmal, V. Laxmi, M. S. Gaur, M. Conti, and M. Rajarajan. Evaluation of Android Anti-malware Techniques against Dalvik Bytecode Obfuscation. In *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2014, Beijing, China, September 24-26, 2014*, pages 414–421. IEEE Computer Society, 2014.

[193] P. Faruki, V. Ganmoor, V. Laxmi, M. S. Gaur, and A. Bharmal. AndroSimilar: robust statistical feature signature for Android malware detection. In *The 6th International Conference on Security of Information and Networks, SIN '13, Aksaray, Turkey, November 26-28, 2013*, SIN'13, pages 152–159. ACM, 2013.

[194] R. Fedler, M. Kulicke, and J. Schütte. An antivirus API for Android malware recognition. In *8th International Conference on Malicious and Unwanted Software: "The Americas", MALWARE 2013, Fajardo, PR, USA, October 22-24, 2013*, MALWARE'13, pages 77–84. IEEE Computer Society, 2013.

[195] R. Fedler, M. Kulicke, and J. Schütte. Native code execution control for attack mitigation on android. In *SPSM'13, Proceedings of the 2013 ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2013, November 8, 2013, Berlin, Germany*, SPSM'13, pages 15–20. ACM, 2013.

[196] A. Feizollah, N. B. Anuar, R. Salleh, and A. W. A. Wahab. A review on feature selection in mobile malware detection. *Digital Investigation*, 13:22–37, 2015.

[197] S. Feldman, D. Stadther, and B. Wang. Manilyzer: Automated Android Malware Detection through Manifest Analysis. In *11th IEEE International Conference on Mobile Ad Hoc and Sensor Systems, MASS 2014, Philadelphia, PA, USA, October 28-30, 2014*, MASS'14, pages 767–772. IEEE Computer Society, 2014.

[198] 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, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, CCS '11, pages 627–638, Chicago, IL, 2011. ACM.

[199] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner. How to Ask for Permission. In *7th USENIX Workshop on Hot Topics in Security, HotSec'12, Bellevue, WA, USA, August 7, 2012*, HotSec'12. USENIX Association, 2012.

[200] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *SPSM'11, Proceedings of the 1st ACM Workshop Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2011, October 17, 2011, Chicago, IL, USA*, pages 3–14. ACM, 2011.

[201] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: user attention, comprehension, and behavior. In *Symposium On Usable Privacy and Security, SOUPS '12, Washington, DC, USA - July 11 - 13, 2012*, SOUPS'12, page 3. ACM, 2012.

[202] A. P. Felt, S. Hanna, E. Chin, H. J. Wang, and E. Moshchuk. Permission re-delegation: Attacks and defenses. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*, San Francisco, CA, 2011.

[203] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Hong Kong, China, Nov. 2014.

[204] P. Ferrara, O. Tripp, and M. Pistoia. MorphDroid: Fine-grained Privacy Verification. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, ACSAC'15, pages 371–380. ACM, 2015.

[205] D. Feth and A. Pretschner. Flexible Data-Driven Security for Android. In *Sixth International Conference on Software Security and Reliability, SERE 2012, Gaithersburg, Maryland, USA, 20-22 June 2012*, SERE'12, pages 41–50. IEEE, 2012.

[206] W. Ford and M. S. Baum. *Secure electronic commerce: building the infrastructure for digital signatures and encryption*. Prentice Hall PTR, 2000.

[207] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey. Modeling and enhancing android's permission system. In *17th European Symposium on Research in Computer Security*, Lecture Notes in Computer Science, pages 1–18, Pisa, Italy, Sept. 2012. Springer Berlin Heidelberg.

[208] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey. Modeling and enhancing android's permission system. In *European Symposium on Research in Computer Security*, pages 1–18. Springer, 2012.

[209] Y. Fratantonio, A. Bianchi, W. K. Robertson, M. Egele, C. Kruegel, E. Kirda, and G. Vigna. On the Security and Engineering Implications of Finer-Grained Access Controls for Android Developers and Users. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*, volume 9148 of *DIMVA'15*, pages 282–303. Springer, 2015.

[210] F. C. Freiling, M. Protsenko, and Y. Zhuang. An Empirical Evaluation of Software Obfuscation Techniques Applied to Android APKs. In *International Conference on Security and Privacy in Communication Networks - 10th International ICST Conference, SecureComm 2014, Beijing, China, September 24-26, 2014, Revised Selected Papers, Part II*, volume 153 of *SecureComm'14*, pages 315–328. Springer, 2014.

[211] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated security certification of android applications. Technical Report Technical Report CS-TR-4991, Department of Computer Science, University of Maryland, College Park, 2009.

[212] J. Gajrani, J. Sarswat, M. Tripathi, V. Laxmi, M. S. Gaur, and M. Conti. A robust dynamic analysis system preventing SandBox detection by Android malware. In *Proceedings of the 8th International Conference on Security of Information and Networks, SIN 2015, Sochi, Russian Federation, September 8-10, 2015*, SIN'15, pages 290–295. ACM, 2015.

[213] D. Gallingani, R. Gjomemo, V. N. Venkatakrishnan, and S. Zanero. Static detection and automatic exploitation of intent message vulnerabilities in Android applications. In *BlackHat, Las Vegas, US, 2014*, BlackHat'14, 2014.

[214] R. Gallo, P. Hongo, R. Dahab, L. C. Navarro, H. Kawakami, K. Galvão, G. Junqueira, and L. Ribeiro. Security and system architecture: comparison of Android customizations. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, New York, NY, USA, June 22-26, 2015*, WISEC'15, pages 12:1–12:6. ACM, 2015.

[215] Gartner Inc. Gartner reveals top predictions for IT organizations and users for 2012 and beyond. `http://www.gartner.com/newsroom/id/1862714`, 2011.

[216] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of android malware using embedded call graphs. In *AISec'13, Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security, Co-located with CCS 2013, Berlin, Germany, November 4, 2013*, AISEC'13, pages 45–54. ACM, 2013.

[217] C. S. Gates, J. Chen, N. Li, and R. W. Proctor. Effective Risk Communication for Android Apps. *IEEE Trans. Dependable Sec. Comput.*, 11(3):252–265, 2014.

[218] C. S. Gates, N. Li, H. Peng, B. P. Sarma, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Generating Summary Risk Scores for Mobile Applications. *IEEE Trans. Dependable Sec. Comput.*, 11(3):238–251, 2014.

[219] D. Geneiatakis, I. N. Fovino, I. Kounelis, and P. Stirparo. A Permission verification approach for android mobile applications. *Computers & Security*, 49:192–205, 2015.

[220] M. Georgiev, S. Jana, and V. Shmatikov. Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, NDSS'14. The Internet Society, 2014.

[221] A. Gianazza, F. Maggi, A. Fattori, L. Cavallaro, and S. Zanero. PuppetDroid: A User-Centric UI Exerciser for Automatic Dynamic Analysis of Similar Android Applications. *arXiv:1402.4826 [cs]*, Feb. 2014.

[222] 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*, TRUST'12, pages 291–307, Vienna, Austria, 2012. Springer-Verlag.

[223] 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, 2011*, pages 21–26. ACM, 2011.

[224] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, volume 8, pages 151–166, 2008.

[225] H. Gonzalez, A. A. Kadir, N. Stakhanova, A. J. Alzahrani, and A. A. Ghorbani. Exploring reverse engineering symptoms in Android apps. In *Proceedings of the Eighth European Workshop on System Security, EuroSec 2015, Bordeaux, France, April 21, 2015*, EuroSec'15, pages 7:1–7:7. ACM, 2015.

[226] H. Gonzalez, N. Stakhanova, and A. A. Ghorbani. DroidKin: Lightweight Detection of Android Apps Similarity. In *International Conference on Security and Privacy in Communication Networks - 10th International ICST Conference, SecureComm 2014, Beijing, China, September 24-26, 2014, Revised Selected Papers, Part I*, volume 152 of *SecureComm'14*, pages 436–453. Springer, 2014.

[227] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-Flow Analysis of Android Applications in DroidSafe. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*, San Diego, CA, 2015.

[228] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, ICSE 2014, pages 1025–1035, Hyderabad, India, 2014. ACM.

[229] M. Graa, N. Cuppens-Boulahia, F. Cuppens, and A. R. Cavalli. Detecting Control Flow in Smarphones: Combining Static and Dynamic Analyses. In *Cyberspace Safety and Security*, CSS'12, pages 33–47, 2012.

[230] M. Graa, N. Cuppens-Boulahia, F. Cuppens, and A. R. Cavalli. Protection against Code Obfuscation Attacks Based on Control Dependencies in Android Systems. In *IEEE Eighth International Conference on Software Security and Reliability, SERE 2014, San Francisco, CA, USA, June 30 - July 2, 2014 - Companion Volume*, SERE'14, pages 149–157. IEEE, 2014.

[231] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *The 10th International Conference on Mobile Systems, Applications, and Services, MobiSys'12, Ambleside, United Kingdom - June 25 - 29, 2012*, pages 281–294, Washington, DC, 2012. ACM.

[232] 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, WISEC 2012, Tucson, AZ, USA, April 16-18, 2012*, WISEC '12, pages 101–112, Tucson, AZ, 2012. ACM.

[233] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, San Diego, CA, 2012.

[234] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang. Characterizing and detecting resource leaks in Android applications. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, ASE'13, pages 389–398. IEEE, 2013.

[235] H.-S. Ham and M.-J. Choi. Analysis of android malware detection performance using machine learning classifiers. In *ICT Convergence (ICTC), 2013 International Conference on, 2013*, pages 490–495. IEEE, 2013.

[236] Y. J. Ham, D. Moon, H.-W. Lee, J. D. Lim, and J. N. Kim. Android mobile application system call event pattern analysis for determination of malicious attack. *International Journal of Security and Its Applications*, 8(1):231–246, 2014.

[237] K. Hamandi, A. Chehab, I. H. Elhajj, and A. I. Kayssi. Android SMS Malware: Vulnerability and Mitigation. In *27th International Conference on Advanced Information Networking and Applications Workshops, WAINA 2013, Barcelona, Spain, March 25-28, 2013*, AINA'13, pages 1004–1009. IEEE Computer Society, 2013.

[238] M. Hammad, H. Bagheri, and S. Malek. Determination and enforcement of least-privilege architecture in android. In *2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3-7, 2017*, pages 59–68, 2017.

[239] H. Han, R. Li, J. Hu, and M. Qiu. Context Awareness through Reasoning on Private Analysis for Android Application. In *IEEE 2nd International Conference on Cyber Security and Cloud Computing, CSCloud 2015, New York, NY, USA, November 3-5, 2015*, CSCloud'15, pages 148–156. IEEE, 2015.

[240] Z. Han, L. Cheng, Y. Zhang, S. Zeng, Y. Deng, and X. Sun. Systematic Analysis and Detection of Misconfiguration Vulnerabilities in Android Smartphones. In *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2014, Beijing, China, September 24-26, 2014*, TrustCom'14, pages 432–439. IEEE Computer Society, 2014.

[241] S. Hanna, L. Huang, E. X. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A Scalable System for Detecting Code Reuse among Android Applications. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 9th International Conference, DIMVA 2012, Heraklion, Crete, Greece, July 26-27, 2012, Revised Selected Papers*, volume 7591 of *DIMVA'12*, pages 62–81. Springer, 2012.

[242] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. SIF: a selective instrumentation framework for mobile applications. In *The 11th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'13, Taipei, Taiwan, June 25-28, 2013*, MobiSys'13, pages 167–180. ACM, 2013.

[243] S. Hao, B. Liu, S. Nath, W. G. J. Halfond, and R. Govindan. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *The 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'14, Bretton Woods, NH, USA, June 16-19, 2014*, MobiSys'14, pages 204–217. ACM, 2014.

[244] M. Harbach, M. Hettig, S. Weber, and M. Smith. Using personal examples to improve risk communication for security & privacy decisions. In *CHI Conference on Human Factors in Computing Systems, CHI'14, Toronto, ON, Canada - April 26 - May 01, 2014*, pages 2647–2656. ACM, 2014.

[245] N. Hardy. The confused deputy:(or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[246] M. Haris, H. Haddadi, and P. Hui. Privacy Leakage in Mobile Computing: Tools, Methods, and Characteristics. *arXiv:1410.4978 [cs]*, Oct. 2014. arXiv: 1410.4978.

[247] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi. ASM: A Programmable Interface for Extending Android Security. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, SEC'14, pages 1005–1019. USENIX Association, 2014.

[248] T.-H. Ho, D. J. Dean, X. Gu, and W. Enck. PREC: practical root exploit containment for android devices. In *Fourth ACM Conference on Data and Application Security and Privacy, CODASPY'14, San Antonio, TX, USA - March 03 - 05, 2014*, CODASPY'14, pages 187–198. ACM, 2014.

[249] J. Hoffmann, S. Neumann, and T. Holz. Mobile malware detection based on energy fingerprints—a dead end? In *Research in Attacks, Intrusions, and Defenses*, pages 348–368. Springer, 2013.

[250] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth. Slicing droids: program slicing for smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, SAC'13, pages 1844–1851. ACM, 2013.

[251] S. Holavanalli, D. Manuel, V. Nanjundaswamy, B. Rosenberg, F. Shen, S. Ko, and L. Ziarek. Flow permissions for android. In *2013 28th IEEE/ACM International*

Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013, pages 652–657, Nov. 2013.

[252] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, CCS '11, pages 639–652, Chicago, IL, 2011. ACM.

[253] W. Hu, D. Octeau, P. D. McDaniel, and P. Liu. Duet: library integrity verification for android applications. In *7th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WiSec'14, Oxford, United Kingdom, July 23-25, 2014*, WISEC'14, pages 141–152. ACM, 2014.

[254] W. Hu, J. Tao, X. Ma, W. Zhou, S. Zhao, and T. Han. MIGDroid: Detecting APP-Repackaging Android malware via method invocation graph. In *23rd International Conference on Computer Communication and Networks, ICCCN 2014, Shanghai, China, August 4-7, 2014*, ICCCN'14, pages 1–7. IEEE, 2014.

[255] C.-Y. Huang, Y.-T. Tsai, and C.-H. Hsu. Performance evaluation on permission-based detection for android malware. *Advances in Intelligent Systems and Applications*, 2:111–120, 2013.

[256] H. Huang, K. Chen, C. Ren, P. Liu, S. Zhu, and D. Wu. Towards Discovering and Understanding Unexpected Hazards in Tailoring Antivirus Software for Android. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, ASIACCS '15, pages 7–18, New York, NY, USA, 2015. ACM.

[257] H. Huang, S. Zhu, K. Chen, and P. Liu. From System Services Freezing to System Server Shutdown in Android: All You Need Is a Loop in an App. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, CCS'15, pages 1236–1247. ACM, 2015.

[258] H. Huang, S. Zhu, P. Liu, and D. Wu. A Framework for Evaluating Mobile App Repackaging Detection Algorithms. In *Trust and Trustworthy Computing - 6th International Conference, TRUST 2013, London, UK, June 17-19, 2013. Proceedings*, volume 7904 of *TRUST'13*, pages 169–186. Springer, 2013.

[259] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang. SUPOR: precise and scalable sensitive user input detection for android apps. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 977–992. USENIX Association, 2015.

[260] 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 *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 1036–1046, Hyderabad, India, 2014.

[261] P. Institute. Big data analytics in cyber defense. Feb. 2013.

[262] T. Isohara, K. Takemori, and A. Kubota. Kernel-based Behavior Analysis for Android Malware Detection. In *Seventh International Conference on Computational Intelligence and Security, CIS 2011, Sanya, Hainan, China, December 3-4, 2011*, CIS'11, pages 1011–1015. IEEE Computer Society, 2011.

[263] R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek. Energy-aware test-suite minimization for android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 425–436, 2016.

[264] R. Jabbarvand, A. Sadeghi, J. Garcia, S. Malek, and P. Ammann. Ecodroid: An approach for energy-based ranking of android apps. In *4th IEEE/ACM International Workshop on Green and Sustainable Software, GREENS 2015, Florence, Italy, May 18, 2015*, pages 8–14, 2015.

[265] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[266] J. Jang, H. Ji, J. Hong, J. Jung, D. Kim, and S. K. Jung. Protecting Android applications with steganography-based software watermarking. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, SAC'13, pages 1657–1658. ACM, 2013.

[267] C. Jeon, W. Kim, B. Kim, and Y. Cho. Enhancing security enforcement on unmodified Android. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, SAC'13, pages 1655–1656. ACM, 2013.

[268] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. android and mr. hide: Fine-grained permissions in android applications. In *SPSM'12, Proceedings of the Workshop on Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2012, October 19, 2012, Raleigh, NC, USA*, SPSM '12, pages 3–14, New York, NY, USA, 2012. ACM.

[269] J. Jeong, D. Seo, C. Lee, J. Kwon, H. Lee, and J. Milburn. MysteryChecker: Unpredictable attestation to detect repackaged malicious applications in Android. In *9th International Conference on Malicious and Unwanted Software: The Americas MALWARE 2014, Fajardo, PR, USA, October 28-30, 2014*, MALWARE'14, pages 50–57. IEEE, 2014.

[270] Y. Jeong, H.-t. Lee, S. Cho, S. Han, and M. Park. A kernel-based monitoring approach for analyzing malicious behavior on Android. In *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, SAC'14, pages 1737–1738. ACM, 2014.

[271] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake. Run-Time Enforcement of Information-Flow Properties on Android. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, volume 8134 of *ESORICS'13*, pages 775–792. Springer, 2013.

[272] H. Jiao, X. Li, L. Zhang, G. Xu, and Z. Feng. Hybrid Detection Using Permission Analysis for Android Malware. In *International Conference on Security and Privacy in Communication Networks - 10th International ICST Conference, SecureComm 2014, Beijing, China, September 24-26, 2014, Revised Selected Papers, Part I*, volume 152 of *SecureComm'14*, pages 541–545. Springer, 2014.

[273] S. Jiao, Y. Cheng, L. Ying, P. Su, and D. Feng. A Rapid and Scalable Method for Android Application Repackaging Detection. In *Information Security Practice and Experience - 11th International Conference, ISPEC 2015, Beijing, China, May 5-8, 2015. Proceedings*, volume 9065 of *ISPEC'15*, pages 349–364. Springer, 2015.

[274] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri. Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, CCS'14, pages 66–77. ACM, 2014.

[275] Y. Jing, G.-J. Ahn, Z. Zhao, and H. Hu. Riskmon: Continuous and automated risk assessment of mobile applications. In *Fourth ACM Conference on Data and Application Security and Privacy, CODASPY'14, San Antonio, TX, USA - March 03 - 05, 2014*, pages 99–110. ACM, 2014.

[276] Y. Jing, G.-J. Ahn, Z. Zhao, and H. Hu. Towards Automated Risk Assessment and Mitigation of Mobile Applications. *IEEE Trans. Dependable Sec. Comput.*, 12(5):571–584, 2015.

[277] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu. Morpheus: automatically generating heuristics to detect Android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, ACSAC'14, pages 216–225. ACM, 2014.

[278] M. M. John, P. Vinod, and K. A. Dhanya. Hartley's test ranked opcodes for Android malware analysis. In *Proceedings of the 8th International Conference on Security of Information and Networks, SIN 2015, Sochi, Russian Federation, September 8-10, 2015*, ICSIN'15, pages 304–311. ACM, 2015.

[279] R. Johnson, M. Elsabagh, A. Stavrou, and V. Sritapan. Targeted DoS on android: how to disable android in 10 seconds or less. In *10th International Conference on Malicious and Unwanted Software, MALWARE 2015, Fajardo, PR, USA, October 20-22, 2015*, MALWARE'15, pages 136–143. IEEE, 2015.

[280] R. Johnson, Z. Wang, C. Gagnon, and A. Stavrou. Analysis of Android Applications' Permissions. In *Sixth International Conference on Software Security and Reliability,*

SERE 2012, Gaithersburg, Maryland, USA, 20-22 June 2012 - Companion Volume, SERE'12, pages 45–46. IEEE, 2012.

[281] C. Jung, D. Feth, and C. Seise. Context-Aware Policy Enforcement for Android. In *IEEE 7th International Conference on Software Security and Reliability, SERE 2013, Gaithersburg, MD, USA, June 18-20, 2013*, SERE'13, pages 40–49. IEEE, 2013.

[282] A. F. A. Kadir, N. Stakhanova, and A. A. Ghorbani. Android Botnets: What URLs are Telling Us. In *Network and System Security - 9th International Conference, NSS 2015, New York, NY, USA, November 3-5, 2015, Proceedings*, volume 9408 of *NSS'15*, pages 78–91. Springer, 2015.

[283] D. Kantola, E. Chin, W. He, and D. Wagner. Reducing attack surfaces for intra-application communication in android. In *SPSM'12, Proceedings of the Workshop on Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2012, October 19, 2012, Raleigh, NC, USA*, CCS'12, pages 69–80. ACM, 2012.

[284] M. Karami, M. Elsabagh, P. Najafiborazjani, and A. Stavrou. Behavioral Analysis of Android Applications Using Automated Instrumentation. In *Seventh International Conference on Software Security and Reliability, SERE 2012, Gaithersburg, Maryland, USA, 18-20 June 2013 - Companion Volume*, SERE'13, pages 182–187. IEEE, 2013.

[285] P. M. Kate and S. V. Dhavale. Two Phase Static Analysis Technique for Android Malware Detection. In *Proceedings of the Third International Symposium on Women in Computing and Informatics, WCI 2015, co-located with ICACCI 2015, Kochi, India, August 10-13, 2015*, WCI'15, pages 650–655. ACM, 2015.

[286] M. Kato and S. Matsuura. A Dynamic Countermeasure Method to Android Malware by User Approval. In *37th Annual IEEE Computer Software and Applications Conference, COMPSAC 2013, Kyoto, Japan, July 22-26, 2013*, COMPSAC'13, pages 730–731. IEEE Computer Society, 2013.

[287] P. G. Kelley, S. Consolvo, L. F. Cranor, J. Jung, N. M. Sadeh, and D. Wetherall. A Conundrum of Permissions: Installing Applications on an Android Smartphone. In *Financial Cryptography and Data Security - FC 2012 Workshops, USEC and WECSR 2012, Kralendijk, Bonaire, March 2, 2012, Revised Selected Papers*, volume 7398 of *FC'12*, pages 68–79. Springer, 2012.

[288] P. G. Kelley, L. F. Cranor, and N. M. Sadeh. Privacy as part of the app decision-making process. In *2013 ACM SIGCHI Conference on Human Factors in Computing Systems, CHI '13, Paris, France, April 27 - May 2, 2013*, pages 3393–3402. ACM, 2013.

[289] R. S. Khune and J. Thangakumar. A cloud-based intrusion detection system for android smartphones. In *Radar, Communication and Computing (ICRCC), 2012 International Conference on*, pages 180–184. IEEE, 2012.

[290] D.-u. Kim, J. Kim, and S. Kim. A malicious application detection framework using automatic feature extraction tool on android market. In *Proceedings: 3rd International Conference on Computer Science and Information Technology (ICCSIT 2013)*, ICCSIT'13, 2013.

[291] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center. ScanDal: Static analyzer for detecting privacy leaks in android applications. In *IEEE Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy, MoST 2012, San Francisco, California, USA, May 24, 2012*, MoST'12, San Francisco, CA, 2012.

[292] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[293] B. Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33:2004, 2004.

[294] 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, SOAP 2014, Edinburgh, UK, Co-located with PLDI 2014, June 12, 2014*, pages 1–6, Edinburgh, UK, 2014. ACM.

[295] X. Kou and Q. Wen. Intrusion detection model based on android. In *2011 4th IEEE International Conference on Broadband Network and Multimedia Technology*, 2011.

[296] S. M. Kywe, C. Landis, Y. Pei, J. Satterfield, Y. Tian, and P. Tague. PrivateDroid: Private Browsing Mode for Android. In *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2014, Beijing, China, September 24-26, 2014*, pages 27–36. IEEE Computer Society, 2014.

[297] M. Kühnel, M. Smieschek, and U. Meyer. Fast Identification of Obfuscation and Mobile Advertising in Mobile Malware. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*, TrustCom'15, pages 214–221. IEEE, 2015.

[298] J.-F. Lalande and S. Wendzel. Hiding Privacy Leaks in Android Applications Using Low-Attention Raising Covert Channels. In *2013 International Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013*, ARES'13, pages 701–710. IEEE Computer Society, 2013.

[299] L. Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[300] B. W. Lampson. Protection. *Operating Systems Review*, 8(1):18–24, 1974.

[301] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4android: a generic operating system framework for secure smartphones. In *SPSM'11, Proceedings of the 1st ACM Workshop Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2011, October 17, 2011, Chicago, IL, USA*, SPSM'11, pages 39–50. ACM, 2011.

[302] S.-H. Lee and S.-H. Jin. Warning system for detecting malicious applications on android system. *International Journal of Computer and Communication Engineering*, 2(3):324, 2013.

[303] Y. K. Lee, J. Y. Bang, G. Safi, A. Shahbazian, Y. Zhao, and N. Medvidovic. A *SEALANT* for inter-app security holes in android. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 312–323, 2017.

[304] L. Lei, Y. Wang, J. Jing, Z. Zhang, and X. Yu. MeadDroid: Detecting Monetary Theft Attacks in Android by DVM Monitoring. In *Information Security and Cryptology - ICISC 2012 - 15th International Conference, Seoul, Korea, November 28-30, 2012, Revised Selected Papers*, volume 7839 of *ICISC'12*, pages 78–91. Springer, 2012.

[305] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo. Don't kill my ads!: balancing privacy in an ad-supported mobile application market. In *2012 Workshop on Mobile Computing Systems and Applications, HotMobile'12, San Diego, CA, USA, February 28-29, 2012*, page 2. ACM, 2012.

[306] D. Li, Y. Lyu, M. Wan, and W. G. J. Halfond. String analysis for Java and Android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, FSE'15, pages 661–672. ACM, 2015.

[307] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, and Y. L. Traon. ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis. In *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings*, volume 455 of *ICT SEC'15*, pages 513–527. Springer, 2015.

[308] L. Li, A. Bartel, T. F. Bissyandé, 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 *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, ICSE'15, pages 280–291. IEEE, 2015.

[309] L. Li, A. Bartel, J. Klein, and Y. L. Traon. Automatically exploiting potential component leaks in android applications. In *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2014, Beijing, China, September 24-26, 2014*, pages 388–397, Beijing, China, 2014.

[310] 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.

[311] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and Y. L. Traon. Static analysis of android apps: A systematic literature review. *Information & Software Technology*, 88:67–95, 2017.

[312] Q. Li and X. Li. Android Malware Detection Based on Static Analysis of Characteristic Tree. In *2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, CyberC 2015, Xi'an, China, September 17-19, 2015*, CyberC'15, pages 84–91. IEEE Computer Society, 2015.

[313] S. Li, J. Chen, T. Spyridopoulos, P. Andriotis, R. Ludwiniak, and G. Russell. Real-Time Monitoring of Privacy Abuses and Intrusion Detection in Android System. In *Human Aspects of Information Security, Privacy, and Trust - Third International Conference, HAS 2015, Held as Part of HCI International 2015, Los Angeles, CA, USA, August 2-7, 2015. Proceedings*, volume 9190 of *HAS'15*, pages 379–390. Springer, 2015.

[314] T. Li, X.-y. Zhou, L. Xing, Y. Lee, M. Naveed, X. Wang, and X. Han. Mayhem in the Push Clouds: Understanding and Mitigating Security Hazards in Mobile Push-Messaging Services. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, SEC'15, pages 978–989. ACM, 2014.

[315] X. Li, G. Bai, B. Thian, Z. Liang, and H. Yin. A Light-Weight Software Environment for Confining Android Malware. In *IEEE Eighth International Conference on Software Security and Reliability, SERE 2014, San Francisco, CA, USA, June 30 - July 2, 2014 - Companion Volume*, SERE'14, pages 158–167. IEEE, 2014.

[316] Y. Li, T. Shen, X. Sun, X. Pan, and B. Mao. Detection, Classification and Characterization of Android Malware Using API Data Dependency. In *Security and Privacy in Communication Networks - 11th International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Revised Selected Papers*, volume 164 of *SecureComm'15*, pages 23–40. Springer, 2015.

[317] S. Liang, A. W. Keep, M. Might, S. Lyde, T. Gilray, P. Aldous, and D. V. Horn. Sound and precise malware analysis for android via pushdown reachability and entry-point saturation. In *SPSM'13, Proceedings of the 2013 ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2013, November 8, 2013, Berlin, Germany*, SPSM'13, pages 21–32. ACM, 2013.

[318] Y.-D. Lin, Y.-C. Lai, C.-H. Chen, and H.-C. Tsai. Identifying android malicious repackaged applications by thread-grained system call sequences. *Computers & Security*, 39:340–350, 2013.

[319] M. Lindorfer, M. Neugschwandtner, and C. Platzer. MARVIN: Efficient and Comprehensive Mobile App Classification Through Static and Dynamic Analysis. In *39th IEEE Annual Computer Software and Applications Conference, COMPSAC 2015, Taichung, Taiwan, July 1-5, 2015. Volume 2*, Taichung, Taiwan, 2015.

[320] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. ANDRUBIS-1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proceedings of the the 3rd International Workshop on Building*

Analysis Datasets and Gathering Experience Returns for Security (BADGERS), 2014, 2014.

[321] M. Lindorfer, S. Volanis, A. Sisto, M. Neugschwandtner, E. Athanasopoulos, F. Maggi, C. Platzer, S. Zanero, and S. Ioannidis. AndRadar: Fast Discovery of Android Applications in Alternative Markets. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 11th International Conference, DIMVA 2014, Egham, UK, July 10-11, 2014. Proceedings*, volume 8550 of *DIMVA'14*, pages 51–71. Springer, 2014.

[322] B. Liu, B. Liu, H. Jin, and R. Govindan. Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2015, Florence, Italy, May 19-22, 2015*, MobiSys'15, pages 89–103. ACM, 2015.

[323] B. Liu, S. Nath, R. Govindan, and J. Liu. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, NSDI'14, pages 57–70. USENIX Association, 2014.

[324] B. Livshits and J. Jung. Automatic Mediation of Privacy-Sensitive Resource Access in Smartphone Applications. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, SEC'13, pages 113–130. USENIX Association, 2013.

[325] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for java. In *Programming Languages and Systems*, pages 139–160. Springer, 2005.

[326] S. Lortz, H. Mantel, A. Starostin, T. Bähr, D. Schneider, and A. Weber. Cassandra: Towards a Certifying App Store for Android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, SPSM@CCS 2014, Scottsdale, AZ, USA, November 03 - 07, 2014*, SPSM'14, pages 93–104. ACM, 2014.

[327] K. Lu, Z. Li, V. P. Kemerlis, Z. Wu, L. Lu, C. Zheng, Z. Qian, W. Lee, and G. Jiang. Checking More and Alerting Less: Detecting Privacy Leakages via Enhanced Data-flow Analysis and Peer Voting. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*, NDSS'15. The Internet Society, 2015.

[328] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 229–240. ACM, 2012.

[329] Z. Lu and S. Mukhopadhyay. Model-Based Static Source Code Analysis of Java Programs with Applications to Android Security. In *36th Annual IEEE Computer Software and Applications Conference, COMPSAC 2012, Izmir, Turkey, July 16-20, 2012*, COMPSAC'12, pages 322–327. IEEE Computer Society, 2012.

[330] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on WebView in the Android system. In *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*, ACSAC'11, pages 343–352. ACM, 2011.

[331] W. Luo, S. Xu, and X. Jiang. Real-time detection and prevention of android sms permission abuses. In *Proceedings of the first international workshop on Security in embedded systems and smartphones, SESP 2013, Hangzhou, China, May 8, 2013*, pages 11–18. ACM, 2013.

[332] Z. Luoshi, N. Yan, W. Xiao, W. Zhaoguo, and X. Yibo. A3: automatic analysis of android malware. In *1st International Workshop on Cloud Computing and Information Security, 2013*, CCIS'13. Atlantis Press, 2013.

[333] B. Ma. How We Found These Vulnerabilities in Android Applications. In *International Conference on Security and Privacy in Communication Networks - 10th International ICST Conference, SecureComm 2014, Beijing, China, September 24-26, 2014, Revised Selected Papers, Part II*, volume 153 of *SecureComm'14*, pages 399–406. Springer, 2014.

[334] K. Ma, M. Liu, S. Guo, and T. Ban. MonkeyDroid: Detecting Unreasonable Privacy Leakages of Android Applications. In *Neural Information Processing - 22nd International Conference, ICONIP 2015, Istanbul, Turkey, November 9-12, 2015, Proceedings, Part III*, volume 9491 of *ICONIP'15*, pages 384–391. Springer, 2015.

[335] S. Ma, Z. Tang, Q. Xiao, J. Liu, T. T. Duong, X. Lin, and H. Zhu. Detecting GPS information leakage in Android applications. In *2013 IEEE Global Communications Conference, GLOBECOM 2013, Atlanta, GA, USA, December 9-13, 2013*, GLOBECOM'13, pages 826–831. IEEE, 2013.

[336] S. Ma, S. Wang, D. Lo, R. H. Deng, and C. Sun. Active Semi-supervised Approach for Checking App Behavior against Its Description. In *39th IEEE Annual Computer Software and Applications Conference, COMPSAC 2015, Taichung, Taiwan, July 1-5, 2015. Volume 2*, COMPSAC'15, pages 179–184. IEEE, 2015.

[337] F. Maggi, A. Valdi, and S. Zanero. AndroTotal: a flexible, scalable toolbox and service for testing mobile malware detectors. In *SPSM'13, Proceedings of the 2013 ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2013, November 8, 2013, Berlin, Germany*, pages 49–54. ACM, 2013.

[338] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou. A whitebox approach for automated security testing of Android applications on the cloud. In *7th International Workshop on Automation of Software Test, AST 2012, Zurich, Switzerland, June 2-3, 2012*, AST'12, pages 22–28. IEEE, 2012.

[339] D. Maier, T. Müller, and M. Protsenko. Divide-and-Conquer: Why Android Malware Cannot Be Stopped. In *Ninth International Conference on Availability, Reliability and Security, ARES 2014, Fribourg, Switzerland, September 8-12, 2014*, ARES'14, pages 30–39. IEEE Computer Society, 2014.

[340] S. Malek, N. Esfahani, T. Kacem, R. Mahmood, N. Mirzaei, and A. Stavrou. A Framework for Automated Security Testing of Android Applications on the Cloud. In *Sixth International Conference on Software Security and Reliability, SERE 2012, Gaithersburg, Maryland, USA, 20-22 June 2012 - Companion Volume*, SERE'12, pages 35–36. IEEE, 2012.

[341] C. Mann and A. Starostin. A framework for static detection of privacy leaks in android applications. In *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*, SAC '12, pages 1457–1462, Riva del Garda, Italy, 2012. ACM.

[342] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun. Analysis of the communication between colluding applications on modern smartphones. In *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*, ACSAC '12, pages 51–60, Orlando, Florida, 2012. ACM.

[343] T. Martin, M. Hsiao, D. S. Ha, and J. Krishnaswami. Denial-of-service attacks on battery-powered mobile computers. In *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom 2004), 14-17 March 2004, Orlando, FL, USA*, pages 309–318. IEEE, 2004.

[344] M. Z. Mas'ud, S. Sahib, M. F. Abdollah, S. R. Selamat, R. Yusof, and R. Ahmad. Profiling mobile malware behaviour through hybrid malware analysis approach. In *9th International Conference on Information Assurance and Security, IAS 2013, Gammarth, Tunisia, December 4-6, 2013*, AIS'13, pages 78–84. IEEE, 2013.

[345] S. Matsumoto and K. Sakurai. A proposal for the privacy leakage verification tool for Android application developers. In *The 7th International Conference on Ubiquitous Information Management and Communication, ICUIMC '13, Kota Kinabalu, Malaysia - January 17 - 19, 2013*, ICUIMC'13, page 54. ACM, 2013.

[346] McAfee, Intel Security. Mobile threat report: What's on the horizon for 2016. `https://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2016.pdf`, 2016.

[347] C. Miller and C. Mulliner. Fuzzing the phone in your phone. In *Black Hat, 2009*, 2009.

[348] J. Milosevic, A. Dittrich, A. Ferrante, and M. Malek. A Resource-Optimized Approach to Efficient Early Detection of Mobile Malware. In *Ninth International Conference on Availability, Reliability and Security, ARES 2014, Fribourg, Switzerland, September 8-12, 2014*, ARES'14, pages 333–340. IEEE Computer Society, 2014.

[349] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek. Reducing combinatorics in GUI testing of android applications. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 559–570, 2016.

[350] M. Mitchell, G. Tian, and Z. Wang. Systematic audit of third-party android phones. In *Fourth ACM Conference on Data and Application Security and Privacy, CODASPY'14, San Antonio, TX, USA - March 03 - 05, 2014*, CODASPY'14, pages 175–186. ACM, 2014.

[351] V. Moonsamy, J. Rong, and S. Liu. Mining permission patterns for contrasting clean and malicious android applications. *Future Generation Comp. Syst.*, 36:122–132, 2014.

[352] V. Moonsamy, J. Rong, S. Liu, G. Li, and L. M. Batten. Contrasting Permission Patterns between Clean and Malicious Android Applications. In *Security and Privacy in Communication Networks - 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers*, volume 127 of *SecureComm'13*, pages 69–85. Springer, 2013.

[353] C. Mulliner, J. Oberheide, W. K. Robertson, and E. Kirda. PatchDroid: scalable third-party security patches for Android devices. In *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, ACSAC'13, pages 259–268. ACM, 2013.

[354] C. Mulliner, W. K. Robertson, and E. Kirda. VirtualSwindle: an automated attack against in-app billing on android. In *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, ASIA'14, pages 459–470. ACM, 2014.

[355] P. Mutchler, A. Doupé, J. Mitchell, C. Kruegel, and G. Vigna. A large-scale study of mobile web app security. In *IEEE Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy, MoST 2015, San Jose, California, USA, May 21, 2015*, 2015.

[356] S. Mutti, E. Bacis, and S. Paraboschi. SeSQLite: Security Enhanced SQLite: Mandatory Access Control for Android databases. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, ACSAC'15, pages 411–420. ACM, 2015.

[357] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna. BareDroid: Large-Scale Analysis of Android Apps on Real Devices. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, ACSAC'15, pages 71–80. ACM, 2015.

[358] Y. Nadji, J. T. Giffin, and P. Traynor. Automated remote repair for mobile malware. In *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*, ACSAC'11, pages 413–422. ACM, 2011.

[359] A. Nadkarni and W. Enck. Preventing accidental data disclosure in modern operating systems. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, CCS'13, pages 1029–1042. ACM, 2013.

[360] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang. UIPicker: User-Input Privacy Identification in Mobile Applications. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, SEC'15, pages 993–1008. USENIX Association, 2015.

[361] F. Nasim, B. Aslam, W. Ahmed, and T. Naeem. Uncovering Self Code Modification in Android. In *Codes, Cryptology, and Information Security - First International Conference, C2SI 2015, Rabat, Morocco, May 26-28, 2015, Proceedings - In Honor of Thierry Berger*, volume 9084 of *C2SI'15*, pages 297–313. Springer, 2015.

[362] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2010, Beijing, China, April 13-16, 2010*, pages 328–332. ACM, 2010.

[363] T. Nelson, S. Saghafi, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Aluminum: Principled scenario exploration through minimality. In *Proc. of ICSE*, pages 232–241, 2013.

[364] S. Neuner, V. van der Veen, M. Lindorfer, M. Huber, G. Merzdovnik, M. Mulazzani, and E. Weippl. Enter Sandbox: Android Sandbox Comparison. *arXiv:1410.7749 [cs]*, Oct. 2014.

[365] Y. Ng, H. Zhou, Z. Ji, H. Luo, and Y. Dong. Which Android App Store Can Be Trusted in China? In *IEEE 38th Annual Computer Software and Applications Conference, COMPSAC 2014, Vasteras, Sweden, July 21-25, 2014*, COMPSAC'14, pages 509–518. IEEE, 2014.

[366] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11:1–11:29, 2011.

[367] Y. Nishimoto, N. Kajiwara, S. Matsumoto, Y. Hori, and K. Sakurai. Detection of Android API Call Using Logging Mechanism within Android Framework. In *Security and Privacy in Communication Networks - 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers*, volume 127 of *SecureComm'13*, pages 393–404. Springer, 2013.

[368] D. Octeau, W. Enck, and P. McDaniel. The ded Decompiler. Technical Report NAS-TR-0140-2010, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, 2010.

[369] D. Octeau, S. Jha, and P. McDaniel. Retargeting android applications to java bytecode. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 6. ACM, 2012.

[370] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *37th*

IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1, ICSE 2015, Florence, Italy, 2015.

[371] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective Inter-Component Communication Mapping in Android: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, SEC'13, pages 543–558. USENIX Association, 2013.

[372] M. Ongtang, K. R. B. Butler, and P. D. McDaniel. Porscha: policy oriented secure content handling in Android. In *Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010*, ACSAC'10, pages 221–230. ACM, 2010.

[373] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. In *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009, Honolulu, Hawaii, 7-11 December 2009*, ACSAC '09, pages 340–349, Honolulu, Hawaii, 2009. IEEE Computer Society.

[374] L. Onwuzurike and E. D. Cristofaro. Danger is my middle name: experimenting with SSL vulnerabilities in Android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, New York, NY, USA, June 22-26, 2015*, WISEC'15, pages 15:1–15:6. ACM, 2015.

[375] X. Pan, Y. Zhongyang, Z. Xin, B. Mao, and H. Huang. Defensor: Lightweight and Efficient Security-Enhanced Framework for Android. In *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2014, Beijing, China, September 24-26, 2014*, TrustCom'14, pages 260–267. IEEE Computer Society, 2014.

[376] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. WHYPER: Towards automating risk assessment of mobile applications. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, SEC'13, pages 527–542, Washington, DC, 2013. USENIX Association.

[377] A. Paturi, M. Cherukuri, J. Donahue, and S. Mukkamala. Mobile malware visual analytics and similarities of Attack Toolkits (Malware gene analysis). In *2013 International Conference on Collaboration Technologies and Systems, CTS 2013, San Diego, CA, USA, May 20-24, 2013*, CTS'13, pages 149–154. IEEE, 2013.

[378] J. Paupore, E. Fernandes, A. Prakash, S. Roy, and X. Ou. Practical Always-on Taint Tracking on Mobile Devices. In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*, HotOS'15. USENIX Association, 2015.

[379] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege separation for applications and advertisers in android. In *7th ACM Symposium on Information,*

Compuer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012, ASIACCS '12, pages 71–72, Seoul, Republic of Korea, 2012. ACM.

[380] N. Peiravian and X. Zhu. Machine Learning for Android Malware Detection Using Permission and API Calls. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013*, ICTAI'13, pages 300–305. IEEE Computer Society, 2013.

[381] H. Peng, C. S. Gates, B. P. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of Android apps. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, CCS'12, pages 241–252. ACM, 2012.

[382] G. Petracca, Y. Sun, T. Jaeger, and A. Atamli. AuDroid: Preventing Attacks on Audio Channels in Mobile Devices. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, ACSAC'15, pages 181–190. ACM, 2015.

[383] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: hindering dynamic analysis of Android malware. In *Proceedings of the Seventh European Workshop on System Security, EuroSec 2014, April 13, 2014, Amsterdam, The Netherlands*, EUROSEC'14, pages 5:1–5:6. ACM, 2014.

[384] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, San Diego, California, 2014.

[385] I. Polakis, S. Volanis, E. Athanasopoulos, and E. P. Markatos. The man who was there: validating check-ins in location-based services. In *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, ACSAC'13, pages 19–28. ACM, 2013.

[386] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: versatile protection for smartphones. In *Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010*, ACSAC'10, pages 347–356. ACM, 2010.

[387] M. Protsenko, S. Kreuter, and T. Müller. Dynamic Self-Protection and Tamperproofing for Android Apps Using Native Code. In *10th International Conference on Availability, Reliability and Security, ARES 2015, Toulouse, France, August 24-27, 2015*, ARES'15, pages 129–138. IEEE, 2015.

[388] M. Protsenko and T. Müller. PANDORA applies non-deterministic obfuscation randomly to Android. In *8th International Conference on Malicious and Unwanted Software: "The Americas", MALWARE 2013, Fajardo, PR, USA, October 22-24, 2013*, MALWARE'13, pages 59–67. IEEE Computer Society, 2013.

[389] C. Qian, X. Luo, Y. Shao, and A. T. S. Chan. On Tracking Information Flows through JNI in Android Applications. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, DSN'14, pages 180–191. IEEE, 2014.

[390] J. Qiu, B. Yadegari, B. Johannesmeyer, S. Debray, and X. Su. A framework for understanding dynamic anti-analysis defenses. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop, PPREW@ACSAC 2014, New Orleans, LA, USA, December 9, 2014*, page 2. ACM, 2014.

[391] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen. AutoCog: Measuring the Description-to-permission Fidelity in Android Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, CCS'14, pages 1354–1365. ACM, 2014.

[392] D. Quan, L. Zhai, F. Yang, and P. Wang. Detection of Android Malicious Apps Based on the Sensitive Behaviors. In *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2014, Beijing, China, September 24-26, 2014*, TrustCom'14, pages 877–883. IEEE Computer Society, 2014.

[393] B. Rashidi and C. Fung. A Survey of Android Security Threats and Defenses. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 6(3):3–35, 2015.

[394] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[395] S. Rasthofer, S. Arzt, M. Kolhagen, B. Pfretzschner, S. Huber, E. Bodden, and P. Richter. Droidsearch: A tool for scaling android app triage to real-world app stores. In *Science and Information Conference (SAI), 2015*, pages 247–256. IEEE, 2015.

[396] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden. DroidForce: Enforcing Complex, Data-centric, System-wide Policies in Android. In *Ninth International Conference on Availability, Reliability and Security, ARES 2014, Fribourg, Switzerland, September 8-12, 2014*, ARES'14, pages 40–49. IEEE Computer Society, 2014.

[397] S. Rasthofer, I. Asrar, S. Huber, and E. Bodden. How Current Android Malware Seeks to Evade Automated Code Analysis. In *Information Security Theory and Practice - 9th IFIP WG 11.2 International Conference, WISTP 2015 Heraklion, Crete, Greece, August 24-25, 2015 Proceedings*, volume 9311 of *WISTP'15*, pages 187–202. Springer, 2015.

[398] V. Rastogi, Y. Chen, and W. Enck. AppsPlayground: Automatic security analysis of smartphone applications. In *Third ACM Conference on Data and Application Security and Privacy, CODASPY'13, San Antonio, TX, USA, February 18-20, 2013*, CODASPY '13, pages 209–220, San Antonio, TX, 2013. ACM.

[399] V. Rastogi, Y. Chen, and X. Jiang. DroidChameleon: evaluating Android anti-malware against transformation attacks. In *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, CCS'13, pages 329–334. ACM, 2013.

[400] V. Rastogi, Y. Chen, and X. Jiang. Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks. *IEEE Trans. Information Forensics and Security*, 9(1):99–108, 2014.

[401] V. Rastogi, Z. Qu, J. McClurg, Y. Cao, and Y. Chen. Uranine: Real-time Privacy Leakage Monitoring without System Modification for Android. In *Security and Privacy in Communication Networks - 11th International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Revised Selected Papers*, volume 164 of *SecureComm'15*, pages 256–276. Springer, 2015.

[402] 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, PPREW@ACSAC 2014, New Orleans, LA, USA, December 9, 2014*, PPREW-4, pages 4:1–4:10, New Orleans, LA, 2014. ACM.

[403] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *ACM European Workshop on Systems Security (EuroSec)*, Prague, Czech Republic, 2013.

[404] C. Ren, K. Chen, and P. Liu. Droidmarking: resilient software watermarking for impeding android application repackaging. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, ASE'14, pages 635–646. ACM, 2014.

[405] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards Discovering and Understanding Task Hijacking in Android. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, SEC'15, pages 945–959. USENIX Association, 2015.

[406] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 49–61. ACM, 1995.

[407] S. S. Response. 2015 internet security threat report. http://www.symantec.com, 2015.

[408] F. Roesner and T. Kohno. Securing Embedded User Interfaces: Android and Beyond. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, SEC'13, pages 97–112. USENIX Association, 2013.

[409] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In

IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA, S&P'12, pages 224–238. IEEE Computer Society, 2012.

[410] F. Rohrer, Y. Zhang, L. Chitkushev, and T. Zlateva. DR BACA: dynamic role based access control for Android. In *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, ACSAC'13, pages 299–308. ACM, 2013.

[411] S. Rosen, Z. Qian, and Z. M. Mao. AppProfiler: a flexible method of exposing privacy-related behavior in android applications to end users. In *Third ACM Conference on Data and Application Security and Privacy, CODASPY'13, San Antonio, TX, USA, February 18-20, 2013*, CODASPY'13, pages 221–232. ACM, 2013.

[412] S. T. A. Rumee and D. Liu. DroidTest: Testing Android Applications for Leakage of Private Information. In *Information Security, 16th International Conference, ISC 2013, Dallas, Texas, USA, November 13-15, 2013, Proceedings*, volume 7807 of *ICS'13*, pages 341–353. Springer, 2013.

[413] G. Russello, B. Crispo, E. Fernandes, and Y. Zhauniarovich. Yaase: Yet another android security extension. In *PASSAT/SocialCom 2011, Privacy, Security, Risk and Trust (PASSAT), 2011 IEEE Third International Conference on and 2011 IEEE Third International Conference on Social Computing (SocialCom), Boston, MA, USA, 9-11 Oct., 2011*, PASSAT'11, pages 1033–1040. IEEE, 2011.

[414] G. Russello, A. B. Jimenez, H. Naderi, and W. v. d. Mark. FireDroid: hardening security in almost-stock Android. In *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, ACSAC'13, pages 319–328. ACM, 2013.

[415] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering (TSE)*, 43(6):492–530, 2017.

[416] A. Sadeghi, H. Bagheri, and S. Malek. Analysis of android inter-app security vulnerabilities using COVERT. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, pages 725–728, 2015.

[417] A. Sadeghi, H. Bagheri, and S. Malek. Analysis of Android Inter-App Security Vulnerabilities Using COVERT. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, ICSE'15, pages 725–728. IEEE, 2015.

[418] A. Sadeghi, N. Esfahani, and S. Malek. Mining the categorized software repositories to improve the analysis of security vulnerabilities. In *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 155–169, 2014.

[419] A. Sadeghi, N. Esfahani, and S. Malek. Mining the Categorized Software Repositories to Improve the Analysis of Security Vulnerabilities. In *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8411 of *Lecture Notes in Computer Science*, pages 155–169. Springer, 2014.

[420] A. Sadeghi, N. Esfahani, and S. Malek. Ensuring the consistency of adaptation through inter- and intra-component dependency analysis. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 26(1):2:1–2:27, 2017.

[421] A. Sadeghi, N. Esfahani, and S. Malek. Mining mobile app markets for prioritization of security assessment effort. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on App Market Analytics, WAMA@ESEC/SIGSOFT FSE 2017, Paderborn, Germany, September 5, 2017*, pages 1–7, 2017.

[422] A. Sadeghi, R. Jabbarvand, N. Ghorbani, H. Bagheri, and S. Malek. A temporal permission analysis and enforcement framework for android. 2017.

[423] A. Sadeghi, R. Jabbarvand, and S. Malek. Patdroid: permission-aware GUI testing of android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 220–232, 2017.

[424] J. Sahs and L. Khan. A Machine Learning Approach to Android Malware Detection. In *2012 European Intelligence and Security Informatics Conference, EISIC 2012, Odense, Denmark, August 22-24, 2012*, EISIC'12, pages 141–147. IEEE Computer Society, 2012.

[425] 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)*, 4(2):55–69, 2014.

[426] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[427] S. Salva and S. R. Zafimiharisoa. Data vulnerability detection by security testing for Android applications. In *2013 Information Security for South Africa, Johannesburg, South Africa, August 14-16, 2013*, ISSA'13, pages 1–8. IEEE, 2013.

[428] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[429] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. A. Maranon. PUMA: Permission Usage to Detect Malware in Android. In *International Joint Conference CISIS'12-ICEUTE'12-SOCO'12 Special Sessions, Ostrava, Czech Republic, September 5th-7th, 2012*, volume 189 of *CISIS/ICEUTE/SOCO'12*, pages 289–298. Springer, 2012.

[430] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, J. Nieves, P. G. Bringas, and G. A. Maranon. Mama: manifest Analysis for Malware Detection in Android. *Cybernetics and Systems*, 44(6-7):469–488, 2013.

[431] B. Sanz, I. Santos, X. Ugarte-Pedrero, C. Laorden, J. Nieves, and P. G. Bringas. Anomaly Detection Using String Analysis for Android Malware Detection. In *International Joint Conference SOCO'13-CISIS'13-ICEUTE'13 - Salamanca, Spain, September 11th-13th, 2013 Proceedings*, volume 239 of *SOCO'13*, pages 469–478. Springer, 2013.

[432] B. Sanz, I. Santos, X. Ugarte-Pedrero, C. Laorden, J. Nieves, and P. G. Bringas. Instance-based Anomaly Method for Android Malware Detection. In *SECRYPT 2013 - Proceedings of the 10th International Conference on Security and Cryptography, Reykjavík, Iceland, 29-31 July, 2013*, SECRYPT'13, pages 387–394. SciTePress, 2013.

[433] B. P. Sarma, N. Li, C. S. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Android permissions: a perspective combining risks and benefits. In *17th ACM Symposium on Access Control Models and Technologies, SACMAT '12, Newark, NJ, USA - June 20 - 22, 2012*, SACMAT'12, pages 13–22. ACM, 2012.

[434] 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 - Proceedings of the 10th International Conference on Security and Cryptography, Reykjavík, Iceland, 29-31 July, 2013*, pages 461–468, 2013.

[435] L. Sayfullina, E. Eirola, D. Komashinsky, P. Palumbo, Y. Miché, A. Lendasse, and J. Karhunen. Efficient Detection of Zero-day Android Malware Using Normalized Bernoulli Naive Bayes. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*, TrustCom'15, pages 198–205. IEEE, 2015.

[436] 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*, 57(6):10:1–10:12, Nov. 2013.

[437] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011.

[438] S. Schmeelk, J. Yang, and A. V. Aho. Android Malware Static Analysis Techniques. In *Proceedings of the 10th Annual Cyber and Information Security Research Conference, CISR '15, Oak Ridge, TN, USA, April 7-9, 2015*, pages 5:1–5:8. ACM, 2015.

[439] B. R. Schmerl, J. Gennari, A. Sadeghi, H. Bagheri, S. Malek, J. Cámara, and D. Garlan. Architecture modeling and analysis of security in android systems. In *Software Architecture - 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28 - December 2, 2016, Proceedings*, pages 274–290, 2016.

[440] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. H. Clausen, O. Kiraz, K. A. Yüksel, S. A. Çamtepe, and S. Albayrak. Static Analysis of Executables for Collaborative Malware Detection on Android. In *Proceedings of IEEE International Conference on Communications, ICC 2009, Dresden, Germany, 14-18 June 2009*, ICC'09, pages 1–5. IEEE, 2009.

[441] A.-D. Schmidt, J. H. Clausen, S. A. Çamtepe, and S. Albayrak. Detecting Symbian OS malware through static function call analysis. In *4th International Conference on Malicious and Unwanted Software, MALWARE 2009, Montréal, Quebec, Canada, October 13-14, 2009*, pages 15–22. IEEE, 2009.

[442] D. Schreckling, J. Posegga, and D. Hausknecht. Constroid: data-centric access control for android. In *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*, SAC'12, pages 1478–1485. ACM, 2012.

[443] D. Schreckling, J. Posegga, J. Köstler, and M. Schaff. Kynoid: Real-Time Enforcement of Fine-Grained, User-Defined, and Data-Centric Security Policies for Android. In *Information Security Theory and Practice. Security, Privacy and Trust in Computing Systems and Ambient Intelligent Ecosystems - 6th IFIP WG 11.2 International Workshop, WISTP 2012, Egham, UK, June 20-22, 2012. Proceedings*, volume 7322 of *WISTP'13*, pages 208–223. Springer, 2012.

[444] S. Schrittwieser, P. Frühwirt, P. Kieseberg, M. Leithner, M. Mulazzani, M. Huber, and E. R. Weippl. Guess Who's Texting You? Evaluating the Security of Smartphone Messaging Applications. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, NDSS'12, 2012.

[445] J. Schütte, R. Fedler, and D. Titze. ConDroid: Targeted Dynamic Analysis of Android Applications. In *29th IEEE International Conference on Advanced Information Networking and Applications, AINA 2015, Gwangju, South Korea, March 24-27, 2015*, AINA'15, pages 571–578. IEEE Computer Society, 2015.

[446] J. Schütte, D. Titze, and J. M. D. Fuentes. AppCaulk: Data Leak Prevention by Injecting Targeted Taint Tracking into Android Apps. In *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2014, Beijing, China, September 24-26, 2014*, TrustCom'14, pages 370–379. IEEE Computer Society, 2014.

[447] S. Seneviratne, H. Kolamunna, and A. Seneviratne. A measurement study of tracking in paid mobile applications. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, New York, NY, USA, June 22-26, 2015*, WISEC'15, pages 7:1–7:6. ACM, 2015.

[448] S.-H. Seo, A. Gupta, A. M. Sallam, E. Bertino, and K. Yim. Detecting mobile malware threats to homeland security through static analysis. *J. Network and Computer Applications*, 38:43–53, 2014.

[449] A. Shabtai, Y. Fledel, and Y. Elovici. Automated Static Code Analysis for Classifying Android Applications Using Machine Learning. In *2010 International Conference on Computational Intelligence and Security, CIS 2010, Nanning, Guangxi Zhuang Autonomous Region, China, December 11-14, 2010*, CIS'10, pages 329–333. IEEE Computer Society, 2010.

[450] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, and S. Dolev. Google Android: A State-of-the-Art Review of Security Mechanisms. *arXiv:0912.5101 [cs]*, Dec. 2009.

[451] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google android: A comprehensive security assessment. *IEEE security and Privacy*, 8(2):35–44, 2010.

[452] 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*, 38(1):161–190, 2012.

[453] H. Shahriar and H. M. Haddad. Content Provider Leakage Vulnerability Detection in Android Applications. In *Proceedings of the 7th International Conference on Security of Information and Networks, Glasgow, Scotland, UK, September 9-11, 2014*, SIN'14, page 359. ACM, 2014.

[454] A. S. Shamili, C. Bauckhage, and T. Alpcan. Malware Detection on Mobile Devices Using Distributed Machine Learning. In *20th International Conference on Pattern Recognition, ICPR 2010, Istanbul, Turkey, 23-26 August 2010*, pages 4348–4351. IEEE Computer Society, 2010.

[455] S. Shao, G. Dong, T. Guo, T. Yang, and C. Shi. Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications. In *IEEE 12th International Conference on Dependable, Autonomic and Secure Computing, DASC 2014, Dalian, China, August 24-27, 2014*, DASC'14, pages 75–80. IEEE Computer Society, 2014.

[456] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang. Towards a scalable resource-driven approach for detecting repackaged Android applications. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, ACSAC'15, pages 56–65. ACM, 2014.

[457] B. Shebaro, O. Oluwatimi, and E. Bertino. Context-Based Access Control Systems for Mobile Devices. *IEEE Trans. Dependable Sec. Comput.*, 12(2):150–163, 2015.

[458] B. Shebaro, O. Oluwatimi, D. Midi, and E. Bertino. IdentiDroid: Android can finally Wear its Anonymous Suit. *Transactions on Data Privacy*, 7(1):27–50, 2014.

[459] S. Shekhar, M. Dietz, and D. S. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, SEC'12, pages 553–567. USENIX Association, 2012.

[460] 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 *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 515–526. ACM, 2014.

[461] T. Shen, Y. Zhongyang, Z. Xin, B. Mao, and H. Huang. Detect Android Malware Variants Using Component Based Topology Graph. In *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2014, Beijing, China, September 24-26, 2014*, TrustCom'14, pages 406–413. IEEE Computer Society, 2014.

[462] W. Shin, S. Kwak, S. Kiyomoto, K. Fukushima, and T. Tanaka. A Small But Non-negligible Flaw in the Android Permission Scheme. In *POLICY 2010, IEEE International Symposium on Policies for Distributed Systems and Networks, Fairfax, VA, USA, 21-23 July 2010*, pages 107–110. IEEE Computer Society, 2010.

[463] W. Shin, S. Kwak, S. Kiyomoto, K. Fukushima, and T. Tanaka. A small but non-negligible flaw in the android permission scheme. In *POLICY 2010, IEEE International Symposium on Policies for Distributed Systems and Networks, Fairfax, VA, USA, 21-23 July 2010*, pages 107–110, 2010.

[464] I. Shklovski, S. D. Mainwaring, H. H. Skúladóttir, and H. Borgthorsson. Leakiness and creepiness in app space: perceptions of privacy and mobile app use. In *CHI Conference on Human Factors in Computing Systems, CHI'14, Toronto, ON, Canada - April 26 - May 01, 2014*, CHI'14, pages 2347–2356. ACM, 2014.

[465] A. Short and F. Li. Android Smartphone Third Party Advertising Library Data Leak Analysis. In *11th IEEE International Conference on Mobile Ad Hoc and Sensor Systems, MASS 2014, Philadelphia, PA, USA, October 28-30, 2014*, MASS'14, pages 749–754. IEEE Computer Society, 2014.

[466] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, NDSS'13. The Internet Society, 2013.

[467] E. Smith and A. Coglio. Android Platform Modeling and Android App Verification in the ACL2 Theorem Prover. In *Verified Software: Theories, Tools, and Experiments - 7th International Conference, VSTTE 2015, San Francisco, CA, USA, July 18-19, 2015. Revised Selected Papers*, volume 9593 of *VSTTE'15*, pages 183–201. Springer, 2015.

[468] F. Song and T. Touili. Model-Checking for Android Malware Detection. In *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, volume 8858 of *APLAS'14*, pages 216–235. Springer, 2014.

[469] Y. Song and U. Hengartner. PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM 2015, Denver, Colorado, USA, October 12, 2015*, pages 15–26. ACM, 2015.

[470] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan. SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, NDSS'14, San Diego, CA, 2014.

[471] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, pages 1808–1815. ACM, 2013.

[472] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in android ad libraries. In *Workshop on Mobile Security Technologies (MoST)*. Citeseer, 2012.

[473] X. Su, M. Chuah, and G. Tan. Smartphone Dual Defense Protection Framework: Detecting Malicious Applications in Android Markets. In *8th International Conference on Mobile Ad-hoc and Sensor Networks, MSN 2012, Chengdu, China, December 14-16, 2012*, MSN'12, pages 153–160. IEEE Computer Society, 2012.

[474] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. B. Alís. Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. *Expert Syst. Appl.*, 41(4):1104–1117, 2014.

[475] 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*, 16(2):961–987, 2014.

[476] Sufatrio, D. J. J. Tan, T.-W. Chua, and V. L. L. Thing. Securing Android: A Survey, Taxonomy, and Challenges. *ACM Comput. Surv.*, 47(4):58, 2015.

[477] M. Sun and G. Tan. NativeGuard: protecting android applications from third-party native libraries. In *7th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WiSec'14, Oxford, United Kingdom, July 23-25, 2014*, WISEC'14, pages 165–176. ACM, 2014.

[478] M. Sun, M. Zheng, J. C. S. Lui, and X. Jiang. Design and implementation of an Android host-based intrusion prevention system. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, ACSAC'15, pages 226–235. ACM, 2014.

[479] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, and L. Xie. Detecting Code Reuse in Android Applications Using Component-Based Control Flow Graph. In *ICT Systems Security*

and Privacy Protection - 29th IFIP TC 11 International Conference, SEC 2014, Marrakech, Morocco, June 2-4, 2014. Proceedings, volume 428 of *IFIP'14*, pages 142–155. Springer, 2014.

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

[481] Symantec. 2015 internet security threat report. Technical Report Vol. 20, Apr. 2015.

[482] Symantec. 2016 internet security threat report. Technical Report Vol. 21, Apr. 2016.

[483] Symantec. 2017 internet security threat report. Technical Report Vol. 22, Apr. 2017.

[484] Symantec Corp. 2012 norton study. `http://www.symantec.com/about/news/release/article.jsp?prid=20120905_02`, Sept. 2012.

[485] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*, San Diego, CA, 2015.

[486] F. Tchakounté and P. Dayang. System calls analysis of malwares on android. *International Journal of Science and Tecnology (IJST) Volume*, 2, 2013.

[487] P. Teufl, M. Ferk, A. Fitzek, D. Hein, S. Kraxberger, and C. Orthacker. Malware detection by applying knowledge discovery processes to application metadata on the Android Market (Google Play). *Security and Communication Networks*, 2013.

[488] D. Tian, X. Li, J. Hu, G. Xu, and Z. Feng. API-Level Multi-policy Access Control Enforcement for Android Middleware. In *Security and Privacy in Communication Networks - 11th International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Revised Selected Papers*, volume 164 of *SecureComm'15*, pages 559–562. Springer, 2015.

[489] D. Titze and J. Schütte. Apparecium: Revealing Data Flows in Android Applications. In *29th IEEE International Conference on Advanced Information Networking and Applications, AINA 2015, Gwangju, South Korea, March 24-27, 2015*, AINA'15, pages 579–586. IEEE Computer Society, 2015.

[490] D. Titze, P. Stephanow, and J. Schütte. App-ray: User-driven and fully automated android app security assessment. *Fraunhofer AISEC TechReport*, 2013.

[491] E. Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications*. PhD thesis, MIT, Feb. 2009.

[492] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *Fundamental Approaches to Software Engineering*, pages 210–225. Springer, 2013.

[493] O. Tripp and J. Rubin. A Bayesian Approach to Privacy Enforcement in Smartphones. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, SEC'14, pages 175–190. USENIX Association, 2014.

[494] Y. Tsutano, S. Bachala, W. Srisa-an, G. Rothermel, and J. Dinh. An efficient, robust, and scalable approach for analyzing interacting android apps. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 324–334, 2017.

[495] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, page 13. IBM Press, 1999.

[496] V. van der Veen, H. Bos, and C. Rossow. *Dynamic analysis of android malware*. PhD thesis, VU University Amsterdam, 2013.

[497] R. Vanciu and M. Abi-Antoun. Finding architectural flaws using constraints. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 334–344, Silicon Valley, CA, 2013. IEEE.

[498] D. Vecchiato, M. Vieira, and E. Martins. A security configuration assessment for android devices. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, SAC'15, pages 2299–2304. ACM, 2015.

[499] T. Vidas and N. Christin. Sweetening android lemon markets: measuring and combating malware in application marketplaces. In *Third ACM Conference on Data and Application Security and Privacy, CODASPY'13, San Antonio, TX, USA, February 18-20, 2013*, CODASPY'13, pages 197–208. ACM, 2013.

[500] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. In *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, pages 447–458. ACM, 2014.

[501] T. Vidas, N. Christin, and L. Cranor. Curbing android permission creep. In *2011 Web 2.0 Security and Privacy Workshop*, volume 2, Oakland, CA, 2011.

[502] T. Vidas, J. Tan, J. Nahata, C. L. Tan, N. Christin, and P. Tague. A5: Automated analysis of adversarial android applications. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, SPSM@CCS 2014, Scottsdale, AZ, USA, November 03 - 07, 2014*, SPSM'14, pages 39–50. ACM, 2014.

[503] L. Vigneri, J. Chandrashekar, I. Pefkianakis, and O. Heen. Taming the Android AppStore: Lightweight Characterization of Android Applications. *arXiv:1504.06093 [cs]*, Apr. 2015.

[504] D. Wang, H. Yao, Y. Li, H. Jin, D. Zou, and R. H. Deng. CICC: a fine-grained, semantic-aware, and transparent approach to preventing permission leaks for Android permission managers. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, New York, NY, USA, June 22-26, 2015*, WISEC'15, pages 6:1–6:6. ACM, 2015.

[505] H. Wang, Y. Guo, Z. Ma, and X. Chen. WuKong: a scalable and accurate two-phase approach to Android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, ISSTA'15, pages 71–82. ACM, 2015.

[506] H. Wang, Y. Zhang, J. Li, H. Liu, W. Yang, B. Li, and D. Gu. Vulnerability Assessment of OAuth Implementations in Android Applications. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, ACSAC'15, pages 61–70. ACM, 2015.

[507] N. Wang, B. Zhang, B. Liu, and H. Jin. Investigating Effects of Control and Ads Awareness on Android Users' Privacy Behaviors and Perceptions. In *Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services, MobileHCI 2015, Copenhagen, Denmark, August 24-27, 2015*, MobileHCI'15, pages 373–382. ACM, 2015.

[508] R. Wang, W. Enck, D. S. Reeves, X. Zhang, P. Ning, D. Xu, W. Zhou, and A. M. Azab. EASEAndroid: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-Scale Semi-Supervised Learning. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 351–366. USENIX Association, 2015.

[509] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: threats and mitigation. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, CCS'13, pages 635–646. ACM, 2013.

[510] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang. Exploring Permission-Induced Risk in Android Applications for Malicious Application Detection. *IEEE Trans. Information Forensics and Security*, 9(11):1869–1882, 2014.

[511] X. Wang, K. Sun, Y. Wang, and J. Jing. DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*, NDSS'15. The Internet Society, 2015.

[512] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du. Compac: enforce component-level access control in android. In *Fourth ACM Conference on Data and Application Security and Privacy, CODASPY'14, San Antonio, TX, USA - March 03 - 05, 2014*, CODASPY'14, pages 25–36. ACM, 2014.

[513] Y. Wang, J. Zheng, C. Sun, and S. Mukkamala. Quantitative Security Risk Assessment of Android Permissions and Applications. In *Data and Applications Security and Privacy XXVII - 27th Annual IFIP WG 11.3 Conference, DBSec 2013, Newark, NJ, USA, July 15-17, 2013. Proceedings*, volume 7964 of *DBSec'13*, pages 226–241. Springer, 2013.

[514] 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, Scottsdale, AZ, USA, November 3-7, 2014*, CCS '14, pages 1329–1341, Scottsdale, AZ, 2014. ACM.

[515] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. ProfileDroid: multi-layer profiling of android applications. In *The 18th Annual International Conference on Mobile Computing and Networking, Mobicom'12, Istanbul, Turkey, August 22-26, 2012*, Mobicom'12, pages 137–148. ACM, 2012.

[516] Z. Wei and D. Lie. LazyTainter: Memory-Efficient Taint Tracking in Managed Runtimes. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, SPSM@CCS 2014, Scottsdale, AZ, USA, November 03 - 07, 2014*, SPSM'14, pages 27–38. ACM, 2014.

[517] T. Werthmann, R. Hund, L. Davi, A.-R. Sadeghi, and T. Holz. PSiOS: bring your own privacy & security to iOS devices. In *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, ASIACCS'13, pages 13–24. ACM, 2013.

[518] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android Permissions Remystified: A Field Study on Contextual Integrity. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, SEC'15, pages 499–514. USENIX Association, 2015.

[519] E. R. Wognsen, H. S. Karlsen, M. C. Olesen, and R. R. Hansen. Formalisation and analysis of Dalvik bytecode. *Sci. Comput. Program.*, 92:25–55, 2014.

[520] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1—-19:36, Oct. 2009.

[521] C. Wu, Y. Zhou, K. Patel, Z. Liang, and X. Jiang. AirBag: Boosting Smartphone Resistance to Malware Infection. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[522] 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 *Seventh Asia Joint Conference on Information Security, AsiaJCIS 2012, Kaohsiung, Taiwan, August 9-10, 2012*, AsiaJCIS'12, pages 62–69. IEEE, 2012.

[523] J. Wu, T. Cui, T. Ban, S. Guo, and L. Cui. PaddyFrog: systematically detecting confused deputy vulnerability in Android applications. *Security and Communication Networks*, 8(13):2338–2349, 2015.

[524] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, CCS '13, pages 623–634, Berlin, Germany, 2013. ACM.

[525] W.-C. Wu and S.-H. Hung. DroidDolphin: a dynamic Android malware detection framework using big data and machine learning. In *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems, RACS 2014, Towson, Maryland, USA, October 5-8, 2014*, RACS'14, pages 247–252. ACM, 2014.

[526] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective Real-Time Android Application Auditing. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, S&P'15, pages 899–914. IEEE Computer Society, 2015.

[527] J. Xiao, H. Huang, and H. Wang. Kernel Data Attack Is a Realistic Security Threat. In *Security and Privacy in Communication Networks - 11th International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Revised Selected Papers*, volume 164 of *SecureComm'15*, pages 135–154. Springer, 2015.

[528] X. Xiao, N. Tillmann, M. Fähndrich, J. d. Halleux, and M. Moskal. User-aware privacy control via extended static-information-flow analysis. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, ASE'12, pages 80–89. ACM, 2012.

[529] X. Xiao, X. Xiao, Y. Jiang, and Q. Li. Detecting Mobile Malware with TMSVM. In *International Conference on Security and Privacy in Communication Networks - 10th International ICST Conference, SecureComm 2014, Beijing, China, September 24-26, 2014, Revised Selected Papers, Part I*, volume 152 of *SecureComm'14*, pages 507–516. Springer, 2014.

[530] L. Xie, X. Zhang, J.-P. Seifert, and S. Zhu. pbmds: a behavior-based malware detection system for cellphone devices. In *Proceedings of the Third ACM Conference on Wireless Network Security, WISEC 2010, Hoboken, New Jersey, USA, March 22-24, 2010*, pages 37–48. ACM, 2010.

[531] Z. Xie and S. Zhu. GroupTie: toward hidden collusion group discovery in app stores. In *7th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WiSec'14, Oxford, United Kingdom, July 23-25, 2014*, WISEC'14, pages 153–164. ACM, 2014.

[532] Z. Xie and S. Zhu. AppWatcher: unveiling the underground market of trading mobile app reviews. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, New York, NY, USA, June 22-26, 2015*, WISEC'15, pages 10:1–10:11. ACM, 2015.

[533] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang. Upgrading Your Android, Elevating My Malware: Privilege Escalation through Mobile OS Updating. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, S&P'14, pages 393–408. IEEE Computer Society, 2014.

[534] J. Xu, Y. Yu, Z. Chen, B. Cao, W. Dong, Y. Guo, and J. Cao. MobSafe: cloud computing based forensic analysis for massive mobile applications using data mining. *Tsinghua Science and Technology*, 18(4):418–427, 2013.

[535] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, Security'12, pages 27–27, Bellevue, WA, 2012. USENIX Association.

[536] W. Xu, F. Zhang, and S. Zhu. Permlyzer: Analyzing permission usage in android applications. In *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*, pages 400–410, Nov. 2013.

[537] Z. Xu and S. Zhu. Semadroid: A privacy-aware sensor management framework for smartphones. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, CODASPY 2015, San Antonio, TX, USA, March 2-4, 2015*, pages 61–72, 2015.

[538] L. Yan, Y. Guo, and X. Chen. SplitDroid: Isolated Execution of Sensitive Components for Mobile Applications. In *Security and Privacy in Communication Networks - 11th International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Revised Selected Papers*, volume 164 of *SecureComm'15*, pages 78–96. Springer, 2015.

[539] L. K. Yan and H. Yin. DroidScope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, Security'12, pages 29–29, Bellevue, WA, 2012. USENIX Association.

[540] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. A. Porras. DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications. In *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part I*, volume 8712 of *ESORICS'14*, pages 163–182. Springer, 2014.

[541] C. Yang, G. Yang, A. Gehani, V. Yegneswaran, D. Tariq, and G. Gu. Using Provenance Patterns to Vet Sensitive Behaviors in Android Apps. In *Security and Privacy in Communication Networks - 11th International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Revised Selected Papers*, volume 164 of *SecureComm'15*, pages 58–77. Springer, 2015.

[542] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan. IntentFuzzer: detecting capability leaks of android applications. In *9th ACM Symposium on Information, Computer*

and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014, ASIACCS'14, pages 531–536. ACM, 2014.

[543] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, ICSE 2015, Florence, Italy, 2015.

[544] T. Yang, Y. Yang, K. Qian, D. C.-T. Lo, Y. Qian, and L. Tao. Automated Detection and Analysis for Android Ransomware. In *17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015, 7th IEEE International Symposium on Cyberspace Safety and Security, CSS 2015, and 12th IEEE International Conference on Embedded Software and Systems, ICESS 2015, New York, NY, USA, August 24-26, 2015*, CSS'15, pages 1338–1343. IEEE, 2015.

[545] W. Yang, J. Li, Y. Zhang, Y. Li, J. Shu, and D. Gu. APKLancet: tumor payload diagnosis and purification for android applications. In *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, ASIACCS'14, pages 483–494. ACM, 2014.

[546] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, ICSE'15, pages 303–313. IEEE, 2015.

[547] 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, 2012*, pages 101–104, Hong Kong, China, Nov. 2012.

[548] 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 *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, CCS '13, pages 1043–1054, Berlin, Germany, 2013. ACM.

[549] H. Ye, S. Cheng, L. Zhang, and F. Jiang. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In *The 11th International Conference on Advances in Mobile Computing & Multimedia, MoMM '13, Vienna, Austria, December 2-4, 2013*, MoMM'13, page 68. ACM, 2013.

[550] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik. A New Android Malware Detection Approach Using Bayesian Classification. In *27th IEEE International Conference on Advanced Information Networking and Applications, AINA 2013, Barcelona, Spain, March 25-28, 2013*, AINA'13, pages 121–128. IEEE Computer Society, 2013.

[551] W. You, B. Liang, J. Li, W. Shi, and X. Zhang. Android Implicit Information Flow Demystified. In *Proceedings of the 10th ACM Symposium on Information, Computer and*

Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015, ASIACCS '15, pages 585–590, New York, NY, USA, 2015. ACM.

[552] W. You, K. Qian, M. Guo, P. Bhattacharya, Y. Qian, and L. Tao. A hybrid approach for mobile security threat analysis. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, New York, NY, USA, June 22-26, 2015*, WISEC'15, pages 28:1–28:2. ACM, 2015.

[553] Y. Yu, P. Manolios, and L. Lamport. Model checking tla$^+$ specifications. In *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, pages 54–66, 1999.

[554] P. Zave. A practical comparison of alloy and spin. Technical report, 2012.

[555] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. ViewDroid: towards obfuscation-resilient mobile application repackaging detection. In *7th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WiSec'14, Oxford, United Kingdom, July 23-25, 2014*, WISEC'14, pages 25–36. ACM, 2014.

[556] L. Zhang, Y. Zhang, and T. Zang. Detecting Malicious Behaviors in Repackaged Android Apps with Loosely-Coupled Payloads Filtering Scheme. In *International Conference on Security and Privacy in Communication Networks - 10th International ICST Conference, SecureComm 2014, Beijing, China, September 24-26, 2014, Revised Selected Papers, Part I*, volume 152 of *SecureComm'14*, pages 454–462. Springer, 2014.

[557] M. Zhang, Y. Duan, Q. Feng, and H. Yin. Towards Automatic Generation of Security-Centric Descriptions for Android Apps. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 518–529. ACM, 2015.

[558] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, CCS'14, pages 1105–1116. ACM, 2014.

[559] M. Zhang and H. Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, NDSS'14, 2014.

[560] M. Zhang and H. Yin. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, ASIACCS'14, pages 259–270. ACM, 2014.

[561] N. Zhang, K. Yuan, M. Naveed, X.-y. Zhou, and X. Wang. Leave Me Alone: App-Level Protection against Runtime Information Gathering on Android. In *2015 IEEE*

Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015, pages 915–930. IEEE Computer Society, 2015.

[562] X. Zhang, A. Ahlawat, and W. Du. AFrame: isolating advertisements from mobile applications in Android. In *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, ACSAC '13, pages 9–18. ACM, 2013.

[563] X. Zhang and W. Du. Attacks on Android Clipboard. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 11th International Conference, DIMVA 2014, Egham, UK, July 10-11, 2014. Proceedings*, volume 8550 of *DIMVA'14*, pages 72–91. Springer, 2014.

[564] Y. Zhang, K. Huang, Y. Liu, K. Chen, L. Huang, and Y. Lian. Timing-Based Clone Detection on Android Markets. In *International Conference on Security and Privacy in Communication Networks - 10th International ICST Conference, SecureComm 2014, Beijing, China, September 24-26, 2014, Revised Selected Papers, Part II*, volume 153 of *SecureComm'15*, pages 375–381. Springer, 2014.

[565] Y. Zhang, M. Yang, G. Gu, and H. Chen. FineDroid: Enforcing Permissions with System-Wide Application Execution Context. In *Security and Privacy in Communication Networks - 11th International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Revised Selected Papers*, volume 164 of *SecureComm'15*, pages 3–22. Springer, 2015.

[566] 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 *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, CCS '13, pages 611–622, Berlin, Germany, 2013. ACM.

[567] K. Zhao, D. Zhang, X. Su, and W. Li. Fest: A feature extraction and selection tool for Android malware detection. In *2015 IEEE Symposium on Computers and Communication, ISCC 2015, Larnaca, Cyprus, July 6-9, 2015*, ISCC'15, pages 714–720. IEEE Computer Society, 2015.

[568] M. Zhao, F. Ge, T. Zhang, and Z. Yuan. AntiMalDroid: An Efficient SVM-Based Malware Detection Framework for Android. In *Information Computing and Applications - Second International Conference, ICICA 2011, Qinhuangdao, China, October 28-31, 2011. Proceedings, Part I*, volume 243 of *ICICA'11*, pages 158–166. Springer, 2011.

[569] S. Zhao, X. Li, G. Xu, L. Zhang, and Z. Feng. Attack Tree Based Android Malware Detection with Hybrid Analysis. In *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2014, Beijing, China, September 24-26, 2014*, TrustCom'15, pages 380–387. IEEE Computer Society, 2014.

[570] 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

7th International Conference on Malicious and Unwanted Software, MALWARE 2012, Fajardo, PR, USA, October 16-18, 2012, pages 135–143, Fajardo, PR, Oct. 2012.

[571] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci. StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, CODASPY 2015, San Antonio, TX, USA, March 2-4, 2015*, CODASPY'15, pages 37–48. ACM, 2015.

[572] Y. Zhauniarovich, O. Gadyatskaya, and B. Crispo. Enabling trusted stores for android. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, CCS'13, pages 1345–1348. ACM, 2013.

[573] Y. Zhauniarovich, G. Russello, M. Conti, B. Crispo, and E. Fernandes. MOSES: Supporting and Enforcing Security Profiles on Smartphones. *IEEE Trans. Dependable Sec. Comput.*, 11(3):211–223, 2014.

[574] 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 *SPSM'12, Proceedings of the Workshop on Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2012, October 19, 2012, Raleigh, NC, USA*, SPSM '12, pages 93–104, New York, NY, USA, 2012. ACM.

[575] M. Zheng, P. P. C. Lee, and J. C. S. Lui. ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-virus Systems. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 9th International Conference, DIMVA 2012, Heraklion, Crete, Greece, July 26-27, 2012, Revised Selected Papers*, volume 7591 of *DIMVA'12*, pages 82–101. Springer, 2012.

[576] M. Zheng, M. Sun, and J. C. S. Lui. Droid Analytics: A Signature Based Analytic System to Collect, Extract, Analyze and Associate Android Malware. In *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2013 / 11th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA-13 / 12th IEEE International Conference on Ubiquitous Computing and Communications, IUCC-2013, Melbourne, Australia, July 16-18, 2013*, TrustCom'13, pages 163–171, Washington, DC, USA, 2013. IEEE Computer Society.

[577] M. Zheng, M. Sun, and J. C. S. Lui. DroidRay: a security evaluation system for customized android firmwares. In *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, ASIACCS'14, pages 471–482. ACM, 2014.

[578] M. Zheng, M. Sun, and J. C. S. Lui. DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability. In *International Wireless Communications and Mobile Computing Conference, IWCMC 2014, Nicosia, Cyprus, August 4-8, 2014*, IWCMC'14, pages 128–133. IEEE, 2014.

[579] Y. Zhongyang, Z. Xin, B. Mao, and L. Xie. DroidAlarm: an all-sided static analysis tool for Android privilege-escalation malware. In *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, ASIACCS'13, pages 353–358. ACM, 2013.

[580] Q. Zhou, D. Wang, Y. Zhang, B. Qin, A. Yu, and B. Zhao. ChainDroid: Safe and Flexible Access to Protected Android Resources Based on Call Chain. In *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2013 / 11th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA-13 / 12th IEEE International Conference on Ubiquitous Computing and Communications, IUCC-2013, Melbourne, Australia, July 16-18, 2013*, TrustCom'13, pages 156–162. IEEE Computer Society, 2013.

[581] W. Zhou, Z. Wang, Y. Zhou, and X. Jiang. DIVILAR: diversifying intermediate language for anti-repackaging on android platform. In *Fourth ACM Conference on Data and Application Security and Privacy, CODASPY'14, San Antonio, TX, USA - March 03 - 05, 2014*, pages 199–210. ACM, 2014.

[582] W. Zhou, X. Zhang, and X. Jiang. AppInk: watermarking android apps for repackaging deterrence. In *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, pages 1–12. ACM, 2013.

[583] W. Zhou, Y. Zhou, M. C. Grace, X. Jiang, and S. Zou. Fast, scalable detection of "Piggybacked" mobile applications. In *Third ACM Conference on Data and Application Security and Privacy, CODASPY'13, San Antonio, TX, USA, February 18-20, 2013*, CODASPY'13, pages 185–196. ACM, 2013.

[584] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Second ACM Conference on Data and Application Security and Privacy, CODASPY 2012, San Antonio, TX, USA, February 7-9, 2012*, CODASPY'12, pages 317–326. ACM, 2012.

[585] X.-y. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, S&P'14, pages 409–423. IEEE Computer Society, 2014.

[586] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 95–109, San Francisco, CA, 2012. IEEE.

[587] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in android applications. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, San Diego, CA, 2013.

[588] Y. Zhou, K. Patel, L. Wu, Z. Wang, and X. Jiang. Hybrid User-level Sandboxing of Third-party Android Apps. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, pages 19–30. ACM, 2015.

[589] Y. Zhou, K. Singh, and X. Jiang. Owner-Centric Protection of Unstructured Data on Smartphones. In *Trust and Trustworthy Computing - 7th International Conference, TRUST 2014, Heraklion, Crete, Greece, June 30 - July 2, 2014. Proceedings*, volume 8564 of *TRUST'14*, pages 55–73. Springer, 2014.

[590] 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 *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012.

[591] Y. Zhou, L. Wu, Z. Wang, and X. Jiang. Harvesting developer credentials in Android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, New York, NY, USA, June 22-26, 2015*, WISEC'15, pages 23:1–23:12. ACM, 2015.

[592] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *Trust and Trustworthy Computing - 4th International Conference, TRUST 2011, Pittsburgh, PA, USA, June 22-24, 2011. Proceedings*, pages 93–107, Pittsburgh, PA, June 2011. Springer.

[593] H. Zhu, H. Xiong, Y. Ge, and E. Chen. Mobile app recommendations with security and privacy awareness. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, KDD'14, pages 951–960. ACM, 2014.

[594] C. Zuo, J. Wu, and S. Guo. Automatically Detecting SSL Error-Handling Vulnerabilities in Hybrid Mobile Web Apps. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, ASIACCS'15, pages 591–596. ACM, 2015.