UNIVERSITY OF CALIFORNIA,
IRVINE

Self-Protection of Android Systems from Inter-Component Communication Attacks

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

Mahmoud M. Hammad

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

2018

DEDICATION

{So high [above all] is Allah, the Sovereign, the Truth. And, [O Muhammad], do not hasten with [recitation of] the Qur'an before its revelation is completed to you, and say, "My Lord, increase me in knowledge."} The Quran 20:114, p.320

This dissertation and all my academic achievements are dedicated
to my inspiring parents, Mohammed Hammad and Amneh Nazzal,
to my best friend and my beloved wife, Salma Suleiman,
and to my little daughters Lamar, Leen, and Layan,
without whom this dissertation would have never been completed.

Contents

List of Figures

List of Tables

ACKNOWLEDGMENTS

First and foremost, I thank Almighty Allah for giving me the strength and patience to work through all these years so that today I can stand proudly with my head held high.

I would like to express my deepest gratitude to my advisor, **Professor Sam Malek** for his endless support, guidance, insightful discussion, and encouragements. I started working with Sam with little knowledge about conducting scientific research, presenting them, and writing research papers, yet Sam was so humble and patient listening to my ideas, discussing them, and politely critiquing them. This dissertation wouldn't have been possible without his inspiration and encouragement every step of the way. One truly cannot ask for a better advisor.

I would like to thank the other members of my dissertation committee, **Professor James Jones** and **Professor Cristina Lopes** for their valuable feedback and devoting time of their busy schedules. I also enjoyed collaborating with **Dr. Joshua Garcia** and **Dr. Hamid Bagheri** and appreciate their contribution in this dissertation. Moreover, I would like to thank **Professor Daniel A. Menasce** and **Professor Hassan Gomaa** for their valuable feedback on my research and my presentations.

During my Ph.D, I worked very closely with Dr. Joshua Garcia. Josh gave me tremendous support and help starting from bouncing ideas all the way to preparing slides to be presented in a conference. Not only Josh helped me to succeed in my academic journey but also he was always available as a close friend. Josh listened to me tirelessly bragging about my various situations and difficulties that I went through during my PhD journey and always did his best to help with the situation. I truly feel privileged to have a friend like Josh that I could both work with and learn from.

I will be forever thankful to **Salma Suleiman**, my wife and my best friend. Her unfailing love, support, understanding, and encouragement has seen me through tumultuous times. I thank her for patiently and simultaneously raising our three beautiful daughters, listening to my rehearsals, enthusiastically giving me many valuable feedback about my ideas and my slides, and bringing joy to our family. I also thank my little sweet daughters, **Lamar**, **Leen**, and **Layan**, for the patience they showed me during this journey. Words would never say how grateful I am for them. I consider myself the luckiest in the world to have such a lovely and caring family, standing beside me with their love and unconditional support.

Last but not least, I deeply thank my parents, **Mohammed Hammad** and **Amneh Nazzal**, my parents-in-low, **Mahmoud Suleiman** and **Hanaa Otair**, for their prayers and their inculcating in me the dedication and discipline to do whatever I undertake well. I also thank my brothers, **Ahmad Hammad** and **Faiq Hammad**, and my sisters, **Lamya Hammad** and **Leema Hammad**, for their support and kind help. I know it has been a tough time for all of them waiting for me, my wife, and my daughters all the past seven years without visiting them even once fearing that I might not get the student visa again and loose my life's biggest dream.

CURRICULUM VITAE

Mahmoud M. Hammad

EDUCATION

Doctor of Philosophy in Software Engineering **2018**
University of California, Irvine *Irvine, CA*

Masters of Science in Software Engineering **2013**
George Mason University *Fairfax, VA*

Bachelor of Science in Computer Science **2005**
Yarmouk University *Irbid, Jordan*

RESEARCH EXPERIENCE

Graduate Research Assistant **2015–2018**
University of California, Irvine *Irvine, California*

Graduate Research Assistant **2014–2015**
George Mason University *Fairfax, VA*

TEACHING EXPERIENCE

Graduate Teaching Assistant **2016–2017**
University of California, Irvine *Irvine, California*

PROFESSIONAL EXPERIENCE

Mobile Apps Developer **2012–2014**
The Helen A. Keller Institute *Fairfax, California*

Software Engineer **2006–2011**
Jordan University of Science & Technology *Irbid, Jordan*

Software Engineer **2005–2006**
JoVal for IT *Amman, Jordan*

REFEREED JOURNAL PUBLICATIONS

DELDroid: Determination and Enforcement of Least-Privilege Architecture in Android. **Oct 2017**
Accepted with revision to the Journal of Systems and Software (JSS)

Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware. **Sept 2017**
ACM Transactions on Software Engineering and Methodology (TOSEM)

REFEREED CONFERENCE PUBLICATIONS

SALMA: Self-protection of Android Systems from Inter-Component Communication Attacks. **September 2018**
International Conference on Automated Software Engineering (ASE)

A Large-Scale Empirical Study on the Effects of Code Obfuscations on Android Apps and Anti-Malware Products. **May 2018**
International Conference of Software Engineering (ICSE)

Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware. **May 2018**
International Conference of Software Engineering (ICSE) – journal-first

Automatic Generation of Inter-Component Communication Exploits for Android Applications. **Sept 2017**
European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)

Determination and Enforcement of Least-Privilege Architecture in Android. **April 2017**
International Conference of Software Architecture (ICSA)

TECHNICAL PAPERS

DELDroid: Determination and Enforcement of Least-Privilege Architecture in Android. **Feb 2018**
Institute for Software Research, University of California, Tech. #UCI-ISR-18-2

ABSTRACT OF THE DISSERTATION

Self-Protection of Android Systems from Inter-Component Communication Attacks

By

Mahmoud M. Hammad

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2018

Associate Professor Sam Malek, Chair

Android is widely used for the development and deployment of autonomous and smart systems, including software targeted for IoT and mobile devices. Security of such systems is an increasingly important concern. Although Android is the predominant mobile platform [26], it is also the most targeted platform by malware authors [24, 25] resulting in millions of malicious apps distributed in numerous app stores [47]. Android relies on a permission model to secure the system's resources and apps. In Android, since the permissions are granted at the granularity of apps, and all components in an app inherit those permissions, an app's components are over-privileged, i.e., components are granted more privileges than they actually need. Systematic violation of *least-privilege principle* in Android is the root cause of many types of Inter-Component Communication (ICC) attacks that can lead to serious security and privacy risks [89, 106, 135, 173].

Due to the increasing use of code obfuscation in Android apps, the current security mechanisms for Android apps, both static and dynamic analysis approaches, are insufficient for detection and prevention of the increasingly dynamic and sophisticated security attacks. Static analysis approaches suffer from false positives whereas dynamic analysis approaches suffer from false negatives. Moreover, they all lack the ability to efficiently analyze systems with incremental changes—such as adding/removing apps, granting/revoking permissions, and

dynamic components' communications. Each time the system changes, the entire analysis needs to be repeated, making the existing approaches inefficient for practical use.

To mitigate these issues, this dissertation presents a novel self-protecting Android software system that automatically determines and continuously maintains the least-privilege architecture of an Android system, incrementally and efficiently analyzes its security posture, and dynamically enforces the maintained least-privilege architecture at runtime. The approach, entitled SALMA, protects the system against ICC attacks at all times in spite of changes at runtime.

The least-privilege architecture limits the privileges granted to apps without the need to modify them or breaking their functionalities. Static program analysis techniques have been utilized to extract the exact privileges each component needs for providing its functionality. A *Multiple-Domain Matrix* representation of the system's least-privilege architecture is then kept in sync with the running system to reason about it at runtime. Every time the system changes, SALMA determines (1) the impacted part of the system, and (2) the subset of the security analyses that need to be performed, thereby greatly improving the performance and the scalability of the approach.

All conducted experiments on hundreds of real-world apps corroborate the scalability and efficiency of the proposed approach in reducing the attack surface of Android systems as well as its ability to detect and prevent security attacks at runtime with minimal disruption.

Chapter 1

Introduction

Android is widely used for the development and deployment of autonomous and smart software systems, including software intended for execution on a variety of mobile devices, as well as software targeted for Internet of Things (IoT) settings, such as smart homes. Security of such systems is an increasingly important concern. Reusability is a major reason behind the meteoric rise in the popularity of the Android platform [26] and the increasing number of apps [48]. To develop rich apps, Android promotes reusability of (1) information and services provided by third-party apps, through its flexible Inter-Component Communication (ICC) model, and (2) sensitive resources protected by a permission-based model.

Permissions form the foundation of security in Android. Android relies on a permission-based model for controlling the resources that each app is allowed to access. Permissions are often granted to an app at the discretion of end user, who makes a decision based on its perceived trustworthiness and expected functionality. Android's permission-based access control model, however, has shown to be ineffective in protecting system resources and apps from security attacks [89]. In Android, all components of an Android app inherit the permissions granted to the app, regardless of whether they need those permissions or not. As a result, a malicious

component inside an app, such as a third-party library, can leverage privileges meant for other components for nefarious purposes [173]. Moreover, by default, a component in Android has significant leeway in terms of the components it can communicate with, both within and outside of its parent app.

The over-privileged nature of components in Android has become the main attack vector for Android apps, which can lead to serious security and privacy risks [89, 106, 135, 173]. These kinds of attacks cannot be prevented by the platform at the moment, as they do not violate the security mechanisms supplied by Android.

Prior research efforts have proposed various solutions to help address certain instances of component-level ICC attacks. Some of the proposed solutions have focused on isolating specific type of component-level threats, caused by for example advertisement [167, 199] or JNI libraries [205]; such approaches are narrowly targeted, and thus, inappropriate for applying comprehensively to other types of component-level threats. Others have proposed component-level permission assignment for third-party components in an app [212, 195], yet they are incapable of controlling communications among components. They also often require application modification or developer intervention, significantly hindering their adoption in practice. Therefore, the current state-of-the-art security mechanisms for Android apps, both static and dynamic analysis approaches, as well as the current state-of-the-practice security mechanisms for Android apps, i.e., the commercial anti-malware products, are all insufficient for detecting and preventing the increasingly sophisticated security attacks.

Static analysis approaches suffer from false positives due to their over-approximation of the analyzed apps, e.g., producing warnings for vulnerabilities that are not executable at runtime. On the other hand, dynamic analysis approaches suffer from false negatives due to the *reachability* problem, where vulnerabilities are missed due to inputs that fail to reach the vulnerable code. Not to mention, the existing commercial anti-malware products can be easily defeated, as we will show in Chapter 4, using *code obfuscation*. Code obfuscation transforms

a code into a form that is difficult to analyze and reverse engineer. These transformations change the syntax of code but not their semantics [93].

Moreover, due to the complex and dynamic nature of Android systems (e.g., adding a new app, removing an existing app, granting/revoking a permission, and dynamic class loading), their security posture continuously changes over time. Simply repeating the entire security analysis of an Android system, either statically or dynamically, every time the system changes is prohibitively expensive for practical use.

To overcome the shortcomings of the current approaches, this dissertation proposes SALMA, a fully automated and novel self-protecting Android software system that (1) automatically determines and enforces the *least-privilege architecture* (LP architecture) of an Android system, (2) continuously monitors the running Android system, (3) incrementally and efficiently analyzes the security posture of the system, and (4) dynamically enforces security policies to prevent security attacks at runtime. An LP architecture is one in which the components are only granted the privileges that they require for providing their functionality [207]. An LP architecture, thus, reduces the risk of an Android system being compromised by limiting its attacks surface. In addition, when a component is compromised, the impact is localized within the scope of that component. A smaller attack surface also facilitates both manual and automated means of inspecting the system's security attributes.

Establishing the least-privilege architecture is quite challenging as it demands mediation of all conceivable channels through which a component may interact with components within and outside its parent app, as well as the underlying system resources. SALMA leverages static program analysis to automatically identify the architectural elements comprising an Android system, as well as the inter-component communication and resource-access privileges each component needs to provide its functionality. It then derives the *initial* LP architecture, i.e., a model, for the system. SALMA models the LP architecture of an Android system as a *Multiple-Domain-Matrix* (MDM) [149]—which provides an elegant, yet

compact, representation of all relationships among principal elements, such as components and permissions, in a system. SALMA further allows a security expert to modify the initial LP architecture as needed to establish the proper privileges for each component. Finally, SALMA enforces automatically obtained or expert-supplied LP architecture at runtime, thus ensuring components are not able to obtain more privileges than that prescribed by the architecture.

Next, SALMA monitors the running system to keep the model synchronized with the running system. Whenever the model changes, SALMA determines (1) the impacted part of the system, and (2) the required security analyses that need to be performed. Finally, SALMA adjusts security policies and enforces them at runtime, thus ensuring the system is safe and protected at all times. Our implementation of the MDM provides a flexible way to load and analyze parts of the system, improving the scalability and efficiency of the overall approach.

By providing an efficient least-privilege determination process associated with a thorough enforcement system, SALMA allows users to focus their analysis efforts on a very narrowed set of interactions in the architecture. This is especially valuable, since at the scale of a single device, the state-of-the-art inter-component communication analysis tools produce an enormous number of potential links between message-passing locations and possible message targets, making manual analysis required to confirm any potential threat rather tedious and error-prone.

SALMA can be used to limit the levels of access available to an app and its components and protect Android systems without modification of the apps' implementation logic, allowing our approach to be applied to all existing Android apps.

Our evaluation of SALMA using hundreds of real-world apps corroborates its ability in significantly reducing the attack surface of Android systems as well as its efficiency and scalability in analyzing evolving Android systems with minimal disruption to apps and

their services while thwarting security threats to keep the system protected at all times. SALMA achieves 94%-99% reduction of attack surface, 70%-84% greater detection of attacks than state-of-the-art approaches, and 45%-85% greater prevention of attacks than those approaches.

1.1 Dissertation Structure

The rest of this dissertation is organized as follows. Chapter 2 provides a background on self-adaptive software systems, Android framework and its privilege management scheme, and the ICC security attacks targeting the Android platform which SALMA detects and mitigates at runtime. Chapter 3 presents the research problem and the scope of this thesis. In Chapter 4, we show, via a large-scale empirical study, the effect of code obfuscation on Android apps and anti-malware products. Chapter 5 presents an Android system to motivate the research and illustrate the approach. Chapter 6 provides a detail description of SALMA and its implementation. The evaluation results of SALMA are presented in Chapter 7. Chapter 8 discusses the related literature effort in light of SALMA. Finally, Chapter 9 concludes this dissertation with discussion of the contributions, limitations and the future work.

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

- Mahmoud Hammad, Hamid Bagheri, and Sam Malek. Self-Protection of Android Systems from Inter-Component Communication Attacks *The 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), September 2018, Montpellier, France.*

- Mahmoud Hammad, Joshua Garcia, and Sam Malek. A Large-Scale Empirical Study on the Effects of Code Obfuscations on Android Apps and Anti-Malware Products. *The*

40th International Conference of Software Engineering (ICSE), May 2018, Gothenburg, Sweden.

- Mahmoud Hammad, Hamid Bagheri, and Sam Malek. DELDroid: Determination and Enforcement of Least-Privilege Architecture in Android. *Accepted with revision to the Journal of Systems and Software (JSS). Submitted on October 2017*

- Mahmoud Hammad, Hamid Bagheri, and Sam Malek. DELDroid: Determination and Enforcement of Least-Privilege Architecture in Android. *University of California, Irvine, Institute for Software Research, technical report # UCI-ISR-18-2, April 2018*

- Joshua Garcia, Mahmoud Hammad, and Sam Malek. Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware. *ACM Transactions on Software Engineering and Methodology (TOSEM). Volume 26, No. 3, Article 11 (January 2018)*

- Joshua Garcia, Mahmoud Hammad, and Sam Malek. Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware. *International Conference of Software Engineering (ICSE), Journal-first track, May 2018, Gothenburg, Sweden.*

- Joshua Garcia, Mahmoud Hammad, Negar Ghorbani, and Sam Malek. Automatic Generation of Inter-Component Communication Exploits for Android Applications. *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), September 2017, Paderborn, Germany.*

- Mahmoud Hammad, Hamid Bagheri, and Sam Malek. Determination and Enforcement of Least-Privilege Architecture in Android. *International Conference of Software Architecture (ICSA 2017), Gothenburg, Sweden, April 2017.*

Chapter 2

Background and Research Motivation

To help the reader follow the discussions that ensue in this dissertation, this chapter provides a brief overview of the self-adaptive software systems, Android framework, the over-privileged nature of Android's access control model, and the ICC security attacks that are threatening the Android platform.

2.1 Self-Adaptive Software System

Over the past decade, researchers and software industrial organizations have invested significant resources on creating *software ecosystems* [132, 186, 196, 77, 155]. A software ecosystem includes a platform for constructing applications—similar to product lines—however, differ in scope due to the intent of releasing the platform for third-party development outside of a single organization's boundaries [77].

From a more human- or process-oriented perspective, ecosystems may also include the developers themselves, and the community of domain experts and users [77]. *Application frameworks* are key enablers of software ecosystems. By providing a variety of APIs, libraries,

and services to third-party developers, application frameworks facilitate the development of new features in the form of apps or plug-ins on top of a platform. Sharing a platform beyond a single organization's boundaries further expands the market share of the organization(s) responsible for the platform at the heart of an ecosystem. Encouraging and supporting an ecosystem also enables customizability (e.g., new features in the form of new applications or plug-ins built on top of a platform).

Android is currently the dominant mobile platform accounting for 85% of the market share [9]. Millions of apps are built on top of the Android platform [48], resulting in a large dynamic software ecosystem, where apps are constantly being added or removed to the repositories making these apps available to users.

Given that each user has nearly 100 apps on a device [19], a single Android device can be considered as a rather complex software system, involving apps from many organizations all running on a single platform. Such a system is highly dynamic, involving a variety of software (e.g., messages being sent across apps) and hardware events (e.g., sensor events). Each of these events can be occurring under different contexts (e.g., while the phone screen is off, while the battery is low, etc.). Managing or adapting such a system is challenging due to the varying contexts and the high dynamism. Furthermore, such a system with a high number of apps, some of which may be of questionable provenance, exposes a user to a variety of security vulnerabilities.

Ideally, such a system would be self-protected, where the system is provided with policies that specify and ensure that the system achieves specific security objectives. For example, a user may wish to ensure that her location is never sent outside of the phone when the device's screen is locked. Self-protecting software, similar to other types of self-* software, relies upon the principle of *separation of concerns* [131]. Specifically, such a system, as depicted in Figure 2.1, separates adaptation logic from application-specific business logic to achieve its objectives at runtime.

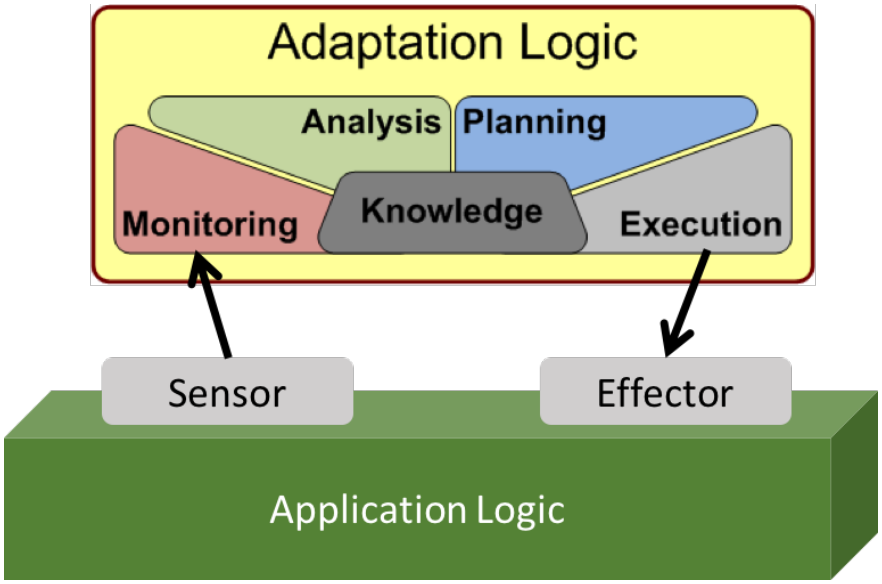


Figure 2.1: A general framework of a self-adaptive software system implementing the MAPE-K control loop.

The adaptation logic realizes a feedback control loop to manage the system. The IBM MAPE-K control loop [139] is the most widely implemented feedback control loop. MAPE-K, as shown in Figure 2.1, consists of four components and a knowledge component. The *Knowledge* contains an abstract representation of the system—often in the form of a component-based architectural model comprising a system's components, their interactions and dependencies.

The *Monitor* component observes the system and keeps the model synchronized with the running system. The *Analyzer* component assesses the system for security threats. The *Planner* component determines the best security policies, a.k.a. adaptation tactics, to be enforced at runtime by the *Executor* component.

2.2 Android Foundations

2.2.1 Android Framework

Figure 2.2 depicts the Android platform stack architecture [1]. Android platform includes a full Linux OS based on the ARM processor, Hardware Abstraction Layer (HAL) which provides standard interfaces that expose device hardware capabilities to the higher-level, system libraries, framework Application Program Interfaces (APIs), and a suite of pre-installed applications.

Android applications (apps) are distributed as an Android Package Kit files (APKs). An APK file is a zipped file that is mainly written in the Java programming language by using a rich collection of APIs provided by the Android Software Development Kit (SDK). Each APK file contains a manifest file, resources (e.g., images), and the app's bytecode. An app's code is compiled into *Dalvik EXecutable* (*DEX*) format, which can be executed on a customized Java Virtual Machine (JVM). There are two JVMs that can execute the DEX format: Android Runtime (ART), introduced in Android version 5 (Lollipop); and Dalvik Virtual Machine (DVM), for older versions.

2.2.2 Application Configuration

Each Android APK includes a mandatory configuration file, called *manifest*. It specifies, among other things, the principal components that constitute the app, including their types and capabilities, as well as required and enforce permissions. The manifest file values are bound to the app at compile time, and cannot be changed afterwards, unless the app is recompiled.

[1] Figure 2.2 is taken from the Android documentation [1]

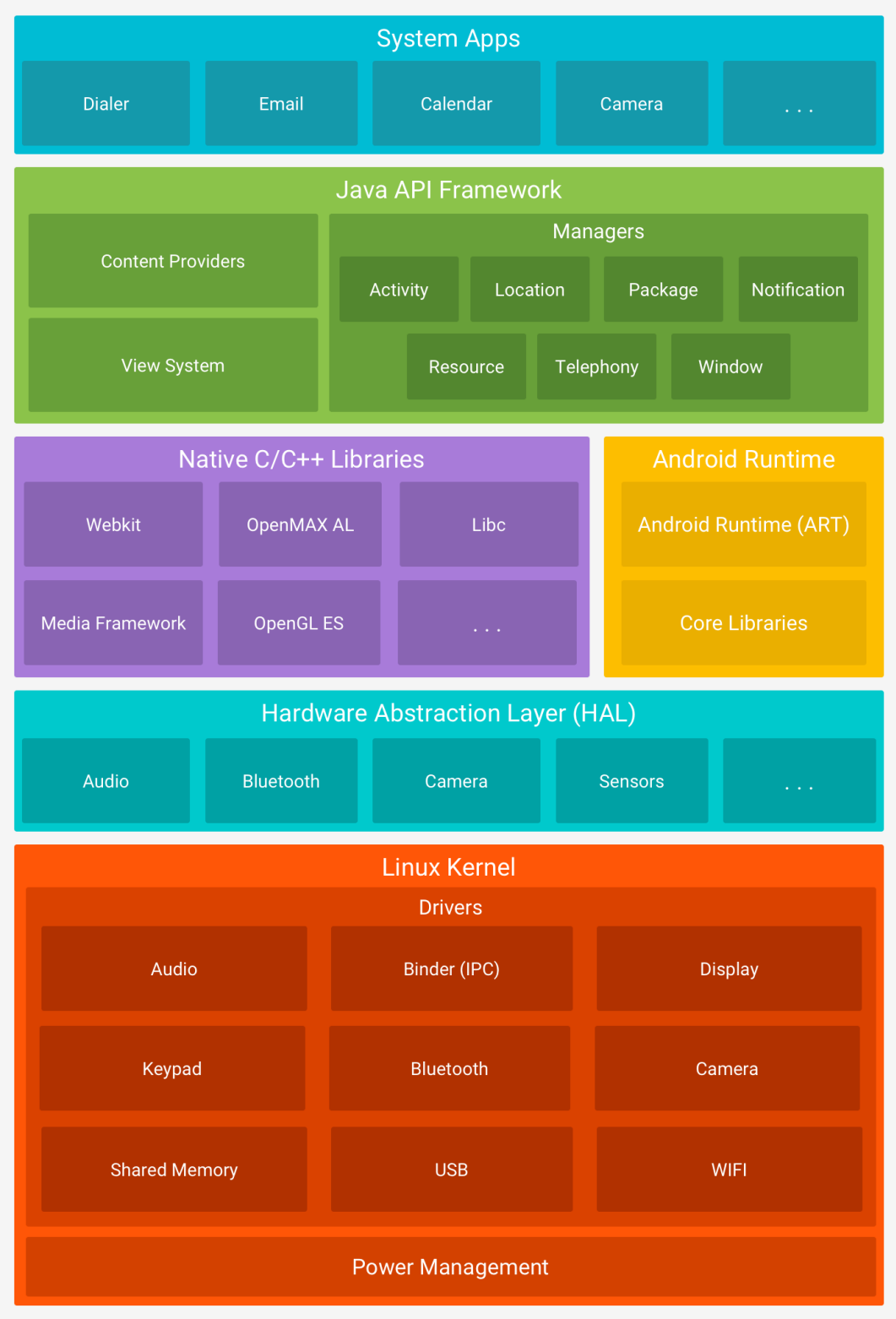


Figure 2.2: The Android software stack taken from the Android documentation [1].

2.2.3 Application Components

Components are basic logical building blocks of apps. Each component can be invoked individually, either by its embodying app or by the system, upon permitted requests from other apps. Android defines four types of components: (1) *Activity* components provide the basis of the Android user interface. Each app may have multiple Activities representing different screens of the app to the user. (2) *Service* components provide background processing capabilities, and do not provide any user interface. Playing a music and downloading a file while a user interacts with another app are examples of operations that may run as a Service. (3) *Broadcast Receiver* components respond asynchronously to system-wide message broadcasts. A receiver component typically acts as a gateway to other components, and passes on messages to Activities or Services to handle them. (4) *Content Provider* components provide database capabilities to other components. Such databases can be used for both intra-app data persistence as well as sharing data across apps. Each component can declare a set of provided interfaces which can be invoked by other components.

2.2.4 Inter-Component Communication

Inter-component communication (ICC) in Android is mainly achieved either by sending *Intents* or using Unified Resource Identifiers (URIs). An Intent message is an event for an action to be performed along with the data that supports that action. Component capabilities are then specified as a set of *Intent Filters* that represent the kinds of requests handled by a given component. Intent Filters are the provided interfaces of a component. Component invocations come in different flavors, e.g., explicit or implicit, intra- or inter-apps, etc. An explicit Intent is delivered to the target component specified in the Intent, whereas an implicit Intent is delivered to a component if the action specified in the Intent matches that specified

in the component's Intent Filter. URIs are used to access or manipulate the encapsulated data in `Content Providers`, the database components in Android apps.

Android's ICC allows for late run-time binding between components in the same or different apps, where the calls are not explicit in the code, rather made possible through event messaging, a key property of event-driven systems.

2.2.5 Android Permissions

Permissions are the cornerstone for the Android security model. Android applies a permission-based model to protect sensitive resources, both system resources and application resources, that each app is allowed to access.

The required permissions stated in the app manifest enable secure access to sensitive resources as well as cross-application interactions. Starting from Android version 6 or Marshmallow, API level 23, Google changed the permission management system in Android from static to dynamic which allows users to grant or revoke permissions at runtime. Before Android version 6, the user has to consent to grant all requested permissions prior to installation. Should the user refuse granting the requested permissions to an app, the app installation is canceled. Besides required permissions, the app manifest may also include enforced permissions that other apps must have in order to interact with this app. In addition to built-in permissions provided by the Android system to protect various system resources, any Android app can also define its own permissions for the purpose of self-protection.

2.3 Android Access Control Model

There are two kinds of privileges a component has: *inter-component communication (ICC) privilege*, allowing a component to communicate with other components in the same or different app, and *resource access privilege*, allowing a component to access the system resources, such as GPS, camera, telephony, etc. Android manages both types of privilege at the app level, meaning that the permissions are granted/revoked at the level of an app and inherited by all components in that app. This causes two kinds of over-privileges, discussed next.

2.3.1 Over-Privileged Resource Access

Android contains a plethora of sensitive system resources (e.g., GPS, camera, account manager, power manager) accessed by obtaining a handle to a system-level, long-running service (e.g., location service, camera service, account service, power manager service). System services are launched by `com.android.server.SystemServer` service, which is started at the boot time of the Android operating system. To use a system service, a component should have the appropriate permission that guards the service. For example, to track the user's location, a component needs to obtain a handle to the location service, which requires the location permission (either `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION`).

The permissions stated in the app manifest enable secure access to sensitive resources. However, a permission granted to an app transfers to all of the components in the app. Android's coarse-grained permission model violates the principle of least-privilege [84, 202], as often not all components of an app need access to the same sensitive resources. The shortcomings of Android's permission model have been widely discussed in the literature [201,

107, 101], and shown to be the root cause of various security attacks, most notably privilege escalation [99, 111].

2.3.2 Over-Privileged Inter-Component Communication

The ICC mechanism in the Android framework provides a flexible component-based development. However, this mechanism gives the components more communication privileges than they actually need and hence violates the principle of least-privilege. Although Android documentation affirms that the sandboxing mechanism, process isolation, is the realization of the least-privilege principle [41], this indicates that the Android security mechanisms treat apps as the minimum security entities and cannot distinguish between their components.

Specifically, Android's ICC mechanism leads to over-privileged architectures, where components needlessly have the ability to use URIs or send Intent messages to invoke services of many other components within and outside their parent apps, and receive a variety of Intent messages implicitly exchanged in the system. A component is allowed to communicate with (1) all components in its parent app, (2) protected components in other apps as long as its parent app has the required permissions, and (3) any public (exported) component in other apps. A component is public if its *VISIBLE* attribute is set to true in the manifest file or declares at least one Intent Filter. Many developers are not aware of the fact that by specifying an Intent Filter for a component, Android by default makes that component public, thus allowing components from other apps to invoke its interfaces [89]. Inter-app communication (IAC) privileges are thus often granted implicitly. Finally, a component does not require a permission to specify an Intent Filter with arbitrary action, thereby allowing that component to receive all implicit Intents exchanged in the system with the specified action.

2.4 Android Security Attacks

The over-privileged nature of components in Android caused by the Android ICC interaction mechanism and the current permission model of Android is the root cause of many security vulnerabilities. It has become a vulnerable attack surface of an Android system which threatens user privacy and has affected millions of users [47]. These ICC attacks are widely discussed in the literature [89, 137, 102, 221, 201, 111, 110, 106, 101, 83, 135].

ICC attacks are security risks facilitated by (1) incorrectly or maliciously using the message-passing system in Android or (2) misusing the permissions in Android. The malicious code to conduct these ICC attacks can be part of an app's implementation logic or even, in a more complicated scenarios, it can be obfuscated or downloaded at runtime using dynamic class loading feature in Android [173]. SALMA provides self-protection against these ICC attacks. This section briefly describes these attacks.

2.4.1 Unauthorized Intent Receipt

In this attack, a malicious component intercepts an implicit Intent by declaring an Intent Filter that matches the sent Intent [89, 137]. In such an attack, a malicious component can access all enclosed data in the intercepted Intent and, possibly perform a phishing attack [51].

There are three different forms of this attack based on the type of the receiver component, i.e., the malicious component [89]: (1) Broadcast theft in which the receiver component can read the content of broadcast Intents without interrupting the broadcast, (2) Activity hijacking in which the receiver component is launched instead of a legitimate Activity, and (3) Service hijacking in which the receiver component is bound to/started instead of a legitimate one. In case a hijacking attack is successful, the sender component may also be a victim of false

response attack [89, 137] in which the receiver component can return a malicious result to the sender component.

As a concrete example of unauthorized Intent receipt attack, consider a legitimate application that processes financial payments. When a user clicks on a "Pay" button, the application sends an implicit Intent to start another Activity that processes the payment. If a malicious Activity hijacks the implicit Intent, then the attacker could receive sensitive information from the user (e.g., card number, billing address, and payment amount). In this Activity hijacking attack, the malicious component can also perform a phishing attack to get even more information from the user after stealing the interface of the legitimate Activity. Phishing attacks cannot be easily determined by users since the Android UI does not specify the currently running application.

2.4.2 Intent Spoofing

In such an attack, a malicious component can communicate with an exported component that is not expecting such communication [137, 89]. If a victim component blindly trusts the received Intent, the malicious component can cause the victim component to perform undesirable actions [115].

There are three different forms of the Intent spoofing attack based on the type of the receiver component, i.e., the victim component [89]: (1) *Malicious Broadcast injection* in which the malicious component can send a malicious broadcast Intent to an exported Broadcast Receiver. Since most Broadcast Receivers act as gateways to other components, and pass messages to Activities and Services [69], the malicious Intent can propagate throughout an app. A more risky scenario can happen if the Broadcast Receiver is registered to receive protected broadcast Intents that only the system can send. In such a scenario, the sender component can send an explicit Intent to the receiver component. If the receiver component

blindly trusts the received Intent without checking the Intent action, it may perform a task that only the system is supposed to trigger. (2) *Malicious Activity launch*, analogous to cross-site request forgeries in websites [74], occurs when a victim component is launched by a malicious component that it does not expect communication from. Since Activities provide GUI interfaces, this attack can be an annoyance to the users. Successfully launching the receiver Activity can cause it to change data in the background using the data enclosed in the malicious Intent sent by the malicious component. (3) *Malicious Service launch* is similar to *malicious Activity launch* except that the interaction between the sender and the receiver components occurs in the background. If a *malicious Activity launch* or a *malicious Service launch* attack is successful, the receiver component may return sensitive information to the malicious component.

As a concrete example of Intent spoofing attack, consider an application that contains an advertisement (ad) library. Once a user clicks on an ad, the application sends an implicit Intent to an Activity, referred to as AdActivity here, which displays details of that ad on a web page. In this case, a malicious component can exploit an Intent spoofing attack by sending a carefully crafted implicit Intent to the AdActivity. If the AdActivity does not properly handle the received implicit Intent, the malicious component can deny the service of AdActivity and crash its app resulting in an *inter-process denial-of-service* (IDOS) attack. Moreover, if the AdActivity blindly trusts the incoming implicit Intent, a malicious component can redirect the user to a web page with malicious JavaScript code resulting in a *cross-application scripting (XAS)* attack. For more descriptions of these kinds of Intent spoofing attacks, we refer the interested readers to our paper [115].

2.4.3 Privilege Escalation

This attack allows a malicious component to indirectly perform a privileged task [111, 83]. In this attack, if a vulnerable component possesses a permission without appropriately protecting its interface, a malicious component can perform a privileged task, such as sending a text message or tracking the location of a user, by interacting with that vulnerable component.

It is worth mentioning that, this security attack is not exploitable unless the vulnerable app is granted the permission that is unsafely used. Therefore, if the vulnerable app is not granted that permission, then the privileged-task is not reachable.

2.4.4 Identical Custom Permission

As we mentioned in Section 2.2, any Android app can also define its own permissions and use them to protect its components. Each permission must define a name and a *protection level*, where each level affects the extent to which a permission can be granted or revoked. The notable protection levels for this chapter are *Normal* and *Signature*. A *Normal* permission is automatically granted to apps that request them without asking for the user's approval and they can not be revoked at runtime. A *Signature* permission is granted to applications that are signed with the same certificate as the app that declared the permission.

The custom permission model of Android contains a vulnerability that is rooted in its design: "if two apps define the same custom permission, whichever app is installed first is the one whose definition is used" [70]. A malicious app can exploit this vulnerability to access a protected component with a custom permission by declaring another custom permission with the same name as that legitimate custom permission.

Figure 2.3 illustrates this attack with an Android system of two apps: `Victim` and `Attacker1` apps. Both apps have defined the same permission, named `P1`, with different protection

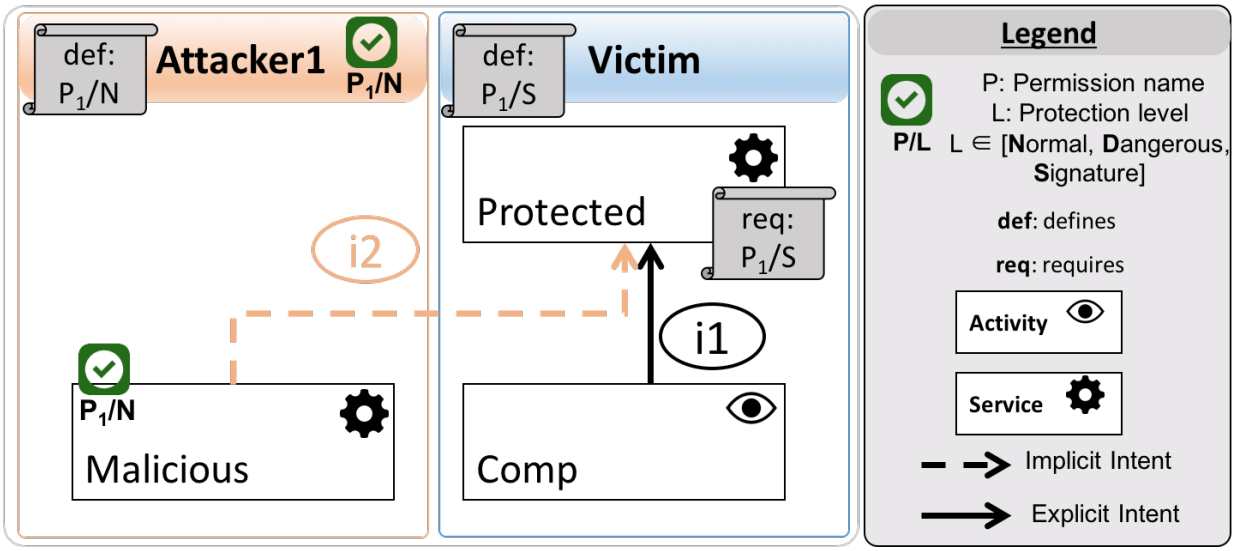


Figure 2.3: Identical Custom Permission Attack

levels. `Victim` app uses `P1` to protect `Protected` component with the protection level *Signature*, meaning that only those apps with the same signature as `Victim` should be able to communicate with `Protected`. The `Attacker1` app, if installed first, can access the `Protected` component by exploiting the identical custom permission vulnerability through defining the permission `P1` with *Normal* protection level which will be granted automatically to it.

Simply uninstalling the `Attacker1` app will not fix the problem, if there is another app that declares the same permission name. As discussed in [70], another malicious app, `Attacker2`, can be installed after installing the `Attacker1` app asking for permission `P1` which would be granted automatically. At this point, even if we uninstall the `Attacker1` app, `Attacker2` can still access the `Protected` component inside the `Victim` app.

2.4.5 Passive Data Leak

Android apps store their sensitive data in database components called `Content Providers`. Content Providers can be used for both intra-app data persistence as well as sharing data

across apps. If the read access to a `Content Provider` is not properly guarded with a permission, other apps can exploit this vulnerability to disclose and leak sensitive data [135].

2.4.6 Content Pollution

This attack is possible when the write access to a `Content Provider` is not properly guarded with a permission [135]. This vulnerability allows a malicious app to manipulate sensitive data managed by a vulnerable app. The manipulated data can cause inadmissible side effects such as altering firewall rules or blocking incoming calls.

Chapter 3

Research Problem

Android's privilege management system has been shown to be ineffective in protecting system resources and apps from security attacks [89, 173]. Since Android manages privileges at the granularity of apps, all components of an Android app inherit the permissions granted to the app, regardless of whether they need those permissions or not. As a result, a malicious component inside an app, such as a third-party library, can leverage privileges meant for other components for nefarious purposes [173]. Moreover, by default, a component in Android has significant leeway in terms of the components it can communicate with, both within and outside of its parent app.

Android's coarse-grained permission model violates the principle of least-privilege [84, 202]. The over-privileged nature of components in Android is the root cause of various Inter-component Communication (ICC) security attacks, most notably Intent spoofing and unauthorized Intent receipt presented in [89], privilege escalation attack discussed in [111], passive data leak and content pollution attacks presented in [135], and identical custom permission attack explained in [70]. These kinds of attacks cannot be prevented by the platform at the moment, as they do not violate the security mechanisms supplied by Android. Moreover,

they cannot be effectively handled by the state-of-the-art security analysis tools, both the static and the dynamic analysis approaches (recall Chapter 1).

3.1 Research Gap

Key to the adaptation logic in the self-adaptive software systems is the *knowledge* component, which contains an architectural model that represents the underlying system and certain user objectives (recall Section 2.1).

Prior research on self-managing systems assume that the model is developed by a software architect or already exists [142, 163, 164, 171, 197]. This assumption does not hold in an Android system, since the software architecture of the system is not known ahead of time. In such a system, a user can add, update, or remove apps while the system is running. Hence, the space of possible software architectures is infinite.

Moreover, as illustrated in Figure 3.1, the current security mechanisms for Android apps, both static and dynamic analysis approaches, are insufficient for detecting and preventing the increasingly dynamic and sophisticated security attacks.

Static analysis approaches [144, 215, 140, 151, 71, 221, 89, 161, 111, 104, 143, 72, 184], see Figure 3.1, suffer from false positives, i.e., false alarms. The high number of false alarms generated by such approaches lower their applicability. Moreover, static analysis approaches face severe limitations when it comes to analyzing obfuscated or dynamically loaded code [173], thus in practice also suffer from false negatives. Precise forms of static analysis also require significant amounts of computing resources and can take a substantial amount of time to execute.

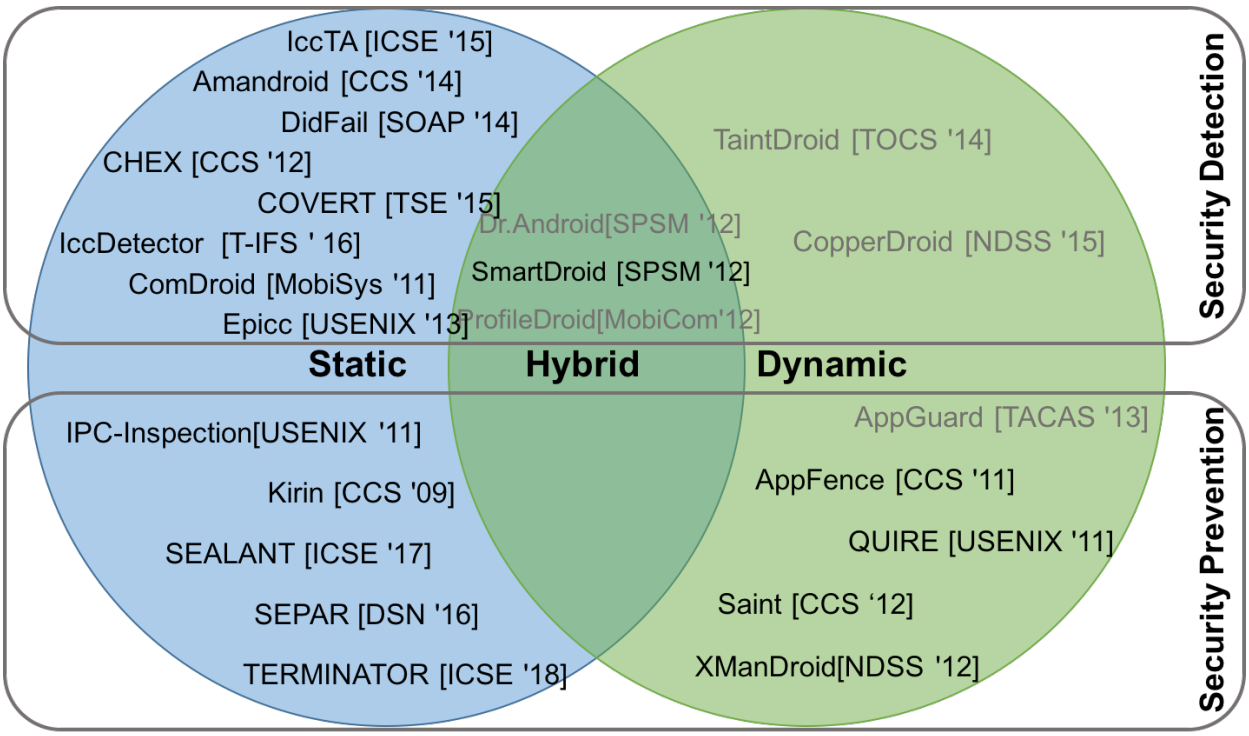


Figure 3.1: The state of the current security mechanisms for Android apps. (Tools with gray color means they do not detect or prevent security attacks instead they monitor the running system, i.e., profiling tools.)

Dynamic analysis approaches [103, 206, 68, 127, 100, 162, 82], see Figure 3.1, are not sound, and are thus prone to false negatives. They require execution of test cases to understand the behavior of an app under analysis. Since the provided test cases are likely to be incomplete, parts of the app's behavior are not discovered. These approaches are susceptible to a variety of anti-debugging and anti-monitoring defenses [190, 172, 176, 210, 178, 64, 136, 114, 97, 90] as well as *time bombs* or *logic bombs* [95], which further decrease their efficacy. Furthermore, dynamic approaches are tedious and time consuming, as exhaustive execution of apps can take a substantial amount of time.

To overcome the limitation of pure static or pure dynamic analysis, Holla and Katti [126] discussed the need for hybrid approaches to protect Android systems. Despite that, few approaches proposed hybrid techniques such as Dr. Android [134], SmartDroid [228], and ProfileDroid [216]. Nevertheless, these tools provide detection capabilities but not prevention

mechanisms. Moreover, they require changes to apps' implementation logic which prevent their practical use.

All of these approaches perform complete analysis of Android systems, and hence lack the ability to efficiently analyze systems as changes occur—such as adding/removing apps, granting/revoking permissions at runtime, or dynamically loading code.

3.2 Problem Statement

The problem caused by the over-privileged nature of components in Android apps due to the current Android privilege management scheme can be summarized as follows:

> Components in Android apps are over-privileged and violate the least-privilege security principle which leads to many types of Inter-Component Communication attacks.

3.3 Thesis Statement

My thesis statement can be summarize as follows:

> The goal of my research is to mitigate inter-component communication security attacks in Android through an automated approach that determines and enforces the LP architecture of an Android system.

Every time the system changes, the self-protecting approach (1) reflects the changes on the maintained *least-privilege* architecture of the system, (2) incrementally and efficiently

analyzes the security posture of the system for potential ICC attacks, and (3) dynamically enforces the least-privilege architecture to prevent security attacks at runtime. Thereby, it ensures the system remains safe and protected at all times.

3.4 Research Hypotheses

This research entails investigating the following research hypotheses:

- The systematic violation of the least-privilege security principle increases the attack surface of an Android system. In such a system, if a component is compromised, the impact might be severe due to the extra privileges that compromised component has. If we were able to determine the least-privilege architecture of an Android system, we could reduce its attack surface and aid in comprehending its security posture. A least-privilege architecture is one in which the components are only granted the privileges that they require for providing their functionality. Determining the least-privilege architecture of an Android system requires determining the exact privileges each component needs to fulfill its task. One solution to this problem is to rely on apps' developers to specify the required privileges for each component. Indeed, such a solution is error prone and labor intensive. However, since we have the implementation logic of these apps in their bytecode, it is plausible to determines the exact privileges each component needs from its implementation logic without human intervention.

Hypothesis 1: An automated approach for determination of least-privilege architecture in Android can be developed to reduce the attack surface without the need to change the implementation logic of Android apps.

- After determining the least-privilege architecture of an Android system, the Android runtime environment needs to be modified to enforce the determined architecture. If a component tries to obtain a privilege that is not specified in the least-privilege architecture, the modified Android runtime environment should prevent it. Such an enforcement of the least-privilege architecture would prevent many ICC attacks facilitated by the extra privileges the components in Android apps currently have.

> **Hypothesis 2:** By enforcing the least-privilege architecture of an Android system, it is possible to prevent inter-component communication security attacks.

- Due to the complex and dynamic nature of Android systems (e.g., adding a new app, removing an existing app, granting/revoking a permission, and dynamic class loading), their architecture and also their security posture continuously change over time. Simply determining the least-privilege architecture at design time and enforcing it at runtime, is not practical since the determined architecture will become an obsolete representation of the system as the system evolves over time. Moreover, repeating the entire security analysis of an Android system, either statically or dynamically, every time the system changes is prohibitively expensive for practical use.

> **Hypothesis 3:** An efficient approach for dynamic monitoring along with incremental security analysis for the least-privilege architecture of an evolving Android system can be devised to keep the system protected against inter-component communication security attacks in spite of changes at runtime.

Chapter 4

The Effects of Code Obfuscations on Android Apps and Anti-Malware Products

The Android platform has been the dominant mobile platform in recent years resulting in millions of apps and security threats against those apps. Anti-malware products aim to protect smartphone users from these threats, especially from malicious apps. However, malware authors use code obfuscation on their apps to evade detection by anti-malware products.

To assess the effects of code obfuscation on Android apps and anti-malware products, we have conducted a large-scale empirical study that evaluates the effectiveness of the top anti-malware products against various obfuscation tools and strategies. To that end, we have obfuscated 3,000 benign apps and 3,000 malicious apps and generated 73,362 obfuscated apps using 29 obfuscation strategies from 7 open-source, academic, and commercial obfuscation tools.

The findings of our study indicate that (1) code obfuscation significantly impacts Android anti-malware products; (2) the majority of anti-malware products are severely impacted by even trivial obfuscations; (3) in general, combined obfuscation strategies do not successfully evade anti-malware products more than individual strategies; (4) the detection of anti-malware products depend not only on the applied obfuscation strategy but also on the leveraged obfuscation tool; (5) anti-malware products are slow to adopt signatures of malicious apps; and (6) code obfuscation often results in changes to an app's semantic behaviors.

4.1 Introduction

Android is the dominant mobile platform holding 85% of the smartphone OS market share [26]. At the same time, the number and sophistication of malicious Android apps are increasing. For instance, McAfee Labs crawled several app stores over six months in 2016 and detected more than 9 million malicious apps [25]. As another example, Kaspersky discovered more than 4 million new malware in 2016 [24].

Many reasons contribute to this meteoric rise of malware apps including: (1) the relative ease of creating a piggybacked app [232, 146, 145, 147, 148], i.e., a mutated version of a legitimate app injected with either malicious code or embedded advertisements; and (2) the prevalence of alternative Android app stores (i.e., app stores other than the official Android app store, Google Play [35]), on which malicious apps may be distributed to users.

To protect mobile devices, users often rely on anti-malware products, which scan apps to determine if they are benign or malicious. However, many malware apps have previously evaded detection by these products. Examples of such malicious apps include Brain Test [22], VikingHorde [27], FalseGuide [34], and DressCode [23]. These apps have infected millions

of users before they were detected. To evade detection, malware authors often rely on *code obfuscation*.

Code obfuscation transforms code into a form that is more difficult for humans, and possibly machines, to read, understand, and reverse engineer. These transformations change the syntax of code but not their semantics [93]. These changes could be small (e.g., inserting unused code) or sophisticated (e.g., performing bytecode encryption)[180]. Although code obfuscations are used by malware authors, they are also used by benign app developers to increase the difficulty of reverse engineering their apps.

To better protect the intellectual property of benign app developers and prevent cloning of their apps, several companies have developed obfuscation tools, or *obfuscators* for short, that implement different code transformations (e.g., identifier renaming, string encryption, reflection, etc.). Given the use of obfuscations by malware authors, the goal of this study is to assess the performance of commercial anti-malware products against various obfuscation tools and strategies.

Although some researchers have studied an individual obfuscation tool's effectiveness on a limited number of anti-malware products [154, 175, 180, 229, 108, 128], no study has performed a large-scale assessment of (1) the effect of individual and combined obfuscation strategies provided by multiple obfuscations tools on anti-malware products, (2) the effect the tools and strategies have on the accuracy of anti-malware products for benign apps and not just malicious apps, (3) the effect of time on obfuscated-app detection by those products, and (4) whether the application of obfuscation strategies result in valid, installable, and runnable apps. Due to the lack of a study regarding the effect of specific and combined obfuscation strategies on anti-malware products, it is unclear which strategies evade such products the most. None of the aforementioned studies determine the extent to which anti-malware products erroneously consider obfuscated, benign apps as malicious, which is undesirable for both anti-malware product vendors and benign app developers.

To determine if the transformations applied by obfuscation tools break an app's semantics, our study investigates the ability of obfuscation tools to generate valid, installable, and runnable apps. An obfuscated app is not useful to a benign app developer or malicious author if it cannot be executed on a device or if its benign, functional behavior changes. To ensure an app's obfuscation is successful, our study further compares the behavior of an obfuscated app with the behavior of its corresponding original app.

Overall, this chapter makes the following contributions:

- We assess the accuracy of over 60 anti-malware products on apps obfuscated using 7 obfuscation tools and 29 obfuscation strategies on 3,000 benign apps and 3,000 malicious apps, totaling over 73,000 obfuscated apps. We further consider the effect of an app's age on that accuracy.
- We evaluate the ability of 7 obfuscation tools to generate Android apps that are valid, installable, and runnable.
- Based on our results, we make suggestions for improving anti-malware products and obfuscation tools.
- To conduct this study, we have implemented a framework for obfuscating Android apps and scanning them using anti-malware products. The framework is reusable, can be extended to include more obfuscation tools and strategies, and is available online [37], along with our resulting dataset of over 73,000 obfuscated apps.

The remainder of this chapter is organized as follows. Section 4.2 covers background information about reverse engineering Android apps and code obfuscation. Section 4.3 discusses the research questions that this study aims to answer. The research methodology of this study is presented in Section 4.4. The results and the findings are reported in Section 4.5. Section 4.6 discusses the results and provides recommendations to enhance anti-malware products and obfuscation tools. Finally, the threats to validity are presented in Section 4.7.

4.2 Background

This section provides a brief overview of reverse engineering Android apps and obfuscation strategies to help the reader understand the rest of the chapter.

4.2.1 Reverse Engineering Android Apps

classes.dex, the main DEX file of an Android app (recall Section 2.2), is a file in the APK generated by *dx*, a utility that converts *.class* files into a DEX file. *classes.dex* can be disassembled by Baksmali [39] into an Intermediate Representation (IR) format which, in turn, can be assembled by Smali [39] to generate a new variant of *classes.dex*. The new *classes.dex* can be repackaged using a tool such as *Apktool* [14], a reverse-engineering tool for Android APK files, to generate a new APK variant (e.g., an obfuscated app).

Different IR formats can be generated from *classes.dex*, including Smali code using Apktool and *.class* files using DARE [159] or dex2jar [32]. Moreover, Soot [208] can generate various IRs such as Baf, Jimple, Shimple, Grimp, or even a low-level IR such as Jasmin.

4.2.2 Obfuscation Strategies

To study the effectiveness of anti-malware products, we applied several different obfuscation strategies on each Android app. We use the term *obfuscation strategy* to refer to a single transformation or multiple transformations applied to an Android app. We consider three types of strategies: *trivial strategies*, *non-trivial strategies*, and *combined strategies*. Table 4.1 presents abbreviations of the trivial and non-trivial obfuscations, which will be used throughout this chapter.

Table 4.1: Obfuscation-strategy abbreviations

Trivial Obfuscation		Non-trivial Obfuscation			
Disassembling/Reassembling	DR	Junk code insertion	JUNK	Identifier renaming	IDR
AndroidManifest transformation	MAN	Class renaming	CR	Control flow	CF
Alignment	ALIGN	Member reordering	MR	Reflection	REF
Repackaging	REPACK	String encryption	ENC		

Trivial obfuscation strategies are code transformations that do not change the app's bytecode. For this study, we examined the following trivial strategies:

- Repackaging (REPACK) involves unzipping the APK file and re-signing it with a different signing certificate. This simple transformation thwarts anti-malware products that rely on the app's certificate to determine if the app is malicious or not. For this transformation, we unzip an APK file using the *zipfile* Python library and resign it with our own signing certificate using *jarsigner* [36], a tool for verifying and generating digital signatures for JAR files.

- Disassembling and Reassembling (DR) involves disassembling the app using a reverse-engineering tool, such as Apktool, reassembling the app, and then signing it. By disassembling and reassembling the app, the items in *classes.dex* will be reordered. Anti-malware products that rely on matching *classes.dex* against signatures of known malicious apps would be broken.

- AndroidManifest transformation (MAN): Each Android app contains a configuration file called *AndroidManifest.xml* file, which specifies the principal components that constitute the application, including their types and capabilities, as well as required and enforced permissions. This transformation changes the manifest by adding permissions or adding components' capabilities, called *Intent Filters* in Android.

- Alignment (ALIGN) realigns all uncompressed data, such as images or raw files, in an APK file. This transformation changes the cryptographic hash of an APK file. Therefore,

if an anti-malware product identifies malicious apps based on their cryptographic hashes (e.g., MD5), this transformation can thwart it.

Non-trivial obfuscation strategies are code transformations that change an app's bytecode. We study the following non-trivial obfuscation strategies:

- Junk code insertion (JUNK) adds code that does not affect the execution of an app. Junk code insertion can add null operations (`nop`), comments, and/or debugging information to a *classes.dex* file.
- String encryption (ENC) encrypts the strings in *classes.dex* and adds a function that decrypts the encrypted strings at runtime. Anti-malware products that rely on the string data in an app to determine if it is malicious will be evaded by this transformation.
- Control-flow manipulation (CF) changes the methods' control-flow graph by adding conditions and iterative constructs. In addition, this transformation changes the app's call graph by adding new methods and fake calls to the newly added methods.
- Members reordering (MR) changes the order of instance variables or methods in a *classes.dex* file, which evades anti-malware products that depend on the sequence of members in a class.
- Identifier Renaming (IDR) renames the instance variables and/or the method names in each Java class with randomly generated names. This transformation changes signatures generated from identifiers and changes the *method table* in Dalvik bytecode.
- Class renaming (CR) renames the classes and/or the packages in an app with randomly generated names. This transformation changes the *method table* in the Dalvik bytecode.
- Reflection (REF) transformations convert direct method invocations into reflective calls using the Java reflection API, which can evade static analyses that rely on direct method calls.

Combined strategies are combinations of the aforementioned obfuscation strategies. Previous work [180, 154] has mentioned that combining obfuscation strategies result in stronger obfuscations. Our study leverages different combined strategies to understand which combinations of transformations will result in better evasion of anti-malware products. The majority of our leveraged combined strategies have not been empirically studied in previous work.

4.3 Research Questions

In this chapter, our primary goal is to provide a large-scale empirical study that evaluates the effectiveness of anti-malware products against various obfuscation tools and strategies. To that end, this section presents and discusses the research questions this study attempts to answer. Moreover, this section shows who will benefit from answering each research question. In the remainder of this section, we introduce and motivate each research question that we study.

RQ1. How is the accuracy of anti-malware products affected by obfuscation strategies?

The use of code obfuscation in Android apps has become popular and is leveraged by both benign and malicious app developers. Given that smartphone users rely on anti-malware products to protect their devices, it is crucial for anti-malware products to distinguish malicious apps from benign ones with high accuracy, while being resilient to obfuscation. RQ1 aims to measure the accuracy of commercial anti-malware products against a broad range of obfuscation strategies. We measure accuracy in this chapter using precision and recall, since these metrics take into account false positives (i.e., benign apps marked as malicious) and false negatives (i.e., malicious apps marked as benign).

Anti-malware providers will benefit from answers to RQ1 in order to improve their products, especially against the obfuscation strategies that thwart their products the most. In addition,

the answers to RQ1 can help smartphone users choose between anti-malware products. Benign app developers will benefit from answers to RQ1 by learning which obfuscation strategies prevent their apps from being flagged as malicious.

RQ2. How is the accuracy of anti-malware products affected by obfuscation tools?

Each anti-malware product's effectiveness likely varies based on the implementations of obfuscation strategies provided by an obfuscation tool. To make that determination, RQ2 measures the accuracy of anti-malware products on obfuscation tools, where accuracy is again measured in terms of precision and recall.

Anti-malware product vendors, benign app developers, and obfuscation tool developers can benefit in several ways from the answers to RQ2. Anti-malware product vendors can use this information to determine which specific implementations of obfuscation strategies may cause false positives (i.e., benign apps marked as malicious) in their products. Similarly, these vendors can benefit from learning which obfuscations result in successful evasion from detection by malicious apps. Answers to RQ2 can aid benign app developers in choosing the obfuscation tools that will prevent their apps from erroneously being marked as malicious. Furthermore, if false positives or false negatives (i.e., malicious apps marked as benign) are due to obfuscation tools, as opposed to the anti-malware products, then this information is useful for correcting obfuscation tools.

RQ3. How is the accuracy of anti-malware products affected by the year an app is created?

RQ3 aims to study the accuracy of anti-malware products on non-transformed and transformed apps over different time periods, where each time period for our study spans two years. We consider transformed apps as belonging to the same time period as their non-transformed versions. For example, if we transform apps created in time period 2012-2013, we still consider the resulting obfuscated apps as created in 2012-2013, for the purposes of RQ3.

This research question allows us to understand the effectiveness of anti-malware products when applied to different time periods and to determine if those products' detection accuracies are affected by time. Anti-malware vendors can use this information to determine the time periods that result in poor accuracy for their products, aiding them with improving results for apps created during those problematic time periods.

RQ4. To what extent do obfuscation tools result in valid, installable, and runnable apps?

Although an obfuscated app does not need to be runnable when scanned by an anti-malware product, developers of benign apps and obfuscation tools rely on those tools to produce valid, installable, and runnable apps. Similar to [128], we consider an APK to be *valid* if an obfuscation tool successfully generates a signed APK package that includes a *classes.dex* file containing correct Dalvik bytecode syntax. An app is *installable* if it can be successfully deployed into the Android runtime. For our purposes, a transformed app is *runnable* if its runtime behavior is similar to its non-transformed version. RQ4 is particularly useful for obfuscation tool developers since answers to that question provide information about transformations that result in malformed apps.

4.4 Research Methodology

This section describes the research methodology that we pursued in terms of our study subjects, selected obfuscation tools, our evaluation framework, and our selected anti-malware products.

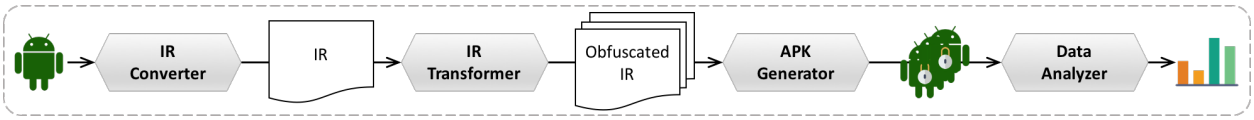


Figure 4.1: Obfuscation study methodology

4.4.1 Study Subjects

We used a dataset of benign apps consisting of 3,000 apps from Google Play and 3,000 malicious apps. To avoid having malicious apps in our ground-truth dataset of benign apps, we obtained benign apps from AndroZoo [57], which is a collection of more than 5.5 million apps collected from several sources, including Google Play. AndroZoo apps have been scanned by commercial anti-malware products using the VirusTotal service [17], a free online service provided by Google that scans URLs, files, and Android apps. Approximately 25,000 Google Play apps out of nearly 2 million apps in AndroZoo are marked as benign by all anti-malware products. From these 25,000 apps, we have randomly selected 3,000 apps for this study. The malicious apps belong to several malware repositories including Android Malware Genome [233], Contagio [20], AndroTotal [152], the Drebin dataset [62] and VirusShare [18]. In addition to these malware repositories, we used the VirusTotal service to include recently discovered malicious apps that belong to the following malware families: BrainTest [22], VikingHorde [27], and FalseGuide [34].

4.4.2 Obfuscation Tools

We have included the following obfuscation tools for our study, whose supported obfuscation strategies are depicted in Table 4.2:

- Allatori [16] is a commercial Java and Android obfuscation tool that supports a wide range of obfuscation strategies. Many companies such as Amazon, Fujitsu, and Motorola rely on Allatori to protect their software systems from being reverse engineered. The

Table 4.2: Obfuscation strategies of each obfuscation tool

	Trivial				Non-trivial						
Obfuscator/Strategy	ALIGN	DR	MAN	REPACK	CF	CR	ENC	IDR	JUNK	MR	REF
Apktool/Jarsigner		✓		✓							
Allatori					✓		✓	✓		✓	
DashO					✓		✓	✓			
DroidChameleon			✓		✓	✓	✓		✓		✓
ADAM	✓				✓		✓	✓	✓		
ProGuard								✓			

providers of this tool, Smardec Inc., provided us with a full version for educational purposes.

- ProGuard [38] is a widely used open-source shrinker, optimizer, obfuscator, and preverifier for Java bytecode. A preverifier performs certain checks on Java bytecode prior to runtime. ProGuard supports identifier renaming and is the default tool in many development environments, including Android Studio [30], the official IDE for Android apps.

- ADAM [229] is a research tool for obfuscating Android apps. It transforms the Smali code of a reversed-engineered app.

- DroidChameleon [180] is a state-of-the-art research tool for obfuscating Android apps which supports a wide range of obfuscation strategies. Compared to ADAM, DroidChameleon supports more complex transformations. Like ADAM, DroidChameleon transforms the Smali code of a reversed-engineered app.

- DashO [31] is a commercial tool for obfuscating Android and Java applications. DashO provides static analysis protection and runtime security control against tampering, unauthorized debugging, and some runtime attack patterns. This tool supports control-

flow, string-encryption, and identifier-renaming transformations. The providers of DashO, PreEmptive Solutions, supplied us with a full free version valid for 30 days.

- Apktool and Jarsigner were used to perform the DR and REPACK obfuscation strategies, respectively. These two transformations often work in tandem because a reassembled APK must be resigned.

We also considered another tool, DexGuard [33], which is an advanced and commercial version of ProGuard. We contacted the providers of DexGuard to obtain an educational or commercial version of their tool to run on our dataset. Unfortunately, they only allow their tool to run on a restricted number of Android apps; and they do not sell licenses for research purposes. Hence, we did not include it in this study.

4.4.3 Evaluation Framework

To conduct our study, we have developed the framework depicted in Figure 4.1, which consists of the following four modules: *IR Converter*, *IR Transformer*, *APK Generator*, and *Data Analyzer*. *IR Converter* takes an Android APK as input and converts its code to Intermediate Representation (IR) formats. *IR Transformer* utilizes all obfuscation tools to transform the IR format using a variety of obfuscation strategies. *APK Generator* repackages each obfuscated IR file and generates an obfuscated APK from that file. *Data Analyzer* scans obfuscated apps using anti-malware products, stores the scanning results in a MySQL database, analyzes the scanning results, and creates various statistical reports.

Our framework is reusable and extendable. A user can add new obfuscation tools and support different obfuscation strategies. Therefore, we make the framework available for researchers and practitioners [37]. The framework is a Python program that consists of more than 5,500 lines of code, not counting the obfuscation tools.

IR Converter. Obfuscation tools do not require source code and they work directly on the IR format. Therefore, this module converts an APK file to two IR formats: *smali* using Baksmali and Java bytecode using *dex2jar*. In our framework, we generate these two IR formats since ADAM and DroidChameleon work on *smali* code while all other obfuscation tools work on Java bytecode.

IR Transformer. This module generates several obfuscated IR files of the original IR file. The framework is configured to leverage twenty nine different obfuscation strategies using seven obfuscation tools (recall Section 4.4.2).

APK Generator. For each obfuscated IR file, this module generates an obfuscated Android app. First, this module leverages the *dx* tool from the Android SDK to convert an obfuscated IR to a *classes.dex* file. Next, it generates an APK file with the new *classes.dex* using Apktool. Finally, the APK file is signed using *jarsigner* with our own certificate, since the original certificate of the app cannot be obtained.

Data Analyzer. This module uses the VirusTotal service to scan apps using anti-malware products. This module uploads the apps to VirusTotal, which scans them using more than 60 up-to-date commercial anti-malware products. For each uploaded app, VirusTotal returns a unique scanning ID, which Data Analyzer uses later to download the scanning reports and stores them in a MySQL database. Data Analyzer queries and processes the database to generate various statistical reports.

4.4.4 Anti-malware Products

We have evaluated the accuracy and the resiliency of 61 commercial anti-malware products against obfuscations. Due to space limitations and to ensure readability, we focus on the

results of the 21 most popular Android anti-malware products in this chapter; however, we make the results for all 61 anti-malware products available online [37].

Table 4.3 shows the anti-malware products evaluated in this study and includes the following information for each product: its number of *Downloads*; its overall user satisfaction score as represented using a star-rating (*Stars*); and the number of users who reviewed the product (*Reviewers*). The numbers in Table 4.3 are obtained from Google Play.

Table 4.3: Anti-malware products (K: Thousand. M: Million)

Product	**Downloads**	**Stars**	**Reviews**	**Product**	**Downloads**	**Stars**	**Reviews**
Ikarus	100K - 500K	4.2	2,862	Trustlook	10M - 50M	4.4	476,671
Emsisoft	100K - 500K	4.2	1,425	McAfee	10M - 50M	4.4	506,491
Fortinet	100K - 500K	4.2	2,086	Avira	10M - 50M	4.5	441,016
AegisLab	100K - 500K	4.2	2,905	Norton	10M - 50M	4.5	946,230
F-Secure	500K - 1M	4.1	12,183	Symantec			
Comodo	500K - 1M	4.6	33,395	ESET-NOD32	10M - 50M	4.7	490,840
GData	1M - 5M	4.0	8,850	Kaspersky	10M - 50M	4.7	2,061,983
Sophos	1M - 5M	4.3	11,816	DrWeb	50M - 100M	4.5	1,044,410
TrendMicro	1M - 5M	4.6	49,977	Antiy-AVL	100M - 500M	4.1	2,166
BitDefender	5M - 10M	4.5	88,809	Avast	100M - 500M	4.5	4,724,478
CAT-QuickHeal	5M - 10M	4.4	204,709	AVG	100M - 500M	4.5	5,785,171

4.5 Data Analysis and Results

For conducting our experiments, we have leveraged a high performance computing cluster (HPC), managed by our organization, that has more than 200 compute nodes with a total of more than 8,000 cores. Each compute node has 264GB-512GB RAM. We utilized HPC to run thousands of jobs simultaneously. On each app, we applied 29 different obfuscation strategies: 4 trivial transformations, 7 non-trivial transformations, and 18 combined transformations. Table 4.4 shows the number of obfuscated apps resulting from applying the 29 obfuscation strategies leveraged by the obfuscation tools. An empty cell indicates an obfuscation strategy

that is not support by a particular obfuscation tool. In total, we have generated 73,362 obfuscated apps from 3,000 benign apps and 3,000 malicious apps.

Table 4.4: Number of obfuscated apps using the obfuscation strategy in the column leveraged by the obfuscator in the row.

Obfuscator/Strategy	Trivial				Non-trivial							Combined Strategies																			Total apps
	ALIGN	DR	MAN	REPACK	CF	CR	ENC	IDR	JUNK	MR	REF	CF_ENC	CF_IDR	CF_MAN	CF_MR	CR_MAN	ENC_IDR	ENC_MAN	JUNK_MAN	MAN_REF	CF_CR_MAN	CF_ENC_IDR	CF_ENC_MR	CF_IDR_MR	CF_REF_MAN	ENC_REF_MAN	CF_ENC_IDR_MR	CF_CR_ENC_MAN_REF	CF_CR_ENC_JUNK_MAN_REF		
Simple Tools		3,693		5,880																											9,573
Allatori					1,612		1,613	1,609		1,612					1,609								1,607	1,607			1,606				12,875
DashO					1,094		1,089	1,097				1,082	1,084				1,077					1,083										7,606
DroidChameleon			3,597		2,593	1,952	687		351		1,487	610		1,594		1,752		679	349	1,385	1,361				993	658		570	314		20,932
ADAM	5,487				0	4,175	2,708	4,119	4,182																						20,671
ProGuard								1,705																							1,705
Total apps	5,487	3,693	3,597	5,880	5,299	6,127	6,097	8,530	4,533	1,612	1,487	1,692	1,084	1,594	1,609	1,752	1,077	679	349	1,385	1,361	1,083	1,607	1,607	993	658	1,606	570	314		73,362

In the remainder of this section, we present the results of our experiments. We measured the effectiveness of anti-malware products at identifying malicious apps in terms of their *precision*, which measures the extent to which benign apps are labeled as malicious, and *recall*, which measures the extent to which malicious apps are labeled as benign. We use the *F-score*, i.e., the harmonic mean of precision and recall, to measure the overall detection rate of anti-malware products.

4.5.1 RQ1. Obfuscation Strategies

We studied the accuracy of anti-malware products with respect to a wide variety obfuscation strategies in two scenarios. In the first scenario (Section 4.5.1), we compare the detection rates of each anti-malware product on the original dataset and the obfuscated dataset. In the second scenario (Section 4.5.1), we measure the detection rate of anti-malware products against each obfuscation strategy.

Detection rate on original and obfuscated apps

Figure 4.2 shows the detection rate of 21 anti-malware products on the original dataset of 6,000 apps, depicted as black bars, and the obfuscated dataset of 73,362 apps, depicted as gray bars. Figure 4.2 demonstrates that the detection rate of anti-malware products on the original dataset is above 85% for 16 products, and between 75% and 85% for 4 anti-malware products. *TrendMicro* exhibits the lowest detection rate, 56%. The average detection rate is 87% on the original dataset. Consequently, prior to application of obfuscation strategies from obfuscation tools, these top anti-malware products are quite effective at protecting Android users; albeit there is room for improvement.

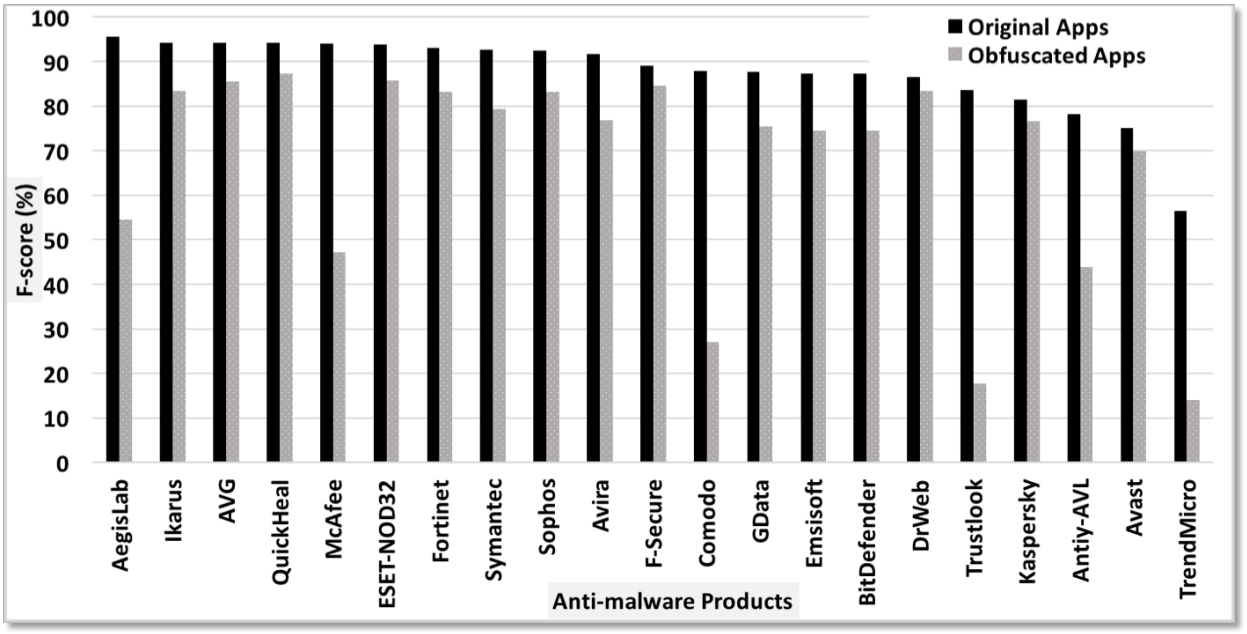


Figure 4.2: Detection rate of 21 anti-malware products on 6,000 original apps and 73,362 obfuscated apps.

Once obfuscation strategies are applied, the detection rates for those anti-malware products decrease significantly, as shown in Figure 4.2. For example, *AegisLab* achieves the highest detection rate on the original dataset, 96%, since it mislabeled only 247 apps in the original dataset. Its detection rate has dropped to 55% on the obfuscated dataset—a 40% decrease—as it mislabeled 27,636 apps. Other anti-malware products are also severely impacted by code obfuscation. While the average detection rate of anti-malware products on the original apps is 87%, the average detection rate on the obfuscated dataset is 67%—a 20% decrease.

Finding 1: Code obfuscation significantly impacts Android anti-malware products. The average detection rate for the top anti-malware products decreases from 87% to 67%—a 20% decrease.

Detection rate for each obfuscation strategy

To better understand the impact of every obfuscation strategy on each anti-malware product, Table 4.5 presents the detection rate of anti-malware products, expressed as the F-score, on the original dataset and the various obfuscation strategies. For example, the detection rate of *Symantec* on the original dataset is 93%. This detection rate has dropped to 64% on obfuscated apps using MAN, to 69% on obfuscated apps using ENC, and to 31% on obfuscated apps using ENC_IDR.

Table 4.5 demonstrates that the majority of anti-malware products are not affected by REF. We consider a transformation's effect on a product's detection rate to be negligible if the detection rate has either improved, decreased by less than 3%, or remains above 85%. In fact, the accuracy of *F-Secure*, *GData*, *BitDefender*, and *Emsisoft* improves on apps obfuscated using REF. This result indicates that the intensive use of *code reflection* makes an app look suspicious to our studied anti-malware products, improving their detection rates. Unfortunately, this phenomenon may result in false positives for certain anti-malware products. For instance, *AVG* erroneously marked 307 benign apps obfuscated using REF as malicious, while also correctly detecting nearly all malicious apps obfuscated using REF.

Finding 2: REF transformations make apps look suspicious, increasing the chance of an app being labeled as malicious.

Perhaps most surprising is that certain trivial obfuscation strategies are quite effective against the top anti-malware products. Notably, the anti-malware products that we studied rely heavily on analyzing an app's manifest file, which contains configuration information. Consequently, these products are often evaded by apps obfuscated using MAN, which involves the trivial addition or modification of permissions or Intent filters. For example, the detection

Table 4.5: (RQ1) Detection rate of anti-malware products, measured by their F-score (%), against each obfuscation strategy.

		Trivial				Non-trivial							Combined Strategies																	
Anti-malware	**Original**	**ALIGN**	**DR**	**MAN**	**REPACK**	**CF**	**CR**	**ENC**	**IDR**	**JUNK**	**MR**	**REF**	**CF_ENC**	**CF_IDR**	**CF_MAN**	**CF_MR**	**CR_MAN**	**ENC_IDR**	**ENC_MAN**	**JUNK_MAN**	**MAN_REF**	**CF_CR_MAN**	**CF_ENC_IDR**	**CF_ENC_MR**	**CF_IDR_MR**	**CF_REF_MAN**	**ENC_REF_MAN**	**CF_ENC_IDR_MR**	**CF_CR_ENC_MAN_REF**	**CF_CR_ENC_JUNK_MAN_REF**
AegisLab	96	84	52	36	92	52	35	53	39	62	66	58	41	11	57	66	37	3	68	61	58	43	4	65	66	66	68	63	58	35
Ikarus	94	95	94	66	96	75	84	81	86	86	93	88	75	77	84	88	85	64	89	86	89	86	60	84	87	93	89	82	92	87
CAT-QuickHeal	94	95	93	92	94	89	**91**	75	88	89	93	94	73	**92**	93	93	91	55	91	90	94	91	54	76	91	94	84	**70**	89	80
AVG	94	75	96	63	96	79	83	85	91	84	97	88	77	**77**	83	97	85	58	90	85	89	85	57	92	96	92	90	**92**	91	85
McAfee	94	90	50	21	93	47	20	45	52	16	73	23	20	41	17	73	20	21	22	20	22	17	22	66	66	15	18	63	14	20
ESET-NOD32	94	94	92	91	94	93	93	80	91	46	94	66	75	92	92	94	93	68	86	46	66	91	61	80	92	68	59	72	80	57
Fortinet	93	94	89	86	93	88	83	78	84	75	91	86	67	79	91	88	83	58	87	77	87	84	50	83	83	89	75	72	72	56
Symantec	93	86	87	64	88	76	84	69	79	84	92	88	63	68	83	92	85	31	90	85	89	85	31	75	90	92	90	73	91	85
Sophos	93	93	91	90	93	89	85	70	79	93	88	92	70	84	93	87	86	52	91	95	92	87	50	66	72	93	91	51	91	85
Avira	92	92	87	84	92	85	84	65	78	78	87	60	61	78	86	86	85	38	80	80	59	83	38	69	85	58	33	63	73	61
F-Secure	89	87	87	85	90	85	82	81	84	95	90	94	73	65	91	90	82	53	93	94	92	84	53	87	88	93	93	84	80	71
Comodo	88	88	27	16	82	22	17	19	24	17	19	15	17	20	11	33	16	20	14	18	16	11	20	20	26	9	14	23	11	18
GData	88	91	84	75	88	79	61	75	77	95	85	91	62	54	79	85	50	46	83	79	81	50	45	79	77	81	82	77	57	41
BitDefender	87	90	84	73	88	78	60	74	75	95	85	91	61	46	76	85	45	44	83	78	78	46	43	79	77	77	83	77	58	41
Emsisoft	87	90	84	73	88	78	60	74	75	95	85	91	61	46	76	85	45	43	83	78	78	46	43	79	77	77	83	77	59	41
DrWeb	87	88	83	81	88	89	86	90	86	93	88	40	92	89	90	88	86	90	94	94	41	90	89	88	87	39	40	87	36	35
Trustlook	84	10	23	0	48	17	0	22	20	0	36	0	2	3	0	38	0	1	0	0	0	0	2	40	39	0	0	40	0	0
Kaspersky	81	83	75	70	82	81	76	70	75	88	81	80	73	77	81	81	77	50	86	89	81	81	50	70	77	83	84	64	91	85
Antiy-AVL	78	79	56	26	80	41	22	47	47	13	68	18	20	25	12	70	21	24	12	8	20	12	25	70	69	12	15	65	8	7
Avast	75	75	63	57	75	73	66	66	66	78	74	75	71	67	78	74	68	46	91	79	76	76	45	60	69	83	91	47	91	86
TrendMicro	56	57	11	7	48	12	7	10	14	6	16	10	9	15	6	16	7	11	5	7	10	5	10	12	14	7	4	12	3	5
AVERAGE	**87**	**83**	**72**	**60**	**85**	**68**	**61**	**63**	**67**	**66**	**76**	**64**	**55**	**57**	**66**	**77**	**59**	**42**	**68**	**64**	**63**	**60**	**41**	**68**	**73**	**63**	**61**	**64**	**59**	**51**

rate of *McAfee* dropped from 94% to 21% for apps obfuscated using MAN. Overall, the average detection rate of anti-malware products fell to 60% from 87% when apps are obfuscated using MAN—a 28% decrease.

Finding 3: MAN, which is a trivial obfuscation strategy, severely impacts many anti-malware products, on average, decreasing a product's detection rate by 28%.

Another interesting, possibly counter-intuitive, conclusion that we can draw from Table 4.5 is that combined transformations are not always superior to individual transformations. For

instance, while the detection rate of *AVG* against CF is 79%, its detection rate against combined transformations that include CF is between 57% and 97%.

Finding 4: In general, combined transformations do not affect detection rates more than single transformations: The average detection rate of anti-malware products is 61% for single non-trivial obfuscations, and 61% for combined obfuscations.

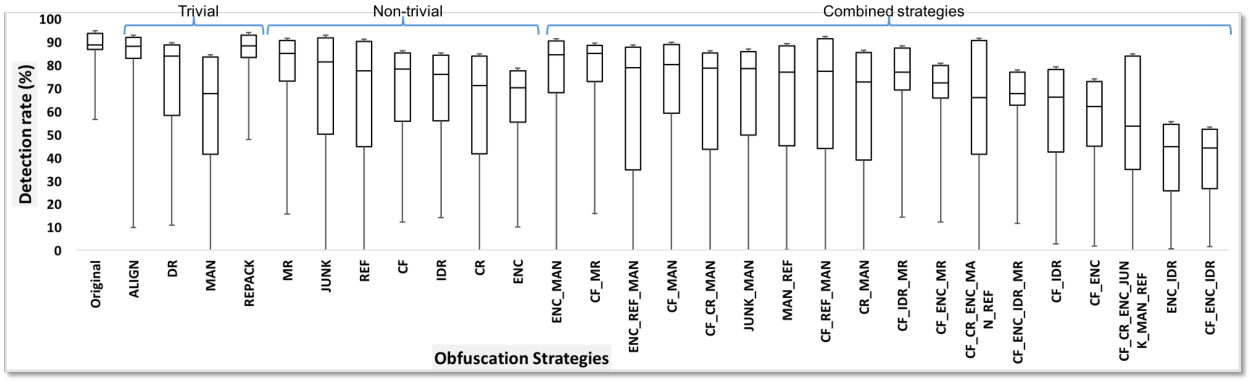


Figure 4.3: (RQ1) The average detection rate of all anti-malware products regarding each obfuscation strategy.

Figure 4.3 contains box-and-whisker plots illustrating the impact of each obfuscation strategy on all anti-malware products. These results suggest that some obfuscation strategies have negligible effects on the majority of anti-malware products. For example, REPACK did not affect 19 anti-malware products. Similarly, the use of the MR transformations did not affect 14 anti-malware products. Lastly, the REF transformation did not thwart the majority of anti-malware products.

Figure 4.3 demonstrates that ENC_IDR and CF_ENC_IDR are very effective in thwarting anti-malware products. In fact, these two transformations evaded all anti-malware products except *DrWeb*.

Finding 5: ENC_IDR and CF_ENC_IDR are the most successful transformations for evading anti-malware products.

4.5.2 RQ2. Obfuscation Tools

For RQ2, we studied the detection rate of anti-malware products on apps transformed using various obfuscation tools. To that end, we analyze the results of each anti-malware product's detection rate on each obfuscation tool. We further assess the overall effect of each obfuscation tool across all studied anti-malware products.

Table 4.6 depicts the detection rate of each anti-malware product on apps transformed using each obfuscation tool. From Table 4.6, we observe that some anti-malware products are severly impacted by all obfuscation tools. For example, the detection rate of *Trustlook* dropped to less than 40% on apps obfuscated using any of our studied tools. Furthermore, *Trustlook* marked all apps obfuscated by the following tools: DroidChameleon, ProGuard, and DashO as benign apps. Likewise, *TrendMicro* and *Comodo* are evaded by all obfuscation tools, except ADAM.

The box-and-whisker plot shown in Figure 4.4 depicts the effect of each obfuscation tool on all anti-malware products. The figure shows that the top anti-malware products are resilient against Apktool/Jarsigner, ADAM, and Allatori. At the same time, DashO evades the top anti-malware products more often than the other products.

We further assessed the variability of each obfuscation tool on our studied anti-malware products, which indicates how consistently each tool affects the accuracy of those products. To that end, we considered the interquartile range (IQR) of the box plots in Figure 4.4. IQR is the difference between the lower bound and the upper bound of a box, which conveys the

Table 4.6: Detection rate of anti-malware products (F-score (%)) against each obfuscation tool

Anti-malware	Original	ADAM	Apktool/ Jarsigner	Allatori	Droid Chameleon	ProGuard	DashO
AegisLab	96	83	79	66	53	9	13
Ikarus	94	95	95	88	82	80	71
CAT-QuickHeal	94	95	94	86	90	80	75
AVG	94	75	96	95	81	90	71
McAfee	94	90	79	68	19	37	31
ESET-NOD32	94	94	93	88	83	90	80
Fortinet	93	94	91	85	85	74	69
SymantecMobileInsight	93	86	88	86	82	72	51
Sophos	93	93	92	75	90	78	69
Avira	92	92	90	80	77	62	60
F-Secure	89	87	89	88	89	92	61
Comodo	88	88	65	25	14	23	20
GData	88	90	86	81	76	90	50
BitDefender	87	90	87	81	74	90	45
Emsisoft	87	90	86	81	74	90	45
DrWeb	87	88	86	88	77	81	89
Trustlook	84	10	37	39	0	0	2
Kaspersky	81	83	79	76	81	70	66
Antiy-AVL	78	79	72	69	16	23	25
Avast	75	75	71	67	77	60	58
TrendMicro	56	57	35	14	7	10	12
AVERAGE	87	83	80	73	63	62	51

central tendency of top anti-malware products against an obfuscation tool. A small IQR indicates that the behavior of an anti-malware product is highly consistent. Figure 4.4 shows that ProGuard has the largest IQR.

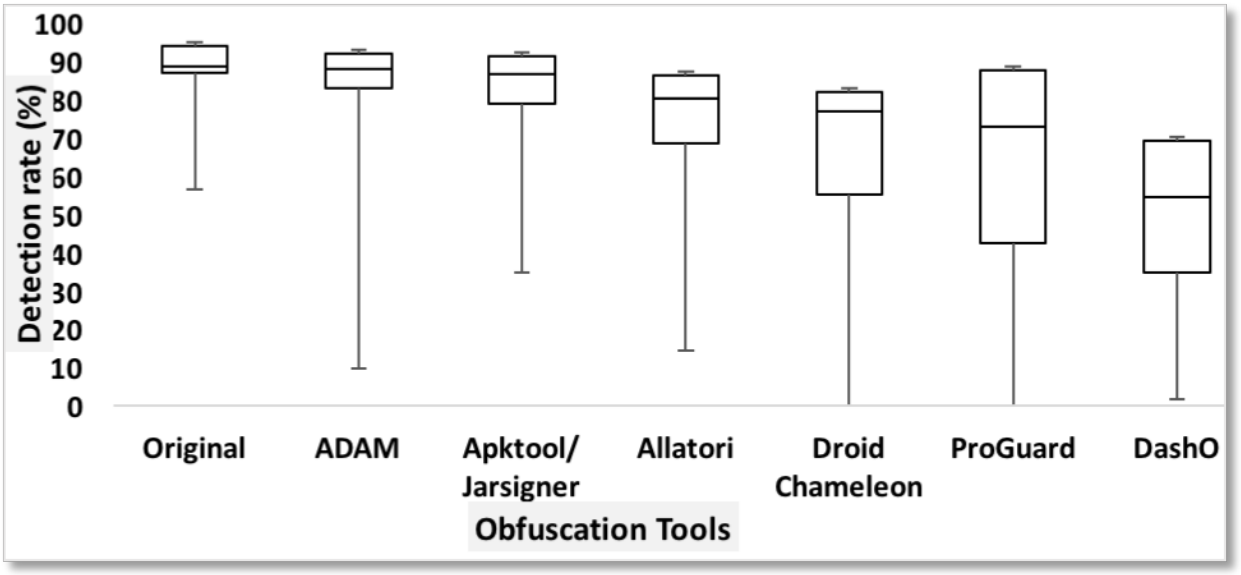


Figure 4.4: (RQ2) The average detection rate of all anti-malware products regarding each obfuscation tool.

ADAM and Apktool/Jarsigner, as shown in Figure 4.4, have a relatively high median, 88% and 86%, respectively, with the lowest IQR, i.e., 9% and 13%, respectively. This indicates that anti-malware products are resilient to these tools. Consequently, apps obfuscated by ADAM and Apktool/Jarsigner work well for benign app developers, who would want to obfuscate apps without having them be falsely reported as malicious, and would be least useful for malware authors.

Finding 6: ADAM and Apktool/Jarsigner produce obfuscations that reduce anti-malware product accuracy the least.

Figure 4.4 suggests that DashO is most successful at evading anti-malware products, which aids malware authors. The average detection rate of anti-malware products on obfuscated apps using DashO is 51% with a median of 58%—a 37% decrease in the average detection rate, and a 32% median decrease.

Finding 7: DashO reduces the accuracy of anti-malware products more than other obfuscation tools in our study.

4.5.3 RQ3. Time-Aware Analysis

A significant factor that may interact with the effect of obfuscations on anti-malware product accuracy is time. For RQ3, we conducted a time-aware analysis that studies the accuracy of anti-malware products on original and obfuscated apps that belong to the same time period for the past 10 years. Figure 4.5 depicts the results of this analysis. We grouped apps into two-year time periods, due to the fact that some years only have a few apps, mainly 2009 with 29 apps, and 2017 with 130 apps. Similar to [55], we consider the year of the last modified date of *classes.dex* in an app as the year from which it originates. We consider any transformed app as belonging to the same year as its original version, in order to determine the actual effect of obfuscation on product accuracy for each time period.

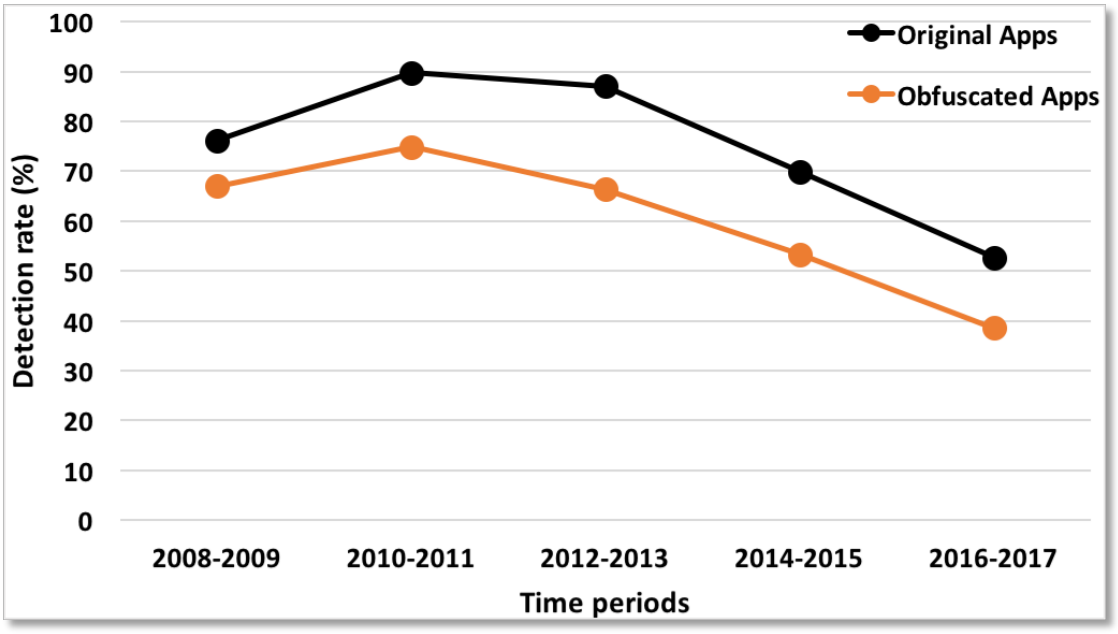


Figure 4.5: (RQ3) Time-aware analysis.

For the original dataset, Figure 4.5 shows that the top anti-malware products maintain similar detection rates for any two consecutive time periods prior to and including 2012-2013. Unfortunately, there are significant decreases in the detection rate starting from the time period between 2012-2013 and 2014-2015; the average detection rate is 70% for 2014-2015, falling from 87% in 2012-2013. By 2016-2017, the average detection rate falls to 53% on the original apps.

For the obfuscated dataset, Figure 4.5 illustrates that the detection rates of anti-malware products decrease in a largely linear fashion. The average detection rate starts at 67% on obfuscated apps from 2008-2009 and decreases to 39% on obfuscated apps from 2016-2017. These results suggest that anti-malware products are slow to adopt signatures of malicious apps.

The average detection rate on original apps from 2010 to 2013 are higher than the average detection rate on older apps. This likely occurs because many malicious apps from 2010-2013 are well-known and highly disseminated among security analysts. For example, Android Malware Genome is a dataset of malicious Android apps that are widely used, are described in a highly cited paper [233], and were released in the 2010-2013 time period.

Finding 8: The average detection rates of anti-malware products tend to decrease over time, indicating that such products are slow to adopt signatures of malicious apps.

4.5.4 RQ4. Valid, installable, and runnable apps

An obfuscated app is not useful for app authors, unless the app can run on a device. For that reason, RQ4 measures the ability of obfuscation tools to generate valid, installable, and runnable apps.

Valid apps

Recall from Section 4.3, a *valid* obfuscated app corresponds to a signed APK package that includes a *classes.dex* file containing the correct Dalvik bytecode syntax. Table 4.7 depicts the ability of obfuscation tools to generate valid obfuscated apps, which we refer to as the *obfuscation rate*. The obfuscation rate is measured using the ratio of the number of valid apps generated by an obfuscation tool to the number of apps successfully retargeted to Java bytecode. For example, ProGuard obfuscated 684 benign apps out of 1,688 successfully retargeted benign apps, resulting in an obfuscation rate of 41% for benign apps. Similarly, ProGuard obfuscated 1,021 malicious apps out of 2,005 retargeted malicious apps; hence, ProGuard's obfuscation rate on malicious apps is 51%.

Table 4.7 shows that DashO has the lowest obfuscation rate (30%) whereas DroidChameleon achieves the highest obfuscation rate (60%). There are many reasons behind these low obfuscation rates, including exceptions raised by obfuscation tools while transforming an app and their inability to produce a valid obfuscated *classes.dex* file. For instance, Allatori raised this exception "*com.allatori.IiIIIiiii: Only final fields may have an initial value!*" on many apps. We contacted the provider of Allatori about this exception, who informed us that this problem has been reported by other users, but could not be reproduced. Consequently, we helped them reproduce it to improve their product. They reported to us that this exception is mainly caused by the use of *dex2jar*, although a fix for the exception is still in progress.

Table 4.7: The ability of obfuscators to generate valid APKs.

	ProGuard	Allatori	DroidChameleon	DashO	ADAM
Benign	40.52%	25.77%	81.57%	13.39%	79.57%
Malicious	50.92%	58.70%	56.10%	43.24%	59.87%
Total	46.17%	43.65%	59.95%	29.60%	69.72%

Installable and runnable apps

To measure an obfuscation tool's ability to generate installable apps, we identified all original apps that have at least one app transformed by each obfuscation tool. For each original app, if there is more than one app transformed using the same obfuscation tool, we randomly select one of them. Using that process, we randomly selected 250 original apps along with their obfuscated versions, resulting in the selection of 1,750 obfuscated apps. We ran this experiment on a MacBook Pro with a 2.2 GHz Intel Core i7 and 16GB RAM, and installed the apps on an Android device. After we confirmed that all 250 original apps were successfully installed on the Android device, we installed the obfuscated apps.

In addition to measuring app *installability after obfuscation*, i.e., the extent to which an obfuscator can generate installable apps, we further measured app *runnability after obfuscation*, i.e., the extent to which an obfuscator can generate runnable apps. For our study, a runnable app can be order-agnostic or order-aware. A runnable app is *order-agnostic* if its obfuscated version exhibits the same *set* of running components and exceptions as its original version; a runnable app is *order-aware* if its obfuscated version exhibits the same *sequence* of running components and exceptions as its original version. To determine app runnability after obfuscation, we recorded the sequence of (1) components that execute and (2) exceptions that occur during execution of an app using *Monkey* [40], a program that generates pseudo-random streams of user events (e.g., clicks, touches, or gestures) and system-level events. We then checked for equality of the sequences of running components and exceptions of an original app and its obfuscated version, using 1,000 events for each app as input. We further ran Monkey using the same random seed for each original app and its obfuscated version, in order to test both app versions using the same sequence of inputs.

To conduct this experiment, we used the 250 original apps and the 1,341 successfully installed apps from the previous experiment, i.e., the total installed apps mentioned in Table 4.8. To

measure whether an obfuscated app runs successfully, we have modified and instrumented the Android framework [29] to include probes for monitoring the running components. We installed our modified Android framework on a Nexus 5X device.

Table 4.8 shows the ability of obfuscation tools to produce installable and runnable apps, including the following information: the total number of obfuscated apps that we *Examined* per obfuscation tool; the number of successfully *Installed* apps; the number of runnable apps that are *Order-Agnostic*; and the number of runnable apps that are *Order-Aware*.

Table 4.8: Installable and runnable apps of each obfuscator.

Obfuscator	Examined	Installed	Order-Agnostic	Order-Aware
Jarsigner	250	249	248	150
Apktool	250	249	246	154
DroidChameleon	250	249	83	31
ProGuard	250	248	237	131
Allatori	250	213	188	122
ADAM	250	84	67	46
DashO	250	49	0	0
Total Apps	**1,750**	**1,341**	**1,069**	**634**

Many obfuscation tools produce installable apps. Our results demonstrate that almost all apps transformed by Apktool/Jarsigner, DroidChameleon, and ProGuard have successfully installed. In addition, only 37 apps obfuscated by Allatori have not installed successfully. Moreover, Table 4.8 shows that most apps obfuscated using ADAM or DashO are not installable. Successfully installed apps obfuscated using ADAM all utilize the ALIGN transformation. All obfuscated apps using the non-trivial obfuscations of ADAM are not installable.

The runnability of apps obfuscated using our studied obfuscation tools varies greatly depending on the tool. Table 4.8 shows that almost all obfuscated apps using *Jarsigner* and *Apktool* are runnable in an order-agnostic fashion. 249 apps obfuscated using DroidChameleon are

installable; only 83 of those installable apps are order-agnostic and runnable; and only 31 of those installable apps are order-aware and runnable. All apps transformed by DroidChameleon using the ENC transformation are missing a function that decrypts encrypted strings, causing these apps to crash at runtime and raise the error `java.lang.NoClassDefFoundError`. All apps that become unrunnable after transformation by DashO raise the same error, i.e., `java.lang.ExceptionInInitializerError`. Given that DashO is not an open-source tool, we could not investigate this problem further. Table 4.8 shows that 95% of the installable apps generated by ProGuard are runnable. Likewise, 88% of the installable apps generated by Allatori are runnable.

Finding 9: The percentage of obfuscated apps that are both installable and runnable in an order-aware fashion with respect to component behaviors varies from 0%-62%. These results suggest a significant need for improving obfuscation tools so that applying their transformations retain an app's original behavior.

4.6 Discussion

For anti-malware product vendors, our study suggests several areas for which anti-malware products in general can be significantly improved. Recall that overall on our obfuscated dataset, anti-malware products experienced a 20% decrease in their detection rate of malicious apps compared to the original dataset (Finding 1). In particular, transformations that mainly involved identifier-name manipulation (i.e., MAN, ENC_IDR, and CF_ENC_IDR) substantially affected obfuscation tools (Findings 3 and 5). Manifest-file transformations (i.e., MAN) that simply involve addition of permissions that are not necessarily used or fake component capabilities resulted in a 28% decrease, on average, for the top-performing anti-malware products (Finding 3). Overall, these results indicate that anti-malware product vendors would

significantly benefit from performing a deeper analysis into the code of an app, potentially focusing on the security-sensitive code used by apps through static or dynamic analysis. The importance of performing deeper analysis is further highlighted by (1) the fact that transformations need not necessarily be combined to evade anti-malware products (Finding 4) and (2) this evasion worsens for newer apps (Finding 8).

For benign-app developers, our study provides some guidance as to how particular obfuscations may be used. Specifically, we have found that reflection transformations tend to increase significantly the possibility of a benign app being labeled as malicious. Therefore, benign app developers may wish to avoid such transformations to avoid this false labeling, and to reduce overhead exhibited by reflection. In the general case, benign-app developers need not be overly concerned about their apps being falsely labeled as malicious when combining obfuscations (Finding 4), except in the case of reflection transformations (Finding 2).

The major finding for obfuscation-tool developers is that our study indicates that many of their transformations result in invalid, non-installable, or unrunnable apps (Finding 9). Although some of that burden lies on benign-app developers to ensure that obfuscations they apply do not adversely affect their apps, obfuscation-tool developers would benefit from aiding benign-app developers in the task of ensuring their apps remain runnable after obfuscation.

We further examine the implications of the interaction between (1) obfuscations tools' ability to produce installable and runnable apps and (2) the anti-malware product detection rate on malicious apps obfuscated using those tools. Figure 4.6 visually depicts the interaction between these two phenomena. Obfuscation tools that lie on the upper-right corner of the figure are preferred tools for benign app developers, obfuscation-tool providers, and anti-malware vendors. These obfuscation tools reliably generate installable and runnable apps while maintaining a high detection rate of malicious apps for anti-malware products. In our study, no tool is above 80% for both its anti-malware detection rate and installability

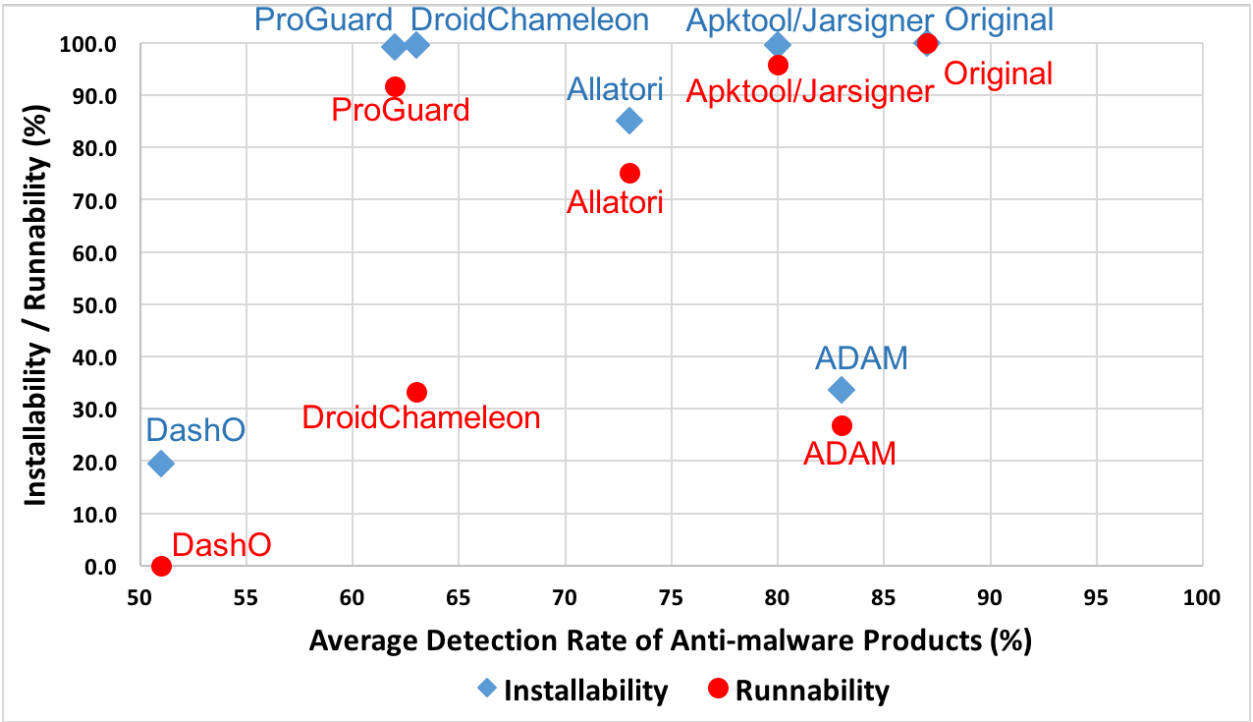


Figure 4.6: Anti-malware detection, installability, and runnability with respect to obfuscators

and runnability after obfuscation. Consequently, significant improvements can be made to these tools along those dimensions.

Obfuscation tools that lie on the upper-left corner of Figure 4.6 are tools that exhibit properties particularly useful for malware authors. These tools reliably generate installable and runnable apps while evading the detection of anti-malware products. ProGuard is an example of such a tool. Although few tools appear close to the upper-left corner of the chart, malware authors are likely to expend the extra effort needed to ensure that their obfuscations result in installable and runnable apps.

4.7 Threats to Validity

This section presents our study's threats to external and construct validity, and the actions we have taken to mitigate them.

External validity measures the extent to which the results of our study can be generalized. One threat to external validity for our study is whether our study's findings can be generalized to other apps outside of our study. To mitigate this threat, we obtained benign and malicious apps from diverse sources that vary across application domains, in terms of app size, and originate from various time periods.

To ensure our findings are likely to generalize to other obfuscation strategies and tools, we employed 29 obfuscation strategies from 7 obfuscation tools—the largest number of strategies and tools utilized to date for a study about app obfuscation. We further obtained obfuscation tools that are academic, open-source, and commercial—aiding in generalizability to these three different sources.

Another threat to external validity is our selection of anti-malware products. To mitigate this threat, we have selected over 60 anti-malware products from VirusTotal, and focused on the most popular and well-rated 21 products for our study. We make the results of our study, along with the complete list of anti-malware products and apps, available online [37]. The findings for the anti-malware products not discussed in this chapter are consistent with the findings in this chapter.

Construct validity is concerned with whether our study's measurements or measurement procedures validly quantify the constructs or concepts we intend to quantify. A threat to construct validity is the metrics and measurement procedures we used to quantify the ability of obfuscation tools to produce runnable apps whose behavior before obfuscation is similar to behavior after obfuscation. To measure these constructs, we compared the set or sequence of running components and thrown exceptions of apps before and after obfuscation. These measurement procedures and associated metrics are sensible given that components are functional units of behavior—making them a sensible means of identifying high-level behavior—and exceptions are the main means of identifying errors in apps whose test cases and oracles are unavailable, which is the case for many apps on Google Play.

Another threat to construct validity is the labeling of our apps as benign or malicious. To mitigate this threat, our dataset of benign apps are marked as benign by over 50 anti-malware products. Similarly, our malicious apps are obtained from repositories containing apps manually labeled as malicious by security experts.

Chapter 5

Illustrative Example

To further motivate our research and illustrate our approach, this chapter provides an example of an evolving Android system that consists initially of two apps: `SuperPhone` and `StayHealthy` apps, illustrated in Figure 5.1.

The `MakeCalls` Activity in the `SuperPhone` allows a user to make phone calls and it stores calls' information in the `CallsDB`, a Content Provider component. The `History` Activity queries the stored calls in `CallsDB` and lists them to a user. Making phone calls requires CALL_PHONE permission. The `SuperPhone` app has this permission, and hence all its components acquire them as well.

`StayHealthy` is a fitness app that allows users to log their daily workouts, via the `Exercises` Activity, and meals, via the `Meals` Activity. Both of these Activities are accessible from the `Home` Activity. The `Home` Activity provides statistical information about the previously recorded workouts and meals and allows a user to navigate to other screens in the app. The `LocTracker` is a service that runs in the background and tracks the user's location upon receiving an Intent. `Exercises` uses `LocTracker` to draw a route map of a user's workout such as walking or cycling routes.

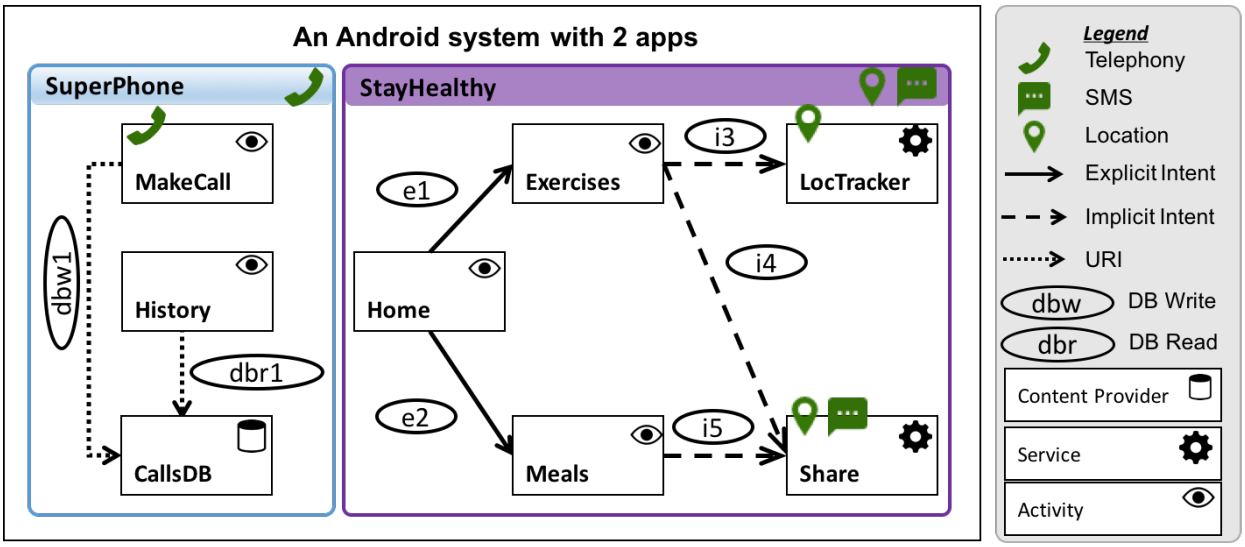


Figure 5.1: Component-based architecture of an Android system consisting of two apps.

`StayHealthy` also allows a user to share his logged activities, either workouts or meals, with friends by sending text messages. The `Share` Service sends spatial data, i.e., tagged data with the current user's location, as a text message to the phone number specified in the received Intent. The `Share` Service is being used by both `Exercises` and `Meals` Activities. Sending spatial text messages requires SMS and LOCATION permissions. The `StayHealthy` app has these permissions, and hence all its components acquire them as well.

Listing 5.1 shows part of the `Share`'s program logic for sending text messages. On lines 5 and 6 of Listing 5.1, the `Share` service extracts the phone number and the message body from the received Intent, respectively. On line 14, the extracted message body is annotated with the current location and stored on the `spatialMsg` object. The extracted phone number and the spatial message information is used in line 18 to send a text message. The `Share` component is vulnerable to a privilege escalation attack since it performs a privileged task, sending spatial text messages, without checking if the caller component has the required SMS and LOCATION permissions to perform the task. An example of such a check is shown in line 4 of Listing 5.1, but in this example it is commented.

Listing 5.1: Vulnerable component, Share Service, sends a text message.

```
1  public class Share extends Service {
2      ...
3      public int onStartCommand(Intent intent, int flags, int startId){
4          //if ((checkCallingPermission("android.permission.SEND_SMS") == PackageManager.
               PERMISSION_GRANTED) && (checkCallingPermission("android.permission.
               ACCESS_FINE_LOCATION") == PackageManager.PERMISSION_GRANTED)) {
5          String phoneNumber = intent.getStringExtra("PHONE_NUMBER");
6          String msg = intent.getStringExtra("MSG_CONTENT");
7          LocationManager locMgr = (LocationManager) this.getSystemService(Context.
               LOCATION_SERVICE);
8          Location myCurrentLoc = locMgr.getLastKnownLocation();
9          StringBuilder spatialMsg = new StringBuilder("-MSG-");
10         spatialMsg.append(msg).append("@");
11         if (myCurrentLoc != null){
12             spatialMsg = spatialMsg.append(String.valueOf(myCurrentLoc.getLatitude()))
13                     .append("/")
14                     .append(String.valueOf(myCurrentLoc.getLongitude()));
15         }
16
17         SmsManager smsManager = SmsManager.getDefault();
18         smsManager.sendTextMessage(phoneNumber, null, spatialMsg.toString(), null, null);
19     //}
20     ...
```

On the other hand, the `LocTracker` is a secure Service since it checks for the granted permissions of the caller component. Such a check in Android can be achieved using the `checkCallingPermission` API. Although `Share` is a vulnerable component in the Android system illustrated in Figure 5.1, the current system is not actively threatened since no component is exploiting this vulnerability.

At a later time, a user installs a new app called `BrainTeaser`, as shown in Figure 5.2. `BrainTeaser` is a malicious app that challenges a user to solve mathematical questions and then measures her intelligence quotient (IQ). The `IQtest` Activity displays questions and communicates with the `Qgenerator` Service to get the next question. `Qgenerator` is a malicious component that, once started, communicates with the `Share` Service of the `StayHealthy` app. Since `Share` does not check if the caller components has the required permissions, i.e., SMS and LOCATION permissions, this component is vulnerable to a privilege-escalation ICC attack. Therefore, the communication between the `Qgenerator` and `Share` results in exploiting this vulnerability which allows `Qgenerator` to leak the user's location to any premium rate number without having the proper permissions to perform such a task.

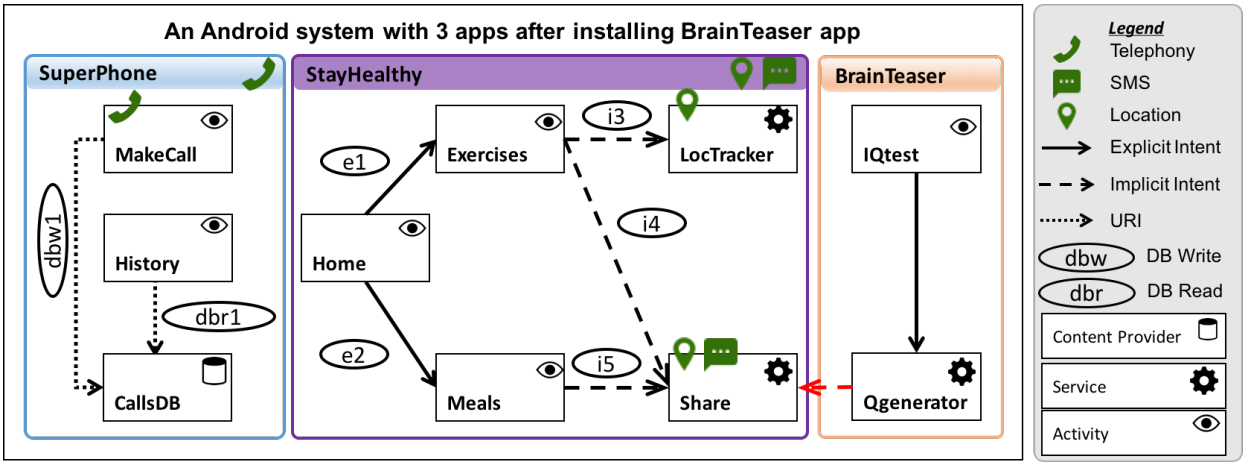


Figure 5.2: Component-based architecture of an evolving Android system after installing `BrainTeaser` app.

The attack described in this section is a legitimate scenario in the current implementation of the Android platform [71]. Moreover, performing a complete analysis of all Android apps in the system every time the system changes is neither efficient nor practical. We show how, through a runtime monitoring and incremental analysis of the least-privilege architecture of the system, SALMA can efficiently and effectively mitigate such a threat.

Chapter 6

A Self-protecting Android Software System

Figure 6.1 depicts a high-level overview of SALMA, a fully automated self-protection Android software system. SALMA contains two layers, the *protected layer* and the *protecting layer*. The protected layer consists of our modified Android framework and a set of apps that a user installs on a device. The protecting layer realizes the IBM MAPE-K control loop [139]. As explained in Section 2.1, MAPE-K consists of four components and a knowledge component. The *Knowledge* contains an architectural model of the system. The *Monitor* observes the system and keeps the model synchronized with the running system. The *Analyzer* assesses the system for security threats. The *Planner* determines the best security policies, a.k.a. adaptation tactics, to be enforced at runtime by the *Executor*.

Figure 6.1 depicts instantiations of each of the MAPE-K components in the protecting layer: *Monitor Extractor & Synthesizer*, which is a *Monitor*; *Incremental Security Analyzer* which is an *Analyzer*; *Policy Synthesizer*, which is a *Planner*; and *Policy Enforcer*, which is an *Executor*.

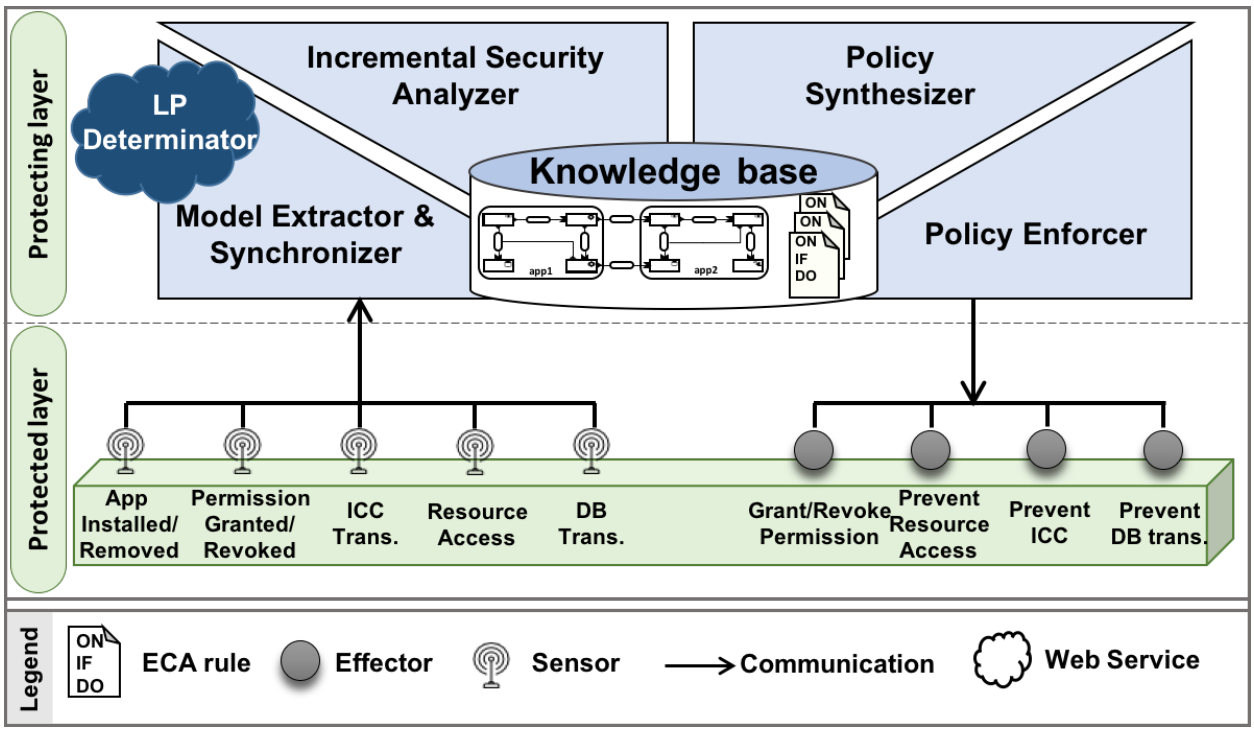


Figure 6.1: Overview of SALMA.

Model Extractor & Synchronizer automatically obtains and maintains a precise runtime least-privilege (LP) architectural model of an Android system. *Model Extractor & Synchronizer* uses the *LP Determinator* web service (see Figure 6.1) to determine the initial LP architecture of an Android system.

The *LP Determinator* web service of *Model Extractor & Synchronizer*, illustrated in Figure 6.2, consists of three steps (1) *Architectural Elements Extractor* uses static program analysis techniques to elicit the system's principal components along with their properties, latent communications, and permissions usages from the apps comprising a system. (2) *Privilege Analyzer* systematically examines each component to comprehensively determine its privileges, the permissions it can use as well as components with which it can communicate, both inside and outside the scope of its hosting app, as permitted by the Android runtime environment. The result of this step is captured in a *Multiple-Domain Matrix (MDM)*, representing the original architecture of system. (3) *Privilege Reducer* determines the exact permissions and

communications each component needs to fulfill its functionality. The derived information is then captured in an *MDM*, representing the least-privilege architecture for the system. Moreover, this step optionally allows a security architect to further modify the architecture as needed to establish the proper privileges for each component.

Once *Model Extractor & Synchronizer* determines the initial LP architecture of an Android system, *Incremental Security Analyzer* performs security analysis to check if the architecture contains ICC attacks. In particular, *Incremental Security Analyzer* focuses on the prominent ICC attacks described in Section 2.4. Next, while the system is running, when a change occurs in the maintained runtime model (e.g., adding a new app, removing an existing app, revoking a permission, etc.), the *Incremental Security Analyzer* (1) determines the impacted part of the system due to that change, (2) runs a subset of security analyses that need to be performed, and (3) updates the security posture of the system by either adding new potential security attacks or removing existing threats.

Policy Synthesizer captures the determined initial LP architecture as a set of *Event-Condition-Action (ECA)* rules. Moreover, *Policy Synthesizer* takes the analysis results computed by the *Incremental Security Analyzer* and constructs security policies in the form of ECA rules.

The *Policy Enforcer* regulates interactions at the granularity of components through enforcing the security policies and the LP architecture at runtime. It relies on various effectors that we have added to the Android runtime environment, e.g., *Prevent ICC*, *Prevent Resource Access*, etc., to check the conformance of ICC and resource-access transactions to the LP architecture. The rest of this chapter describes SALMA in detail.

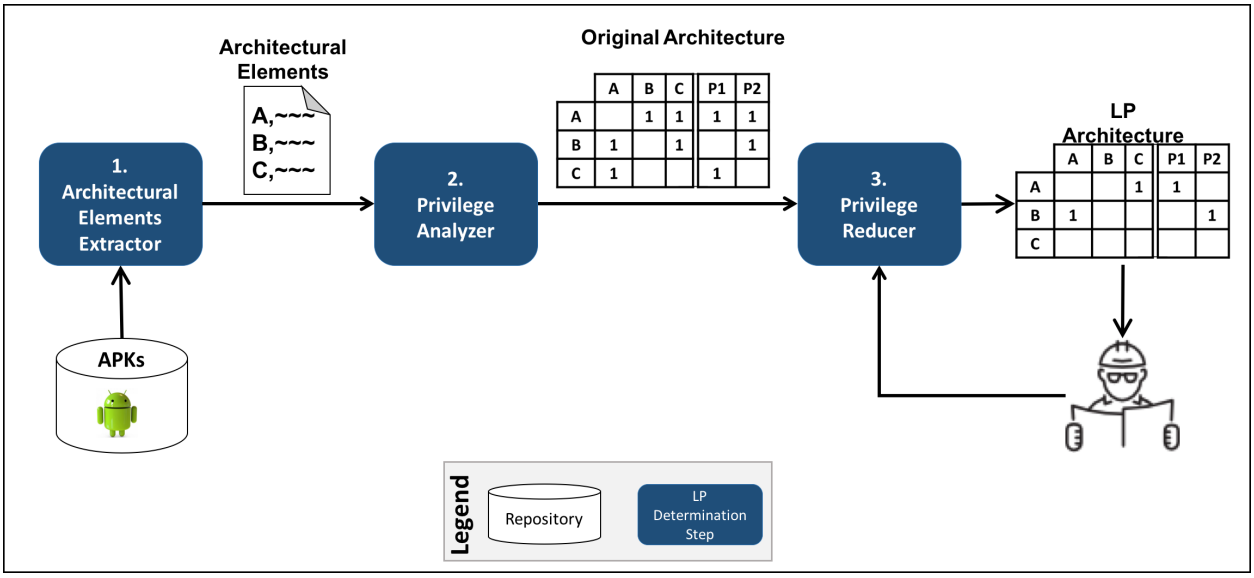


Figure 6.2: The steps of *LP Determinator* of *Model Extractor & Synchronizer* to determine the LP architecture of an Android system. An Android system could be compromising a set of Android apps or even a single app.

6.1 Model Extractor & Synchronizer

Similar to other self-* software systems, SALMA leverages an abstract representation of the software to manage and adapt the system at runtime. Prior research assumes these models are developed prior to the deployment of software. Given the rich app ecosystem of Android, this assumption does not hold, since users can install a variety of apps that are unknown a priori. To address this challenge, *Model Extractor & Synchronizer* utilizes static and dynamic analysis techniques to automatically obtain and maintain a precise LP architectural runtime model of an Android system.

Unlike the default privilege management scheme in Android, where each component inherits all privileges that are granted to its parent app, *Model Extractor & Synchronizer* grants each component the exact privileges it needs to fulfill its functionality. Therefore, the LP architectural model derived by *Model Extractor & Synchronizer* minimizes the attack surface of an Android system. Hence, it is an ideal representation of an Android system to be maintained at runtime.

6.1.1 Determining the LP architecture

To derive the LP architectural representation of an Android system, *Model Extractor & Synchronizer* asynchronously interacts with the *LP Determinator*, as shown in Figure 6.1, a cloud-based web service that consists of three steps as shown in Figure 6.2 and further discussed below.

Step 1: Architectural Elements Extractor

To obtain the system's architecture, *LP Determinator* first needs to determine the principal components that constitute the system, their properties, communication interfaces, and permission usages. Such information is obtained from two sources, an app's manifest file and its bytecode.

LP Determinator utilizes *APKtool* [14], a reverse-engineering tool for Android APK files, to recover an app's manifest file. *LP Determinator* then parses the file to extract the components of the app, their properties, their provided interfaces, the permissions that the app requires, and the permissions that the app defines, if any.

Table 6.1 partially shows the extracted information corresponding to our running example (recall Chapter 5). The *Component Type* column represents the particular type of a component, which could be either Activity, Service, Broadcast Receiver, or Content Provider. The *Exported* column indicates whether a component can be launched from outside its hosting app or not. The *Intent Filter* column shows the interfaces provided by a component. Finally, the *Granted* column shows the permissions requested by an app, and subsequently granted by Android to all of its component. Among others, the five components of the `StayHealthy` app all have access to both the SMS (`android.permission.SEND_SMS`) permission and the LOCATION

Table 6.1: The extracted architectural elements for the Android system shown in Figure 5.1

ID	App	Component Name	Component Type	Exported	Intent Filter	Permissions			Intent	DB Write (DBW)	DB Read (DBR)
						Granted	Used	Enforced			
1	SuperPhone	MakeCall	Activity	Yes		{PHONE}	{PHONE}			{dbw1}	
2	SuperPhone	History	Activity	No		{PHONE}					{dbr1}
3	SuperPhone	CallsDB	Content Provider	Yes		{PHONE}					
4	StayHealthy	Home	Activity	Yes	MAIN	{LOCATION, SMS}			{e1, e2}		
5	StayHealthy	Excercises	Activity	Yes		{LOCATION, SMS}			{e3, e4}		
6	StayHealthy	Meals	Activity	Yes		{LOCATION, SMS}			{e5}		
7	StayHealthy	LocTracker	Service	Yes	LOC_ROUTE	{LOCATION, SMS}	{LOCATION}	{LOCATION}			
8	StayHealthy	Share	Service	Yes	SEND_SMS	{LOCATION, SMS}	{LOCATION, SMS}				

(`android.permission.ACCESS_FINE_LOCATION`) permission, given that the `Messaging` app acquires the SMS and the LOCATION permissions.

Parsing the manifest file is not enough to obtain a system's architecture, since a large amount of information is latent in the app's bytecode. For example, Broadcast Receivers can be registered in code without declaring them in the manifest file. Components can also programmatically define Intent Filters in code. In addition, all ICCs are latent in the app's bytecode. Components can communicate with one another in two ways: (1) using Unified Resource Identifiers (URIs) to access the encapsulated data in Content Providers, and (2) by sending Intents, either explicitly or implicitly. SALMA utilizes IC3 [160] to analyze each app in the system and extract such latent information from its bytecode. IC3 is the state-of-the-art static program analysis tool for Android.

For each Intent in bytecode, *LP Determinator* extracts the sender component, receiver component, action, categories, and data. Moreover, the type of each extracted Intent is indicated in the *Intent* column with prefix$e$ for explicit Intent or i for implicit Intent. Similarly, for each extracted URI in bytecode, *LP Determinator* extracts the sender component, receiver Content Provider, and the type of the request. The URI request type can be either (1) a database manipulation request, e.g., delete, insert, or update the encapsulated data in a Content Provider, shown in the `DBW` column of Table 6.1 or (2) a database read request, e.g., querying the encapsulated data in a Content Provider, shown in the `DBR` column of Table 6.1. Table 6.1 shows the remaining information collected in this way for our running example. It is worth mentioning that if a communication between two components cannot be statically extracted, i.e., a hidden communication either due to the use of code obfuscation (recall Chapter 4) or dynamic class loading [173], this hidden communication will not be part of the architectural elements table, i.e., Table 6.1.

LP Determinator also identifies the permissions actually used by components. These are the permissions that a component uses for (1) accessing a protected Content Provider, or (2) calling a protected API. For the former, we have created a mapping between protected Content Providers and the required permissions. For example, to read the contacts information from Android's Contacts Content Provider, a component needs `android.permission.READ_CONTACTS` permission. Using this mapping and the accessed Content Providers, our approach determines the actually used permissions for a component. Since IC3 does not extract the permissions used through API calls, for the latter case, *LP Determinator* leverages PScout permission map [66], one of the most recently updated and comprehensive permission maps available for the Android framework. It specifies mappings between Android API calls/Intents and the permissions required to perform those calls. For example, `Share` component in `StayHealthy` app uses the `sendTextMessage()` API for sending text messages (see line 18 of Listing 5.1), which requires SMS permission. We thus consider this to be a permission that is actually used by this component, as shown in the `Used` column of Table 6.1.

Finally, *LP Determinator* builds on a prior work [192] to extract the permissions enforced by a component at two levels. While the coarse-grained permissions specified in the manifest file are enforced by the Android runtime environment over an entire component, it is possible to add permission checks, such as *checkCallingPermission*, throughout the code controlling access to specific parts of a component (see line 4 of Listing 5.1). *LP Determinator* identifies both types of checks. Since the `Share` component in the Android system illustrated in Figure 5.1 does not perform any checks (line 4 of Listing 5.1 is commented out), the `Enforced` column in Table 6.1 is empty for the `Share` component. Notice that since `LocTracker` is a secure Service (recall Chapter 5), column `Enforced` of Table 6.1 show that this component is enforcing the `LOCATION` permission.

Step 2: Privilege Analyzer

The next step is to derive the overall system architecture from the information obtained for individual components in the previous step. This is called the *Original* system architecture, as it represents the architecture of system if it were to be deployed on the official Android runtime environment. SALMA captures the architecture of an Android system as a *Multiple-Domain-Matrix (MDM)*[149], which is a matrix representation of all relationship types (i.e., domains) among principal elements, such as components and permissions, in a system. An MDM consists of multiple *Design-Structured Matrices (DSMs)* [203]. Each domain is modeled as a DSM—a simple matrix that captures the relationships of one type.

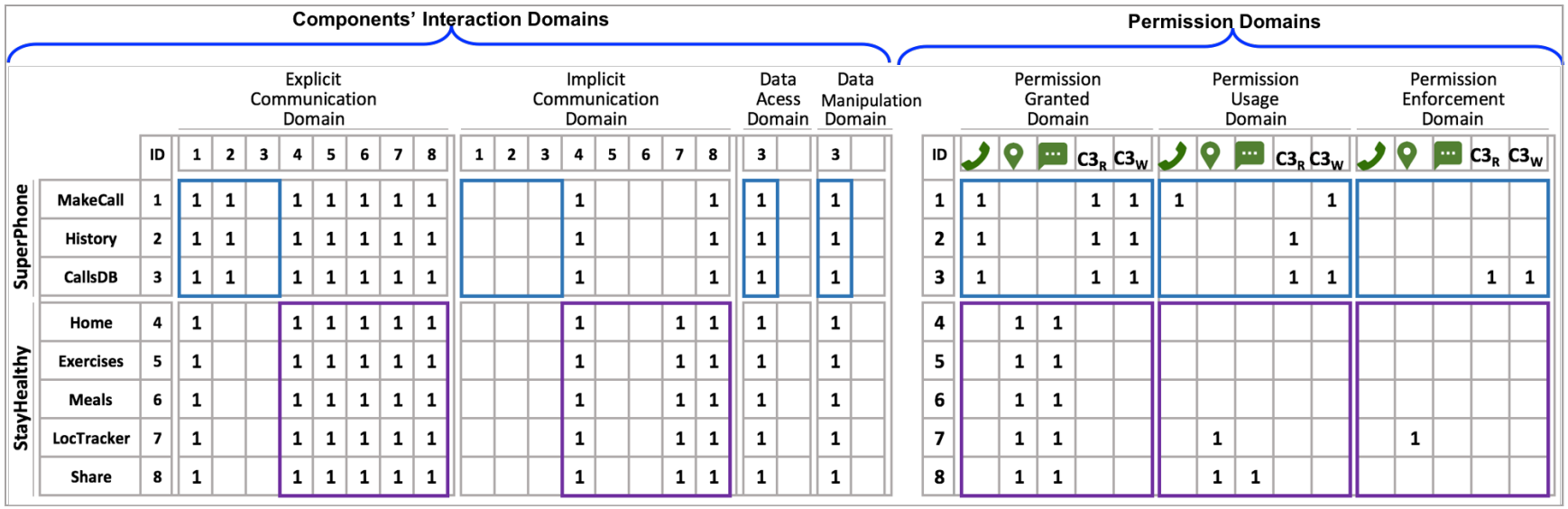
For the purpose of security analysis, SALMA models an Android system using seven domains, four component interaction domains and three permission domains. The four component interaction domains in the MDM model of Figure 6.3 shows all potential component-to-comp onent communications. Each non-empty cell in these domains indicates the fact that the architecture of system allows for potential interaction between two components, either by sending Intents or using URIs to access the encapsulated data in Content Providers. Rows represent sender components; columns represent receiver components.

The *explicit* communication domain shows all potential component-to-component interactions using explicit Intents. Allowed explicit communications are derived using the following rule.

Definition 1 (Allowed Explicit Communication)**.** *Let E be a set of all exported components, and c_1 and c_2 be two arbitrary components in the system. We say that c_1 can explicitly communicate with c_2, if either both components belong to the same app or c_2 is an exported component and c_1 is granted the permissions enforced by c_2:*

$$canCommunicate_e(c_1, c_2) \equiv (app_{c1} = app_{c2}) \vee (c_2 \in E \wedge enforced_{c2} \subseteq granted_{c1})$$

Figure 6.3: The Original architecture derived from the Android system described in Chapter 5.



The Explicit Communication Domain in Figure 6.3 shows the result of applying Definition 1 to Table 6.1. According to the explicit communication domain, components 4, 5, 6, 7, and 8 can communicate with one another because they belong to the same app, as well as component 1 since it is exported, but not component 2. Components 1 and 2 can also communicate with all the other components in the system.

The *implicit* communication domain shows all potential component-to-component interactions using implicit Intents. Allowed implicit communications are derived using the following rule.

Definition 2 (Allowed Implicit Communication)**.** *Let F be a set of all declared public provided interfaces, i.e., Intent filters, and c_1 and c_2 be two arbitrary components in the system. We say that c_1 can implicitly communicate with c_2, if c_2 defines a public provided Interface and either both components belong to the same app or c_1 is granted the permissions enforced by c_2:*

$$canCommunicate_i(c_1, c_2) \equiv c_2.filters \subseteq F \wedge (app_{c1} = app_{c2} \vee enforced_{c2} \subseteq granted_{c1})$$

The Implicit Communication Domain in Figure 6.3 shows the result of applying Definition 2 to Table 6.1. According to the implicit communication domain, all components in the system can communicate with component 4 and component 8. Component 8 declares a public provided interface for sending spatial text messages without enforcing any permission. Component 4 is the main entry point for *StayHealthy* app, i.e., declares a public Intent filter with *android.intent.action.MAIN* action. Moreover, none of the components in the `SuperPhone` app, i.e., components 1, 2, and 3, can communicate with component 7 since its enforcing the `LOCATION` permission to be granted to the caller component.

Note that the communication domain also includes interactions between the Android framework and components of third-party apps. Android provides over 230 protected broadcast

Intents that can only be sent by the system to the registered components. For example, when a user installs an app, the system sends a broadcast Intent including the package name of the newly installed app to all components that listen to the PACKAGE_ADDED broadcast Intent action. Figure 6.3 shows no such interactions with the system, as no component in our running example is registered to receive protected broadcast Intents.

The data *access* domain shows potential component-to-content provider interactions for reading (i.e., querying) stored data. Allowed data access communications are derived using the following rule.

Definition 3 (Allowed Data Access)**.** *Let E be a set of all exported components, and c be an arbitrary component in the system, i.e., $c \in E$, cp be a content provider, i.e., $cp \in E$. We say that c can access the enclosed data in cp, if either both components belong to the same app or cp is an exported content provider and c_1 is granted the read access permission enforced by cp:*

$$canAccess(c, cp) \equiv (app_{c1} = app_{cp}) \vee (cp \in E \wedge read_{cp} \subseteq granted_{c1})$$

The Data Access Domain in Figure 6.3 shows the result of applying Definition 3 to Table 6.1. According to the data access communication domain, only the components in the `SuperPhone` app can query the enclosed data in the `CallsDB` Content Provider since its enforcing the $C3_R$ read access permission.

Finally, the data *manipulation* domain shows potential component-to-content provider interactions for modifying (i.e., updating, inserting, or deleting) stored data. Allowed data manipulation communications are derived using the following rule.

Definition 4 (Allowed Data Manipulation)**.** *Let E be a set of all exported components, and c be an arbitrary component in the system, i.e., $c \in E$, cp be a content provider, i.e., $cp \in E$. We say that c can manipulate the enclosed data in cp, if either both components belong to the same app or cp is an exported content provider and c_1 is granted the write access permission enforced by cp:*

$$canManipulate(c, cp) \equiv (app_{c1} = app_{cp}) \vee (cp \in E \wedge write_{cp} \subseteq granted_{c1})$$

The Data Manipulation Domain in Figure 6.3 shows the result of applying Definition 4 to Table 6.1. According to the data manipulation communication domain, only the components in the `SuperPhone` app can modify, i.e., insert, update or delete, the enclosed data in the `CallsDB` Content Provider since its enforcing the $C3_W$ write access permission.

The three permission domains in the MDM model of Figure 6.3 represent the component-to-permission relationships. Each non-empty cell corresponds to a permission that is either (1) granted to a component, meaning that the component has that permission, as its hosting app has requested the permission in its manifest file, (2) used by a component, meaning that the component is actually making API calls, using protected URIs, or interacts with other apps that require the permission, or (3) enforced by a component, meaning that either the Android runtime environment or the component itself check the permission of callers (as you may recall from Section 6.1.1 there are two ways of enforcing permissions in Android). The permission domains in the MDM are populated based on the information obtained in the first step (i.e., Granted, Used, and Enforced columns of Table 6.1). For example, the MDM shown in Figure 6.3 indicates that the first three components are granted the PHONE permission, while components 4, 5, 6, 7, and 8 are granted the location and the SMS permissions.

Step 3: Privilege Reducer

The Original architecture derived in the previous step clearly violates the principle of least-privilege. This step aims to derive the LP architecture by granting only the privileges required by each component to fulfill its tasks.

LP Determinator uses the extracted inter-component communications (information in the *Intent* and *URI* columns of Table 6.1) to determine the communication privileges that are needed for each component to provide its functionality, and removes communication privileges that are unnecessary. Then it models the LP architecture of an Android system in an MDM of seven domains.

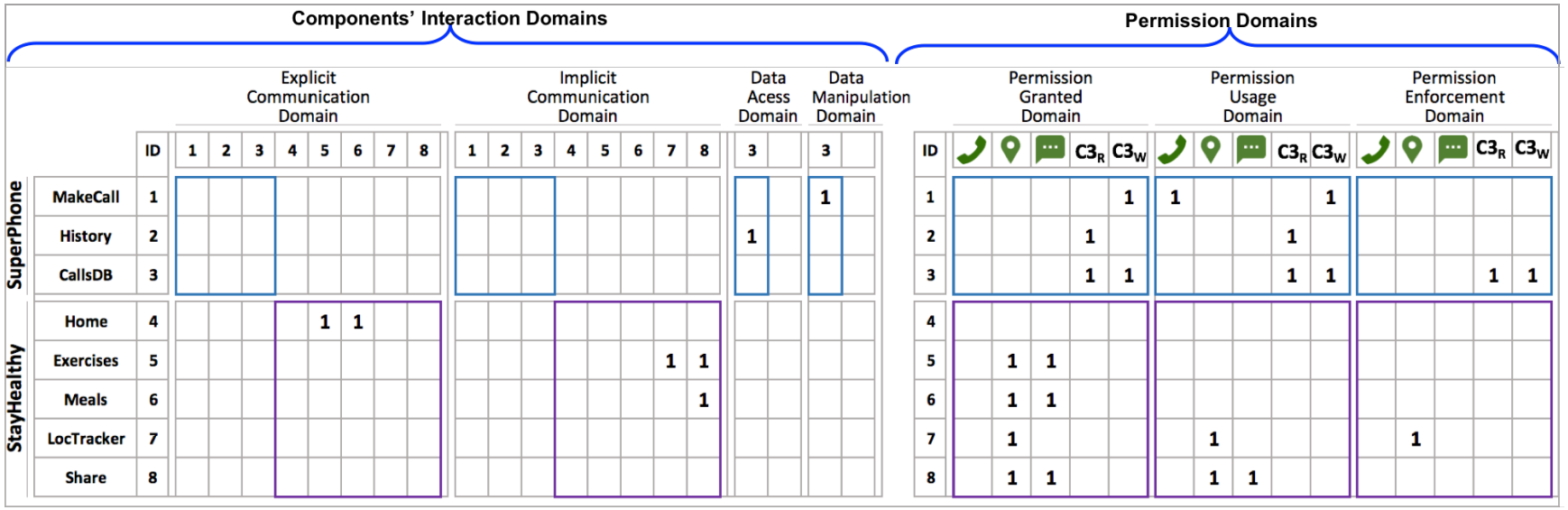


Figure 6.4: An MDM representation of the system illustrated in Figure 5.1. Each colored box in the MDM corresponds to the matching colored app in Figure 5.1.

Moreover, *LP Determinator* reduces the granted permissions for each component in the Permission Granted Domain of the LP architecture using the following rule:

Definition 5 (Required Permission)**.** *Let c_1 be a component, and $used_{c1}$ be a set of permissions directly used by component c_1. We define the required permissions for c_1 as permissions either directly used by c_1 or used by component c_2 with which c_1 communicates:*

$$requiredPermissions_{c1} = \{p : Permission \mid \exists\ c_2 : Component \bullet p \in$$
$$used_{c1} \vee ((communicate_e(c_1, c_2) \vee communicate_i(c_1, c_2)) \wedge p \in used_{c2} \wedge p \in granted_{c1})\}$$

According to Definition 5, a component legitimately needs a permission in two cases: (1) the permission is directly used by the component through, among other things, making protected API calls or using protected URIs; (2) another component with which the given component is interacting is using that permission. The latter may be a legitimate case, since a component that uses a permission may require the calling component to also have that permission. In fact, failing to check if the calling component has the necessary permission may result in a privilege escalation attack.

In our running example, *LP Determinator* determines that the `Share` component has a legitimate reason to hold the *SMS* and the *LOCATION* permissions, since it uses them. The `Exerciser` component also has a legitimate reason to hold the *LOCATION* permission, since the app it belongs to has these permissions and it communicates with the `Share` component that uses these permissions. The `MakeCall` component, on the other hand, has a legitimate reason to hold the *PHONE* permission since it uses that permission, while `History` and `CallsDB` do not need it. The `History` component, however, does not need the *LOCATION* permission, since it neither uses it nor does it communicate with a component that uses that permission.

The four component interaction domains in the MDM model of the LP architecture, shown in Figure 6.4, represent the various component-to-component communications. Each non-empty cell in these domains indicates that there is a communication between two components. The *explicit* and the *implicit* communication domains show all allowed component-to-component interactions using explicit and implicit Intents, respectively. Similarly, the data *access* and the data *manipulation* domains show allowed component-to-content provider interactions for reading (i.e., querying) and modifying (i.e., updating, inserting, or deleting) stored data, respectively.

The *explicit* communication domain is derived using the following rule.

Definition 6 (Explicit Communication)**.** *Let E be a set of all exported components in the system, I is a set of all extracted Intents in the system. Let c_1 and c_2 be two arbitrary components in the system, i.e., $\{c_1, c_2\} \subseteq E$, i be an Intent, i.e., $i \in I$. We say that c_1 can explicitly communicate with c_2, if i is sent by c_1, i.e., $i.sender = c_1$, and c_2 is explicitly specified in the Intent i as a target component, i.e., $i.target = c_2$, and either both c_1 and c_2 belong to the same app or c_1 is granted the permissions enforced by c_2:*

$$communicate_e(c_1, c_2) \equiv \exists i \in I \mid i.sender = c_1 \wedge i.target = c_2 \wedge (app_{c_1} = app_{c_2} \vee enforced_{c_2} \subseteq granted_{c_1})$$

The *explicit* communication domain in Figure 6.4 shows the results of applying definition 6 to Table 6.1. For instance, as shown in Figure 6.4, the LP architecture allows the `Home` component to communicate with the `Exercise` component (indicated by " 1" in row 4, column 5 of Explicit Communication Domain) since there is an explicit Intent between them as illustrated in Figure 5.1. On the other hand, the LP architecture prohibits the `Home` component to communicate with the `MakeCall` component.

Similarly, the communications in the *implicit* communication domain are derived using the following rule.

Definition 7 (Implicit Communication)**.** *Let E be a set of all exported components in the system, I is a set of all extracted Intents in the system. Let c_1 and c_2 be two arbitrary components in the system, i.e., $\{c_1, c_2\} \subseteq E$, i be an Intent, i.e., $i \in I$. We say that c_1 can communicate with c_2, if i is sent by c_1, i.e., $i.sender = c_1$, and c_2 is exporting an Intent Filter that can handle i, i.e., $match(i, c_2.f)$, and either both c_1 and c_2 belong to the same app or c_1 is granted the permissions enforced by c_2:*

$$communicate_i(c_1, c_2) \equiv \exists i \in I \mid i.sender = c_1 \wedge match(i, c_2.f) \wedge (app_{c_2} = app_{c_1} \vee enforced_{c_2} \subseteq granted_{c_1})$$

The $match(i, c_2.f)$ function in Definition 7 performs Intent resolution [43] to check if there is an *Intent Filter* declared by c_2 that can handle the Intent i. The *implicit* communication domain in Figure 6.4 shows the results of applying Definition 7 to Table 6.1. According to Definition 7, component 5 implicitly communicates with component 7 since there is an implicit Intent sent by `Exercises` in which `LocTracker` can handle (recall Figure 5.1.

The *data access* and the *data manipulation* domains are derived using the following rules.

Definition 8 (Data Access)**.** *Let E be a set of all exported components in the system, DBR is the set of all extracted database read requests in the system. Let c be an arbitrary component in the system, i.e., $c \in E$, cp be a content provider, i.e., $cp \in E$, and dbr be a database read (query) request in the system, i.e., $dbr \in DBR$. We say that c can access the stored data in cp, if c sends a database query (dbr) where the authority attribute of dbr*

matches the authority name of cp, and either c and cp belong to the same app or c is granted the enforced read access permission by cp.

$$access(c, cp) \equiv \exists dbr \in DBR \mid dbr.sender = c \wedge dbr.authority = cp.authority \wedge (app_c = app_{cp} \vee read_{cp} \subseteq granted_c)$$

To illustrate an instance of Definition 8 on the extracted database requests, i.e., DBR, Figure 5.1 shows that component 2, `History`, accesses the stored data in component 3, `CallsDB`—which is also reflected in Figure 6.4.

Definition 9 (Data Manipulation)**.** *Let E be a set of all exported components in the system, DBW is the set of all extracted database write requests in the system. Let c be an arbitrary component in the system, i.e., $c \in E$, cp be a content provider, i.e., $cp \in E$, and dbw be a database write (insert, delete, or update) request in the system, i.e., $dbw \in DBW$. We say that c can access the stored data in cp, if c sends a database manipulation request (dbw) where the authority attribute of dbw matches the authority name of cp, and either c and cp belong to the same app or c is granted the enforced write access permission by cp.*

$$manipulate(c, cp) \equiv \exists dbw \in DBW \mid dbw.sender = c \wedge dbw.authority = cp.authority \wedge (app_c = app_{cp} \vee write_{cp} \subseteq granted_c)$$

As an example of Definition 9, Figure 5.1 depicts component 1, `MakesCall`, updates the stored data in component 3, `CallsDB`—which is further shown in Figure 6.4.

Finally, a security architect can adjust the resulting architecture by manually granting and revoking permissions in the MDM. For example, a security architect can revise the privileges

granted to apps and their components based on their reputation. This capability could also be useful in a forward-engineering setting, where an Android system is developed from scratch.

The amount of privilege reduction achieved through enforcing LP architecture can be quantified by calculating the distance between the LP architecture (L) and the Original architecture (O) as shown in Equation 6.1.

$$Reduction(O, L) = 1 - \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} L_{ij}}{\sum_{i=1}^{n} \sum_{j=1}^{m} O_{ij}} \tag{6.1}$$

In Equation 6.1, i and j represent the ith column and jth row of an MDM with n rows (components) and m columns (components and permissions). In our running example, comparing the Original architecture (cf. Figure 6.3) with the LP architecture (cf. Figure 6.4) shows 76.9% reduction in granted privileges.

6.1.2 Synchronizing the LP architecture

Extracting an architectural model of an Android system, as described in Section 6.1.1, and use it to reason about the running system is not practical since Android systems are highly dynamic software systems where a user can install/remove an app, grant/revoke permissions to apps, etc. In all of these system changes, the determined LP architectural model becomes

Table 6.2: The Significant Events that SALMA Monitors.

ID	Event	ID	Event
1	ADD_APP	5	NEW_IMPLICIT_COMM
2	REMOVE_APP	6	NEW_EXPLICIT_COMM
3	GRANT_PERMISSION	7	NEW_DATA_ACCESS
4	REVOKE_PERMISSION	8	NEW_DATA_MANIPULATION

an obsolete representation of the running system. Therefore, to keep the maintained model a valid representation of the running system in spite of changes at runtime, *Model Extractor & Synchronizer* maintains the model synchronized with the running system. Every time the system changes, *Model Extractor & Synchronizer* reflects that change in the maintained model.

Table 6.2 shows a list of the events (i.e., changes in the system) that *Model Extractor & Synchronizer* tracks. In this paper, we refer to these events as *significant events*. For each significant event, *Model Extractor & Synchronizer* receives a notification from the system and updates the model accordingly. For example, when a user installs a new app, *Model Extractor & Synchronizer* receives a system notification with the ACTION_PACKAGE_ADDED Intent action. In this case, *Model Extractor & Synchronizer* communicates with *LP Determinator* to obtain the architecture of the new app, i.e., LP_{newApp}; merges LP_{newApp} with LP; and applies Definitions 6–9 to add the new app to the current MDM. To avoid substantial analysis time caused by running static analysis tools, the *LP Determinator* can analyze Android apps in advance without waiting for a user to install an app.

In our running example, after a user installs the `BrainTeaser` app (see Figure 5.2), *Model Extractor & Synchronizer* updates the maintained model. The MDM illustrated in Figure 6.5 displays the results of merging `BrainTeaser` with the current MDM, presented in Figure 6.4. Figure 6.5 shows that `IQtest` explicitly communicates with `Qgenerator`, which implicitly communicates with `Share` (see Figure 5.2).

To synchronize the runtime model with the system, *Model Extractor & Synchronizer* relies on receiving system notifications from the sensors in Figure 6.1, indicating significant changes that occur in the system. Some system notifications are already implemented in the Android framework, such as `ADD_APP` and `REMOVE_APP` events. For these events, the framework sends broadcast Intents with `ACTION_PACKAGE_ADDED` and `ACTION_PACKAGE_REMOVED` actions, respectively. While in all other significant events, see Table 6.2, the framework silently

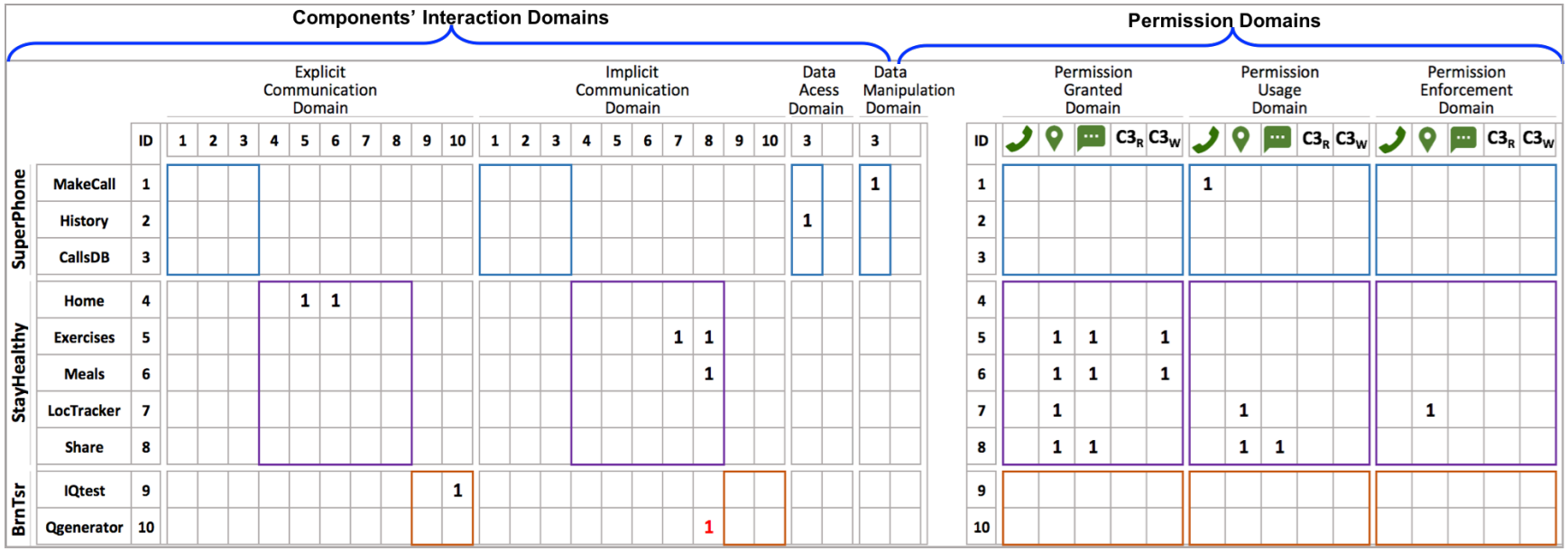


Figure 6.5: An MDM representation of the system illustrated in Figure 5.2 . Each colored box in the MDM corresponds to the matching colored app in Figure 5.2.

executes the event. Therefore, we have introduced new system-generated broadcast Intents to the Android platform. The new broadcasts inform *Model Extractor & Synchronizer* of certain events whenever they occur in the system. Each system-generated broadcast Intent contains information about a particular event. For example, in case a user grants a permission to an app, the framework sends a `GRANT_PERMISSION` broadcast Intent with the permission name and the application package name.

6.2 Incremental Security Analyzer

Once *Model Extractor & Synchronizer* determines the initial LP architecture of an Android system as described in Section 6.1.1, *Incremental Security Analyzer* performs security analysis to check if the architecture contains ICC attacks. In particular, *Incremental Security Analyzer* focuses on prominent security vulnerabilities due to the interaction of multiple apps (recall Section 2.4). After that, since Android systems are highly dynamic software systems, reanalyzing the entire system every time a change occurs is neither efficient nor scalable.

Therefore, SALMA incrementally analyzes the system whenever a change in the system occurs. Our approach leverages the fact that a change in the system (1) impacts only part of the system and (2) often requires running a subset of the security analyses on the impacted part of the system. In Section 6.2.1, we describe the security rules that SALMA applies (1) on the entire system for the first time or (2) on the impacted parts of the system after a change in the system occurs to detect the potential security attacks presented in Section 2.4. Section 6.2.2 describes how our approach determines the impacted parts of the system after a change occurs.

6.2.1 Security Rules

This section describes the security rules that our approach applies on the impacted parts of the system. Each rule when applied on an interaction between two components would reveal if that interaction is vulnerable to a given security attack.

Security Rule 1 (Unauthorized Intent Receipt)**.** *Let c_m be a malicious component, c_v be a vulnerable component, and c_x be a component that c_v intends to send an implicit Intent i to. c_v and c_x belong to the same app, and c_x declares a provided interface, i.e., an Intent filter, through which c_v aims to communicate with c_x using i. In an unauthorized Intent receipt, c_m can intercept i from c_v by declaring a provided interface similar to the one declared by c_x. As such, c_m may gain access to all enclosed data in any matching Intents meant to be received by c_x.*

$$\exists\ communicate_i(c_v, c_m) \mid (app_{c_v} \neq app_{c_m}) \wedge \exists\ communicate_i(c_v, c_x) \wedge (app_{c_v} = app_{c_x})$$

Security Rule 2 (Intent Spoofing)**.** *Let c_m be a malicious component, c_v be a vulnerable component, and c_x be a component intending to communicate with c_v. c_v and c_x belong to the same app. c_v declares a provided interface, i.e., an Intent filter, through which it aims to communicate with c_x. In Intent Spoofing, c_m can send an Intent to c_v over the Intent filter and force c_v to perform a nefarious action upon receipt of the Intent.*

$$\exists\ (communicate_e(c_m, c_v) \vee communicate_i(c_m, c_v)) \mid (app_{c_v} \neq app_{c_m}) \wedge \exists\ communicate_i(c_x, c_v) \wedge (app_{cv} = app_{cx})$$

Security Rule 3 (Privilege Escalation)**.** *Let p be a permission, c_m be a malicious component that is not granted p, and c_v be a vulnerable component that is granted and uses p but does not enforce the use of p as requested by other components. In privilege escalation, c_m is able to indirectly obtain p by interacting with c_v.*

$$\exists\ (communicate_e(c_m, c_v) \vee communicate_i(c_m, c_v)) \mid p \in used(c_v) \wedge p \notin granted(c_m) \wedge p \notin enforced(c_v)$$

Security Rule 4 (Identical Custom Permission)**.** *Let p_m be a custom permission defined by malicious app app_m, i.e., $p_m.definedBy = app_m$, and p_v be a custom permission defined by the vulnerable app app_v, i.e., $p_v.definedBy = app_v$. Both p_v and p_m have the same permission name, i.e., $p_v.name = p_m.name$. c_m is a malicious component in app_m that is granted p_m. c_v is a vulnerable component in app_v that enforces p_v. In an identical custom permission, c_m can communicate with c_v since $p_v.name = p_m.name$, even if p_v and p_m are semantically different permissions.*

$$\exists\ (communicate_e(c_m, c_v) \vee communicate_i(c_m, c_v) \vee access(c_m, c_v) \vee manipulate(c_m, c_v)) \mid app_{c_m} \neq app_{c_v} \wedge p_m \in granted(c_m) \wedge p_v \in enforced(c_v) \wedge p_v.name = p_m.name \wedge p_m.definedBy \neq p_v.definedBy$$

Security Rule 5 (Passive Data Leak)**.** *Let cp_v be a vulnerable* `Content Provider` *that does not enforce a read access permission, c_m be a malicious component that accesses (queries) the stored data in cp_v. In a passive data leak, c_m can passively disclose the sensitive data stored in cp_v.*

$$\exists\ access(c_m, cp_v) \mid enforce_r(cp_v) = \emptyset$$

In Definition 5, $enforce_r(cp_v)$ refers to the read access permission enforced by the `Content Provider` cp_v. In our approach, for each `Content Provider` component, we add two columns in the permission domains of the MDM: one for the read access permission and the other one for the write access permission. For example, each permission domain in the MDM illustrated in Figure 6.5 contains two permissions for the `CallsDB` component, $C3_R$ for the read permission and $C3_W$ for the write permission.

Security Rule 6 (Content Pollution)**.** *Let cp_v be a vulnerable* `Content Provider` *that does not enforce a write access permission, c_m be a malicious component that changes*

(inserts, updates, or deletes) the stored data in cp_v. In the content pollution attack, c_m can inappropriately manipulate the sensitive data stored in cp_v.

$$\exists manipulate(c_m, cp_v) \mid enforce_w(cp_v) = \emptyset$$

In Definition 6, $enforce_w(cp_v)$ refers to the write access permission enforced by the `Content Provider` cp_v.

It is worth mentioning that all violations to the determined LP architecture are recorded and accessible to the security architect through an Android app that we have developed, not shown in Figure 6.1 to reduce the clutter in the figure. This app allows a security architect to understand the running system and adjust the architecture as needed.

6.2.2 Change Impact Analysis

This analysis consists of two steps: (1) determining the impacted parts of the MDM and (2) identifying the subset of the security analysis rules, formally specified in Section 6.2.1, that need to be considered. More specifically, in step (1), *Incremental Security Analyzer* determines the affected parts of the system by calculating $\Delta MDM_e = MDM_{t2} - MDM_{t1}$, where $t2$ is a time after an event e and $t1$ is a time before e.

Each cell in ΔMDM has a value of either -1, 0, or 1. -1 means a relationship in the previous system has been removed after e, e.g., e is the revocation of a permission. 0 means there is no change in that relationship before and after e. 1 indicates that a new relationship is introduced due to e. For example, e may be the introduction of a new communication between two components appearing at runtime due to installing a new app, updating an existing app, dynamically loaded code, or execution of obfuscated code. From the affected relationships, *Incremental Security Analyzer* determines the impacted domains. Applying

ΔMDM to our running system, described in Section 5.2, reveals that the communications in rows 9 and 10 have been added to the system which belong to the *explicit* and the *implicit* communication domains. Note that since `BrainTeaser` does not have `Content Providers` nor does it interact with other `Content Providers` in the example system; as a result, both the `Data Access` and `Data Manipulation` domains are not affected, and as such will not be considered for the incremental analysis.

In step (2), *ISA* determines the subset of the security rules that need to be considered in light of the affected domains. To that end, *ISA* uses Table 6.3, which is a lookup table that maps each *Security Analysis* with the *Involved Domains* in that analysis. This table also shows the security analyses that need to be performed when a specific domain changes. For example, if the *EXPLICIT* domain changes, then *Incremental Security Analyzer* needs to perform only 3 security analyses instead of all 6 security analyses.

In our running example where only the *explicit* and the *implicit* domains have been changed after installing `BrainTeaser`, Table 6.3 indicates that the security posture of the system should be checked against the following security attacks: *Intent Spoofing*, *Unauthorized Intent Receipt*, *Privilege Escalation*, and *Identical Custom Permission*. Therefore, *Incremental Security Analyzer* applies the security rules 1, 2, 3, and 4 (see Section 6.2.1) to all interactions in rows and columns 9 and 10. Running these rules, mainly rule 3 on the communication between `Qgenerator` and `Share`, reveals that the implicit communication in row 10 and column 8 of Figure 6.5 is vulnerable to privilege escalation attack.

6.3 Policy Synthesizer

The *Policy Synthesizer* efficiently creates *context-sensitive* security policies to be executed at runtime that capture (1) the initial LP architecture of the system (recall Section 6.1.1) and (2)

Table 6.3: Security analyses lookup table.

Security Analysis	**Involved Domain(s)**
Intent Spoofing	Explicit Implicit
Unauthorized Intent Receipt	Implicit
Privilege Escalation	Explicit Implicit Granted Usage Enforcement
Identical Custom Permission	Explicit Implicit Enforcement Granted
Passive Data Leak	Data Access Read Permission
Content Pollution	Data Manipulation Write Permission

the security vulnerabilities determined by the *Incremental Security Analyzer* (recall Section 6.2.2). The created security policies, in our approach, follow the Event-Condition-Action (ECA) rules paradigm suitable for rapid evaluation as the system executes. SALMA creates ECA rules that, based on a particular system context, will be executed to prevent security threats. More specifically, SALMA creates ECA rules to prevent the communication between two components that are either (1) based on the LP architecture, not allowed to communicate with one another or (2) involved in an identified security vulnerability. Moreover, SALMA creates ECA rules to prevent a component from accessing protected system resources in case that component is not granted sufficient permissions.

6.3.1 Efficiently Generating ECA Rules

Event-condition-action (ECA) rules allow the system to automatically perform actions in response to events given that the stated conditions hold. Each ECA rule reads as follows: "when an event occurs, check the condition, if it holds, execute the action". ECA rules make the system efficiently adapt while the rules are stored in a single rule base instead of encoding them in many modules, thus improving the maintainability and the manageability of the system. ECA rules have been widely used in the literature, including self-adaptive systems [130, 76, 142], active database [217, 166], implementing business processes [53, 86, 81], and in the web technology [165, 75].

Since the identified LP architecture will be stored and monitored in resource-constrained mobile devices in terms of a set of ECA rules, it is significantly important for such rules to be efficient in a way that would minimize the number of required ECA rules. A naïve approach for generating ECA rules that capture an LP architecture of n rows and m columns would result in $n \times m$ ECA rules, where each cell is captured by an ECA rule. However, such an approach results in the generation of a large number of rules, many of which are very similar.

SALMA generates ECA rules more efficiently. As for ICC ECA rules, i.e., the rules that capture the explicit and implicit communication domains of an LP architecture, if a component has no legitimate reason to communicate with any component of another app, SALMA generates only one ECA rule that entirely prevents that particular component from communicating with that app. This, in turn, reduces the number of generated ECA rules from the number of components in the target app to merely one ECA rule. Similarly, if no component of an app is allowed to communicate with any component of another app, SALMA generates just one ECA rule that prevents all components of the former app from communicating with components of the latter app. Generating ECA rules in this way not only reduces the number of generated rules but also makes the search process for an ECA rule

governing a specific component or a specific app faster. Once SALMA finds a coarse-grained ECA rule, i.e., a rule that restricts one app from communicating with another app, SALMA stops the search and executes the action specified in that ECA rule.

For example, the following ECA rule is produced, from the LP architecture shown in Figure 6.4, to prevent the `Qgenerator` component from communicating with the `LocTracker` component:

Event: $i \in ICC$ occurs
Condition: $i.senderPkg$ = `BrainTeaser` $\wedge$ $i.senderComp$ = `Qgenerator` $\wedge$ $i.receiverPkg$ = `StayHealthy`
Action: *prevent*

In the case of resource access ECA rules, i.e., ECA rules that capture the Permission Granted Domain, SALMA generates resource access ECA rules only for the granted permissions, i.e., ECA rules that capture only the "1"s in the Permission Granted Domain. It is worth mentioning that, in Android, it is possible for one permission to protect more than one system resource. In such a case, SALMA generates more than one resource access ECA rule per granted permission. For example, the `android.permission.READ_PHONE_STATE` permission is required to request `CARRIER_CONFIG_SERVICE` in order to access the carrier configuration values, and the same permission is required to request the `TELEPHONY_SERVICE` to access the `TelephonyManager`, which provides access to information about the telephony services on a device.

As a concrete example, the following ECA rule is produced, from the LP architecture shown in Figure 6.4, to grant `LocTracker` permission to access the *LOCATION* service:

Event: *resourceaccessrequest*

Condition: $requester = \texttt{ListMsgs} \wedge service = \texttt{Context.LOCATION\_SERVICE}$

Action: *allow*

SALMA further tries to minimize the disruption that the security policies may cause. For example, in the case of a potential privilege escalation attack, SALMA creates a security policy to prevent a vulnerable communication instead of revoking the escalated permission from the vulnerable app, as proposed in [184]. The later solution disrupts all components in the vulnerable app from using that permission which may stop crucial services provided by the disrupted components such as sending text messages or getting driving directions.

As a concrete example, since the communication between Qgenerator and `Share` is marked as potential privilege escalation attack, SALMA creates the following ECA rule.

Event: $i \in ICC$ occurs

Condition: $i.senderPkg = \texttt{BrainTeaser} \wedge i.senderComp =$

$\texttt{Qgenerator} \wedge i.receiverPkg = \texttt{StayHealthy} \wedge i.receiverComp = \texttt{Share}$

Action: *prevent*

6.4 Policy Enforcer

Policy Enforcer administers security policies at runtime through various effectors that we have added to the Android runtime environment, as shown in Figure 6.1. *Policy Enforcer* applies security policies by intercepting the ICCs (both the Intent-based and the URI-based communications) and the resource access transactions to check if they are allowed or not.

For Intent-based communication, *Policy Enforcer* can prevent or allow transactions. For the URI-based ICC transactions, *Policy Enforcer* can prevent a component from accessing or manipulating either (1) the entire `Content Provider` specified in the URI or (2) a specific table or file in that `Content Provider`

The *Policy Enforcer* component extends the capabilities of the Android framework by intercepting each ICC transaction passed to the *ActivityManager*—an Android component that administers the ICC transactions—to check whether the transaction is allowed to run or not. In case an ICC is prevented, *Policy Enforcer* records the transaction for further inspection by a security analyst.

For example, at runtime, when `Qgenerator` tries to communicate with `Share`, the Android framework passes the request to the *ActivityManager* which sends the ICC transaction's information (sender, receiver, and the Intent's attributes) to the *Policy Enforcer* component. After that, *Policy Enforcer* vets the ICC transaction in light of the stored ECA rules. If a matched ECA rule is found, *Policy Enforcer* prompts the *ActivityManager* to execute the associated action, i.e., *prevent* the communication in this particular example (recall Section 6.3.1).

As we explained in Section 2.2, components need permissions to access various system resources. Such system resources are accessed via the *Context* component, an Android component that holds information about the application environment and controls access to resources. SALMA modifies *Context* to extract information from each resource access request, and passes it to the *Policy Enforcer* to check whether the requester is allowed to access the requested service.

For instance, when the `Share` component try to track user's location, it tries to obtain a handle to the *LocationManager* service, Line (7) of Listing 5.1. The Android framework dispatches the request to the *Context*, which then sends the request to the *Policy Enforcer*.

Upon receiving the resource access request, *Policy Enforcer* checks it against the ECA rules and performs the corresponding action, i.e., *allows* the request in this particular case (recall Section 6.3.1).

6.5 Implementation Details

As depicted in Figure 6.1, SALMA is a self-protecting Android software system that automatically determines the LP architecture of an Android system and maintains it synchronized with the running system. After that, when a change in the system occurs, SALMA incrementally and efficiently analyzes the security posture of the system and creates security policies to mitigate ICC attacks at runtime. We have implemented SALMA and its constituent components for our experiments and make it available online for reproducibility and reuse purposes [50].

As described earlier, the architecture extraction capability was built on top of several prior static program analysis tools [160, 66, 71]. Each tool provides specific information that SALMA uses to tailor the LP architecture. The derived LP architecture and results of analysis are stored in a comma separated values (CSV) file. The implementation of SALMA consists of more than 10,000 lines of code (LOC), not counting the existing tools on which it relies.

The monitoring and the enforcement capabilities are implemented on top of the Android Open-Source Project (AOSP) [29] version 6 (Marshmallow), API level 23. AOSP is the open-source repository for the Android system maintained by Google. The enforcement mechanism introduced a new package in the Android runtime environment. We also modified other components such as *ActivityManager*, *ContextWrapper*, *ContentProvider*, and *PackageManager*. The total framework changes account for approximately 600 LOC. The

changes were made such that any existing Android app could continue to run in our version of Android runtime environment without modification. Moreover, our modifications to the Android version 6 are not restricted only to this version and can be applied without technical difficulties to any Android version.

We built the modified AOSP on an Ubuntu server with a 64-core AMD processor and 264GB RAM. It took about an hour to complete the build process. We have successfully installed the modified Android system image on a Nexus 5X phone and an Android emulator using Android Fastboot tools [45] and Android debug bridge [42].

To make the MDM implementation more efficient and scalable, we have developed a *Dynamic Multiple-Domain-Matrix* (*D-MDM*), which is our enhanced variant of the traditional *MDM*. Using a D-MDM, we are able to load and analyze a subset of the domains that are relevant to the particular analysis at hand.

Finally, since the Android framework components do not provide graphical user interface (GUI), we implemented an Android app that provides a GUI to (1) list the vulnerabilities in the system, (2) allow a user to optionally add new security policies, and (3) allow a user to visualize the synchronized model of an Android system in *realtime* using FireBase[49], a Google realtime NoSQL cloud-hosted database service.

Chapter 7

Experimental Evaluation

Our evaluation addresses the following research questions:

- **RQ1. (Attack Surface Reduction)** How effective is SALMA in reducing the attack surface of Android systems and aiding the architect with understanding their security posture?
- **RQ2. (Efficiently Generating ECA Rules)** How efficient is SALMA in generating ECA rules that capture the determined LP architecture?
- **RQ3. (Incremental Analysis Efficiency)** How efficient is SALMA at incrementally analyzing the security posture of Android systems compared to a complete analysis approach?
- **RQ4 (Disruption)** How effective is SALMA at reducing the unnecessary disruption caused by the enforcement of security policies to prevent permission-induced ICC attacks?
- **RQ5. (Attack Detection and Prevention)** How effective is SALMA at detecting and preventing security attacks in real-world apps?

- **RQ6. (Performance)** What is the performance of SALMA?

We constructed datasets of benign, malicious, and vulnerable Android apps as shown in Figure 7.1(a). The benign dataset is a collection of 370 apps, randomly selected from the Google Play store, the official Android app market [35].

To prevent any bias in the results, we did not use any particular criteria, such as high ranking or high downloads, in selection of the Google Play apps. Therefore, these apps vary in terms of their 5-star ranking, as depicted in Figure 7.2 (a), as well as their number of downloads, as depicted in Figure 7.2 (b).

The second dataset is a collection of 389 vulnerable apps identified in prior literature [144]. Finally, the malware dataset contains 225 apps obtained from various malware repositories [233, 20, 153]. Figure 7.1(b) illustrates the distribution of apps from various malware repositories that were used in our experiments.

7.1 RQ1. Attack Surface Reduction

By reducing the privileges granted to software components, SALMA focuses its analysis effort on a narrowed set of interactions and it also helps security architects in comprehending the security posture of Android systems. To evaluate the degree to which SALMA reduces the attack surface of Android systems, we ran SALMA on 10 bundles of apps, each containing 30 non-overlapping apps. Each bundle contains apps randomly selected from the app datasets as follows: 24 benign apps, 3 vulnerable apps, and 3 malicious apps. Figure 7.3 depicts a histogram of the Google Play categories of the benign apps. Table 7.1 shows the structure of the bundles.

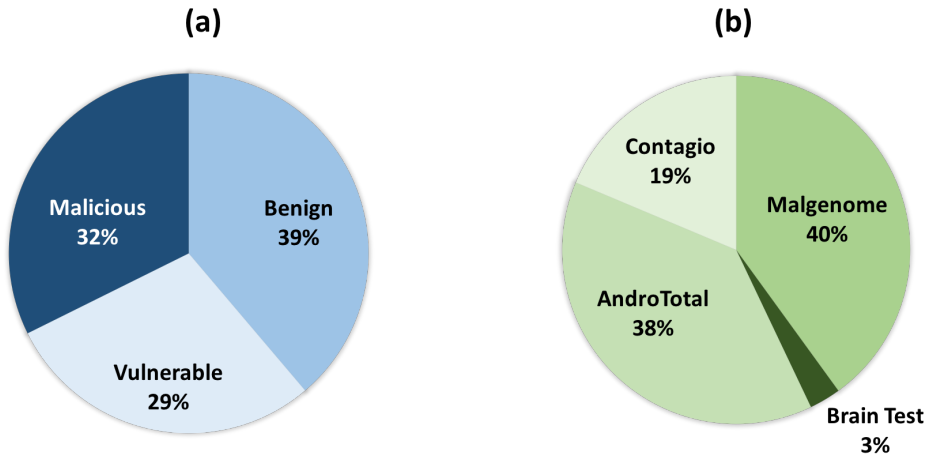


Figure 7.1: (a) Distribution of the entire experimental subjects across various repositories from which the subject apps are downloaded; (b) distribution of apps from various malware repositories that were used in our experiments.

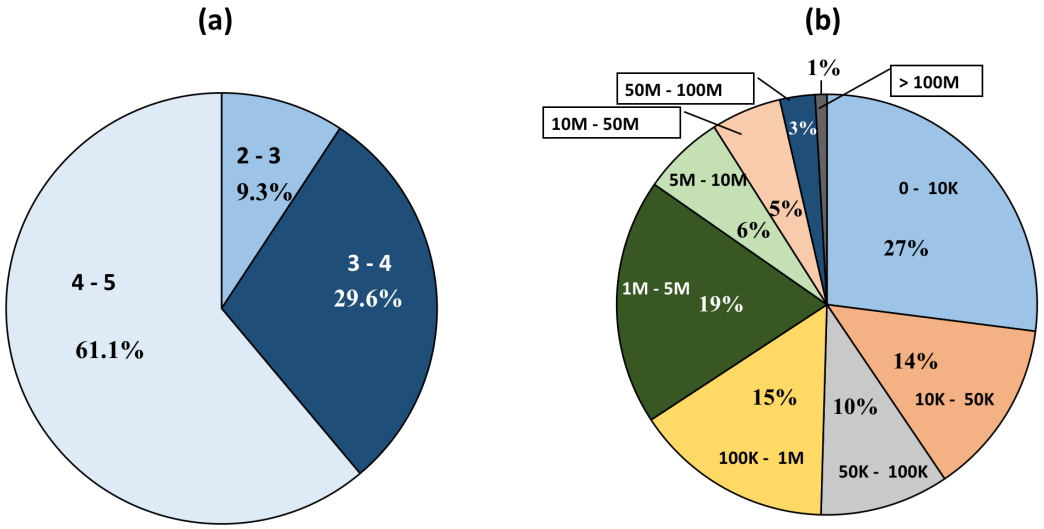


Figure 7.2: The popularity of the Google Play apps in terms of their (a) 5-star ranking and (b) number of downloads as of June of 2018.

Table 7.1: Summary of app bundles, each bundle contains 30 apps.

Bundle	Components	Intent		Intent
		Explicit	Implicit	Filter
Bundle 1	306	344	79	176
Bundle 2	432	468	379	287
Bundle 3	422	574	212	200
Bundle 4	449	348	370	511
Bundle 5	353	304	277	292
Bundle 6	541	890	476	4919
Bundle 7	562	412	38	324
Bundle 8	362	417	267	242
Bundle 9	265	180	98	166
Bundle 10	421	322	1231	185
Average	411.3	425.9	342.7	730.2
Avg. (per app)	13.7	14.2	11.4	24.3

Table 7.2: The Original and the LP architecture obtained from running SALMA over the bundles.

Bundle	Communication Domain			Permission Granted Domain		
	Original	LP	Reduction (%)	Original	LP	Reduction (%)
Bundle 1	29,031	42	99.86	1,642	178	89.16
Bundle 2	78,237	625	99.20	2,954	143	95.16
Bundle 3	65,709	173	99.74	2,510	109	95.66
Bundle 4	80,372	205	99.74	4,234	146	96.55
Bundle 5	56,868	345	99.39	1,536	81	94.73
Bundle 6	85,556	661	99.23	4,461	329	92.63
Bundle 7	82,863	137	99.83	1,577	109	93.09
Bundle 8	50,208	250	99.50	1,946	92	95.27
Bundle 9	25,817	129	99.50	1,568	57	96.36
Bundle 10	50,001	74	99.85	2,386	127	94.68
Average	60,466.2	264.1	99.58	2,481.4	137.1	94.33
Avg. (per app)	2,015.5	8.8	99.56	82.7	4.6	94.47

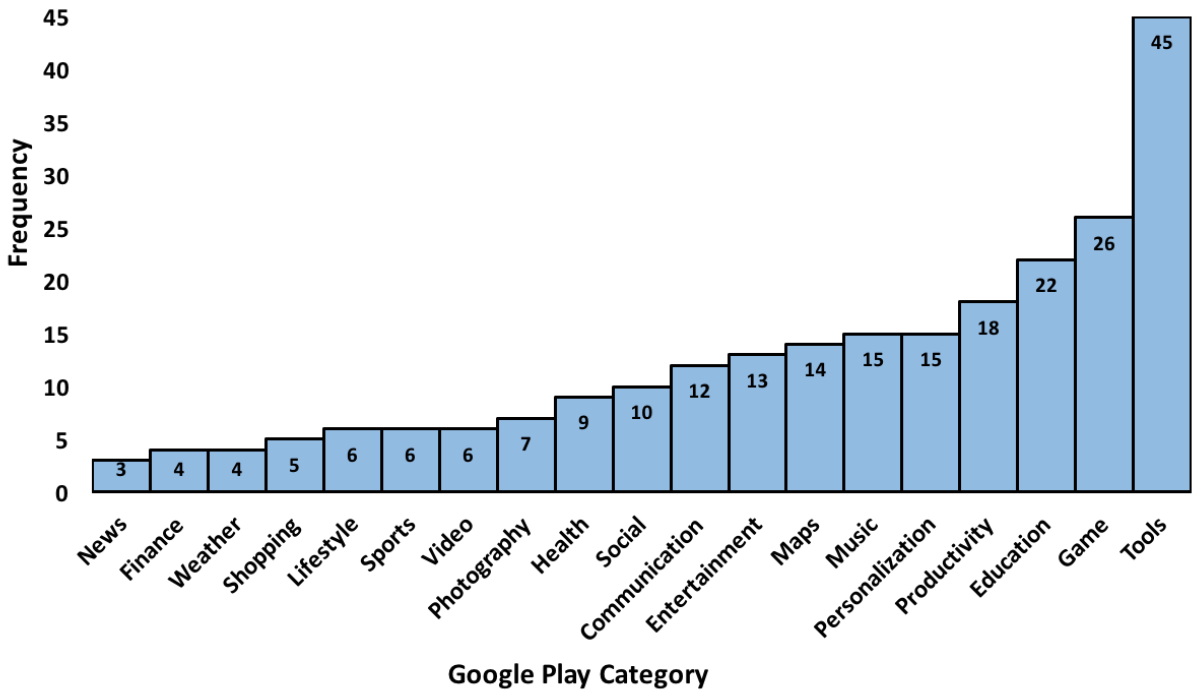


Figure 7.3: Histogram of Google Play categories.

Table 7.2 shows the number of entries in the Communication Domains as well as the Permission Granted Domain for both the Original and LP architectures. To measure the degree to which SALMA reduces the attack surface of Android systems, we used Equation 6.1. For example, in bundle 1, the LP architecture contains 42 inter-app communication (IAC) and 178 resource access permissions, whereas the Original architecture contains 29,031 IAC and 1,642 resource access privileges. On average, across all bundles, 99.56% of IAC and 94.47% of resource access privileges are reduced.

Recall from Section 6.2 that SALMA analyzes the initial architecture and pinpoints potential ICC attacks including privilege escalations, unauthorized Intent receipts, and Intent spoofing attacks. Tables 7.3 shows the number of potential privilege escalation ICC attacks, in both the Original and LP architectures. For example, in bundle 5 in Table 7.3, the Original architecture contains 26,914 possible privilege escalation attacks, whereas the LP architecture contains only 2 such attacks that need investigation. On average, an analyst needs to verify

14 potential privilege escalation security issues for a bundle of 30 apps using our approach. In fact, in the case of bundles 1 and 4 in Table 7.3, all potential privilege escalation attacks are already resolved with the LP architecture, eliminating the need for further investigation.

Similar patterns can be observed for Intent spoofing. Tables 7.4 shows the number of potential Intent spoofing ICC attacks, in both the Original and LP architectures. For example, in bundle 10, the Original architecture contains 2,015 potential Intent spoofing ICC attacks, whereas the LP architecture contains only 1 potential Intent spoofing attack that needs investigation. On average, an analyst needs to investigate 28 potential Intent spoofing for a bundle of 30 apps using our approach.

In a similar fashion, Tables 7.4 shows the number of potential unauthorized Intent receipt ICC attacks, in both the Original and LP architectures. For example, in bundle 10, the Original architecture contains 214 potential unauthorized Intent receipt ICC attacks, whereas the LP architecture contains only 3 potential unauthorized Intent receipt attacks that need investigation. On average, an analyst needs to investigate 8 potential unauthorized Intent receipt for a bundle of 30 apps using our approach.

Note that an analyst needs to verify less than 2 security issues per app on average. Even in some cases, such as in bundle 1, all potential ICC attacks are already resolved with the LP architecture, entirely eliminating the need for further investigation.

The results confirm the effectiveness of our approach in reducing the attack surface and hence reducing the effort required to assess the security properties of an Android system.

Table 7.3: Summary of Privilege Escalation ICC attack surfaces in both Original and LP architectures across app bundles.

Privilege Escalation			
Bundle	Original	LP	Reduction (%)
Bundle 1	25,944	0	100.00
Bundle 2	35,601	110	99.69
Bundle 3	22,721	2	99.99
Bundle 4	33,551	0	100.00
Bundle 5	26,914	2	99.99
Bundle 6	24,745	2	99.99
Bundle 7	15,503	1	99.99
Bundle 8	27,663	14	99.95
Bundle 9	19,428	8	99.96
Bundle 10	16,953	3	99.98
Average	24,902.3	14.2	99.94
Avg. (per app)	498.0	0.3	99.94

Table 7.4: Summary of Intent Spoofing ICC attack surfaces in both Original and LP architectures across app bundles.

Intent Spoofing			
Bundle	Original	LP	Reduction (%)
Bundle 1	2,242	0	100.00
Bundle 2	1,980	65	96.72
Bundle 3	3,132	0	100.00
Bundle 4	4,020	57	98.58
Bundle 5	12,402	24	99.81
Bundle 6	1,416	17	98.80
Bundle 7	1,077	1	99.91
Bundle 8	6,283	115	98.17
Bundle 9	4,638	4	99.91
Bundle 10	2,015	1	99.95
Average	3,920.5	28.4	99.28
Avg. (per app)	130.7	0.9	99.28

Table 7.5: Summary of Unauthorized Intent Receipt ICC attack surfaces in both Original and LP architectures across app bundles.

Unauthorized Intent Receipt			
Bundle	Original	LP	Reduction (%)
Bundle 1	297	0	100.00
Bundle 2	204	21	89.71
Bundle 3	299	7	97.66
Bundle 4	599	4	99.33
Bundle 5	1,646	7	99.57
Bundle 6	33	24	27.27
Bundle 7	78	0	100.00
Bundle 8	297	4	98.65
Bundle 9	371	10	97.30
Bundle 10	214	3	98.60
Average	403.8	8	98.02
Avg. (per app)	13.5	0.3	98.02

Table 7.6: Comparing the number of generated ECA rules between SALMA and the Naïve approach.

Bundle	Communication ECA rules			Pemission granted ECA rules		
	Naïve	SALMA	Improvement (%)	Naïve	SALMA	Improvement (%)
Bundle 1	93,636	1,035	98.89	1,917	211	88.99
Bundle 2	186,624	1,534	99.18	3,573	257	92.81
Bundle 3	178,084	893	99.50	3,094	115	96.28
Bundle 4	201,601	1,416	99.30	5,556	161	97.10
Bundle 5	124,609	1,238	99.01	1,840	99	94.62
Bundle 6	292,681	1,687	99.42	5,593	344	93.85
Bundle 7	315,844	1,027	99.67	2,046	151	92.62
Bundle 8	131,044	1,039	99.21	2,307	92	96.01
Bundle 9	70,225	1,051	98.50	1,964	69	96.49
Bundle 10	177,241	1,069	99.40	2,794	172	93.84
Average	177,159	1,199	99.21	3,068	167.10	94.26

7.2 RQ2. Efficiently Generating ECA Rules

Table 7.6 compares the numbers of generated ECA rules by SALMA and the Naïve approach (recall Section 6.3.1). For example, in bundle 1, the Naïve approach would generate 93,636 ICC ECA rules, whereas SALMA generates 1,035 ICC ECA rules showing more than 98% reduction in the number of rules that need to be monitored. On average, for an Android system with 30 apps, the Naïve approach would generate 177,159 ICC ECA rules, whereas SALMA generates 1,199 ICC ECA rules to capture the communication domains in the LP architecture.

Similarly, the Naïve approach would generate 1,917 resource access ECA rules for bundle 1, whereas SALMA generates 211 resource access ECA rules for the same bundle. On average, for an Android system with 30 apps, the Naïve approach would generate 3,068 resource access ECA rules, whereas SALMA generates 167 resource access ECA rules to capture the Permission Granted domain.

The results presented in Table 7.6 confirm the efficiency of SALMA in generating ECA rules to capture an LP architecture and hence reducing the time required to validate components' communications and resource access requests at runtime.

7.3 RQ3: Incremental Analysis Efficiency

To measure the efficiency of SALMA's incremental analysis, we compared the performance of SALMA with DELDroid [123, 124], a prior approach that similar to our work analyzes the architecture of an Android system for ICC vulnerabilities and enforces the determined architecture at runtime. However, unlike SALMA, DELDroid is not capable of continuous monitoring and incremental analysis of an evolving Android system.

Figure 7.4 contains box-and-whisker plots comparing the analysis time of each approach as Android apps are added to the system. We started with an Android system of 120 apps and added one app at a time until the system contained 150 apps. We randomly selected 120 apps from the benign dataset, 15 apps from the vulnerable dataset, and 15 apps from the malicious dataset.

Every time an app is added to the system, SALMA incrementally analyzes the system whereas DELDroid reanalyzes the entire system. As illustrated in Figure 7.4, the analysis time of SALMA takes, on average, 2 seconds to incrementally analyze an Android system whenever a new app is installed. On the other hand, DELDroid takes, on average, 75 seconds.

Figure 7.5 compares the analysis time of each approach with a decreasing number of apps. We started with a bundle of 150 apps, then we removed one app at a time until the system contained 120 apps. The average analysis time of SALMA is 0.2 seconds while the average analysis time of DELDroid is 35.3 seconds. Note that Figures 7.4 and 7.5 show the analysis

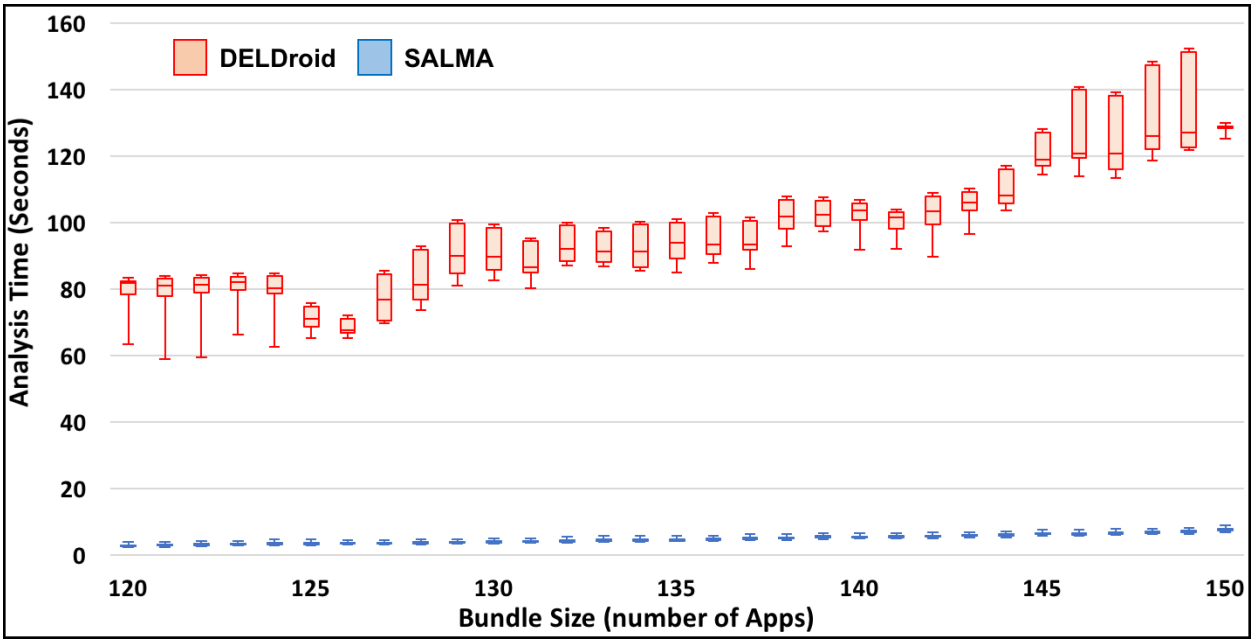


Figure 7.4: The analysis time of SALMA and DELDroid as Android apps are added to the system.

results of adding/removing 30 apps, however, the project's website [50] contains the analysis results of an experiment of adding/removing 80 apps.

Due to the use of code obfuscation and dynamic class loading in Android apps, not all communications can be discovered using static analysis tools. As a result, some communication appears only at runtime, e.g., a new explicit or implicit communication. In such scenarios, SALMA also incrementally reanalyzes the security posture of the system to determine if the new communication poses any threat to the system. If so, SALMA prevents the new communication. In addition to `ADD_APP` and `REMOVE_APP`, Figure 7.6 compares the efficiency of SALMA and DELDroid with respect to other system events mentioned in Table 6.2. SALMA takes, on average across all events, 1.6 milliseconds while DELDroid takes, on average across all events, 63.8 seconds.

Overall, these results corroborate the efficiency and the scalability of SALMA in incrementally analyzing Android systems.

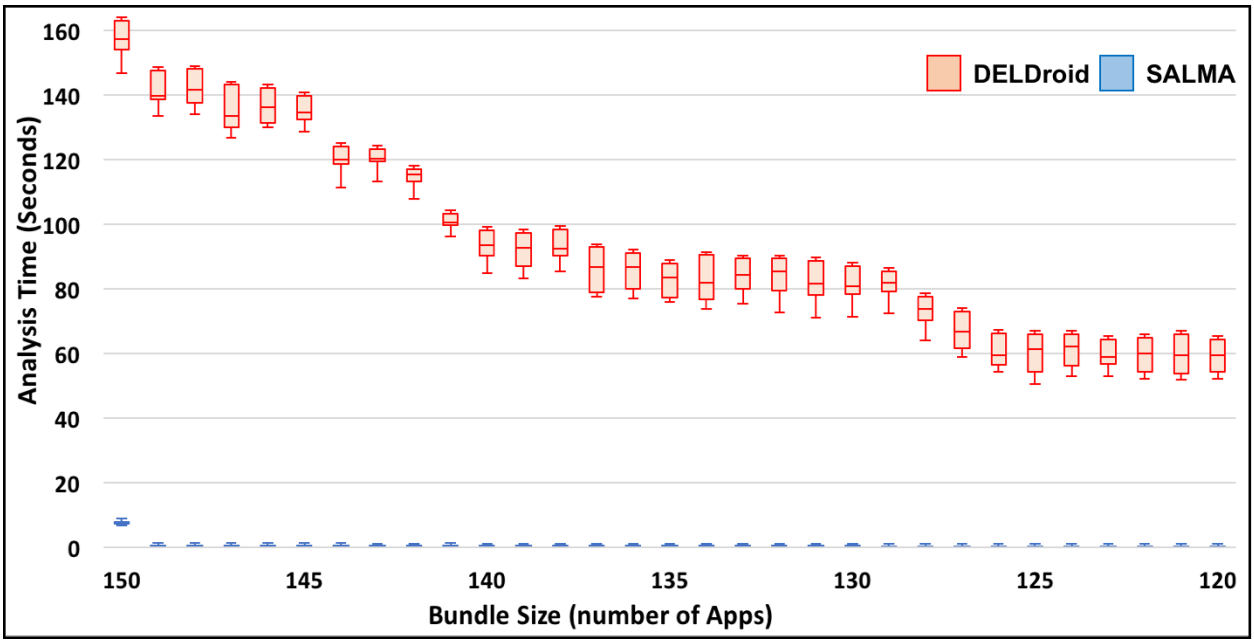


Figure 7.5: The analysis time of SALMA and DELDroid as Android apps are removed from the system.

7.4 RQ4: Disruption

Enforcing security policies at runtime by preventing permission-induced ICC attacks may disrupt benign behaviors of an app. Permission-induced attacks are security breaches enabled by permission misuse, i.e., privilege escalation, identical custom permissions, content pollution, and passive data leaks. Preventing permission-induced attacks can be applied at install-time or runtime [106]. Install-time approaches, such as Kirin [104], prevent the installation of vulnerable apps. Runtime prevention approaches can either (1) prevent only the malicious communication whenever it occurs, as performed in SALMA, DELDroid [124], SEALANT [143], and SEPAR [72]; or (2) revoke permissions of vulnerable apps at runtime, as in TERMINATOR [184] and AppGuard [68].

For this experiment, we analyzed a bundle of 150 apps, the apps used in RQ3, and found that 40 apps are vulnerable to various permission-induced attacks. We then computed the *disruption* in each vulnerable app caused by the enforcement of the various security policy

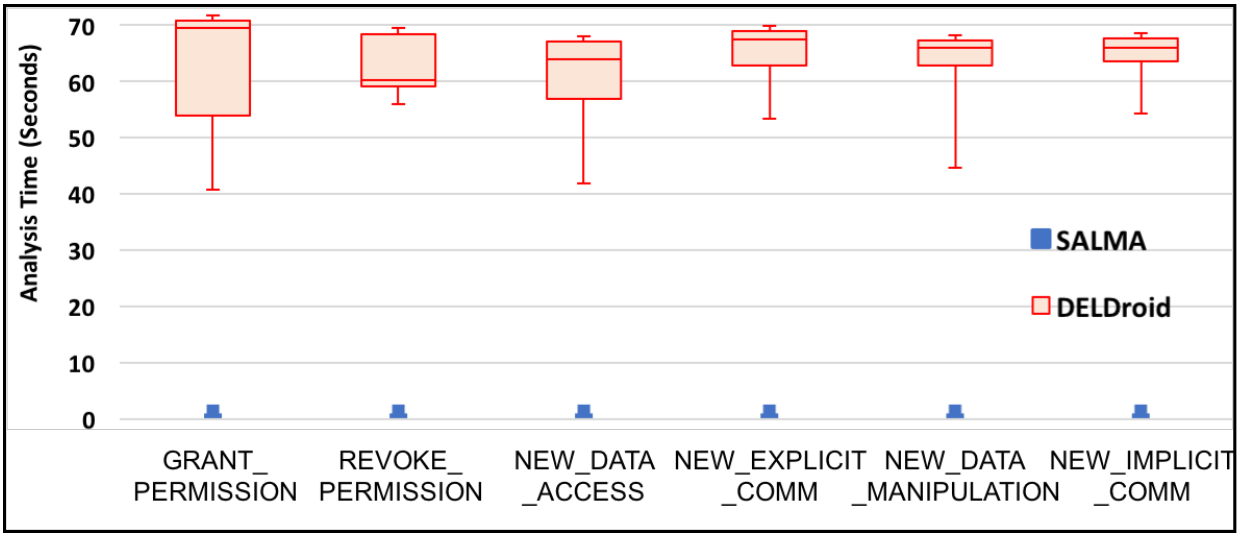


Figure 7.6: The analysis time of SALMA and DELDroid with respect to the significant system events mentioned in Table 6.2 other than ADD_APP and REMOVE_APP events.

mechanisms discussed earlier. Disruption of an app a is computed using the following equation:

$$disruption(a) = \frac{|comps_{disr}(a)|}{|comps_{tot}(a)|} \times 100$$

Where $comps_{disr}(a)$ is the set of components in app a that are disrupted and $comps_{tot}(a)$ are the set of all components in app a. We consider a component c to be disrupted if c uses a permission p involved in a permission-induced attack, since c will be unable to provide its full services if p is revoked.

As an example, consider an app a_v with 5 components where 3 of its components use permission p to provide their services. One component using p is vulnerable to a privilege-escalation attack. In this case, to protect the user, the install-time approaches prevent the installation of a_v, disrupting all of its components, i.e., $disruption(a_v) = 100\%$. On the other hand, approaches that revoke permission will revoke p to prevent the attack, resulting in 60% disruption of that app, i.e., 3 components will not be able to provide their full services due to the lack of the required permission p. However, SALMA, which only prevents malicious

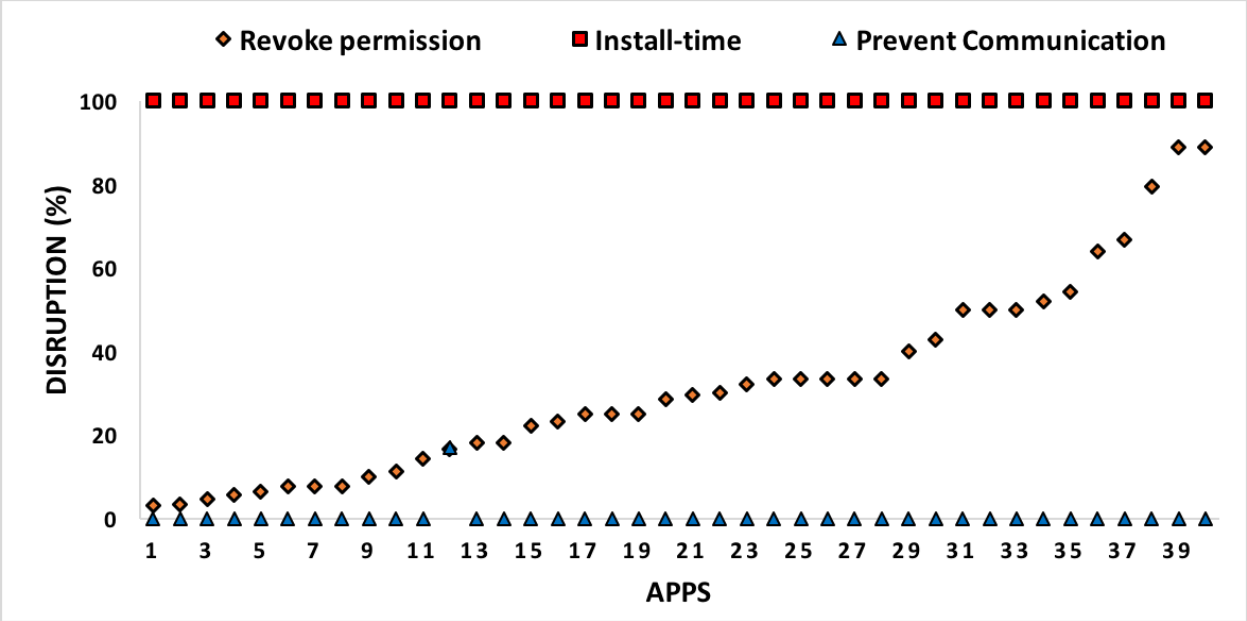


Figure 7.7: Disruption results for each app

communication when it occurs, results in 0% disruption, since all components will be able to provide their full services while keeping the system protected.

Figure 7.7 compares the three different permission-induced prevention mechanisms. The diagram shows that SALMA has 0.4% disruption, meaning that SALMA does not disturb components from providing their services except in one identical custom permission case. In that case, SALMA created a security policy to revoke a custom permission from the malicious app so it will not be able to access the vulnerable app. On the other hand, the install-time approach performs the worst (100%), as it does not allow installation of a vulnerable app. Finally, revoking permissions at runtime to prevent permission-induced attacks would result, on average per app, in 32% disruption. Meaning that, on average, 32% of the components in a vulnerable app will not be able to provide their full services due to the lack of required permissions even though some of these components are not vulnerable or involved in any vulnerability. Moreover, revoking permissions from apps at runtime lead to crashes or unexpected behaviors due to inappropriate handling of dynamic permissions in Android [185].

DELDroid, SEPAR, SEALANT, and SALMA all attempt to prevent malicious communication whenever it occurs. However, unlike SALMA, the other three approaches assume that all permissions are granted to all apps indefinitely. This assumption increases those approaches false positives which, in turn, increases unnecessary disruption. For example, a privilege-escalation vulnerability is not exploitable unless the escalated permission is granted to the vulnerable app. However, the three approaches prevent vulnerable communication at all times, while SALMA prevents vulnerable communication only when the system is at risk, i.e., the permission is granted to the vulnerable app.

7.5 RQ5: Attack Detection and Prevention

To evaluate SALMA's ability to detect and prevent security threats, we conducted a thorough evaluation using malicious and vulnerable real-world apps with known security attacks, and compared the detection and prevention results of SALMA with state-of-the-art approaches. We included state-of-the-art approaches that are (1) publically available, (2) provide detection and prevention mechanisms, and (3) extend the Android framework. To that end, we included DELDroid [123, 124], SEPAR [72], and SEALANT [143]. DELDroid determines the least-privilege architecture of an Android system and enforces it at runtime. SEPAR provides an automatic scheme for formal synthesis and enforcement of Android ICC security policies. SEALANT is a technique that combine static analysis with dynamic monitoring to detect security vulnerabilities in Android apps and prevent ICC attacks.

To conduct this experiment, we used 188 malicious and vulnerable open-source apps for which the steps and inputs required to create the attacks were known and documented. These apps have been used in the evaluation of the included approaches. To validate the attacks, we manually reviewed the code and affirmed the existence of security issues. In total, the subject apps contain 94 ICC attacks where 45 of them are hidden attacks, i.e., the malicious

code is not part of the apps' bytescode but instead is loaded at runtime using the dynamic class loading feature as described in [173], and the rest (49 attacks) are not hidden attacks, i.e., the malicious code is part of the apps' bytecode. We ran each approach on the subject apps, then deployed the apps on the Android environment, and manually exercised all known attacks. We report the number of detected and prevented attacks for each approach.

The *Attack Detection* column in Table 7.7 show the evaluation results of each approach for detecting the security attacks. For example, SALMA and DELDroid detected all of the 20 privilege-escalation instances that are not dynamically loaded, i.e., not hidden attacks, whereas SEPAR and SEALANT detected only 12 and 14 attacks, respectively. According to Table 7.7, SALMA is able to detect all 94 attacks, including the hidden attacks, with no false positives or false negatives, while the detection rate of the other approaches ranges from 16% to 30%.

Given the reliance of the included approaches on static program analysis to detect security risks, all of them are unable to detect hidden attacks launched via dynamically loaded code, see the gray area in Table 7.7. However, since SALMA incrementally analyzes the security posture of the system in response to system changes, i.e., new inter-app communications added at runtime as explained in Section 7.3, SALMA is able to detect these attacks at runtime.

The *Attack Prevention* column in Table 7.8 shows the evaluation results of each approach for thwarting the security attacks at runtime. SALMA is able to prevent all security attacks in Table 7.8 at runtime while the prevention rate of the other approaches ranges from 15% to 55%. Interestingly, DELDroid is able to prevent some of the ICC attacks that it did not detect, because it prevents all communications that are not part of the statically determined architecture.

Table 7.7: The ability of SALMA in detecting security attacks compared to the state-of-the-art approaches.

Attack Type (Count)	Security Attack	# Attacks	**Attack Detection**			
			DELDroid	SEPAR	SEALANT	**SALMA**
Not hidden (49)	Intent Spoofing	3	3	3	2	3
	Unauthorized Intent Receipt	5	5	0	1	5
	Privilege Escalation	20	20	12	14	20
	Identical Custom Permission	7	0	0	1	7
	Content Pollution	7	0	0	0	7
	Passive Data Leak	7	0	0	0	7
Hidden (45)	Intent Spoofing	13	0	0	0	13
	Unauthorized Intent Receipt	2	0	0	0	2
	Privilege Escalation	9	0	0	0	9
	Identical Custom Permission	7	0	0	0	7
	Content Pollution	7	0	0	0	7
	Passive Data Leak	7	0	0	0	7
Total attacks		94	28	15	18	94
Detection Rate			30%	16%	19%	100%

Table 7.8: The ability of SALMA in preventing security attacks compared to the state-of-the-art approaches.

Attack Type (Count)	Security Attack	# Attacks	**Attack Prevention**			
			DELDroid	SEPAR	SEALANT	**SALMA**
Not hidden (49)	Intent Spoofing	3	3	3	2	3
	Unauthorized Intent Receipt	5	5	0	1	5
	Privilege Escalation	20	20	12	14	20
	Identical Custom Permission	7	0	0	1	7
	Content Pollution	7	0	0	0	7
	Passive Data Leak	7	0	0	0	7
Hidden (45)	Intent Spoofing	13	13	0	0	13
	Unauthorized Intent Receipt	2	2	0	0	2
	Privilege Escalation	9	9	0	0	9
	Identical Custom Permission	7	0	0	0	7
	Content Pollution	7	0	0	0	7
	Passive Data Leak	7	0	0	0	7
Total attacks		94	52	15	18	94
Prevention Rate			55%	16%	19%	100%

7.6 RQ6. Performance

We measured the execution time of running SALMA on the 10 bundles of apps shown in Table 7.1. These experiments were conducted on a MacBook Pro with 2.2 GHz Intel Core i7 processor and 16 GB DDR3 RAM. We repeated our experiments 33 times to achieve a 95% confidence interval.

Table 7.9 summarizes the results. On average, for an Android system with 30 apps, it takes less than 70 minutes to execute SALMA and obtain the ECA rules, but the great majority of this time is spent in the one-time effort of recovering the architecture of an Android system from its implementation artifacts. A less precise but more efficient forms of program analysis could be substituted for architecture recovery, at the expense of a higher rate of false positives. Since the security analysis is realized in terms of a set of mathematical operations on a numerical matrix, it takes, on average, 2 milliseconds only to analyze the architecture.

After determining the initial LP architecture of an Android system, SALMA incrementally determines the architecture of the running system. Table 7.10 shows the performance of SALMA in merging an app to the runtime architectural model as well as removing an app from the runtime architectural model. Note that, the performance time reported in the *Merge app to the model* column of Table 7.10 does not include the time required to statically analyze an app, which takes, on average, 2.3 minutes. To further improve SALMA's efficiency at

Table 7.9: SALMA's offline performance to determine, analyze, and capture the initial LP architecture in ECA rules for an Android system with 30 apps.

	Recovery (min)	LP Determination (sec)	Analysis (sec)	ECA Rules (sec)
Average	69.5	1.61	0.002	0.45
Std Dev	2.7	0.69	0.001	0.99

Table 7.10: SALMA's runtime performance.

	Merge app to the model (second)	Remove app from the model (second)	Validating ICC trans. (second)
Average	0.024	0.026	0.0075
Std Dev.	0.027	0.028	0.0045

runtime, the static analysis time can be performed in advance without waiting for a user to install an app.

Finally, to evaluate the runtime overhead of SALMA, we measured the time it takes to check the ECA rules for an intercepted ICC transaction on a Nexus 5X phone. To that end, we created a script that sends 363 requests (e.g., start an app, click a button) to an Android system, simulating its use. Each request causes the system to perform an ICC of some sort. We found that, on average, the performance overhead is 7.73 milliseconds with 4.5 milliseconds standard deviation, i.e., an 4.99% performance overhead as depicted in Figure 7.8. In Figure 7.8, the green area shows the ICC transactions time whereas the red area shows the performance overhead. Most users cannot perceive delays of this magnitude, per Android development guidelines [46], and thus, we believe SALMA poses an acceptable overhead.

7.7 Threats to Validity

This section presents the threats to validity of our experimental setup and our evaluation results and the actions we have taken to mitigate these threats.

One threat to validity of our work is weather our results can be generalized to apps outside our study. To mitigate this threat, we obtained benign, vulnerable, and malicious apps from diverse sources. Benign apps vary across application domains (see Figure 7.3), application popularity (see Figure 7.2), and in terms of app size [50]. The size of the apps vary from

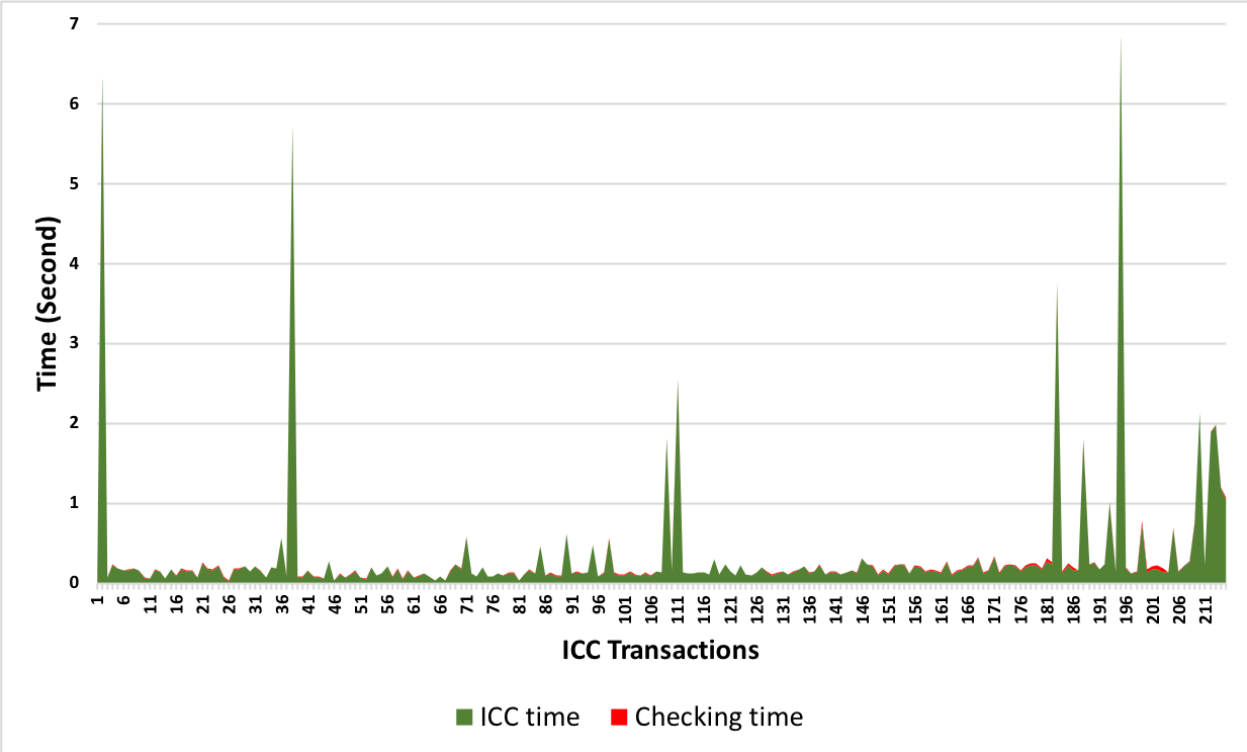


Figure 7.8: The performance overhead for validating ICC transactions.

1MB to 57MB as in the case of *Gemmy Lands* app [5]. Vulnerable apps have been discovered and verified in a previous study [144]. Similarly, our malicious apps drawn from repositories containing apps manually labeled as malicious by security experts.

Our selection of Nexus 5X phone to measure the runtime performance of SALMA can be seen as a threat to validity, since the runtime performance using another Android device might be different. However, since this device has been released in 2015, it is not the most advanced Android device. Therefore, we believe that the reported performance would be similar or even better using the currently available Android devices in the market.

Finally, the ability of SALMA in detecting and preventing security attacks depends on the dataset that we used as a ground truth with known security attacks. To reduce this threat and challenge SALMA, we did not use benchmarks with hand-crafted toy apps such as

DroidBench [65] or ICC-Bench [6], instead we used a dataset with real-world Android apps with known vulnerabilities created by experts from outside our research group.

Chapter 8

Related Work

A large body of research has focused on Android security. This chapter provides a discussion of the related efforts in light of our research.

8.1 The Effect of Code Obfuscation on Anti-malware Products

We divide previous work related to our study presented in Chapter 4 into four categories: (1) studies about similarity of a repackaged app with its original version, (2) obfuscation strategies for PC and desktop software, (3) obfuscation tools specifically designed for Android, and (4) studies about the effects of obfuscation on anti-malware products. In the remainder of this section, we discuss each of these areas of related work and conclude the section with the key differences between our study and the most similar related work.

Researchers have studied the similarity between original apps and repackaged apps. Huang et al. [128] used Androguard [15], an Android reverse-engineering framework for malware

analysis, to study the obfuscation resilience of repackaging detection algorithms. Faruki et al. [108] compared the performance of anti-malware products and Androguard's code similarity with AndroSimilar [109], their tool for detecting obfuscated apps. Crussell et al.[96] proposed DNADroid, a tool for detecting Android apps cloning. Similarly, Zhang et al. [224] proposed ViewDroid, a user interface based approach for detecting repackaged apps. Wang and Rountev describe an approach for determining which obfuscation tool was applied to an obfuscated app [213].

A few studies have considered the application of obfuscation strategies in the context of PC and desktop software. Collberg et al. produced a taxonomy of transformations for obfuscation with a focus on Java [94]. Collberg et al. [93] also implemented a tool called SandMark for evaluating the effectiveness of code obfuscation to protect Java-based software systems from piracy, tampering, and reverse engineering. Christodorescu and Jha [91] evaluated the resilience of anti-malware products against code obfuscation applied to Visual Basic programs and proposed a semantics-aware malware-detection algorithm in [92].

Previous work has produced a few obfuscation tools specifically designed for Android apps. Zheng et al. [229] proposed ADAM, a framework for obfuscating Android apps and testing them on anti-malware products. They evaluated ADAM's effectiveness for evading 10 anti-malware products on 222 transformed, malicious apps. Rastogi et al. [180] presented DroidChameleon, a tool for obfuscating Android apps, and assessed obfuscations on six apps from Android Malware Genome [233].

Another set of studies focused on the effects of obfuscations on anti-malware products, without proposing new obfuscation tools. Maiorca et al. [154] studied the effects of code obfuscated by a single tool on 13 anti-malware products. Pomilia [175] studied the performance of 9 anti-malware products on a dataset obfuscated using Allatori. Morales et al. [156] studied the resilience of 4 anti-malware products after transforming 2 viruses on Windows Mobile OS.

All the aforementioned previous work that have studied either obfuscation tools or the effects of obfuscation strategies on anti-malware products focus on a single obfuscation tool and a small number of anti-malware products and apps. None of these studies have performed a large-scale empirical study considering the effects that occur due to the concurrent utilization of anti-malware products, various obfuscation tools, and their supported obfuscation strategies. None of these studies assessed these effects on benign apps; the effects of combining obfuscation strategies; and the ability of obfuscation tools to produce valid, installable, and runnable apps.

8.2 Security Attack Detection

Numerous static analysis approaches have been proposed in the literature for detecting ICC attacks in Android systems [71, 140, 144, 89, 215, 151, 221, 161].

COVERT [71] presents an approach for compositional analysis of Android inter-app vulnerabilities. DidFail [140] introduces an approach for tracking data flows between Android components. Similarly, IccTA [144] leverages an Intent resolution analysis to identify inter-component privacy leaks. Amandroid [215] is a taint static analysis tool that builds Inter-component Data Flow Graph (ICDFG) and Data Dependency Graph (DDG) and use them for detecting Intent-based data leak and data injection. It does not analyze `Content Providers` or consider some calls such as `bindService` or `startActivitiesForResults`. CHEX [151] is a static analysis tool mainly to detect component hijacking, a type of unauthorized Intent receipt, vulnerabilities in apps. IccDetector [221] is an ICC-based malware detection in Android. It statically extracts ICC information and uses them as features to train a machine-learning classifier. Epicc [161] computes ICC call graph after retargeting the Dalvik bytecode to java bytecode using Dare [159]. FlowDroid [65] is another precise static taint analysis approach for Android apps. Chin et al. [89] discussed several ICC attacks that can

be achieved through receiving an Intent by unauthorized receipt or spoofing an Intent, and they have provided ComDroid, a tool that is meant to be used by developers to analyze their apps before releasing them. ScanDroid [113] is a data-centric static analysis tool for reasoning about the data flow in Android apps; it creates security specifications from the app's manifest file. More recently, LetterBomb [115] presents an approach for automatic exploit generation for vulnerabilities exposed in an Android app's Intent-based interface.

Other dynamic analysis tools also have been proposed to help detecting ICC vulnerabilities in Android apps such as TaintDroid [103] and CopperDroid [206]. TaintDroid [103] is a dynamic analysis tool that tracks data from sensitive sources to sensitive destinations and provide these information to users so they can understand how their sensitive data are being used by various apps. CopperDroid [206] is an automatic virtual machine introspection or VMI-based dynamic tool for reconstruction of Android malware behavior. It provides behavioral profiling of malicious Android apps.

Moreover, other researchers proposed hybrid approaches to facilitate detecting ICC vulnerabilities including SmartDroid [228], Dr.Android [134] and ProfileDroid [216]. SmartDroid [228] is a hybrid approach for detecting graphical user interface (GUI) actions that triggers behavior similar to a malicious behavior found in a previously known Android malware. Dr. Android [134] proposed a fine-grained permission system instead of the current coarse-grained permission system in Android. It requires modification to apps' implementation logic. ProfileDroid [216] proposed a design for profiling Android apps in four layers: static analysis, user layer, OS layer, and network layer.

Finally, Schmerl et al. [192] describe an architectural style for Android in ACME [118] that, among other capabilities, supports analysis of certain security properties. This work utilizes COVERT for performing security checking and hence they share the same limitation. Unlike SALMA, their work does not provide a mechanism for determining the LP architecture, nor does it provide any runtime enforcement mechanism.

While these research efforts are concerned with the analysis of information/permission leakage between Android apps, they do not really address the problem that we are addressing, namely the automated determination and the dynamic analysis and enforcement of least-privilege architecture in Android. Unlike SALMA, all of these approaches cannot detect ICC attacks conducted through dynamic class loading. Moreover, they do not generate nor enforce security policies, as performed by SALMA. SALMA, to our knowledge, is the first tool with this capability.

8.2.1 Security Attack Prevention

DELDroid [124, 123] is an approach that determines the least-privilege architecture of an Android system and enforces it at runtime. Similar to SALMA, DELDroid analyzes the architecture of an Android system for ICC vulnerabilities and modifies the Android platform to enforce the determined architecture. Unlike SALMA, DELDroid is a design-time solution that (1) does not change the derived architecture as the system evolves; (2) assumes that all permissions are granted to apps indefinitely, which increases disruption; and (3) assumes that all hidden communications are malicious, which further contributes to disruption.

Other approaches, such as [106, 104, 72, 143, 184], statically analyze Android apps and dynamically enforce security policies to prevent ICC attacks. Felt et al. [111] studied permission re-delegation security attacks (aka, privilege escalation) in mobile systems and web browsers; they showed the wide spread of this attack and provided an IPC inspection mechanism to prevent such attacks. IPC-Inspection is an OS mechanism for preventing privilege-escalation attacks by reducing the permissions assigned to an app when it communicates with an app having fewer privileges. Kirin [104] extends the application installer component of Android's middleware to check the permissions requested by applications against a set of security rules. These predefined rules are aimed to prevent unsafe combination of permissions that may

lead to insecure data flows. Kirin detects security vulnerabilities by only analyzing an app's configuration file and preventing the installation of vulnerable apps.

SEPAR [72] is a tool for automatic synthesis and enforcement of security policies allowing the end-users to safeguard the apps installed on their devices from ICC attacks. SEPAR's policy enforcement relies on the Xposed framework [52] that requires root access to the device. SEPAR [72] and SEALANT [143] rely on the analysis results generated by COVERT [71] to prevent ICC attacks at runtime. TERMINATOR [184] performs temporal analysis for preventing permission-induced ICC attacks. All of these tools do not update their models once the system changes. Moreover, unlike SALMA, these approaches cannot prevent malicious hidden behaviors.

While the above techniques rely on static program analysis to detect security risks and prevent them at runtime, another set of approaches leverage dynamic analysis techniques to detect and prevent security attacks [127, 100, 162, 82]. AppFence [127] prevents apps from exfiltration of data outside the device. Especially for advertisement components that ship a user's data outside the device. It allows a user to mock the sensitive data or block network transmission. Saint [162] extends the functionality of Kirin to allow for install-time permission assignment and their run-time use as specified in the policies provided by an app's developer. XManDroid [82] presents a solution for privilege-escalation attacks by restricting communication at runtime between applications that could lead to dangerous information flows based on Chinese Wall-style policies [79] (e.g., forbidding communication between an app with GPS privileges and an app with Internet access).

Kynoid [193] performs a dynamic taint analysis over a modified version of Dalvik VM. DeepDroid [211] presents an enforcement extensions based on dynamic memory instrumentation of system processes. ASM (Android Security Modules) [125] is a framework that provides a programmable interfaces for defining reference monitors for Android similar to the proposed reference monitors for Linux [157] and TrustedBSD [214].

While SALMA automatically analyzes the system and creates security policies to prevent ICC attacks, all of these dynamic analysis tools depend upon defining security policies by developers and they require modifications to apps' implementation logic. Defining security policies by developers are error prone and time consuming.

The research effort presented in this section share with ours the emphasis on dynamic enforcement of security policies. SALMA differs fundamentally in its emphasis on both providing an architectural solution and automatically adjusting the privileges at the architectural level as needed at runtime.

8.3 Enforcing the Least-Privilege Principle

The importance of enforcing the principle of least-privilege was introduced in the seminal work of Saltzer et al. [187], and is well recognized by many researchers. Notably, Scandariato et al. [191] lays the formal definition of the least-privilege violation and provides a technique to identify such violation in UML models. To the best of our knowledge, SALMA is the first solution capable of automatically recovering the architecture of an Android system to derive and enforce an LP variant of it.

The importance of limiting the privileges assigned to Android components have also been discussed in the literature [137, 198, 212, 195, 100, 200, 168]. Kantola et al.[137] described heuristics to allow the Android framework distinguish between inter-app and intra-app communications and hence detect any unintentional inter-app communication. Unlike SALMA, the proposed heuristics are not totally backward compatible with the existing apps and they require modifications by the apps' developers. Shehab and AlJarrah [198] proposed a policy-based approach for controling the access of different pages in web-based Android apps to mitigate potential attacks. However, unlike SALMA, their approach requires source

code and it is limited only to web-based multi-page apps generated by the Apache Cordova framework [44]. Wang et al. [212] proposed Compac, an approach for reducing the permissions assigned for third-party components in an app. Similar to Compac [212], FlexDroid [195] is an Android security model and isolation mechanism for limiting the permissions granted to third-party libraries.

Dietz et al. [100] presented Quire, an approach that adds two security mechanisms into Android to prevent privilege escalation attack. The first security mechanism tracks the inter-process communications (IPCs) in a device to either allow an app to run with reduced privilege of its caller or with its full privileges by acting explicitly on its own behalf. The second security mechanism allows an app to create a signed statement that can be verified by any app on the same phone. Shekhar et al. developed AdSplit [200] on top of Quire. AdSplit is an approach that runs an advertising library and its hosting app in separate processes with different user identifiers. This separation eliminate the need for an app and its advertising library to share the same permissions. Similar to AdSplit, AdDroid [168] introduces advertising API and corresponding advertising permissions as part of the Android platform. AdDroid allows for permission separation between advertising libraries and their hosting apps. Unlike SALMA, these approaches do not control interactions among components and they also require developer intervention to modify their apps, significantly hindering their adoption in practice.

8.4 Modeling Architectures using Matrices

The other relevant thrust of research has focused on studying DSM for modeling and analysis of complex systems. The study on matrices (DSM [203] and MDM [149]) for modeling the architecture of a complex system is also related to our work.

Xiao et al. [188] used the DSM to capture the architecture of a complex system and analyzes it to find architectural debts. Browning [80] discussed various usage of the static and time-based DSMs for decomposing and integrating a system. Sangal et al. [189] implemented the LDM tool that applies the DSM to manage complex software architectures. Similar to SALMA, LDM extracts the architecture using conventional static analysis tools and captures it using a set of 'design rules'. Unlike SALMA, LDM targets conventional software systems not a modern mobile software system such as Android, hence the challenges and the approach are different.

Besides the aforementioned differences, all the security analysis approaches mentioned in this chapter take substantial amount of time to run every time a change in the system occurs and hence they all lack the ability to efficiently analyze systems with incremental changes.

Chapter 9

Conclusion

The systematic violation of the least-privilege principle in Android is the root cause of many types of Inter-Component Communication (ICC) attacks. These attacks have been widely discussed in the literature [89, 173, 99, 111, 70, 135]. Unfortunately, the Android platform at the moment cannot prevent these kinds of attacks, as they do not violate the security mechanisms supplied by Android. Moreover, these attacks cannot be effectively handled by the state-of-the-art security analysis tools, both the static and the dynamic analysis approaches, since malware authors use sophisticated tactics to obfuscate their malicious code to evade their detection.

To effectively address the systematic violation of the least-privilege principle, this dissertation presents a novel approach for automatically determining the least-privilege architecture of an Android system and its enforcement at runtime without the need to modify the existing apps. The approach combines static program analysis techniques to determine the least-privilege architecture of an Android system and use it to reason about the running system, with dynamic monitoring to maintain the architecture synchronized with the running system and enforce it at runtime. This approach has been implemented in a tool called SALMA.

SALMA is an automated self-protecting Android system that monitors itself and adapts its behavior at runtime to prevent ICC security risks. SALMA maintains a precise least-privilege architecture, represented as a *Multiple-Domain-Matrix (MDM)*, and incrementally and efficiently analyzes an Android system in response to incremental system changes. The maintained architecture is used to reason about the running Android system. Every time the system changes, e.g., adding a new app, removing an existing app, granting/revoking a permission, etc, SALMA determines (1) the impacted part of the system, and (2) the subset of the security analyses that need to be performed, thereby greatly improving the performance of the approach.

The least-privilege architecture narrows the attack surface of an Android system, making it easier to evaluate its security posture, and thwarts certain class of ICC security attacks. SALMA leverages static program analysis to determine the exact privileges each component needs to fulfill its task from its implementation logic. SALMA adds a privilege management layer to the Android platform to continuously enforce the maintained LP architecture.

Our experimental results on hundreds of real-world apps corroborate the efficiency and scalability of SALMA as well as its ability to detect and prevent ICC security attacks with minimal disruption. My research artifacts, including tools, presentation slides, and evaluation data, are available publicly [10].

9.1 Research Contributions

To summarize, this dissertation makes the following contributions:

- *Theory.* This dissertation shows that the over-privileged nature of Android is the root cause of many ICC security attacks. These ICC attacks cannot be effectively handled by the current Android platform nor by the state-of-the-art security analysis tools,

both the static and the dynamic analysis approaches. To the best of our knowledge, SALMA is the first self-protecting Android system that can monitor, incrementally analyze, and automatically enforce the least-privilege architecture of an Android system to prevent a wide range of ICC attacks at runtime.

- *Formal description.* Formally describes the least-privilege architecture of an Android system, specified in relational logic, and models it as a *Multiple-Domain-Matrix (MDM).* Moreover, this dissertation provides formal descriptions of ICC security attacks which SALMA uses to analyze the MDM and identifies potential security threats.

- *Experiments.* Empirical evaluation of SALMA on hundreds of real-world Android apps demonstrating its scalability and efficiency in incrementally analyzing Android systems and alleviating their security threats with minimum disruption.

- *Tool.* the research artifact, including tools and evaluation data, available publicly [50];

- *Dataset.* To evaluate SALMA, we used hundreds of benign, malicious, vulnerable, and obfuscated Android apps. We make this dataset available for researchers and practitioners [50].

- *Required Android Platform Modifications.* This dissertation describes the needed modifications to the current Android platform to maintain, analyze, and enforce the least-privilege architecture of an Android system.

9.2 Future Work

Android components are increasingly shipped with native binaries that are shown to have memory-based vulnerabilities (e.g., buffer overflow) [59]. Modeling native code in MDMs, building associated security rules for native-code vulnerabilities, and modeling the interaction among managed and native code in MDMs can provide further attack detection and prevention,

but complicate analyses and may lead to scalability issues. Such challenges are interesting avenues of future work.

Moreover, beside intent-based and URI-based communications, components can launch ICC attacks using either Remote Procedure Call (RPC) or issuing notifications from a component in a handheld device (e.g., a smartphone) to a component in a wearable device (e.g., smartwatch). Detecting and preventing ICC attacks based on RPC or notification-based communications is beyond the scope of this dissertation but indeed an interesting avenue of future work.

To determine the exact privileges each component needs, SALMA relies on the information that are latent in apps' bytecode and assumes that all apps have the same level of trust. Knowing the trustworthy of apps and use these information to tailor the least-privilege architecture would enhance the architecture and reduces possible false positives. My future research would involve implementing an approach that can determine the trustworthy of components with high accuracy.

Finally, since the theoretical contribution of determining and enforcing the least-privilege architecture in Android is applicable to many frameworks with a privilege-based scheme, one can investigate the applicability of the pesented approach in this dissertation to other platforms that use privilege-based security model, such as Chromium and Firefox web browsers.

Bibliography

[1] Android platform architecture. https://developer.android.com/guide/platform/,.

[2] Android trojan looks, acts like windows malware. http://www.snoopwall.com/android-trojan-looks-acts-like-windows-malware/.

[3] Bitcoin-mining malware reportedly found on google play. http://www.cnet.com/news/bitcoin-mining-malware-reportedly-discovered-at-google-play/.

[4] Cisco 2014 annual security report. http://www.cisco.com/web/offers/lp/2014-annual-security-report/index.html.

[5] Gemmy lands app. https://play.google.com/store/apps/details?id=com.nevosoft.mylittleplanet.

[6] Icc bench. https://github.com/fgwei/ICC-Bench.

[7] RevealDroid. http://tiny.cc/revealdroid.

[8] Server-side polymorphic android applications. http://www.symantec.com/connect/blogs/server-side-polymorphic-android-applications.

[9] Smartphone os market share, 2017 q1. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[10] Software engineering and analysis lab (seal). https://www.ics.uci.edu/~seal/projects.html.

[11] The Drebin Dataset. http://user.informatik.uni-goettingen.de/ darp/drebin/.

[12] Threat description trojan:android/oldboot.a. https://www.f-secure.com/v-descs/trojan_android_oldboot_a.shtml.

[13] VirusTotal. https://www.virustotal.com/.

[14] Apktool. https://ibotpeaches.github.io/Apktool/, 2010.

[15] Androguard: Reverse engineering and malware analysis of android apps by blackhat. https://github.com/androguard, 2011.

[16] Allatori obfuscator. http://www.allatori.com/, January 2012.

[17] Virustotal-free virus, malware and url scanner. https://www.virustotal.com/en, 2012.

[18] Virusshare. http://virusshare.com/, August 2013.

[19] Android users have an average of 95 apps installed on their phones, according to yahoo aviate data. http://thenextweb.com/apps/2014/08/26/android-users-average-95-apps-installed-phones-according-yahoo-aviate-data/#gref, 2014.

[20] Contagio malware repository. http://contagiodump.blogspot.it, 2015.

[21] Quick Heal Annual Threat Report 2015. http://www.quickheal.co.in/resources/threat-reports, January 2015.

[22] Brain test lookout report. https://blog.lookout.com/blog/2016/01/06/brain-test-re-emerges/, 2016.

[23] Dresscode android malware. http://blog.checkpoint.com/2016/08/31/dresscode-android-malware-discovered-on-google-play/, 2016.

[24] Kaspersky security bulletin. https://kasperskycontenthub.com/securelist/files/2016/12/Kaspersky_Security_Bulletin_2016_Review_ENG.pdf, 2016.

[25] Mcafee mobile threats report. https://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2016.pdf, 2016.

[26] Smartphone os market share, 2017 q1. international data corporation. http://www.idc.com/promo/smartphone-market-share/os, 2016.

[27] Vikinghorde android malware. http://blog.checkpoint.com/2016/05/09/viking-horde-a-new-type-of-android-malware-on-google-play/, 2016.

[28] 1.5. Stochastic Gradient Descent — scikit-learn 0.18.2 documentation. http://scikit-learn.org/stable/modules/sgd.html, 2017.

[29] Android open source project. https://source.android.com/, July 2017.

[30] Android studio. https://developer.android.com/studio/build/shrink-code.html, 2017.

[31] Dasho. https://www.preemptive.com/, 2017.

[32] Dex2jar: Tools to work with android. dex and java. class files. https://github.com/pxb1988/dex2jar, 2017.

[33] Dexguard. https://www.guardsquare.com/en, 2017.

[34] Falseguide android malware. http://blog.checkpoint.com/2017/04/24/falaseguide-misleads-users-googleplay/, 2017.

[35] Google play app store. https://play.google.com/store?hl=en, 2017.

[36] jarsigner - jar signing and verification tool. https://docs.oracle.com/javase/6/docs/technotes/tools/windows/jarsigner.html, 2017.

[37] Obfuscation study framework. http://www.ics.uci.edu/~seal/projects/obfuscation/index.html, August 2017.

[38] Proguard. https://www.guardsquare.com/en/proguard, 2017.

[39] Smali/backsmali. https://github.com/JesusFreke/smali, 2017.

[40] Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey.html, August 2017.

[41] Android application fundamentals. https://developer.android.com/guide/components/fundamentals.html, Accessed February 2018.

[42] Android debug bridge. https://developer.android.com/studio/command-line/adb.html, Accessed February 2018.

[43] Android documentation: Intent resolution. https://developer.android.com/guide/components/intents-filters.html#Resolution, Accessed February 2018.

[44] Apache cordova. https://cordova.apache.org/, Accessed February 2018.

[45] Fastboot. https://source.android.com/source/running.html, Accessed February 2018.

[46] Keeping your app responsive. https://developer.android.com/training/articles/perf-anr.html, Accessed February 2018.

[47] Nokia threat intelligence report. https://onestore.nokia.com/asset/200492/Nokia_Threat_Intelligence_1H2016_Report_EN.pdf, Accessed February 2018.

[48] Number of available apps in the google play store. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, Accessed February 2018.

[49] Realtime nosql database. https://firebase.google.com/, Accessed February 2018.

[50] Salma: Self-protection of android systems from inter-component communication attacks. https://sites.google.com/view/incremental-analysis/, April 2018.

[51] So many apps, so much more time for entertainment. http://www.nielsen.com/us/en/insights/news/2015/so-many-apps-so-much-more-time-for-entertainment.html, Accessed February 2018.

[52] Xposed module repository. `http://repo.xposed.info/`, Accessed February 2018.

[53] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. *Journal of Computer and System Sciences*, 61(2):236–269, 2000.

[54] M. Alazab, V. Monsamy, L. Batten, P. Lantz, and R. Tian. Analysis of malicious and benign android applications. In *Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on*, pages 608–616. IEEE, 2012.

[55] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Are your training datasets yet relevant. In *International Symposium on Engineering Secure Software and Systems*, pages 51–67, Milan, Italy, March 2015. Springer.

[56] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. *Are Your Training Datasets Yet Relevant?*, pages 51–67. Springer International Publishing, Cham, 2015.

[57] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471, Austin, Texas, May 2016. IEEE.

[58] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 468–471. IEEE, 2016.

[59] B. Aloraini and M. Nagappan. Evaluating state-of-the-art free and open source static analysis tools against buffer errors in android apps. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 295–306. IEEE, 2017.

[60] M. Aly. Survey on multiclass classification methods. *Neural Netw*, pages 1–9, 2005.

[61] A. Apvrille and R. Nigam. Obfuscation in android malware, and how to fight back. *Virus Bulletin*, 2014.

[62] 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 *Proceeding of Network and Distributed System Security Symposium*, San Diego, California, February 2014.

[63] 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 *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2014.

[64] 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 (Workshops)*, pages 107–122. Citeseer, 2013.

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

[66] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *ACM CCS*, pages 217–228, Raleigh, NC, October 2012.

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

[68] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. Appguard–enforcing user requirements on android apps. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 543–548. Springer, 2013.

[69] 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.

[70] H. Bagheri, E. Kang, S. Malek, and D. Jackson. Detection of design flaws in the android permission protocol through bounded verification. In *International Symposium on Formal Methods*, pages 73–89, Oslo, Norway, June 2015. Springer.

[71] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek. Covert: Compositional analysis of android inter-app permission leakage. *IEEE Transactions on Software Engineering*, 41(9):866–886, September 2015.

[72] H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek. Practical, formal synthesis and automatic enforcement of security policies for android. In *Int'l Conf. on Dependable Systems and Networks (DSN)*, pages 514–525, Toulouse, France, June 2016. IEEE.

[73] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 27–38. ACM, 2012.

[74] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 75–88, Alexandria, VA, October 2008. ACM.

[75] E. Behrends, O. Fritzen, W. May, and F. Schenk. Combining eca rules with process algebras for the semantic web. In *Rules and Rule Markup Languages for the Semantic Web, Second International Conference on*, pages 29–38. IEEE, 2006.

[76] N. Bencomo, S. Hallsteinsen, and E. S. De Almeida. A view of the dynamic software product line landscape. *Computer*, 45(10):36–41, 2012.

[77] J. Bosch. From software product lines to software ecosystems. In *Proceedings of the 13th International Software Product Line Conference*, pages 111–119. Carnegie Mellon University, 2009.

[78] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and regression trees*. CRC press, 1984.

[79] D. F. Brewer and M. J. Nash. The chinese wall security policy. In *Security and privacy, 1989. proceedings., 1989 ieee symposium on*, pages 206–214. IEEE, 1989.

[80] T. R. Browning. Applying the design structure matrix to system decomposition and integration problems: a review and new directions. *IEEE Transactions on Engineering management*, 48(3):292–306, 2001.

[81] F. Bry, M. Eckert, P.-L. Pătrânjan, and I. Romanenko. Realizing business processes with eca rules: Benefits, challenges, limits. In *International Workshop on Principles and Practice of Semantic Web Reasoning*, pages 48–62. Springer, 2006.

[82] 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.

[83] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, volume 17, page 19. Citeseer, 2012.

[84] S. Bugiel et al. Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In *USENIX Security Symposium*, Washington DC, August 2013.

[85] G. Cauwenberghs and T. Poggio. Incremental and decremental support vector machine learning. In *Advances in neural information processing systems*, pages 409–415, 2001.

[86] S. Ceri and P. Fraternali. *Designing database applications with objects and rules: the IDEA Methodology*. Addison-Wesley, 1997.

[87] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. Mast: Triage for market-scale mobile malware analysis. In *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '13, pages 13–24, New York, NY, USA, 2013. ACM.

[88] 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)*, pages 659–674, Washington, D.C., Aug. 2015. USENIX Association.

[89] 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*, pages 239–252. ACM, 2011.

[90] H. Cho, J. Lim, H. Kim, and J. H. Yi. Anti-debugging scheme for protecting mobile apps on android platform. *The Journal of Supercomputing*, 72(1):232–246, 2016.

[91] M. Christodorescu and S. Jha. Testing malware detectors. *International Symposium on Software Testing and Analysis (ISSTA'04)*, July 2004.

[92] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, pages 32–46, Oakland, CA, May 2005. IEEE.

[93] C. Collberg, G. Myles, and A. Huntwork. Sandmark-a tool for software protection research. *IEEE security & privacy*, 99(4):40–49, 2003.

[94] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report TR148, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[95] K. Coogan, S. Debray, T. Kaochar, and G. Townsend. Automatic static unpacking of malware binaries. In *Working Conference on Reverse Engineering*, Washington, DC, October 2009. IEEE.

[96] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In *ESORICS: European Symposium on Research in Computer Security*, volume 12, pages 37–54. Springer, 2012.

[97] A. Danielescu. Anti-debugging and anti-emulation techniques. *CodeBreakers Journal*, 5(1), 2008.

[98] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro. Droidscribe: Classifying android malware based on runtime behavior. In *Security and Privacy Workshops (SPW), 2016 IEEE*, pages 252–261. IEEE, 2016.

[99] L. Davi et al. Privilege escalation attacks on android. In *Int'l Conf. on Information Security*, Boca Raton, FL, October 2010.

[100] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, volume 31, page 3, 2011.

[101] A. Egners, U. Meyer, and B. Marschollek. Messing with Android's permission model. In *Int'l Conf. on Trust, Security and Privacy in Computing and Communications*, pages 505–514, Liverpool, United Kingdom, June 2012. IEEE.

[102] K. O. Elish, D. Yao, and B. G. Ryder. On the need of precise inter-app icc classification for detecting android malware collusions. In *Proceedings of IEEE mobile security technologies (MoST), in conjunction with the IEEE symposium on security and privacy*, 2015.

[103] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[104] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245, Chicago, Illinois, November 2009. ACM.

[105] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 235–245. ACM, 2009.

[106] Z. Fang, W. Han, and Y. Li. Permission based Android security: Issues and countermeasures. *Computers & Security*, 43:205–218, June 2014.

[107] Z. Fang, W. Han, and Y. Li. Permission based Android security: Issues and countermeasures. *Computers & Security*, 43:205–218, June 2014.

[108] 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 *International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 414–421, Beijing, China, September 2014. IEEE.

[109] P. Faruki, V. Laxmi, A. Bharmal, M. S. Gaur, and V. Ganmoor. Androsimilar: Robust signature for detecting variants of android malware. *Journal of Information Security and Applications*, 22:66–80, June 2015.

[110] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[111] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, volume 30, page 88, San Francisco, California, August 2011.

[112] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 576–587, New York, NY, USA, 2014. ACM.

[113] A. P. Fuchs et al. Scandroid: Automated security certification of android. *University of Maryland, Tech. Rep. CS-TR-4991*, November 2009.

[114] M. N. Gagnon, S. Taylor, and A. K. Ghosh. Software protection through anti-debugging. *IEEE Security & Privacy*, 5(3), 2007.

[115] J. Garcia, M. Hammad, N. Ghorbani, and S. Malek. Automatic generation of inter-component communication exploits for android applications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 661–671. ACM, 2017.

[116] J. Garcia, M. Hammad, and S. Malek. Lightweight, obfuscation-resilient detection and family identification of android malware. Technical Report UCI-ISR-16-2, Institute for Software Research, Irvine, California, 2016.

[117] J. Garcia, M. Hammad, B. Pedrood, A. Bagheri-Khaligh, and S. Malek. Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. Technical Report GMU-CS-TR-2015-10, Department of CS, George Mason University, Fairfax, Virginia, 2015.

[118] D. Garlan, R. Monroe, and D. Wile. Acme: An architecture description interchange language. In *CASCON First Decade High Impact Papers*, pages 159–173, Toronto, ON, Canada, November 2010. IBM Corp.

[119] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, AISec '13, pages 45–54, New York, NY, USA, 2013. ACM.

[120] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1025–1035, New York, NY, USA, 2014. ACM.

[121] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, pages 281–294. ACM, 2012.

[122] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[123] M. Hammad, H. Bagheri, and S. Malek. DELDroid: Determination and enforcement of least-privilege architecture in android. *Institute for Software Research, University of California, Irvine. UCI-ISR-18-2.*

[124] M. Hammad, H. Bagheri, and S. Malek. Determination and enforcement of least-privilege architecture in android. In *IEEE International Conference on Software Architecture (ICSA)*, pages 59–68, Gothenburg, Sweden, April 2017. IEEE.

[125] S. Heuser et al. Asm: A programmable interface for extending android security. In *USENIX Security Symposium*, San Diego, California, August 2014.

[126] S. Holla and M. M. Katti. Android based mobile application development and its security. *International Journal of Computer Trends and Technology*, 3(3):486–490, 2012.

[127] 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*, pages 639–652. ACM, 2011.

[128] H. Huang, S. Zhu, P. Liu, and D. Wu. A framework for evaluating mobile app repackaging detection algorithms. In *International Conference on Trust and Trustworthy Computing*, pages 169–186, London, UK, June 2013. Springer.

[129] 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 *Proceedings of the 36th International Conference on Software Engineering*, pages 1036–1046, New York, NY, USA, 2014. ACM.

[130] M. C. Huebscher and J. A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys (CSUR)*, 40(3):7, 2008.

[131] W. L. Hürsch and C. V. Lopes. Separation of concerns. 1995.

[132] S. Hyrynsalmi, A. Suominen, and M. Mäntymäki. The influence of developer multi-homing on competition between software ecosystems. *Journal of Systems and Software*, 111:119–127, 2016.

[133] N. Japkowicz and S. Stephen. The class imbalance problem: A systematic study. *Intelligent data analysis*, 6(5):429–449, 2002.

[134] 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 *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2012.

[135] Y. Z. X. Jiang and Z. Xuxian. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2013.

[136] 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*, pages 216–225. ACM, 2014.

[137] D. Kantola, E. Chin, W. He, and D. Wagner. Reducing attack surfaces for intra-application communication in android. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 69–80. ACM, 2012.

[138] J. C. J. Keng, T. K. Wee, L. Jiang, and R. K. Balan. The case for mobile forensics of private data leaks: Towards large-scale user-oriented privacy protection. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, page 6. ACM, 2013.

[139] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[140] W. Klieber et al. Android taint flow analysis for app sets. In *International Workshop on the State of the Art in Java Program Analysis*, Edinburgh, United Kingdom, June 2014. ACM.

[141] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell. The shellcoder's handbook. *Edycja polska. Helion, Gliwice*, 2004.

[142] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *2007 Future of Software Engineering*, pages 259–268. IEEE Computer Society, 2007.

[143] 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 *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, pages 312–323. IEEE, 2017.

[144] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 280–291, Piscataway, NJ, USA, 2015. IEEE Press.

[145] L. Li, T. F. D. A. Bissyande, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and Y. Le Traon. Static analysis of android apps: A systematic literature review. Technical report, SnT, 2016.

[146] L. Li, D. Li, T. F. Bissyandé, J. Klein, H. Cai, D. Lo, and Y. L. Traon. Automatically locating malicious packages in piggybacked android apps. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft '17)*, pages 170–174, Buenos Aires, Argentina, May 2017. IEEE Press.

[147] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics and Security*, 12(6):1269–1284, June 2017.

[148] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. L. Traon, D. Lo, and L. Cavallaro. Understanding android app piggybacking. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 359–361, Buenos Aires, Argentina, May 2017. IEEE Press.

[149] U. Lindemann and M. Maurer. Facing multi-domain complexity in product development. In *The future of product development*. Springer, Berlin, Germany, 2007.

[150] B. Livshits, J. Whaley, and M. S. Lam. *Programming Languages and Systems: Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005. Proceedings*, chapter Reflection Analysis for Java, pages 139–160. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

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

[152] F. Maggi, A. Valdi, and S. Zanero. Andrototal: a flexible, scalable toolbox and service for testing mobile malware detectors. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*, pages 49–54, Berlin, Germany, November 2013. ACM.

[153] F. Maggi, A. Valdi, and S. Zanero. Andrototal: A flexible, scalable toolbox and service for testing mobile malware detectors. In *Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 49–54, Berlin, Germany, November 2013.

[154] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, March 2015.

[155] K. Manikas and K. M. Hansen. Software ecosystems–a systematic literature review. *Journal of Systems and Software*, 86(5):1294–1306, 2013.

[156] J. A. Morales, P. J. Clarke, Y. Deng, and B. G. Kibria. Testing and evaluating virus detectors for handheld devices. *Journal in Computer Virology*, 2(2):135–147, 2006.

[157] J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *USENIX Security Symposium*, Berkeley, CA, August 2002. ACM.

[158] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 231–245. IEEE, 2007.

[159] D. Octeau, S. Jha, and P. McDaniel. Retargeting android applications to java bytecode. In *International Symposium on the Foundations of Software Engineering*, page 6, Cary, North Carolina, November 2012. ACM.

[160] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Int'l Conf. on Software Engineering*, pages 77–88, Florence, Italy, May 2015. IEEE.

[161] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *USENIX Sec. Symp.*, Washington DC, Aug. 2013.

[162] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. *Security and Communication Networks*, 5(6):658–673, 2012.

[163] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th international conference on Software engineering*, pages 177–186. IEEE Computer Society, 1998.

[164] P. Oreizy, N. Medvidovic, and R. N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *Companion of the 30th international conference on Software engineering*, pages 899–910. ACM, 2008.

[165] G. Papamarkos, A. Poulovassilis, and P. T. Wood. Event-condition-action rule languages for the semantic web. In *Proceedings of the First International Conference on Semantic Web and Databases*, pages 294–312. Citeseer, 2003.

[166] N. W. Paton and O. Díaz. Active rules in database systems. In *Active Rules in Database Systems*, pages 3–27. Springer, 1999.

[167] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 71–72. ACM, 2012.

[168] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 71–72, Seoul, Republic of Korea, May 2012. Acm.

[169] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[170] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 241–252. ACM, 2012.

[171] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[172] 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*, page 5. ACM, 2014.

[173] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS*, volume 14, pages 23–26, San Diego, California, February 2014.

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

[175] M. Pomilia. A study on obfuscation techniques for android malware, 2016.

[176] 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*, page 2. ACM, 2014.

[177] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *The Network and Distributed System Security Symposium 2016*, 2016.

[178] S. Rasthofer, I. Asrar, S. Huber, and E. Bodden. How current android malware seeks to evade automated code analysis. In *IFIP International Conference on Information Security Theory and Practice*, pages 187–202. Springer, 2015.

[179] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334, Hangzhou, China, May 2013. ACM.

[180] V. Rastogi, Y. Chen, and X. Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security*, 9(1):99–108, 2014.

[181] V. Rastogi, Y. Chen, and X. Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *Information Forensics and Security, IEEE Transactions on*, 9(1):99–108, Jan 2014.

[182] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *European Workshop on Systems Security (EuroSec), April*, 2013.

[183] S. Roy, J. DeLoach, Y. Li, N. Herndon, D. Caragea, X. Ou, V. P. Ranganath, H. Li, and N. Guevara. Experimental study with real-world data for android app security analysis using machine learning. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 81–90. ACM, 2015.

[184] A. Sadeghi, R. Jabbarvand, N. Ghorbani, H. Bagheri, and S. Malek. A temporal permission analysis and enforcement framework for android. In *International Conference of Software Engineering (ICSE '18)*, Gothenburg, Sweden, May 2018. IEEE.

[185] A. Sadeghi, R. Jabbarvand, and S. Malek. Patdroid: permission-aware gui testing of android. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*, pages 220–232. ACM, September 2017.

[186] M. H. Sadi and E. Yu. Designing software ecosystems: How can modeling techniques help? In *Enterprise, Business-Process and Information Systems Modeling*, pages 360–375. Springer, 2015.

[187] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *IEEE Computer Society Press*, 63(9):1278–1308, April 1975.

[188] P. Sandhu. Legislation and the current provisions for specific learning disability in india-some observations. *Journal of Disability Studies*, 1(2):85–88, 2016.

[189] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *ACM Sigplan Notices*, volume 40, pages 167–176. ACM, 2005.

[190] 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*, volume 96435, 2013.

[191] R. Scandariato et al. Automated detection of least privilege violations in software architectures. In *European Conference on Software Architecture*, Copenhagen, Denmark, August 2010.

[192] B. Schmerl, J. Gennari, A. Sadeghi, H. Bagheri, S. Malek, J. Cámara, and D. Garlan. Architecture modeling and analysis of security in android systems. In *European Conference on Software Architecture*, pages 274–290, Copenhagen, Denmark, November 2016.

[193] D. Schreckling et al. Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android. *Information Security TR.*, 17(3):71–80, February 2013.

[194] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[195] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim. Flexdroid: Enforcing in-app privilege separation in android. In *The Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[196] A. Serebrenik and T. Mens. Challenges in software ecosystems research. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, page 40. ACM, 2015.

[197] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*, volume 1. Prentice Hall Englewood Cliffs, 1996.

[198] M. Shehab and A. AlJarrah. Reducing attack surface on cordova-based hybrid mobile apps. In *Proceedings of the 2nd International Workshop on Mobile Development Lifecycle*, Portland, Oregon, October 2014.

[199] S. Shekhar, M. Dietz, and D. S. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In T. Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, SEC'12, pages 553–567. USENIX Association, 2012.

[200] S. Shekhar, M. Dietz, and D. S. Wallach. Adsplit: Separating smartphone advertising from applications. In *USENIX Security Symposium*, volume 2012, Bellevue, WA, August 2012.

[201] W. Shin, S. Kwak, S. Kiyomoto, K. Fukushima, and T. Tanaka. A Small But Non-negligible Flaw in the Android Permission Scheme. In *Int'l Symp. on Policies for Distributed Systems and Networks*, pages 107–110, Fairfax, VA, July 2010.

[202] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS*, San Diego, California, February 2013. The Internet Society.

[203] D. V. Steward. The design structure system: A method for managing the design of complex systems. *IEEE transactions on Engineering Management*, (3):71–74, 1981.

[204] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications*, 41(4):1104–1117, 2014.

[205] M. Sun and G. Tan. NativeGuard: protecting android applications from third-party native libraries. In G. Ács, A. Martin, I. Martinovic, C. Castelluccia, and P. Traynor, editors, *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.

[206] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*, 2015.

[207] R. N. Taylor et al. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[208] 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*, CASCON '99. IBM Press, November 1999.

[209] 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*, page 13. IBM Press, 1999.

[210] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458. ACM, 2014.

[211] X. Wang et al. Deepdroid: Dynamically enforcing enterprise policy on android devices. In *NDSS*, San Diego, California, February 2015.

[212] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du. Compac: Enforce component-level access control in android. In *ACM CODASPY*, 2014.

[213] Y. Wang and A. Rountev. Who changed you? obfuscator identification for android. May 2017.

[214] R. N. Watson. Adding trusted operating system features to freebsd. In *USENIX Technical Conference*, Boston, MA, June 2001.

[215] 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*, CCS '14, pages 1329–1341, New York, NY, USA, 2014. ACM.

[216] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Profiledroid: multi-layer profiling of android applications. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 137–148. ACM, 2012.

[217] J. Widom and S. Ceri. *Active database systems: Triggers and rules for advanced database processing*. Morgan Kaufmann, 1996.

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

[219] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective real-time android application auditing. In *IEEE Symposium on Security and Privacy*, 2015.

[220] E. P. Xing, M. I. Jordan, R. M. Karp, et al. Feature selection for high-dimensional genomic microarray data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, volume 1, pages 601–608. Citeseer, 2001.

[221] K. Xu, Y. Li, and R. H. Deng. Iccdetector: Icc-based malware detection on android. *IEEE Transactions on Information Forensics and Security*, 11(6):1252–1264, 2016.

[222] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 303–313, May 2015.

[223] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[224] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the ACM conference on Security and privacy in wireless & mobile networks*, pages 25–36, Oxford, United Kingdom, July 2014. ACM.

[225] 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*, pages 1105–1116. ACM, 2014.

[226] T. Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *ICML 2004: PROCEEDINGS OF THE TWENTY-FIRST INTERNATIONAL CONFERENCE ON MACHINE LEARNING. OMNIPRESS*, pages 919–926, 2004.

[227] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 611–622, New York, NY, USA, 2013. ACM.

[228] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104. ACM, 2012.

[229] M. Zheng, P. P. Lee, and J. C. Lui. Adam: an automatic and extensible platform to stress test android anti-virus systems. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 82–101, Heraklion, Crete, Greece, July 2012. Springer.

[230] M. Zheng, P. P. Lee, and J. C. Lui. Adam: an automatic and extensible platform to stress test android anti-virus systems. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 82–101. Springer, 2013.

[231] M. Zheng, M. Sun, and J. Lui. Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, pages 163–171. IEEE, 2013.

[232] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 185–196, San Antonio, TX, February 2013. ACM.

[233] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy*, pages 95–109, San Francisco, California, May 2012. IEEE.

[234] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

[235] 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 *Annual Network &*

Distributed System Security Symposium (NDSS), volume 25, pages 50–52, San Diego, CA, February 2012.

Appendix A

Malware Detection and Family Identification Using Machine Learning

Chapter 4 shows, using a large-scale empirical study, that the average detection rate of anti-malware products decreases by 20% and up to 90% due to the use of code obfuscation. Therefore, the current anti-malware products are insufficient in protecting smartphone users against the increasing number and sophisticated malicious apps.

To address these limitations, this chapter presents a novel machine learning-based Android malware detection and family identification approach, RevealDroid, that operates without the need to perform complex program analyses or to extract large sets of features. Specifically, our selected features leverage categorized Android API usage, reflection-based features, and features from native binaries of apps.

The evaluation of RevealDroid using a large dataset consisting of more than 54,000 malicious and benign apps shows the accuracy and the resiliency of RevealDroid against code obfuscation. The experimental evaluations show that RevealDroid achieves an accuracy of 98% in detection of malware and an accuracy of 95% in determination of their families.

A.1 Introduction

Mobile devices have become ubiquitous, and are still growing quickly. Among such devices, Android has become the dominant platform and is deployed on hundreds of millions of devices around the world. With this widespread usage, an increasing number of malware applications (*apps*) have been found on such devices and the repositories that distribute mobile apps (e.g., Google Play). These malware increasingly resemble their counterparts in Desktop PC environments [4, 2], demonstrating the growing sophistication of mobile malware. Consequently, a significant amount of effort has been expended on producing techniques to detect Android malware.

Existing work on Android malware detection has focused on distinguishing between benign and malicious apps. A number of these approaches utilize permissions requested or used by an app to identify Android malware, including through the use of custom signatures [105, 235, 231] or machine learning [218, 63]. Other techniques identify malicious apps by ranking their riskiness [170, 121]. Alazab et al. and Adagio [54, 119] use graph structures to identify malware. Other techniques identify malicious apps by comparing program behavior with other aspects of an app, including the user interface [129] and app descriptions on app markets [120]. These approaches have made significant and important steps toward identifying malicious apps from individual devices and app markets.

Although accurately identifying if an app is benign or malicious is an important step towards fighting the growing prevalence of malware on Android devices, simply declaring an app as malicious and removing it is not enough to address the damage it may have done once deployed [158]. Engineers that assess the impact of a malware app must determine if other apps, files, or settings may have been damaged or altered; whether there are any remaining malicious or problematic services or processes that have been compromised; if any sensitive data has been stolen or leaked; if any unlawful or illegitimate financial charges have been made

due to the malware's presence; etc. To make such a determination, a security engineer can significantly benefit from *identifying the specific family to which an Android malware belongs*. The family of a malware app can be coarse-grained (e.g., Trojan, virus, worm, spyware, etc.) or finer-grained, where more specific families (e.g., DroidKungFu [234], DroidDream [234], Oldboot [12], etc.) are identified. Knowledge of the family to which an Android malware belongs can help an engineer determine the specific steps that need to be taken to mitigate or undo damage caused by the malware.

Complicating the detection and family identification of Android malware are transformations that obfuscate apps in order to evade detection and family identification by anti-malware software [8, 61, 179]. For example, a variety of malware uses reflection to obfuscate security-sensitive behaviors [177]. A recent study of Android malware obfuscation has demonstrated that simple transformations can prevent ten popular anti-malware products from detecting any of the transformed malware samples, even though prior to the transformations those products were able to detect those malware samples [179]. Thus, malware detection must be designed to *defeat these evasion techniques*. To achieve this goal, malware detection techniques can utilize program analyses that focus on the key semantics and behavior performed by a malware (i.e., behavior as represented by control flow or data flow of a program), particularly in its interactions with the system APIs and libraries that are external to the app, rather than just on syntactic aspects of its implementation (e.g., identifier name or string constants). However, the extent to which recent Android-malware detection techniques are resilient to modern transformation attacks is not well-understood. Existing studies have largely applied their techniques to malware that do not use any, or very limited, obfuscation [204, 225, 63, 119]. These techniques use features that are not resilient to obfuscations. For example, some features utilized by existing approaches are based on control flow [204, 119], which are susceptible to control-flow obfuscations (e.g., addition of junk code or call indirection). As another example, features involving constant strings [225, 63] are susceptible to encryption or renaming obfuscations.

To further reduce Android malware propagation and damage, detection or family identification of such malware should be *scalable*. Some state-of-the-art techniques run into scalability issues and can take hours or up to an entire day to analyze even a single app [129, 67]. Cumulatively, this delayed analysis can allow Android apps to propagate undetected for a longer period of time and, thus, cause more damage. Furthermore, it can prevent users from scanning apps directly on their Android devices, which is important given that Android markets have relatively poor vetting processes [235, 88]. Consequently, it is desirable to utilize features that can be extracted efficiently for detection and family identification of Android malware apps, even obfuscated ones.

In this chapter, we introduce *RevealDroid*, a lightweight machine learning-based approach for detecting malicious Android apps and identifying their families. RevealDroid leverages a set of features selected to achieve obfuscation resiliency, efficiency of analysis, and accuracy. It does not require complex program analyses (e.g., data-flow analysis [67, 225]) or large sets of features (e.g., hundreds of thousands of features [63, 119]), which can lead to scalability problems. More specifically, our selected machine-learning features are based on Android-API usage, including resolution of APIs invoked using reflection, and function calls (e.g., system calls) made by native binaries within an Android app. No previous work has included native-code feature extraction to detect malware. Including features based on reflection and native code significantly aids RevealDroid with achieving obfuscation resiliency.

RevealDroid is capable of accurately detecting malicious apps with a 98% accuracy, and identifying their families with a 95% accuracy, in under 90 seconds on average. RevealDroid can maintain high accuracy even for obfuscated apps. We evaluate RevealDroid's detection and family identification accuracy by comparing its ability to correctly identify malware and classify its family on a dataset of over 24,600 benign apps and over 30,000 malicious apps from two different malware repositories. We further compare RevealDroid's detection and family-identification accuracy against state-of-the-research approaches: Adagio [119],

Drebin [63], and MUDFLOW [67], both of which are approaches for malware detection; and Dendroid [204], an approach for malware-family identification. RevealDroid has an overall greater accuracy by about 11%-25% and mislabels 25%-54% fewer benign apps as malicious than MUDFLOW; RevealDroid achieves up to 23% greater accuracy than Adagio and up to 60% greater accuracy than Drebin. Additionally, RevealDroid achieves a 24%-70% higher classification rate than Dendroid.

This chapter makes the following contributions:

- RevealDroid demonstrates that highly lightweight analyses that extract API-based features—including those based on reflection—and native code features combined with machine learning, can achieve high accuracy, scalability, and obfuscation resiliency.
- We construct an updated dataset of over 27,900 malware apps labeled with their 447 malware families and assess RevealDroid's family-identification accuracy on that dataset. We make this updated dataset available for researchers and practitioners [7].
- To evaluate RevealDroid's obfuscation resiliency, we apply several transformations to malware apps in order to obfuscate them and assess our ability to detect and identify families of those transformed apps. Using these transformed apps, we compare RevealDroid's accuracy for detection against Adagio, Drebin, and MUDFLOW, and for family identification against Dendroid. We also make the transformed dataset available online [7].
- We assess the efficiency of RevealDroid's feature extraction and machine-learning classification. We show that RevealDroid's features can be more than 13 times faster than information-flow feature extraction—which are features used in a variety of Android malware detection tools [225, 67, 227]—while still exhibiting obfuscation resiliency and accuracy. We further demonstrate that RevealDroid can produce classifiers efficiently, as compared to other state-of-the-research tools.

By utilizing machine-learning, RevealDroid is capable of detecting zero-day malware, as opposed to just already-known malware. We assess the efficiency of our different types of features through determining (1) the number of apps for which each feature can be extracted in a short amount of time and (2) the actual runtime of a selection of apps. We further illustrate RevealDroid's ability to identify zero-day malware by utilizing it to detect several families of such malware. Lastly, we leverage an expanded dataset of malware samples from three malware repositories, discovered across several years, to demonstrate RevealDroid's efficacy.

The remainder of this chapter is structured as follows. Section A.2 introduces RevealDroid and its design. Section A.3 covers our evaluation design, the research questions we study, evaluation results, and RevealDroid's limitations. The last sections cover work related to RevealDroid (Section A.4), and conclude the chapter (Section A.5).

A.2 RevealDroid

Malware detection and family identification can be placed into two categories: signature-based and machine learning-based [225]. For signature-based methods, security engineers must produce (often, manually) specifications that match against key properties of a malware family. For learning-based classification, techniques utilize machine learning to automatically determine whether an app is benign or malicious. Each Android app is an *instance* represented by *features* used to distinguish between apps supplied to learning algorithms (e.g., Android API methods or permissions used). A dataset is a set of instances along with their features.

To classify Android apps as benign, malware, or a specific malware family, we leverage *supervised* learning algorithms. For supervised learning, each instance is given a label; in the case of malware detection, the labels chosen are often simply "benign" or "malicious". The

dataset is split into a *training* and *testing* set. A learning algorithm is applied to the training set in order to produce a *classifier*, which can then label apps as "benign" or "malicious". The testing set is passed as input to the classifier to assess its accuracy.

Signature-based methods are highly reliable for detecting known malware, but are often constructed manually and unreliable for detecting variants of known malware or zero-day malware. Learning-based methods require a sizeable dataset and properly selected features to ensure accuracy, but are more likely to generalize in their findings, making them particularly well-suited for identifying variants of known malware or zero-day malware. In this chapter, we utilize learning-based methods.

To properly leverage learning-based methods, we must select features that are likely to distinguish both benign apps from malicious ones and different families of malware apps (e.g., DroidDream from DroidKungFu). Android malware detection and family identification can benefit significantly from the utilization of the Android platform itself to represent features of apps. In particular, the Android API methods, the system calls, and other low-level library calls invoked by an Android app vary significantly between malware families, in order to perform different types of malicious behavior (e.g., sending SMS messages to premium-rate numbers, stealing location and identifier information, acting as a bot, listening for different activation triggers, etc.). We leverage this insight about distinguishing between and identifying Android malware to design an approach for classifying Android malware families. By focusing on framework-, system-, and library-level invocations, which in different combinations tend to be malicious or benign, RevealDroid is capable of achieving obfuscation resilience.

In the rest of this section, we discuss the features utilized by RevealDroid, the labeling of apps and RevealDroid's use of supervised learning to produce classifiers for detecting malware and identifying their families, and other features we considered but ultimately excluded from RevealDroid.

A.2.1 Features Chosen for Learning

To construct RevealDroid, we explored a variety of statically extractable features, both those previously used by other researchers and novel ones. Our goal when designing RevealDroid is to select features that meet three criteria: *accuracy* since any malware detection or family identification should be as correct as possible; *efficiency*, in order to quickly detect malware and its malicious behaviors before it propagates widely; and *obfuscation resiliency* to address different ways malware may evade detection. No malware detection or family identification technique is obfuscation proof, i.e., capable of identifying malware or its family for all possible evasion techniques. However, RevealDroid's aim is to be resilient to as many obfuscation techniques as possible.

To achieve accuracy, efficiency, and obfuscation resiliency, RevealDroid contains the following four types of features: package-level Android API usage (PAPI), method-level Android API usage (MAPI), reflection, and native code. We describe each of these features in more detail in the following paragraphs.

Android API invocations or accesses have been used as features [63, 218]. MAPI features in our formulation are the number of invocations of a specific Android API method. An example of a MAPI feature value is simply the number of times `TelephonyManager.getSimNumber()` is invoked. Categorization of API usage has been shown to be useful in previous malware detection work [67]. To obtain categories of APIs, we simply used PAPI features since packages are specified by Android framework developers themselves.

Increasingly, Android malware are relying on reflection, i.e., the ability of a program to modify or inspect itself at runtime, in order to perform malicious behaviors or obfuscate such behaviors [174]. At the same time, benign apps utilize such behaviors to perform legitimate operations (e.g., update an app with the latest features or bug fixes without having to re-install the entire app). Due to the increasing prevalence of reflection, we included it as

part of RevealDroid. Furthermore, given that obfuscating the target (e.g., method or field) of a reflective call is an indicator of suspicious behavior, it is possible—as our evaluation will demonstrate—to identify malicious reflective usage without needing to fully resolve all reflective calls.

An Android app can use native code to improve the performance of the app, which is often used for games, or to make use of shared native libraries. However, malware authors can utilize native binaries to package exploits, hide behavior from anti-malware techniques that do not scan native binaries, or perform other malicious functionalities [234]. Native code is often ignored as part of Android malware analysis, especially if that analysis is static. Consequently, we included it in RevealDroid.

A.2.2 Labeling and Classifier Selection

Through supervised learning, RevealDroid aims to utilize the aforementioned features to determine which combinations of them indicate malicious behavior and the specific family most likely to exhibit that behavior. As a result, RevealDroid can detect whether an app is benign or malicious, or determine the family to which a malware belongs. RevealDroid can produce different classifiers to perform these functions. The classifier constructed by RevealDroid depends on the labels used when the classifier is trained.

To that end, RevealDroid can build multiple n-way classifiers, where n is the number of labels for Android apps. To detect whether an app is malware, the training set of Android apps can simply contain $n = 2$ labels: *benign* or *malicious*. For malware family identification, the number of labels correspond to the number of malware families in the training set. For example, Android Malware Genome contains 48 malware families, resulting in $n = 48$ for a malware classifier trained on Malware Genome. Given that SVMs (Support Vector Machines) are inherently two-way classifiers [60], we select a linear SVM for malware detection. For

family identification, RevealDroid produces a CART (Classification and Regression Trees) classifier [78], which is a type of decision tree classifier that handles multiclass classification effectively [60]. We demonstrate the efficacy of our choice of classifiers in Section A.3.

The number of labels for family identification significantly increases the difficulty of correctly labeling an Android app, as compared to the 2-way classification when distinguishing between benign and malicious apps. Nevertheless, as our evaluation results will demonstrate, RevealDroid is capable of achieving high accuracy for identifying families of malicious apps.

A.2.3 Android API-Usage Extraction

The Android API contains security-sensitive functionality (e.g., sending SMS messages, and accessing private repositories or location information). We leverage two means of representing Android API usage: the number of Android API method invocations, representing MAPI features, and the number of method invocations for specific Android API packages, representing PAPI features. For example, in the case of PAPI, `android.account` provides APIs for handling account information; `android.media` contains APIs for managing media interfaces to audio and video. These features have been shown to be useful for distinguishing malware families when manually specifying their signatures [112]. Consequently, we chose to include such features for detecting and identifying families of Android malware using machine learning.

To that end, we built the *Android API-Usage Extractor*, which determines the number of API invocations per Android package and the number of invocations per Android API method. As an example of the number of package-level invocations, if three methods of classes in the `android.telephony` package are invoked, then the feature corresponding to that package obtains a value of 3. Formally, the feature vector $PAPI_a = (p_1, ..., p_i, ..., p_{|P|})$, where

$p_i = |\{m \bullet m \in methodPkgs(i)\}|$, P is the set of Android API packages, $methodPkgs(i)$ are the set of methods in package i, and m is an invocation of a method in an Android app a.

To illustrate how such features can help distinguish malware families, Table A.1 depicts features from three Android malware families. Each feature is denoted by a package name within the `android.*` top-level Android API package. For example, in Table A.1, jSMSHider accesses sqlite APIs twice and the telephony package 8 times. The table shows that a supervised learning algorithm can determine that Geinimi samples access location APIs significantly more than jSMSHider or BaseBridge. The learning algorithm can also determine that both jSMSHider and BaseBridge access the telephony and sqlite packages. However, jSMSHider accesses telephony packages more than BaseBridge does; and BaseBridge accesses sqlite packages more than telephony packages.

Table A.1: Example package API features from known Android malware families

	telephony	location	sqlite	Family
mal1	8	0	2	jSMSHider
mal2	0	12	0	Geinimi
mal3	2	0	7	BaseBridge

In the case of method-level invocations, if `telephony.TeleMgr.listen` is invoked by an app twice, then that method obtains a value of 2. Formally, the feature vector $MAPI_a = (m_1, ..., m_i, ..., m_{|M|})$, where $m_i = |\{m\}|$, M is the set of Android API methods, m is an invocation of a method $\mu \in M$. Table A.2 depicts the example in Table A.1 where packages are expanded into methods.

Table A.2: Example method-level API features from known Android malware families

	telephony.TeleMgr .listen	location.LocMgr .rmUpdates	location.LocMgr .reqLocUpdates	sqlite.Db .execSQL	Family
mal1	8	0	0	2	jSMSHider
mal2	0	6	6	0	Geinimi
mal3	2	0	0	7	BaseBridge

A.2.4 Reflective Feature Extraction

A type of feature often ignored by existing Android malware detection and classification techniques are those that involve reflection and, as a result, dynamic class loading. Dynamic class loading through reflection allows an app to modify or inspect itself during runtime, and violate certain language constructs related to information hiding (e.g., allow access to the private members of a class). At the same time, Android malware are increasingly utilizing reflection to obfuscate their malicious behaviors [174]. To address this issue, RevealDroid extracts statically attainable information about reflection. Specifically, RevealDroid determines if a method is reflectively invoked, and the extent to which reflective APIs are used. RevealDroid further separates reflective invocations into three categories [150]:

- *fully resolved:* Both the reflectively invoked method and class names (e.g., `TelephonyManager.getSimNumber()`) can be statically determined;
- *partially resolved:* Only the invoked method name (e.g., `getSimNumber()`) can be statically determined
- *unresolved:* Neither the method name nor the class name can be determined statically (e.g., a non-constant string provided as input during runtime).

Partially extracted reflective invocations occur in cases where non-constant strings, or inputs, are used as target methods of a reflective call. Additionally, as we have observed in Android malware, a high number of unresolvable, reflective method invocations (e.g., reflective calls whose target is encrypted) tend to be malicious. Although reflection enables useful abilities, such as allowing an app to update itself so that users can have the latest features or bug fixes, reflection when used in excess is a strong indicator of malicious behaviors.

Reflective invocation of a method, for both constructor and non-constructor methods, occurs in three stages: (1) class procurement (i.e., a class with the method of interest is obtained) (2) method procurement (i.e., the method of interest to be invoked is identified), and (3) the

method of interest is actually invoked. *Reflection Extractor* attempts to identify information at each stage for the three types of reflective invocations described above.

```
1  ClassLoader cl = MyClass.getClassLoader();
2  try { Class c = cl.loadClass("MyActivity");
3  ...
4  Method m = c.getMethod("onPause",...);
5  ...
6  m.invoke(...); }
7  catch { ... }
```

Figure A.1: Simple reflective method invocation example

A simple example, based on those found in real-world apps, of reflective method invocation, not involving constructors, is depicted in Figure A.1. In this example, a `ClassLoader` for `MyClass` is obtained (line 1), which is responsible for loading classes. The `MyActivity` class is loaded using that `ClassLoader` (line 2). The `onPause` callback of `MyActivity` (line 4)—which pauses a component that has been running—is retrieved and eventually invoked using reflection (line 6). Apps that try to alter the standard Android lifecycle, by invoking callbacks, are indicators of potentially malicious behaviors. Furthermore, any security-sensitive behavior invoked using reflection is, at the least, suspicious.

Our analysis identifies reflectively invoked methods using a backwards analysis. That analysis begins by identifying all reflective invocations (e.g., line 6 in Figure A.1). The analysis currently does not consider the various different parameters or arguments that can be passed to a method invoked reflectively. However, it does track the number of times a particular method is reflectively invoked, which is used as a feature for supervised learning.

Next, the analysis follows the use-def chain of the invoked `java.lang.reflect.Method` instance (e.g., `m` on line 6) to identify all possible definitions of the `Method` instance (e.g., line 4). Our analysis considers various methods that return `Method` instances, i.e., using `getMethod` or `getDeclaredMethod` of `java.lang.Class`. The analysis then records each identified method name. If the analysis cannot resolve the name, this information is also recorded.

At that point, the analysis attempts to identify the class name that is being invoked. Similar to the resolution of method names, the analysis follows the use-def chain of the `java.lang.Class` instance from which a `java.lang.Class` is retrieved (e.g., following the use-def chain of `c` on line 4). We model various means of obtaining a `java.lang.Class` instance. For example, the class may be loaded by name using a `ClassLoader`'s `loadClass(...)` method (e.g., line 2), using `java.lang.Class`'s `forName` method, or through a class constant (e.g., using `MyClass.class`). The analysis then records the class name it can find statically, or stores that it could not resolve that name. Note that our analysis considers any subclass of `ClassLoader`, including the Android-specific `DexClassLoader` that allows dynamic loading of classes stored in the Android Dalvik Executable format. Our reflection analysis involving constructors works in a similar manner by analyzing invocations of `java.lang.reflect.Constructor` and invocations of its `newInstance` method.

Overall, for the three categories of reflective invocations described previously (full, partial, or unresolved), the analysis obtains the following feature information for each app: the full or partial method names invoked; the number of times the full or partial method name is invoked; and the total numbers of fully, partially, or unresolved reflective invocations. As an example, Table A.3 shows samples of features for a fully resolved method name (`SmsManager.sendTextMessage`), a partially resolved one (`getSimSerialNumber`), and unresolved method names.

Table A.3: Sample reflection features from real Android malware apps

	SmsManager.sendTextMessage	getSimSerialNumber	UNRESOLV
mal1	6	0	0
mal2	0	4	0
mal3	0	0	6

A.2.5 Native Call Extraction

A capability of Android apps that is almost never taken into account is use of native code. In particular, to the best of our knowledge, no analysis that utilizes machine learning and static analysis examines the internal behaviors of an app's native binaries. This allows malware authors to package malicious payloads in native binaries, since they are largely ignored. To address this issue, RevealDroid includes a *Native Call Extractor (NCE)* that records calls (e.g., system calls and calls to shared libraries) made by a native binary to entities outside of it, and the extent to which these calls are invoked.

To extract information about security-sensitive invocations in native binaries, NCE must disassemble binaries in a popular binary format for Unix-like systems called the Executable and Linkable Format (ELF). A typical ELF binary, in Android, consists of headers describing meta-data about the binary (e.g., address format, sections of the binary, memory layout information, etc.). After the header, the binary file is divided into sections containing code, data, and potentially other extra information.

To identify malicious behaviors, we focus on calls that the binary may make external to itself and represent key semantics of security-sensitive behavior. In particular, the NCE extracts system calls and other calls that the binary makes to external binaries (e.g., shared libraries). While the main code segment of a native binary on Android is relatively easy to obfuscate (e.g., storing data at non-standard addresses, or adding dead code), these external calls are not easy to obfuscate, particularly system calls.

To identify external binary calls, NCE must identify any call within the code segment of a native binary, and the appropriate assembly instructions that realize a function call. Within an ELF binary in Android, this information is stored in the Procedure Linkage Table (PLT) of the binary. Simply put, the PLT is used to determine the address of external functions not known at linking time.

```
1  00008b54 <sendmsg@plt>:
2      8b54: e28fc600 add ip, pc, #0, 12
3      8b58: e28cca02 add ip, ip, #8192
4      8b5c: e5bcf61c ldr pc, [ip, #1564]!
5  00008af4 <chmod@plt>:
6      8af4: e28fc600 add ip, pc, #0, 12
7      8af8: e28cca02 add ip, ip, #8192
8      8afc: e5bcf65c ldr pc, [ip, #1628]!
```

Figure A.2: PLT of a GingerBreak sample

```
1      99ec: e59d0010 ldr r0, [sp, #16]
2      99f0: e59f13c0 ldr r1, [pc, #960]
3      99f4: ebfffc3e bl  8af4 <chmod@plt>
```

Figure A.3: Code segment where `chmod` is invoked

As an example, consider the disassembled PLT section of a native binary containing the GingerBreak root exploit, shown in Figure A.2 and reduced due to space limitations. In that section, the location of two security-sensitive system calls are identified: `sendmsg` for sending messages over sockets (starting at address `8b54`), and `chmod` (starting at address `8af4`) for modifying the permissions of a file. Each sequence of instructions modifies the program counter so that the machine begins executing at the address of the appropriate system call. For example, lines 2-4 of Figure A.2 first computes an address at which the `sendmsg` code resides, and then loads that address to the program counter (`pc`), so that the code will execute. This binary is stored in the app's **assets/** directory, intended to contain raw resources of an Android app, of the package containing the archive, and is named *gbfm.png* to make the file appear to be simply an image. To identify binaries obfuscated in that manner, NCE scans every file in the package containing the app, i.e., the APK, and checks the format of the file to see if it matches that of an ELF binary.

To properly identify the binary as an Android ELF file, it must be analyzed using the appropriate matching Application Binary Interface (ABI), which is the analog of an Application Programming Interface at the binary level. An ABI defines the manner in which an application's machine code interacts with the system or other binaries. Android supports a variety of hardware architectures (e.g., ARM, MIPS, and x86) built against an Android-specific C

library; each of these architectures use a different ABI. Disassembly and proper identification of the particular ABIs requires use of the Android toolchain for that ABI. Incorrect selection of an ABI or toolchain (e.g., using standard GNU ARM disassembly for Android ARM binaries) will result in incorrectly disassembled code, which may appear to look correct.

Identification of system calls, or other external calls, actually invoked in a binary requires analyzing the `.text` section of an Android ELF binary, which contains its executable code. In such a binary, branching instructions realize invocations of external calls. Specifically, NCE scans the `.text` section of every native binary within an Android app for branch, branch with link, and branch with link and exchange instructions. For each instruction, our analysis determines if the instruction references a label for a function in the binary's PLT.

To illustrate, Figure A.3 depicts an invocation of the `chmod` system call. The initial two instructions prepare the first (`r0`) and second (`r1`) arguments that are passed to `chmod`. The final instruction invokes `chmod` using the address of the external call in the PLT (`8af4` in Figure A.2).

Overall, NCE records each external call of every binary in an Android app, and the number of times each external call is invoked, which together serve as feature types for supervised learning. By scanning every binary, our analysis ensures that no code is missed, even if the code is not invoked using Android's native code interface. For example, this behavior is common for Android root exploits (e.g., executing the binary by using the `Process` or `Runtime` Java classes). An example of three system call features from three different Android malware apps is depicted in Table A.4.

Table A.4: Native call features from real Android malware apps

	chmod	rename	unlink
mal1	6	9	13
mal2	4	6	0
mal3	3	0	6

A.2.6 Other Features Considered

To construct RevealDroid, we explored a variety of statically extractable features, both those previously used by other researchers and novel ones. Table A.5 depicts the various features we considered in comparison with our three criteria of interest (accuracy, efficiency, and obfuscation resiliency) but did not include since they did not meet one or more of those criteria. The various features include the following : *Perm*issions, *Comp*onent names, and Intent Filters (*IFilters*) attainable from an Android app manifest; security-sensitive data *Flows*; and Intent actions (*IActions*). ✓indicates the feature meets the criterion in question; ✘ indicates that it does not. Next, we discuss each feature, and our reasons for discarding it.

1. *Android manifest properties.* An *Android application archive (APK)*, i.e., a compressed archive containing an installable Android app, is distributed with an XML manifest file that contains a variety of metadata about an app. Extracting information from a manifest file can be highly efficient, since it requires simply parsing an XML file. Among the information available in an app's manifest file, we considered using the following as features. However, we discard them due to either not contributing significantly to accuracy or for not being obfuscation resilient, as demonstrated in previous studies [181].
    - *Permissions.* Before Android 6.0, Android apps needed to request permissions at installation time. Starting with Android 6.0, users can revoke or grant permissions at app runtime. However, app permissions are highly granular. Although an app may

Table A.5: Considered features and desired approach criteria: ✓indicates the feature meets the criterion; ✘ indicates that it does not

	Perm	**Comp**	**IFilters**	**Flows**	**IActions**
Acc	✘	✘	✘	✓	✓
Eff	✓	✓	✓	✘	✓
Obf	✓	✘	✘	✘	✘

even request more permissions than it actually uses, it may simply be requesting extra permissions in anticipation of its use in future versions.

- *App components.* A variety of component types, with specific functionalities (e.g., components for providing GUIs, and others for running background services) are declared within an Android app's manifest. However, presence of particular components, especially simply tracking their name, as conducted by some approaches [181], can be obfuscated easily through renaming.
- *Intent filters.* An app's manifest often declares messages, called *Intents*, it can receive and process through filters indicating Intent properties of interest. Although this information can be useful for identifying malware (e.g., those that listen for Intents indicating system actions), an app may simply declare filters in code, allowing for another form of obfuscation.

2. *Security-sensitive data flows.* A few approaches for Android malware detection [225, 67] use data flows between security-sensitive Android interfaces to determine if an app is malicious. Tracking this form of information is particularly useful for identifying privacy leaks, but can be computationally expensive to compute [67]. For that reason, we exclude this feature from RevealDroid. Section A.3.6 further examines efficiency issues with such flows. Furthermore, our experimentation demonstrated that call indirection actually affects data-flow analyses which, in turn, obfuscates privacy leaks, particularly in the case of family identification [117].
3. *Intent actions.* Android malware are known to rely upon tracking the actions of an Intent (e.g., whether a package is installed, or if a device has recently completed booting) to determine when to perform a malicious behavior [234, 112]. In fact, these features are particularly useful for distinguishing between different malware families. Unfortunately, these features are relatively easy to obfuscate, due to the fact that Intent actions are stored as strings, which can be encrypted. In fact, we found that such features can cause a

classifier to miss up to 27% of malware obfuscated using custom encryption transformations [116]. Thus, we exclude such a feature in RevealDroid.

A.3 Evaluation Design and Results

To evaluate RevealDroid, we study its accuracy, efficiency, and resiliency to transformations intended to obfuscate malware. Furthermore, we compare RevealDroid to another state-of-the-research Android malware-family identification approach, Dendroid, and three other detection approaches, MUDFLOW, Drebin, and Adagio. Specifically, we answer the following research questions:

- **RQ1**: How accurate is RevealDroid for distinguishing between benign and malicious Android apps in a time-agnostic and time-aware scenarios?
- **RQ2**: How accurate is RevealDroid for identifying the specific family of a malicious Android app?
- **RQ3**: How does RevealDroid's detection accuracy compare to other detection approaches?
- **RQ4**: How does RevealDroid's family identification capability compare to another state-of-the-research malware-family identification approach?
- **RQ5**: Which features were selected and account for the detection or family identification capabilities of RevealDroid?
- **RQ6**: What is RevealDroid's run-time efficiency? How does this run-time efficiency compare to other learning-based approaches for malware detection?

We implemented RevealDroid in Java and Python. To construct the *Android API-Usage Extractor* and *Reflection Feature Extractor*, we leveraged Soot [209], a static analysis framework, and Dexpler [73], a translator from Android Dalvik Bytecode to Soot's intermediate repre-

sentation. For *Native Call Extractor*, we utilized the Android ABI toolchain to disassemble binaries and identify Android ELF binaries, and constructed a custom-built extractor using Python. For machine learning, we selected Scikit-learn [169], a widely used machine-learning toolkit for Python. For our experiments, we used a machine with 64 cores and 256GB RAM.

To assess RevealDroid's accuracy, we constructed a dataset of both benign and malicious Android apps. To obtain benign apps, we utilized AndroZoo [58], which is a repository of more than 5.5 million apps collected from several sources, including Google Play, the official Android market—and scanned by commercial anti-malware products from VirusTotal [13], an online service provided by Google that scans URLs, files, and Android apps to determine if they are malicious or benign. After scanning the AndroZoo dataset, we found over 24,600 Google Play apps, out of nearly 2 million apps, that are marked as benign by all 55 anti-malware products.

We obtained malware samples from four Android malware repositories: the Android *Malware Genome* project [234], the Drebin dataset [11], *VirusShare* [18], and VirusTotal [13]. Malware Genome contains over 1,200 Android malware apps from 48 different malware families. We utilized 22,592 Android malware samples from VirusShare. We further leveraged 5,538 samples from the Drebin dataset, which includes the samples from the Android Malware Genome project. The remaining apps were obtained from VirusTotal.

A.3.1 RQ1: Detection Accuracy

To answer RQ1, we assess how accurate RevealDroid is for detecting whether an app is benign or malicious in both *time-agnostic* and *time-aware* scenarios. In a time-agnostic scenario, training and testing as part of machine learning is conducted without considering the age of apps in the dataset. This scenario has been utilized to evaluate an overwhelming majority of machine learning-based Android malware-detection approaches [56]. A time-aware scenario

uses the modification date of apps to determine training and testing sets, which avoids training on apps from the future to test on apps from the past.

To evaluate RevealDroid in a time-agnostic scenario, we utilized our entire dataset of Android apps. Table A.6 depicts results for a 10-fold cross-validation, which includes the following: *Prec*ision indicates the extent to which the classifier produces false positives; *Rec*all shows the extent to which the classifier produces false negatives; *F1* score is the weighted harmonic mean of precision and recall; the *No.* of *Apps* used; averages (*Avg.*) for precision, recall, and the F1 score; and the *Total* number of apps.

Table A.6: Detection results for time-agnostic scenario

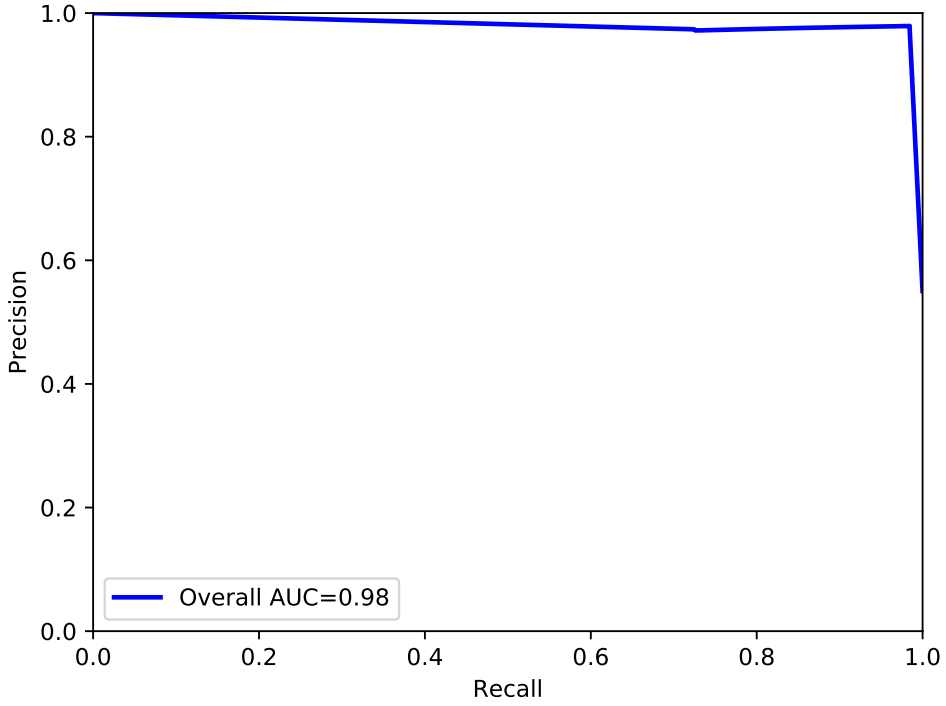
	Precision	Recall	F1	No. Apps
Benign	98%	97%	98%	24,679
Malicious	98%	98%	98%	30,203
Avg./Total	98%	98%	98%	54,882

The table illustrates that RevealDroid achieves high accuracy across the board, for both benign and malicious apps, with an average F1 score of 98%. For just benign apps, RevealDroid obtains a 98% F1 score. For malicious apps alone, RevealDroid attains a 98% F1 score. These consistently high results across multiple measures demonstrates RevealDroid's ability to detect malicious apps with high accuracy.

Figure A.4 shows the detection results for the time-agnostic scenario as part of a precision-recall (PR) curve. This PR curve is nearly perfect, as shown by being close to the upper right corner and having on overall area under the curve (AUC) of .98, where the ideal is 1.00.

To evaluate RevealDroid in a time-aware scenario, we followed the methodology described by Allix et al. [56]. Specifically, we extracted the modification date of the `classes.dex` file in each app's APK file. `classes.dex` contains the compiled implementation classes of the app. The date at which the file is modified allows us to determine the age of the app, which is used to split our datasets into training and testing.

Figure A.4: Precision-recall curve for detection in the time-agnostic scenario

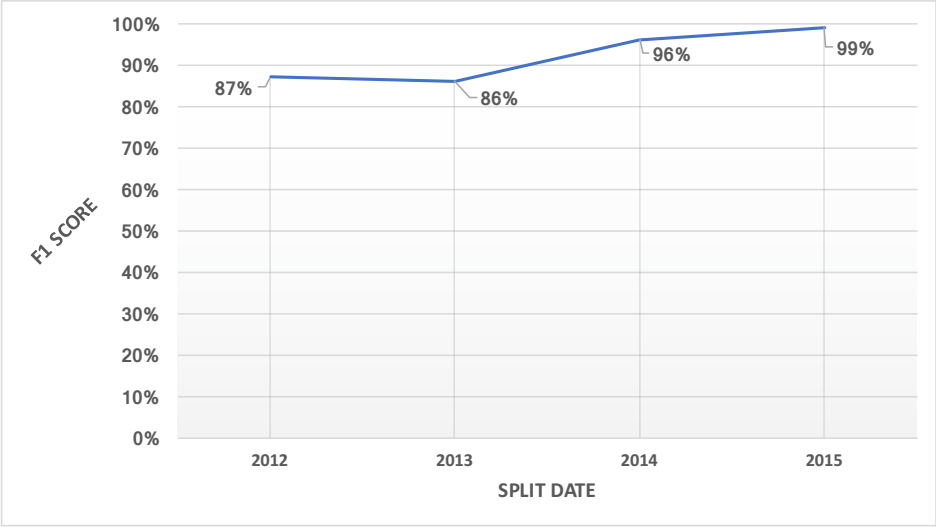


We split our apps into training and testing for a particular date as follows: For each date, any apps older than that date are assigned to the training set; the remaining apps are assigned to the testing set. We selected dates as the first day of each year from 2012-2015. For example, we selected apps created prior to 1/1/2012 as training, the remaining apps are for testing.

Figure A.5 depicts the results obtained for each year with selected dates for splitting from 2012 to 2015. For the first day of 2012, RevealDroid obtains an 87% F1 score and similar results for 2013. However, RevealDroid results only improve for the following years to 96% for 2014 and 99% for 2015. These results show that RevealDroid is able to obtain high accuracy, even when the age of apps is taken into account.

These results are particularly notable since previous work has demonstrated that machine learning-based Android malware detection was unable to obtain an F1 score higher than 70% in a time-aware scenario [56]. In that work, dates newer in time resulted in lower F1 scores; however, RevealDroid actually improves to as high as 99%. Consequently, RevealDroid

Figure A.5: Detection results for time-aware scenario



exhibits a strong ability to obtain high detection results in both time-aware and time-agnostic scenarios.

A.3.2 RQ2: Family Identification

Identifying an Android app as malware is insufficient for dealing with the damage it may cause. Once a malicious app is deployed, it may install other apps, steal information, modify settings, etc. Thus, determining the family to which an app belongs can aid engineers and end users with determining how to deal with the malicious app, besides simply removing it.

Android Malware Genome. To determine RevealDroid's ability to classify Android malware apps into families, we assessed RQ2 by utilizing the Android Malware Genome (AMG) [234], which contains over 1,200 apps and 48 malware families, labeled by other researchers.

Figure A.6 depicts a histogram of malware families in AMG. Notice that no family constitutes more than 25% of the apps in the dataset. Consequently, a naive classifier that labels all

Figure A.6: Histogram of AMG Families

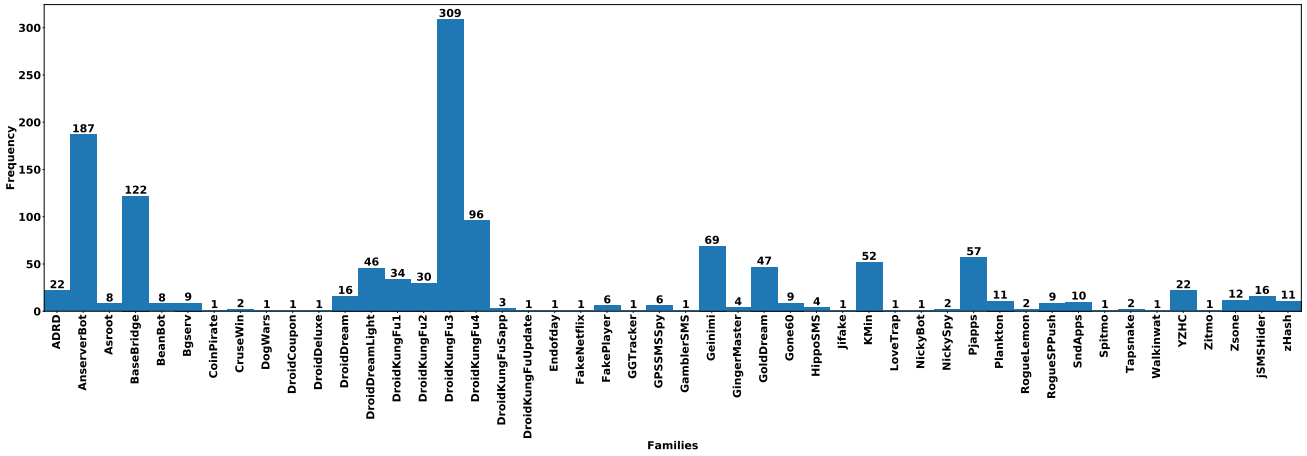
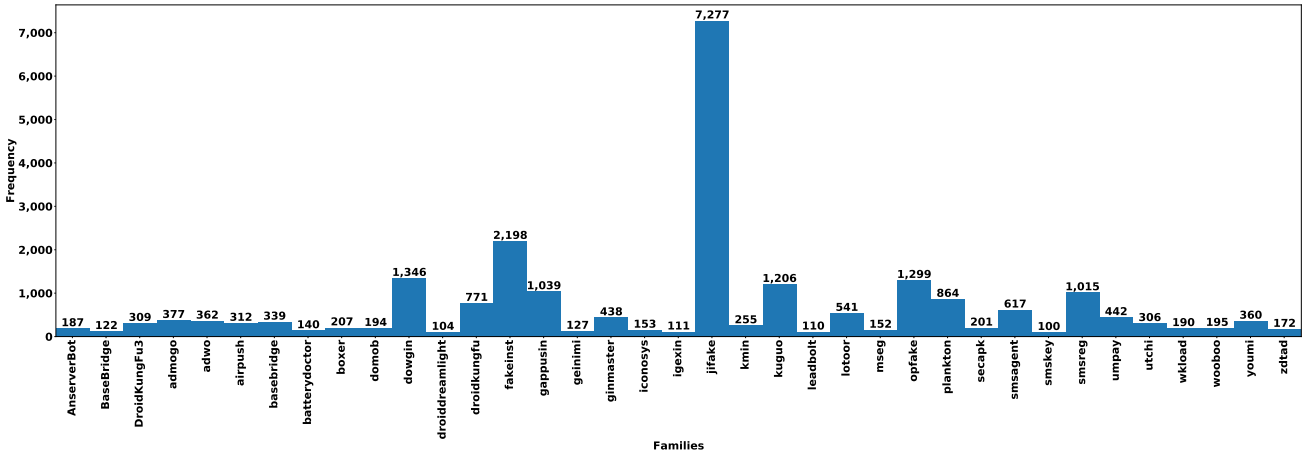


Figure A.7: Histogram of Expanded Families



samples with the most frequently appearing label would only obtain an accuracy of 25%. The histogram indicates that this particular classification task requires a sophisticated classifier.

We used RevealDroid to construct a classifier with 48 different labels, one for each family in AMG. For this experiment, we conducted a 10-fold cross-validation to assess the accuracy of our classifier.

On the AMG dataset, RevealDroid's malware-family classifier obtains a 95% correct classification rate, far above the 25% correct classification rate for a naive classifier. These results showcase RevealDroid's ability to identify a malicious app's family with high accuracy. This outcome indicates that our features are well-chosen for discriminating between malware families.

RevealDroid's classifier did not reach perfect correctness due to a lack of samples for certain malware families: Malware families with less than 10 samples obtained lower results, since our cross-validation uses 10 folds. Ideally, when performing a cross-validation by selecting folds, the number of labels should be greater than or equal to the number of folds.

Expanded Families. To assess RevealDroid's classifiers' effectiveness on more recent malware families, we evaluated those classifiers on a much larger set of malware samples from Drebin, VirusShare, and VirusTotal. To produce a ground truth of families for malicious apps beyond those found in AMG, we again leveraged VirusTotal and a tool called AVClass [194], which provides an algorithm for identifying the malware family label of a malicious app using VirusTotal labels. We uploaded every sample in our malicious dataset to VirusTotal to obtain labels from all the anti-malware products on it. These labels are then passed to AVClass which then provides a family label for each malicious app. If AVClass could identify a family for the malicious app, we utilized it for this experiment. This process resulted in a dataset of 447 families and 27,979 malware samples.

Figure A.7 shows the histogram of the 37 families among the 447 families that contain 100 or more samples. A random classifier would obtain only about a 0.22% correct classification rate; and a naive classifier that simply marks every app with the most frequent family label (jifake) would only obtain a 26% classification rate. As with the AMG dataset, this expanded family dataset poses a challenging classification problem, requiring a sophisticated classifier.

RevealDroid's family identification for this set of apps achieves an 84% correct classification rate, which is far above the 26% classification rate for a naive classifier. This result is particularly useful given the choice of 447 families to which an individual malicious sample may belong.

Note that a major contributing factor to the accuracy of RevealDroid from this expanded family-identification experiment is the use of AVClass and VirusTotal labels. These labels are obtained by an automated technique that relies on (1) heuristics and (2) labels from anti-malware products in VirusTotal, which often disagree with each other. As a result, those labels are likely less accurate than the manually curated AMG labels. That lower quality of family labels for AVClass is likely affecting the correct classification rate of RevealDroid for the expanded family dataset.

A.3.3 RQ3: Detection Comparison

To assess RevealDroid against state-of-the-research approaches for Android malware detection, we compared it against three research prototypes: MUDFLOW, Adagio [119], and Drebin [63]. Besides MUDFLOW, we attempted to obtain state-of-the-research tools, DroidSIFT and Drebin [63], by contacting their respective authors. Drebin is another machine learning-based Android malware detection approach. Unfortunately, both tools are unavailable, preventing us from comparing against them directly. However, in the place of Drebin, its authors suggested we use their other tool, Adagio, which achieves similar accuracy and efficiency results, and

also utilizes machine learning. Adagio operates by constructing function call graphs and encoding them as features used for machine learning.

Although the original Drebin implementation is unavailable, we decided to assess the extent to which Drebin's features for machine learning are useful for detection of Android malware. In particular, we selected three key features that are unique to Drebin: network addresses, requested permissions, and used permissions. Network addresses include URLs, IP addresses, and valid hostnames. Requested permissions are permissions an app requests at install time; used permissions are permissions that an app actually utilizes in its code, as determined by static analysis.

For MUDFLOW, we downloaded its implementation and consulted with its authors to verify that we are using their implementation correctly by re-running MUDFLOW to replicate their results on their original dataset. We further computed method-level flows from FlowDroid and verified that we can replicate the high accuracy results from MUDFLOW's original study on a subset of apps from their dataset. We performed a similar verification in the case of Adagio and Drebin. Due to space limitations, we omit a comparison we conducted with 60 commercial anti-virus products. However, RevealDroid met or exceeded the detection rates of those products. The results of that comparison are available online [7].

We compared Adagio, MUDFLOW, Drebin, and RevealDroid in the following two scenarios: one involving only the original untransformed apps, and another involving apps transformed using DroidChameleon [179, 181], a tool that transforms apps in order to obfuscate them. In the scenario with no transformed apps, we split a dataset consisting of 7,989 malicious apps and 1,742 benign apps into a training set that has half of the benign apps and half of the malicious apps; the testing set has the remaining apps. For the other scenario, the training set consists of 7,995 malicious apps and 878 benign apps; the testing set contains (1) 1,188 malicious AMG apps to which DroidChameleon transformations are applied, and (2) 869

benign apps. For classifier selection, we used the most accurate classifiers of MUDFLOW, Adagio, and Drebin.

DroidChameleon transformations are designed to prevent anti-malware tools from detecting apps to which those transformations are applied. These transformations are based on obfuscations seen in the wild, and have previously been shown to prevent 10 commercial antivirus products from detecting the resulting transformed apps [181]. Another alternative obfuscation tool for Android we considered is ADAM [230]. However, DroidChameleon provides a wider variety of obfuscations, has composite transformations, and has demonstrated the ability to completely evade anti-malware detection. We selected apps from the original AMG to assess RevealDroid's, MUDFLOW's, Drebin's, and Adagio's obfuscation resiliency. Using AMG allows us to assess both the malware detection and family identification abilities of RevealDroid for obfuscation resiliency.

Table A.7 depicts the *sets of transformations* we applied: *call indirection*, where a method invocation is moved into a new method which, in turn, is invoked in place of the original method; *renaming of classes*, where the identifier of classes is changed, which may prevent detection or family identification that searches for specific class names; and *encrypting arrays* and *strings* if they are used by an app. We selected these transformations because they have been shown to evade anti-virus products [181], can be combined to produce stronger obfuscations, and actually result in apps that are still usable. We manually tested several malicious apps, after applying transformations, to verify that the obfuscations resulted in runnable, usable apps.

For each malicious app in AMG, we attempted to apply transformation sets in the following order (ts0, ts1, ts2, ts3), where we try each transformation set in that sequence until a set results in an installable app. For example, we first attempt to apply ts0 and if that fails we then try ts1. We continue in that manner until we have tried all four transformation sets.

Table A.7: Sets of transformations attempted or applied

Trans. Set	Call Indirection	Rename Classes	Encrypt Arrays	Encrypt Strings
ts0	X	X	X	X
ts1	X	X	X	
ts2	X	X		
ts3	X			

Table A.8 showcases the *Pr*ecision, *Re*call, and *F1* score results for each approach and both scenarios, i.e., without transformations ($\neg\ T$) and with transformations (T). For each of those metrics, the table depicts results for *Ben*ign apps and *Mal*icious ones.

Table A.8: Detection comparison, where each numeric result is expressed as a percentage[1]

	MUDFLOW						**RevealDroid**						**Adagio**						**Drebin**					
	$\neg\ T$			T			$\neg\ T$			T			$\neg\ T$			T			$\neg\ T$			T		
	Pr	**Re**	**F1**	**Pr**	**Re**	**F1**	**Pr**	**Re**	**F1**	**Pr**	**Re**	**F1**	**Pr**	**Re**	**F1**	**Pr**	**Re**	**F1**	**Pr**	**Re**	**F1**	**Pr**	**Re**	**F1**
Ben	85	34	49	98	47	64	90	88	89	91	72	80	90	76	83	54	73	62	97	100	98	42	100	59
Mal	87	99	93	72	99	84	97	98	98	82	95	88	95	98	96	73	54	62	100	99	100	0	0	0
AVG	**86**	**66**	**71**	**85**	**73**	**74**	**96**	**96**	**96**	**86**	**85**	**85**	**92**	**87**	**90**	**63**	**63**	**62**	**99**	**99**	**99**	**18**	**42**	**25**

Overall, RevealDroid's classifier outperforms MUDFLOW's two-way classifier. With no tranformation, RevealDroid obtains an average F1 score of 96% compared to MUDFLOW's 71%. For the obfuscations scenario, RevealDroid obtains an average F1 score of 85% compared to MUDFLOW's 74%. The reason MUDFLOW's results improve or remain unchanged overall is likely due to the fact that transformations applied by DroidChameleon are based on transformations seen in the wild. Thus, the approach is likely learning about combinations of feature values that indicate obfuscations.

The most striking difference between MUDFLOW's and RevealDroid's results for both scenarios is each classifier's recall for benign apps. In the scenario with obfuscations, RevealDroid achieves a 72% recall for benign apps compared to MUDFLOW's 47%. For benign apps in the other scenario, RevealDroid obtains a 88% recall compared to MUDFLOW's 34% recall.

[1]Note that RevealDroid's accuracy changes from Section A.3.1 due to differing dataset sizes and splitting strategies.

These results indicate that MUDFLOW's classifier has a strong tendency to mark benign apps as malicious, unlike RevealDroid's classifier.

Adagio obtains a 6% lower F1 score than RevealDroid in the scenario with no DroidChameleon obfuscations. Furthermore, with the DroidChameleon obfuscations, RevealDroid significantly outperforms Adagio by 23%. This low obfuscation resiliency for Adagio is particularly due to the use of call-indirection transformations, which changes the expected call graph that Adagio utilizes to identify malware.

Drebin obtains a 3% higher F1 score than RevealDroid in the scenario with no DroidChameleon transformations. However, in the scenario with transformations, Drebin exhibits the least obfuscation resiliency compared to the other approaches, obtaining a 60% lower F1 score than RevealDroid. The drastic change in accuracy is due to Drebin's heavy reliance on network addresses, which account for most of the features utilized by Drebin. Network addresses are constant strings which are susceptible to identifier renaming and encryption transformations, utilized by DroidChameleon.

In summary, RevealDroid obtains obfuscation resiliency and accuracy for detection, as compared to three state-of-the-research malware detection approaches.

A.3.4 RQ4: Family-Identification Comparison

To demonstrate the improvement in accuracy of RevealDroid's family identification over the state-of-the-art, we compare RevealDroid against a state-of-the-art Android-malware family-identification approach, Dendroid [204], which also utilizes machine learning to classify malware. Dendroid uses features that represent each method of an app as a sequence of typed statements. We contacted the authors of another approach, DroidSIFT [225], which also identifies families. However, DroidSIFT's authors are unable to share their implementation.

Consequently, we could not compare against it. Note that neither MUDFLOW, Adagio, nor Drebin perform family identification.

We closely consulted with the authors of Dendroid to ensure we obtain the most accurate results using their tool as possible. To that end, we replicated their evaluation and verified the accuracy of our results with Dendroid's authors. To compare Dendroid and RevealDroid, we assessed both approaches using AMG. Specifically, we split AMG apps into a training and testing set of approximately equal size. Given that 15 families in AMG only have a single sample, we selected families which had at least two samples, resulting in 33 families in total. For each family, half of the samples were placed into the test set and half into the training set. For families with odd-numbered samples, the remaining sample was added to the training set. This splitting strategy resulted in a training set of 626 apps and a testing set of 607 apps.

Using that experimental setup, Dendroid correctly classified 73% of the test apps, while RevealDroid achieves a 97% correct classification rate. Although our replicated results for Dendroid are significantly lower than the Dendroid authors' original results [204], we verified our results with those authors and discovered an error in their experiment.

We further compared RevealDroid's and Dendroid's obfuscation resiliency. To that end, we trained both Dendroid and RevealDroid using the training set consisting of half of AMG. We then replaced apps in the test set with their obfuscated versions—transformed as discussed in Section A.3.3. The resulting test set contains 590 apps.

RevealDroid demonstrated overwhelmingly greater obfuscation resiliency than Dendroid: RevealDroid obtains a 97% correct classification rate, while Dendroid's classification rate falls to 27%. This low result for Dendroid is unsurprising since it relies on the structure of a method as features. Given that the call indirection transformation that we applied to the test apps alters that structure, the transformation prevents proper classification by Dendroid.

A.3.5 RQ5: Feature Selection

To obtain a better understanding of the extent to which RevealDroid's manually chosen features (i.e., method-level, package-level, reflection-based, and native code-based features) affect its classifiers, we used automated feature selection to identify the features that affect RevealDroid's results most. Additionally, feature selection allows for faster creation of classifiers, reduced training and testing time, and reduces the possibility of overfitting [122, 220]. Specifically, we focused on the dataset of 54,882 apps used for RQ1, where each malicious app is labeled using its family name and every benign app is labeled as such. We obtain family labels using the methodology described in Section A.3.2. By performing automated feature selection using such labeling, we are better able to understand RevealDroid's ability to identify malware families, rather than just its ability to distinguish benign apps from malicious ones. Our initial dataset contains over a million features. Consequently, to perform feature selection, we used a stochastic gradient descent (SGD) classifier, which is a classifier based on an optimization method for unconstrained optimization problems [226, 28] that supports incremental learning [85]. Incremental learning allows a machine learning algorithm to build a classifier incrementally in order to avoid storing all data in memory; given the number of features and apps we utilize, storing all of them in memory is intractable.

Through the feature-selection process described above, a total of 1,054 features were chosen. The resulting features included 595 method-level and package-level features, 454 native call features, and 5 reflection features. The five reflection features selected were features that aggregate information about specific Android APIs that are reflectively invoked. These features are the number of partially resolved API invocations; the number of fully resolved API invocations; the number of reflective API invocations, where the invoked class cannot be statically determined; the number of unresolvable reflective API invocations; and the total number of reflective invocations in the app.

For method-level and package-level features, the selection process chose a variety of security-sensitive API (SAPI) methods and UI-oriented API methods. The use of both security-sensitive and UI-oriented methods to distinguish between benign and malicious apps makes sense since having both types of information allows a classifier to identify the context necessary to decide whether an app's usage is malicious. For example, this intuition has been used by techniques that do not leverage machine learning to identify malicious apps or behavior, but instead identify mismatches among an app's UI and program behaviors [129, 88]. SAPI methods selected include those related to the sending and receiving of Intents, notifications, and other types of inter-process communication; access and manipulation of security-sensitive data stores, such as a Content Provider (i.e., a type of Android component that stores app-specific data) or SQLite database; preferences and settings of the app or the entire Android system; location information of the device (e.g., GPS coordinates); telephone functionality, including sending SMS messages and listening for changes of telephony state; and low-level Android operating system functionality (e.g., asynchronous task running, threading, power management, process killing, and process priority modification).

An assortment of UI-oriented API method types were selected, including the following: methods for operating on different types of standard Android widgets (e.g., images, pop-up windows, dialog windows, progress bars, etc.); web-based UI widgets (e.g., methods of the Android `WebView` class); methods of the Android component type representing a single UI screen (i.e., the Android `Activity` class) and its sub-components (i.e., Android `Fragments`); and Android graphics rendering and animation methods.

The native code features extracted include a variety of functions associated with exploits and security-sensitive functionality—and features associated with benign functionality. In terms of exploits or security-sensitive functionality, the following types of functions are selected: encryption and decryption functions (e.g. RSA functions); compression and decompression functions (e.g., for BZ2 compression); stack unwinding (e.g., used in return-

oriented programming attacks); file and memory manipulation; concurrency control (e.g., threading and mutual-exclusion mechanisms); external application frameworks (e.g., the Mono platform implementation of Microsoft's .NET Framework); and exception handling and manipulation, which is critical for writing exploit code [141].

In terms of benign functionality, selected native code functions include graphics rendering libraries (e.g., OpenGL libraries) and image manipluation (e.g., JPEG and PNG manipulation). In such cases, native code is sensible to use due to improved performance from executing code compiled for a specific hardware architecture.

A.3.6 RQ6: Run-Time Efficiency

The number of both benign and malicious Android apps is growing very quickly [21] making it increasingly important that Android malware analysis scales so that such malware does not remain undetected long enough to do major damage, or even any damage. A slow analysis of Android apps can allow malware to propagate undetected longer. Furthermore, an efficient analysis of malware apps is particularly beneficial for Android end users, since they can protect themselves further by using RevealDroid's classifiers and extractors on their Android devices.

To assess RevealDroid's efficiency, we measured run-times for both (1) feature extraction and (2) classifier training and testing. Note that once a classifier is trained, classifying an app using it—whether for malware detection or family identification—is practically instantaneous. Consequently, feature extraction and classifier training are the key bottlenecks for machine learning-based malware detection and family identification.

General Feature-Extraction. To determine RevealDroid's general run-time efficiency for extracting features, we selected 100 apps in the following manner. We first created a

histogram of app sizes with five bins, as depicted in Figure A.8. From each bin, we randomly sampled 20 apps, resulting in 100 apps in total to be used to measure RevealDroid's feature extraction run-time efficiency. We then ran our three types of feature extractors on each app. Recall that both package-level and method-level feature extraction occur concurrently. Such an experiment allows us to assess the general run-time efficiency of each type of feature ensuring that we have sampled from apps with a variety of sizes.

Figure A.8: Histogram of app sizes from our dataset

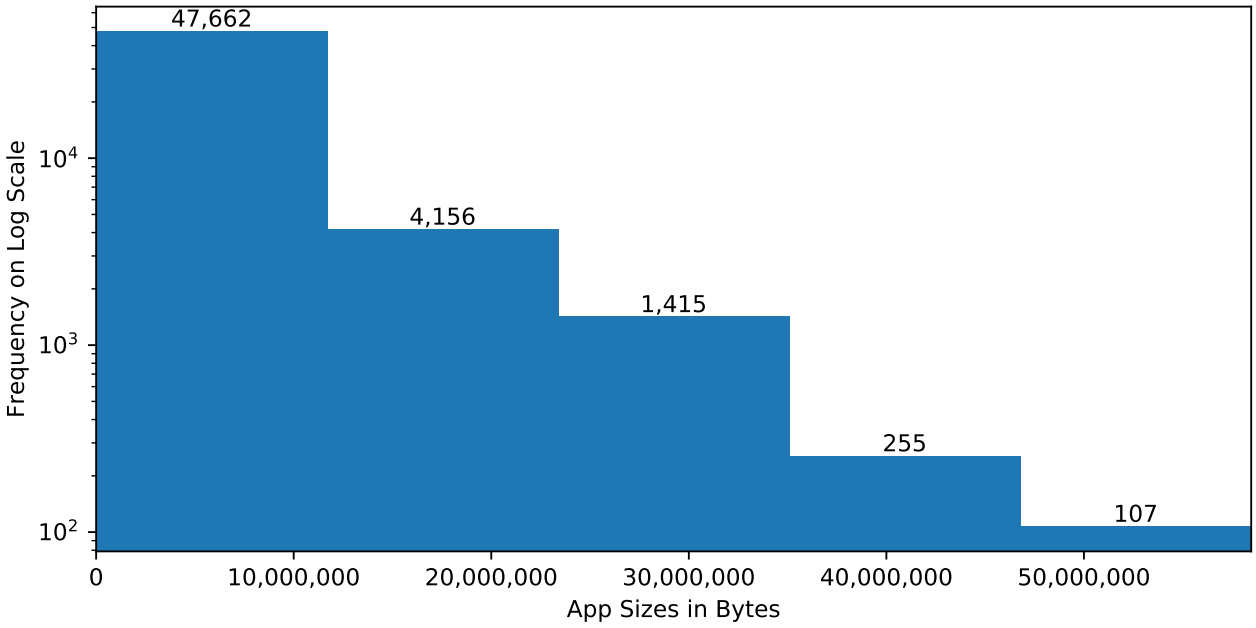


Table A.9 shows the results of our feature-extraction efficiency analysis. Each feature takes under 90 seconds on average to compute. Furthermore, RevealDroid is designed to extract features in parallel, making the total feature extraction, on average, also under 90 seconds. This runtime is reasonable for practical malware detection and family identification that is obfuscation-resilient and accurate.

Table A.9: Average feature-extraction times for each type of RevealDroid feature in seconds.

	Native	**Reflection**	**PAPI/MAPI**
Average (s)	45.79	89.89	79.48

Feature Extraction and Classification. Another bottleneck for learning-based malware detection and family identification is the time it takes for a supervised-learning algorithm to train a classifier and, subsequently, test it. In a practical setting, classifiers need to be regularly updated and re-trained in order to maximize the possibility that such a classifier detects new Android malware.

Table A.10: Feature extraction and classification run-times in hours

	RD-CART	**RD-LSVM**	**Adagio**	**MUDFLOW**	**Drebin**
Feature Extraction	84.79	84.79	56.12	1101.28	817.67
Classification	–	–	21.59	0.20	–
Total	84.79	84.79	77.70	1101.48	817.67

Table A.10 depicts execution times, in hours, for both feature extraction and classification on 9,731 apps. The – indicates that classification takes under 2 seconds to run. For this experiment, we compared RevealDroid's CART (*RD-CART*) and linear SVM (*RD-LSVM*) classifiers with Adagio's, Drebin's, and MUDFLOW's classifiers. Each approach was run on our experiment machine, using the same hardware configuration. MUDFLOW took approximately 46 days to execute; Drebin took approximately 34 days to execute; RevealDroid's CART and SVM classifiers each take 3.5 days to execute; and Adagio's classifier requires about 3 days to execute. Given RevealDroid's superior obfuscation resiliency and its family identification capability, along with its high accuracy and efficiency, RevealDroid achieves its three main non-functional goals.

A.3.7 Discussion and Limitations

One of the major goals of RevealDroid is to aid in the selection of features that are obfuscation-resilient, highly accurate, and highly efficient. Our results demonstrate that these three qualities are achieved, in tandem, using RevealDroid.

Limitations of the dataset utilized by RevealDroid represents a threat to external validity. The number of apps in our dataset affect the generalizability of our results. To maximize our study's generalizability, we used a relatively large dataset, consisting over 50,000 apps, to assess RevealDroid. These apps range from 2011 to 2016.

Internal validity issues mainly arise due to the labeling of apps as benign, malicious, and belonging to a particular malware family. To mitigate this labeling threat, we carefully selected apps to maximize the probability that they are correctly marked as benign or malicious (see the preamble of Section A.3). We further utilized family labels already verified by security experts (see Section A.3.2). Moreover, machine-learning algorithms themselves are partially self-corrective, through statistical methods, for errors in the datasets.

Our choice of using DroidChameleon transformations to evaluate obfuscation resiliency of RevealDroid, and other approaches, is a threat to construct validity. In particular, DroidChameleon may not apply the most effective, realistic obfuscations to malware. This threat is alleviated by DroidChameleon's demonstrated ability to evade existing anti-virus products; its wide variety of transformations, including those inspired by obfuscations observed in the wild; and its composite transformations. Moreover, some apps in our dataset use obfuscations, further mitigating this threat.

We further conducted a study where we trained on our entire app dataset and tested on 1,109 apps transformed with reflection transformations using Droid Chameleon. RevealDroid successfully detected all these apps as malicious. In future work, we intend to include reflection transformations as part of comparing RevealDroid with other malware detection approaches.

Although RevealDroid does extract reflection-related features, static analysis is limited in terms of its ability to extract information related to reflection. To reduce the affect of this

limitation, we directly account for the partiality of our reflection features by representing the degree to which a reflective call can be resolved by our analysis.

One possible way to obfuscate native calls is to utilize the dynamic linker along with the associated `dlopen` and `dlsym` functions to load dynamically-linked functions. To obfuscate this behavior, a malware author can encrypt names in a native binary and decrypt the names during runtime. To handle this case, which has not been observed in Android malware so far, we can create features similar to those that RevealDroid uses for reflection: RevealDroid can determine the extent to which an invoked native call is encrypted. Given that including this type of feature worked well for reflection-based obfuscation (see Section A.3.5), these features should aid in detecting malicious native-code obfuscations that leverage the dynamic linker. To further aid in detecting these types of malicious behaviors, RevealDroid can also include `dlopen` and `dlsym` as native-call features directly.

For family identification, we primarily chose a CART classifier due to the improved performance gain compared with an SVM, with no loss of accuracy. Specifically, we obtained approximately the same F1 score for an SVM and CART classifier, i.e., about 95% for the AMG dataset. However, while the SVM classifier takes 3,539 seconds to run, which is nearly an hour, the CART classifier only takes 194 seconds—which is 18 times faster. For that experiment, we selected an SVM with a linear kernel, a penalty parameter $C = 1.0$, square of the hinge loss as the loss function, and 1,000 as the maximum number of iterations.

One interesting aspect of our experiments, particularly compared to others, is the manner in which we sample benign apps and malicious apps to conduct machine learning. Some approaches choose imbalanced datasets [63]. Other approaches use significantly more malicious apps than benign apps [67]. At least one study has examined a balanced dataset and an imbalanced dataset where the majority class represents benign apps [183].

For our malware-detection experiment (Section A.3.1), we chose to balance the samples. There are two key reasons for choosing a balanced dataset. First, different Android markets have varying levels of malicious apps compared to benign apps [88]. For example, certain Android markets have been known to have a ratio around 60% benign to 40% malicious apps [88]. Second, previous experiments have often chosen imbalanced datasets, which are not necessarily representative of Android markets in general, and may result in less accurate classifiers [133]. Consequently, given the varying degrees of ratios of benign apps to malicious apps on different Android markets, and to avoid biasing the classifier toward either malicious or benign apps, we chose to balance our dataset.

A.4 Related Work

We provide an overview of the current state of Android malware detection and family identification. We first discuss the techniques that solely aim to detect malicious Android apps. We then cover signature-based and machine learning-based techniques that aim to identify the family of such apps.

Many non-machine learning-based Android malware detection approaches have been created. Some approaches mainly use Android-app permissions [105, 235]. Others focus on a variety of other risk factors to rank apps according to their suspiciousness [121, 170, 87]. A significant number of approaches focus on data leakage using taint analyses including dynamic taint analysis [103], combined static and dynamic analysis [219], taint analysis that focuses on user intention or user actions in association with data leaks [223, 138], or analysis of leaks that occur through inter-component communication [215, 144]. Other techniques leverage virtualization to monitor [144], reconstruct [206, 182], or trigger [144] malicious Android app behaviors.

Besides MUDFLOW and Adagio, other approaches have used machine learning for distinguishing between benign and malicious Android apps. DroidMat [218] distinguishes between benign and malicious apps through various features extracted using static analysis and clustering. Furthermore, it relies on easily obfuscatable features (e.g., names of component classes). We contacted the authors of DroidMat multiple times to obtain its implementation so that we can compare against it. However, none of the authors ever responded to our queries.

AppContext [222] utilizes extensive analyses and machine learning involving information flow, Intent filters, Intent actions, and other context factors (e.g., conditions guarding security-sensitive behaviors). Although we considered including AppContext in our study, we could not set up a controlled experiment in the form we used to compare against MUDFLOW and Adagio for two key reasons. First, AppContext is not distributed with its source code, preventing us from modifying its training set as we did with MUDFLOW and Adagio. Second, this limitation further prevents us from comparing with AppContext in terms of its training execution time. However, our analysis takes about 30 seconds on average to analyze a single app; the AppContext study reports an average analysis time of 647 seconds [222]. Furthermore, as discussed in Section A.2.1, Intent actions are likely to significantly reduce obfuscation resiliency due to their susceptibility to encryption transformations.

Drebin [63] is designed to detect Android malware directly on an Android device and uses machine learning. Drebin also uses pre-defined templates to display potentially useful information about what makes an app malicious. Unlike RevealDroid, Drebin relies heavily on features based on constant strings (e.g., names of components) that are obfuscatable using basic automated transformations (e.g., renaming and encrypting identifiers and string values), as demonstrated in our evaluation. Furthermore, their feature space is very large, containing about 545,000 features, as compared to RevealDroid's feature space of about 1,000 features,

which allows our classification—and potentially our feature extraction—to be significantly more efficient and scalable.

ViewDroid [224] and MassVet [88] are capable of detecting malicious Android apps and both focus on repackaging detection. Both techniques leverage graphs based on UI widgets of an Android app. Due to the use of control flow-based graphs, both of these techniques are potentially susceptible to control flow-based obfuscations. RevealDroid is not vulnerable to such obfuscations, due to the fact that it does not rely on any program-analysis graph representations. Unlike in the case of RevealDroid, no automated transformations were applied to existing malicious apps to assess MassVet; automated transformations were applied to benign apps for MassVet. However, as discussed in the MassVet paper [88], obfuscations of malicious methods may be problematic for MassVet. Additionally, whether the transformed benign apps utilized combinations of transformations was not discussed. Unlike MassVet and ViewDroid, RevealDroid is capable of accurately identifying the family to which a malware belongs, and not just identifying an app as malicious. Furthermore, RevealDroid is not limited to only detecting and identifying families of repackaged malicious apps.

A variety of other techniques use different mechanisms for detecting Android malware. DroidAnalytics [231] provides an automated workflow for the collection and signature generation of Android malware by analyzing apps at the opcode level. AsDroid [129] detects stealthy behaviors of possibly malicious apps characterized by mismatches between program behavior and the UI. Poeplau et al. [174] construct a static analysis tool for identifying unsafe and malicious dynamic code loading. HARVESTER [177] extracts features relevant to anti-analysis techniques (e.g., obfuscations and emulator detection techniques) from Dalvik bytecode using static and dynamic analyses. Unlike HARVESTER, RevealDroid aims to utilize lightweight static analysis and machine learning to identify malicious apps, and directly analyzes apps' native binaries.

Besides not identifying malware families, most of the above techniques rely on heavyweight program analysis, unlike RevealDroid's lightweight analysis.

Several approaches focus on identifying specific malware families. Apposcopy [112] provides a language to specify malware signatures and a static analysis to identify apps matching those signatures. For Apposcopy, security engineers must manually construct malware signatures, which is a time-consuming and error-prone task.

A few approaches automatically identify the family of Android malware. Dendroid [204] utilizes text-mining techniques and control-flow features to identify families of malicious apps. DroidSIFT [225] employs extracted dependency graphs to determine whether an app is benign or malicious, and the family of a malicious app.

Two approaches that automatically identify the family of Android malware using static analysis—Dendroid and DroidSIFT—are both limited, when compared to RevealDroid, in three key ways: (1) they have limited or no reflection features, (2) they have no native-code features and (3) they perform a highly limited assessment for obfuscation resiliency, or no such assessment at all. Both approaches are evaluated on a limited number of malware families and apps. On the other hand, we evaluate RevealDroid on a dataset consisting of tens of thousands of more apps, and several hundred more malware families studied as part of the DroidSIFT paper. Additionally, DroidSIFT utilizes flow features, which are heavyweight to extract, as demonstrated in our experiments, and do not account for statically unresolvable or partially resolvable reflective calls.

Both techniques have limited obfuscation resiliency, and rely on representations (e.g., control-flow features or constant strings) that can be evaded by using standard automated transformations. Furthermore, DroidSIFT is only assessed using unstated obfuscations applied to a small number of apps from a single malware family.

A third approach, DroidScribe [98], uses dynamic analysis and machine learning to identify the family to which a malicious app belongs. However, it does not determine whether an app is benign or malicious.

None of the aforementioned approaches extract native-code features by actually analyzing an app's native binaries. Furthermore, RevealDroid is the only Android malware detection and family-identification approach that combines machine learning with static analysis extraction of features based on Android API usage, reflection, and native code.

A.5 Conclusion

This chapter has introduced RevealDroid, a machine learning-based approach for Android malware detection and family identification that is accurate, efficient, and obfuscation resilient. RevealDroid relies on features involving security-sensitive Android andAPI calls; reflective calls categorized according to the degree to which invoked methods can be resolved; and invocations in native binaries to external functions (e.g., system calls or shared library calls) and functions within the binaries. We have compared RevealDroid with state-of-the-art tools for Android malware detection and family identification. For Android malware detection, RevealDroid obtains an 11%-60% superior accuracy compared to state-of-the-art tools. In the case of family identification, RevealDroid attains a 24%-70% higher classification rate. Our experiments showcase RevealDroid's high accuracy and efficiency (e.g., a 98% F1 score for 54,882 apps and an app extraction time of 90 seconds on average), with particularly high accuracy under various obfuscations. We further compared RevealDroid to a state-of-the-art family-identification approach, demonstrating significantly higher accuracy—95% accuracy on a high quality Android malware family dataset—especially in the face of obfuscations.

In the future, we intend to explore feature characteristics of emerging malware apps—such as those that infect an Android device's Master Boot Record [12] and stealthily utilizing devices to mine cryptocurrency services [3]—in order to detect and identify the families of those malware. To enable replication of our results and improvement over RevealDroid, we make our RevealDroid prototype and data available online at [7].