AUTOMATIC PROGRAM STATE EXPLORATION TECHNIQUES FOR SECURITY ANALYSIS OF ANDROID APPS

by

Ryan Johnson

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

______________________	Dr. Angelos Stavrou, Dissertation Director
______________________	Dr. Sanjeev Setia, Dissertation Co-Director
______________________	Dr. Jeff Offutt, Committee Member
______________________	Dr. Paul Ammann, Committee Member
______________________	Dr. James H. Jones Jr., Committee Member
______________________	Dr. Deborah J. Goodings, Associate Dean
______________________	Dr. Kenneth Ball, Dean, Volgenau School of Engineering
Date: ______________________	Fall Semester 2019 George Mason University Fairfax, VA

Automatic Program State Exploration Techniques for Security Analysis of Android Apps

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Ryan Johnson
Master of Science
George Mason University, 2010
Bachelor of Science
George Mason University, 2007

Director: Dr. Angelos Stavrou, Professor
Co-Director: Dr. Sanjeev Setia, Professor
Department of Computer Science

Fall Semester 2019
George Mason University
Fairfax, VA

Dedication

I dedicate this dissertation to my daughter Alexis ♡.

Acknowledgments

I would like to thank the Drs. Angelos Stavrou, Sanjeev Setia, Jeff Offutt, Paul Ammann, and James H. Jones Jr for their valuable feedback and for serving on my committee. I would also like to thank Dr. Mohamed Elsabagh for his advice and research collaboration. I would like to give a special thanks to my family and friends for their patience with me and their positive influence on my life.

Table of Contents

List of Tables

List of Figures

Abstract

AUTOMATIC PROGRAM STATE EXPLORATION TECHNIQUES FOR SECURITY ANALYSIS OF ANDROID APPS

Ryan Johnson, PhD

George Mason University, 2019

Dissertation Director: Dr. Angelos Stavrou

Dissertation Co-Director: Dr. Sanjeev Setia

The usage and ownership of mobile devices is increasing globally. Our reliance on mobile devices and the apps they run warrant novel techniques to explore the behavior of both downloaded and pre-installed apps. Mobile apps are increasing in size and complexity, making them more challenging to design and test. Focusing on Android, the most popular mobile platform, I present methodologies to automatically analyze the states of Android apps without access to source code and from a security perspective. Primarily, my research suggests approaches to overcome the limitations of current binary code analysis techniques to also include external and environmental inputs. I explain how utilizing this augmented set of inputs we can discover unsafe app states that violate end-user security and privacy when abused by an adversary.

To that end, I designed and implemented a novel program analysis technique for Android called Forced-Path Execution (*FPE*). *FPE* forces execution of code independent of the program state according to an execution strategy exposing program states that are deemed safety critical. Applying *FPE* on Android apps, I was able to discover unsafe use of sensitive Android Programming Interfaces (APIs) and "leaking" of Personally Identifiable Information

(PII) including access to text messages and system logs, among others. In addition, I explore the security and reliability of inter-app communications via the Android Inter-Process Communication (IPC) mechanism, namely the use of Intents. I systematically stress-test this Android IPC mechanism to uncover design flaws within apps and the Android Operating System (OS) itself. My approach scales to scan thousands of apps from Google Play and the official Android Open Source Project (AOSP) code. As a result, I discovered thousands of Intent input validation faults in apps from Google Play and multiple faults in a critical AOSP system process for both the smartphone and embedded Android platforms.

Chapter 1: Introduction

Mobile devices are an integral part of daily life. It is estimated that 5 billion people own mobile devices, and more than half of these mobile devices are smartphones [1]. Over the past eight years, mobile devices have eclipsed traditional Personal Computers (PC) in the number of units shipped [2], number of active daily users on Facebook [3], number of Google searches [4], and volume of web browsing traffic [5]. Furthermore, estimates predict that in 2019 people will spend more time on mobile devices than on PCs, and by 2025 72.6% of all users will use solely a mobile device to access the Internet [6]. Greater dependence on mobile devices is likely to continue [7]. Mobile devices have become so ubiquitous and integrated with our daily activities to such an extent that research exists to study the effects of *nomophobia*, the fear of being without a mobile device [8–10].

Mobile device users can quickly and easily extend the functionality of their device by downloading and installing software applications (subsequently referred to as *apps*) to obtain new content and capabilities. Users have greatly benefited from the large and readily available corpora of apps that are hosted on app distribution platforms that offer a seamless experience downloading and installing billions of apps. Globally, 194 billion apps were downloaded in 2018, and consumers spent $101 billion through in-app purchases, paid app downloads, and in-app subscriptions [11]. The official app distribution platform for Android, Google Play, hosted more than 2.7 million apps as of August 2019 [12]. Since 2011 Android has been the dominant mobile OS and recently surpassed Windows to become the most widely used OS [13]. Android apps are not just limited to smartphones, they can run on streaming media players, wearable devices, smart TVs, Internet of Things (IoT) devices, and infotainment systems [14].

Occasionally, apps downloaded through official and unofficial app stores can cause security and privacy issues [15–20]. These issues can also arise in pre-installed apps in Android devices that user did not choose to install themselves [21–26]. Pre-installed apps provide the base functionality for a device, including downloading additional apps. Pre-installed apps can obtain privileges from the system, which are unavailable to apps that the user downloads and installs, generally referred to as *third-party apps*. Since pre-installed apps are present on the device from the beginning, they present a potent attack vector, especially since some cannot be disabled or uninstalled by the user.

Mobile apps provide expanded functionality to the user. The degree to which an app has access to resources on a mobile device is generally regulated by the use of permissions. The Android platform declares permissions that an app can request to access specific resources, such as obtaining a list of the user's contacts. These protected resources include user data such as the user's email address, phone number, call log, text messages, Global Positioning System (GPS) coordinates, unique device identifiers, photos, downloaded files, and more. Some of these protected resources qualify as *Personally Identifiable Information* (PII), which can be used by itself or in concert with other data points to identify the user. User data can be utilized in an app to provide a rich user experience, but the user data can also be mishandled, both intentionally and unintentionally.

On devices running Android version 6.0 and higher, the user can decide if an app is allowed to obtain certain permissions. Permissions that provide access to user data require the user to directly grant the permission to the app. After the user grants a permission to an app, the app can then access the corresponding resource directly, such as using the GPS subsystem to obtain the device's physical location. Users can see what permissions an app requests, but they may not understand all the ways that their personal data can be used. For example, the typical user is likely unaware if or when GPS data is sent from their device to a remote location; to whom the data was sent and under what conditions; and if the GPS data is sent only if the user is a member of a certain group of interest. I aim to address these issues by simultaneously simulating different environments to an app while recording its behavior.

Typically, the use of permissions by apps provides a measurable boundary of an app's functionality, although this is not always the case. In certain circumstances, a less privileged app can use an open interface of a more privileged app to have it perform permission-protected functionality on its behalf, manifesting as a *capability leak*. Android apps can provide interfaces for other apps to share data and communicate. Therefore, an app with no permissions may be able to indirectly access permission-protected resources through an app that has access to them. This means that resident vulnerabilities in other apps can be locally exploited by an app co-located on the device. Attacking locally from an Android app greatly expands its access to the system and other apps, giving leverage to remote attacks, which can result in privilege escalation [21, 23], spyware [22, 27], ransomware [28, 29], and Denial of Service (DoS) attacks [30–33]. The proliferation of intentionally and unintentionally insecure apps warrants novel techniques to detect and thwart these issues using a proactive approach to security.

1.1 Android App Analysis

A program *state* is a snapshot of a program during execution that encompasses the program's memory, representing the current values of its variables. Currently, the two primary methods for determining the states that a mobile app can occupy, and thus its behavior, are static analysis and dynamic analysis. *Static analysis* examines artifacts of the app and reasons about its behavior without executing it. *Dynamic analysis* involves executing the app and recording aspects of the app's precise runtime behavior. Recording app behavior is generally achieved using *instrumentation* which adds extra code to interpose on an app during its execution. Numerous dynamic analysis frameworks and techniques exist for Android [34–44]. These approaches are often paired with a tool to programmatically interact with the app being analyzed to increase code coverage and facilitate automation. The majority of dynamic analysis platforms for Android depend on leveraging interactions with the app's Graphical User Interface (GUI), such as programmatic injection of random user and system events into a running app.

Android apps generally use data obtained from the environment, which, in turn, affects the behavior of an app. By the *environment*, I mean data that is obtained from outside of the app code such as from the Android OS, user, file system, or network. The environment in which an app executes can be expected to vary. I have discovered multiple real-word apps that require quite specific runtime conditions to exhibit a behavior that is generally harmful to the user (see Chapter 4). I propose that it is constructive to simulate different environments to make an Android app exhibit a greater range of behavior than what would be otherwise displayed in a single environment. In addition, Android apps can employ anti-analysis techniques in which they change their behavior when they detect that they are being analyzed [45, 46]. Some of the anti-analysis approaches inspect the environment with the aim of detecting an emulator [45–47], detecting `root` user management utilities [48,49], watermarking analysis platforms [50], and detecting the use of automated exploration techniques [51].

1.2 Forced-Path Execution

To enable full code analysis of Android apps, many challenges have to be addressed. To that end, I have designed and implemented a novel analysis methodology that relies on a new custom Android analysis tool using forced-path execution. *Forced-path execution* (*FPE*) explores both branches of selected conditional statements based on execution strategies in order to enumerate the possible behavior of an Android app in different environments. A *conditional statement* contains a mathematical or logical expression that is evaluated using its input(s) to determine which code branch to execute. A logical or mathematical expression, known as a *predicate*, consists of a relation among inputs or a Boolean value that is evaluated to produce a Boolean result. An *execution strategy* is a set of rules that determine which conditional statements should be explored based on a high-level goal (e.g., complete exploration, exploring the input domains). Various dynamic analysis approaches have difficulty achieving code coverage [38, 52]. Current tools and frameworks to explore the

GUI are not sufficient to enter code branches that may require specific external input to execute. This includes system and network events in addition to conditions that depend on the configuration of the app execution environment. An app *execution environment* refers to external sources of data that an app accesses which can affect the evaluated outcome of conditional statements within the app. For example, some items comprising the execution environment are the network, OS version, user input, location, language setting, time, date, presence and versions of supporting software, to name a few. These externalities naturally vary and can differ from one environment to the next.

FPE relies on execution strategies to inform the framework under what conditions to explore all branches of a conditional statement. Depending on the execution strategy, *FPE* may track data from the environment to eliminate some infeasible paths from the analysis that cause false positives, though the problem is formally undecidable. This is accomplished by limiting the forcing of conditional statements to only when values related to the execution environment are evaluated in a conditional statement. Since values related to the execution environment can be expected to vary, all branches of a conditional statement will be taken to examine an app's functionality. For example, an app may obtain the current timestamp and check if a certain amount of time has passed by comparing it to saved timestamp when making a branching decision. As time passes, the necessary condition will naturally arise for the branch that is control-dependent on a timestamp comparison to be entered, although the time threshold to enter this branch can be set by the developer to a longer period than most dynamic analysis frameworks spend analyzing an app [53].

I also present another research area that focuses on the security and reliability with regard to the usage of Intent objects. An *Intent* is an object in the Android Application Programming Interface (API) that provides a message-like abstraction that is asynchronously sent within an app or between apps [54]. Android apps are compartmentalized into app components to encourage code reuse and data sharing. Intent objects are the primary communication mechanism between app components. A single Intent can support different information flows, including being sent to multiple destinations using a broadcast Intent.

Sending Intents require no access permissions; therefore, they can be leveraged by any app co-located on the Android device.[1] Through my research, I have demonstrated that Intents can be used as an effective medium for launching local DoS attacks against the system and other apps. By sending Intents at specified rate of transmission, an app can create memory pressure and cause the system to terminate another app of its choosing. Therefore, an app can set itself as the gatekeeper that decides if another app can execute. Using Intent objects, a local app can also cause the device to be unresponsive to the user, denying the user all productive functionality of the device. In addition, sending Intents at an accelerated rate can cause a user space system crash.[2] A user space system crash occurs when user space components of the Android OS crash and then restart while the underlying Linux kernel continues execution. I applied these DoS attacks on a range of embedded Android devices to evaluate their mechanisms to cope with a persistent, local DoS attack. Although resiliency to the DoS attacks among the devices were varied, in the most severe cases, a zero-permission Android app could disable certain embedded Android devices and make them functionally useless.[3] I also proposed some platform defenses and implemented them in open-source defense apps to mitigate the DoS attacks [55, 56].

Furthermore, I demonstrate that many Android apps and the Android OS itself make assumptions in their code about the presence of certain embedded data in Intents without first checking to see if the data exists at runtime. Even if apps do perform input validation, some apps are unable to gracefully handle the absence of anticipated data. The use of Intents with empty data fields can lead to a program crash, user space system crash, or permission leaks in privileged pre-installed apps. I developed an open-source tool that systematically discovers and tests the exposed app components on various Android devices and determines their effects [57]. If a user space system crash is caused repeatedly by an app, it can result in a local DoS attack against the availability of the device. A fault occurring within Android OS code is generally a significant event and some devices will write the system logs to a

[1] There are some limitations on the Intents that an instant app can send. *Instant apps* are ephemeral apps launched by clicking on an HTTP(S) link, which provides a preview of an app without having to install it.

[2] This is also informally known as a *soft reboot*, a term that also appears in the research literature.

[3] I responsibly reported the DoS attacks to Google and the affected vendors for remediation.

world-readable file, resulting in potential data disclosure.

1.3 Contributions of this Work

1.3.1 Thesis Statement

1. *Problem Statement*: Current program analysis techniques evaluate the target program in a single or limited set of execution environments that depend on immutable predicates and lack modeling of external and environmental inputs and adversarial behavior.

2. *Thesis Statement*: Exploring and controlling the evaluation of predicates where artifacts of the execution environment are used as inputs can offer comprehensive analysis of software in different execution environments and uncover states that may not be reached under *normal* execution.

To answer the question, "How can I comprehensively uncover unwanted functionality that can be expressed as unwanted states or program conditions that can lead to program exploitation or failure in terms of reliability?" I show that if I attempt to identify these program states using existing testing techniques, under certain conditions known to an adversary, it fails to reach them. This can happen for a number of reasons that have to do with theoretical bounds of existing testing methodologies and the limitations of practical implementations on actual systems.

The aim of my research is to extend the analysis and testing of programs to eliminate potential *exposure* to attacks or corruption. I do so by analyzing the program with *FPE*, which identifies states and code paths that are undesirable from a security or reliability perspective. I then create different and all-encompassing *what if* scenarios that can lead to those states that might be reachable from a set of inputs. In some cases, the unwanted functionality can be triggered with a simple Inter-Process Communication (IPC) message, which in Android takes place through Intents. In other cases, a simple file change might be sufficient. Finally, I prune paths by using execution strategies to eliminate paths that are truly non-reachable either because their preconditions cannot be met under any condition or because the changes required are not easy to achieve.

1.3.2 Summary of Contributions

This dissertation introduces a novel program analysis technique for Android that examines the potential behavior of an Android app under a wide array of input and environmental conditions. In addition, I examine a commonplace IPC mechanism, Intents, and the methods in which they can be leveraged to launch local DoS attacks against other apps and the Android OS. In summary, my contributions are as follows:

- I created and implemented an Android app analysis technique, *FPE*. This generic framework can be used for examining behavior and extended for use in other domains.
- I demonstrate the utility of the *FPE* framework by examining its findings on a sample of Android apps. I also highlight successful applications of *FPE* in the following domains: detecting Android app clones and discovering capability leakages and PII leakages.
- I developed various Intent-based local DoS attacks. These concrete attacks were reported to Google, who made changes to the official codebase of Android.[4] I also proposed framework defenses for the DoS attacks and developed open-source apps to mitigate the attacks [55, 56].
- I evaluated the resiliency of a range of embedded Android devices to a persistent, local system crash DoS attack performed by a low-privilege Android app. I discovered that this attack, launched from an unprivileged app, can leave certain embedded Android

[4]Of the seven bug reports I made to Google, only one is publicly available: (`https://issuetracker.google.com/issues/37061958`). Many of the issues were marked as infeasible to fix, presumably due to the engineering effort involved in preventing them systematically. They did fix the following Intent-based DoS attacks in *their current form* as presented in Chapter 7. When I inquired if they could be made public, their response the following: "Thanks for your inquiry. Per our policy, reports filed in this component generally remain private. However, we have no objection to you publicly disclosing the information in this ticket, as you choose."

devices permanently disabled since it creates a ceaseless crash loop that is inescapable because they lack any known recovery mechanisms.

- I created open-source software, *Daze* [57], to test app components for certain classes of omission errors. I used *Daze* to test all apps on 32 Android devices. This also resulted in discovering system crash vulnerabilities in the official codebase of Android (which affected all vendors) for both the smartphone and Android TV branches. In total, *Daze* triggered 4,972 unique app crashes and 64 unique system crashes across all devices.

- I performed a longitudinal study of Android inter-app vulnerabilities covering 18,583 popular apps from Google Play, which provided meaningful insight into the vulnerability exposure window in terms of number of versions and time duration. During this study, *Daze* discovered 14,413 fatal exceptions in the set of 18,583 popular apps.

Below is a list of published manuscripts stemming from the research in this dissertation:

1. R. Johnson, Z. Wang, C. Gagnon, and A. Stavrou, "Analysis of Android Applications Permissions," in *2012 IEEE Sixth International Conference on Software Security and Reliability Companion*, June 2012, pp. 45-46 [58].

2. R. Johnson, Z. Wang, A. Stavrou, and J. Voas, "Exposing Software Security and Availability Risks for Commercial Mobile Devices," in *2013 Proceedings Annual Reliability and Maintainability Symposium (RAMS)*, Jan 2013, pp. 1-7 [59].

3. R. Johnson and A. Stavrou, "Forced-Path Execution for Android Applications on x86 Platforms," in *2013 IEEE Seventh International Conference on Software Security and Reliability Companion*, June 2013, pp. 188-197 [60].

4. R. Johnson, M. Elsabagh, A. Stavrou, and V. Sritapan, "Targeted DoS on Android: How to Disable Android in 10 Seconds or Less," in *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, Oct 2015, pp. 136-143 [61].

5. R. Johnson, M. Elsabagh, and A. Stavrou, "Why Software DoS Is Hard to Fix: Denying Access in Embedded Android Platforms," in *Applied Cryptography and Network Security*. Springer International Publishing, 2016, pp. 193-211 [32].

6. R. Johnson, M. Elsabagh, A. Stavrou, and J. Offutt, "Dazed Droids: A Longitudinal Study of Android Inter-App Vulnerabilities," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ser. ASIACCS 18. New York, NY, USA: ACM, 2018, pp. 777-791 [33].

7. M. Elsabagh, R. Johnson, and A. Stavrou, "Resilient and scalable cloned app detection using forced execution and compression trees," in *2018 IEEE Conference on Dependable and Secure Computing (DSC)*, Dec 2018, pp. 1-8 [62].

In addition, the list below contains published manuscripts that were not included in this dissertation:

1. Z. Wang, R. Johnson, and A. Stavrou, "Attestation & Authentication for USB Communications," in *2012 IEEE Sixth International Conference on Software Security and Reliability Companion*, 2012, pp. 43-44 [63].

2. R. Johnson, N. Kiourtis, A. Stavrou, and V. Sritapan, "Analysis of Content Copyright Infringement in Mobile Application Markets," in *2015 APWG Symposium on Electronic Crime Research (eCrime)*, 2015, pp. 1-10 [64].

3. R. Johnson, A. Stavrou, and V. Sritapan, "Improving Traditional Android MDMs with Non-Traditional Means," in *2016 IEEE Symposium on Technologies for Homeland Security (HST)*, 2016, pp. 1-6 [65].

4. R. Johnson, R. Murmuria, A. Stavrou, and V. Sritapan, "Pairing Continuous Authentication with Proactive Platform Hardening," in *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 2017, pp. 88-90 [66].

Lastly, below is a list of information security conferences that I presented at which are also referenced in this dissertation:

1. R. Johnson and A. Stavrou, “Resurrecting the READ_LOGS Permission on Samsung Devices,” presented at *Black Hat Asia 2015* [23].
2. R. Johnson, A. Benameur, and A. Stavrou, “All Your SMS & Contacts Belong to Adups & Others,” presented at *Black Hat USA 2017* [22].
3. R. Johnson and A. Stavrou, “Vulnerable Out of the Box: An Evaluation of Android Carrier Devices,” presented at *DEF CON 26 (2018)* [21].

1.4 Dissertation Structure

Chapter 1 of this dissertation provides context and the high-level motivation for the research. In addition, it also outlines the research questions and summarizes the contributions of the dissertation. Chapter 2 introduces fundamental concepts of Android apps and the Android ecosystem. Chapter 3 covers the necessary background on program analysis techniques, such as symbolic execution, concolic execution, *FPE*, and the differences between them. Chapter 3 offers additional details on *FPE* including a formal high-level definition of the *FPE* algorithm using an execution strategy that explores the input domain. Chapter 4 provides theoretical and real-world uses cases where externally defined program inputs directly control the exfiltration of sensitive user data.

Chapter 5 discusses conceptual and implementation details of *FPE* for Android. Chapter 6 discusses applications of *FPE* such as detecting PII leakages, capability leakages, identifying app clones, and detecting insecure programming practices. Chapter 7 examines the usage of Intent objects for low-privilege, local DoS attacks and their effects on other apps and the Android OS itself. These attacks cover the most common IPC mechanism to make the Android OS unresponsive to the user, target and kill other running apps, and cause a system crash. I intensified the system crash DoS attack by making it perpetual and investigated its effects on a range of Android devices. Notably, embedded Android devices, including an Android TV device, can be rendered useless by a low-privilege Android app.

Chapter 8 discusses software faults and inadequate error handling present in numerous Android apps and the Android OS with regard to their processing of received Intents. These faults, when manifested as errors, result in a system crash and can be leveraged to perform local DoS attacks. I also discovered that as a side effect, a system crash can also result in user data disclosure. I provide a method to programmatically detect concrete Intent input validation faults for apps and the Android OS and perform a longitudinal fault study involving apps from Google Play. Chapter 9 concludes with a succinct overview of my approach to answering the research questions posed in Chapter 1.

Chapter 2: Overview of Android

This chapter introduces and explains fundamental concepts related to Android that serve as a basis to contextualize the research in this dissertation. Section 2.1 introduces Android, its architecture, and some of its core software components. Section 2.2 provides details of the structure and contents of typical Android apps. Section 2.2.3 discusses the permission model imposed on apps by the Android OS. Section 2.3 briefly explains the primary mechanism that apps use to communicate within and between apps. Section 2.3.1 outlines some mechanisms Android uses to increase the overall availability and stability of the platform.

2.1 Android Open Source Project

Google has been the custodian and developer of Android since it purchased it in 2005. Android is generally considered to be one of Google's most successful acquisitions [67, 68]. Google has further developed Android into the OS with the largest global market share of any OS, overtaking Windows [13]. Google publicly releases the Android source code via the Android Open Source Project (AOSP). AOSP is a Google-led project that contains the source code for different versions of Android and also information on how to build, modify, and port the software to different platforms. In addition to locally cloning Google's online repositories using its `repo` tool, the source code files can be searched and viewed online via a web browser through official and unofficial online resources [69, 70]. The openness of Android lends itself to research, innovation, customization, expert users, and a significant diversity of *Original Equipment Manufacturers* (OEMs) that create Android devices.[1] Android OEMs create Android devices according to their own design specifications and branding. At the

[1]Interestingly, some OEMs modify the standard GUI of Android to mimic the experience provided by that of Apple's iOS [71]. Effectively, attempting to create an iPhone clone using Android.

same time, this openness has also resulted in disparate app marketplaces with differing levels of security vetting of apps [72, 73], pre-installed spyware and malware [22, 74], and many Android devices running older versions of the OS [75, 76].

In addition to OEMs, an *Original Device Manufacturer* (ODM) creates Android devices that will be subsequently rebranded or licensed and sold by another organization. Therefore, an ODM may manufacture a specific device for multiple vendors. An Android *vendor* is the entity in the supply chain that is responsible for the branding and direct or indirect selling of the Android device to the consumer. The Android vendor is generally the OEM that manufactured the Android device, although this is not always the case due to the presence of ODMs in the market.[2] Google provides a list of OEMs and ODMs that are partners in their Google Play Protect device initiative [77]. *Google Play Protect* is a local service on the device that scans the resident apps and then programmatically uninstalls or disable detected apps that are deemed by Google to be harmful to the user.

Google created a series of Android smartphone, tablet, and streaming media player devices under the brand name of *Google Nexus* [78]. The Nexus device models were manufactured by various ODMs (i.e., LG, Asus, Samsung, Huawei, Motorola, and HTC) to showcase a primarily unadulterated version of Android (i.e., AOSP) that contained minimal modification by the ODMs and carriers, allowing for faster OS updates and thus greater platform security.[3] Google appears to have ceased production of Nexus devices through external ODMs and has become an OEM itself, purveying a line of tablets and smartphones with a moniker of Pixel. The Pixel devices, like previous Nexus devices, allow the user to manually update the Android OS using firmware images. *Firmware images* contain the software for the Android OS and its necessary components. Google hosts firmware images for Nexus and Pixel devices that allow the user to easily upgrade or downgrade the Android OS version using command line tools that are included with the Android Software Development Kit (SDK). Some Android devices allow the user to *flash* unofficial firmware images that are

[2]In this dissertation, vendor and OEM will be used interchangeably, except where otherwise noted.
[3]Although these are major Android OEMs, they are acting as ODMs for Google Nexus devices.

not signed by the OEM. *Flashing* describes the process of writing a firmware image to a particular internal memory location on the device, overwriting the current firmware image to update or change its functionality. This fostered the development of several unofficial Android distributions that are referred to as *custom ROMs*. The term ROM, an acronym for *Read Only Memory*, is generally used interchangeably with the term *firmware images*.[4]

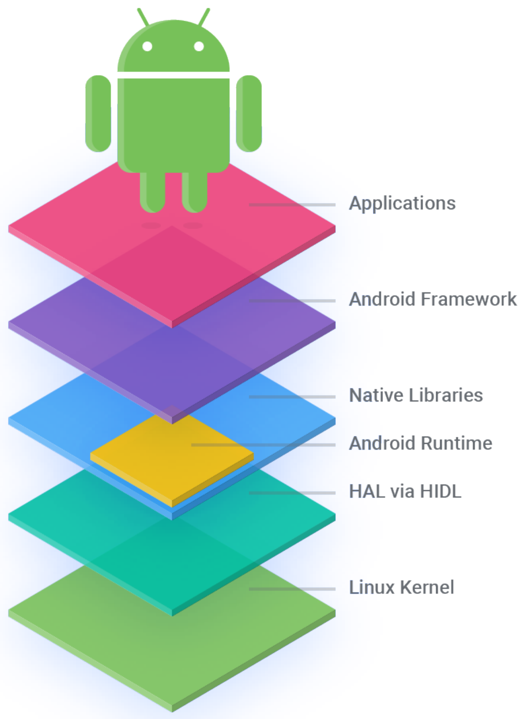


Figure 2.1: Google's depiction of the Android software stack.[5]

2.1.1 Android Architecture

The Android architecture can be decomposed into various interacting logical layers in a software stack where each layer has specific responsibilities. Figure 2.1 depicts Google's

[4]Some of the firmware images on an Android device are mounted on read-only partitions.
[5]This image is borrowed from Google's AOSP website: `https://source.android.com/`.

illustration of the Android software stack. At the top of the Android software stack are apps. Android apps come pre-installed from the device vendor and can also installed directly by the user. Android apps are primarily developed in the Java programming language, although they can also contain native code that is pre-compiled for different Central Processing Unit (CPU) architectures. Recently, Google has encouraged developers to utilize the Kotlin programming language instead of Java.[6]

The user generally spends most of their time using the device's native hardware (e.g., touchscreen) to interact with apps, as the remaining layers are somewhat abstracted from the user. The general user's view and understanding of the system may be limited to an extensible set of apps that provide functionality and utility. Android apps use the Android API to access device resources and capabilities through the Android framework.[7] The Android framework is primarily written in Java, although it does use native code libraries, mostly written in C and C++, to interact with the Linux kernel and hardware devices. During the development of an Android app, a Java `jar` file, generally named `android.jar`, from the Android SDK locally fulfills a compile-time external dependency for the Android API by containing implementation stubs. The `jar` file is specific to a particular Android API level since developers must choose the lowest Android OS version that can support the app with the required APIs. An *API level*, in this context, refers to a particular version of the Android OS and its corresponding API.

Android apps use a runtime environment called Android Runtime (ART) on recent Android OS versions. ART is the successor to the previous runtime environment named the Dalvik Virtual Machine (DVM). ART executes the app's bytecode using Ahead-of-time (AOT) and Just-in-time (JIT) compilation.[8] ART primarily uses AOT and the DVM uses JIT. The official switch from DVM to ART occurred in Android 5.0 (Lollipop) [79]. There is

[6]The interested reader can find a comparison of Java and Kotlin here: `https://kotlinlang.org/docs/reference/comparison-to-java.html`.

[7]The current Android API packages are located here: `https://developer.android.com/reference/packages`.

[8]Additional information about how AOT and JIT are used can be found here: `https://source.android.com/devices/tech/dalvik/jit-compiler`.

a trade-off between ART and the DVM. Notably, ART has better runtime performance, as the app is pre-compiled into native code, although this is at the expense of a larger memory footprint at runtime and a longer app installation time when compared to DVM.

Some of the API methods employed by apps use Remote Procedure Calls (RPCs) to interact with service threads residing in system processes using a kernel module named *binder* to enable expeditious communication between processes. A *Remote Procedure Call* is a Java language feature that allows the caller to obtain an interface to an object in a separate process and call its methods as if it is a local object in its process. The `ActivityManager` class in the Android API provides an illustrative example. When an app uses the `ActivityManager` class, it uses binder-enabled RPCs to interact with the `ActivityManagerServce` service thread that resides in a system process. Google does not provide extensive documentation for binder directly since app developers don't necessarily need to know the implementation of the underlying IPC mechanism.

The Android framework contains various native C/C++ libraries. The framework classes written in Java use the Java Native Interface (JNI) to execute native code. *JNI* provides a mechanism for managed code to call and execute native code. Many of the software projects comprising Android are open-source such as Webkit, OpenGL, Bionic Lib C, SQLite, OpenSSL, the Linux kernel, and others. Some of the Android framework API calls serve simply as wrappers to JNI calls to native code. In addition, the Android framework can interact with other processes using domain sockets, shared memory, and files. The Android framework contains high-level APIs that apps use to initiate communication with low-level sensors (e.g., camera, fingerprint sensor, GPS subsystem, etc.) on the device. This behavior is appropriately abstracted from the user of the API.

The native libraries either perform system calls directly or use the wrapper functions in the Bionic Lib C to interact with the Linux kernel. In addition, the native libraries use defined interfaces to interact with the Hardware Abstraction Layer (HAL) for accessing hardware devices.[9] Android provides a HAL with defined interfaces that must be implemented by

[9]Additional information on HAL can be found here: `https://source.android.com/devices/architecture/hal`.

Android vendors. HAL sits above the kernel and hardware and provides a unified interface for accessing underlying hardware drivers. The native libraries, ART, and HAL serve as *middleware* in between apps and the kernel. *Middleware* has a range of definitions depending on the context, although a broad definition is *software glue* residing between the client apps and the kernel, providing some logical abstraction.

Android uses a modified Linux kernel that contains extra facilities that are not present in a stock Linux kernel. These differences from the stock Linux kernel that are suited to the mobile environment are sometimes called *Androidisms*. Some of the Androidisms include Anonymous Shared Memory (ASHMEM), binder, Low Memory Killer (LMK), and Wakelocks [80, 81].[10]

2.1.2 Android Application Framework

The Android platform provides an extensive set of APIs for app developers to use in their apps. Each major release of Android provides additional features, security improvements, and changes to the Android API. The overall size of the API typically increases with each release, although methods and classes can become deprecated. All major releases contain an integer that denotes the particular Android SDK level, the platform version, and a code name.[11] The code names correspond to a dessert or treat that starts with a letter of the alphabet that increases with each subsequent version [82]. For example, the previous version of Android, 9.0, has a code name of *Pie*. Google announced that they were ending this tradition and instead of using a treat starting with Q, they will call the next release Android 10 [83]. With each Android OS release, Google provides updates on the major changes to the Android API.[12] The API modifications while providing additional functionality have also caused compatibility issues when app developers do not update their apps quickly to account for these changes [75, 84]. In addition to the Android SDK, Google also provides developers with a Native Development Kit (NDK) to enable developers to include native code in their

[10]The DoS attack described in Section 7.3.2 leverages LMK to target and kill external apps.

[11]Code names for Android releases have been provided since Android 1.5.

[12]For example, see `https://developer.android.com/about/versions/pie/android-9.0-changes-all` for the API changes in Android 9.0.

apps by pre-compiling binaries for different CPU architectures. The Android NDK may be used for integrating game engines, porting apps, and achieving high-performance.

During the creation of an Android app, a developer explicitly specifies the minimum SDK level that the app can support. Specifically, the developer declares a numeric value for the `android:minSdkVersion` attribute in the app's manifest file. This value denotes the lowest version of the SDK that can properly run the app. Each Android device is running a certain version of the Android OS that has a corresponding API level. If the `android:minSdkVersion` of an app is greater than the SDK level of the device, then the device will not be able to install the app since the app may contain APIs that the Android OS version does not support. On the other hand, Android devices are backwards compatible so that they can install and run any app that has an `android:minSdkVersion` attribute value that is not greater than the SDK level of the Android device. This behavior ensures that all the APIs used in the app will be present on the device at runtime.

Table 2.1: Distribution of Android device versions as of May 7, 2019.

OS Version	**Codename**	**Distribution**
2.3.3 - 2.3.7	Gingerbread	0.3%
4.03 - 4.0.4	Ice Cream Sandwich	0.3%
4.1.x - 4.3	Jelly Bean	3.2%
4.4	Kit Kat	6.9%
5.0 - 5.1	Lollipop	14.5%
6.0	Marshmallow	16.9%
7.0 - 7.1	Nougat	19.2%
8.0 - 8.1	Oreo	28.3%
9.0	Pie	10.4%

Due to the Android fragmentation problem [75, 76], app developers tend to target lower versions of the Android SDK to be able to reach the widest number of potential devices [85]. The Android fragmentation problem is illustrated in Table 2.1. This table was created from data provided by Google's *Distribution dashboard* webpage that contained the distribution of Android versions as of May 7, 2019.[13] The *Android fragmentation problem* is the result of having multiple vendors that do not update their devices in a timely manner to the most recent stable version of the Android OS. Therefore, the devices from different vendors may be running different major and minor versions of Android depending on the vendor's level of support in updating the device OS version and keeping it current. Google provides periodic snapshots of the current SDK levels of devices that connect to Google servers with their *Distribution dashboard* webpage. App developers have a financial incentive to have their app be compatible with the largest number of devices to maximize any potential ad revenue and paid downloads. An app with an older (i.e., lower) SDK version running on a device with a more recent (i.e., higher) SDK version can still use newer features provided by the device by using Java Reflection APIs to call them at runtime. In practice, this takes effort to maintain due to the changes in behavior and presence of APIs among the different API levels and can also cause security weaknesses due to old behavior being utilized [76].

2.1.3 System Server

System server is a critical system process that provides services to apps through the Android framework.[14] The system server process is not well documented on the *Android Developers* website since this process is mostly abstracted from the developer. System server executes as the `system` user and provides various service threads that are externally exported and available to local processes on the device.[15] System server employs a client-server architecture where apps interact with its interfaces to obtain functionality. Since the client and server reside in different processes, Android uses an IPC mechanism named *binder* (see Section 2.1.1). Sensitive functionality in Android is guarded by permissions that an app must possess at runtime to perform certain protected behavior. For the permissions defined by the Android framework, an app generally uses IPC to interact with the system server process to access

[13]This table was recreated from the table provided here: `https://developer.android.com/about/dashboards`.

[14]The system server process has a name of `system_server` in the process status list, so this name is also used in the dissertation to reference this process.

[15]The services running within system server can be listed with the following command: `service list`.

sensitive functionality. Prior to performing the desired behavior, system server will check to see if the calling process has been granted the permission(s) required. System server keeps track of the permissions granted to each app using its package manager (i.e., the `PackageManagerService` class).

When apps use the functionality of system server, they generally obtain a handle to a service interface using the `android.content.Context.getSystemService(java.lang.String)` API method call, which returns an `IBinder` instance, dictated by the `String` parameter, to the desired service in system server. Listing 2.1 provides a concrete Java source code example of an app using RPCs to interact with system server to set an alarm to start an app every 60 seconds. In line 1 of Listing 2.1, the `IBinder` instance is cast to an object type in the Android API (e.g. the `AlarmManager` in the local app that interacts with `AlarmManagerService` in the system server process). Then the local app can interact with the system server process using RPCs, so that it appears that the app is simply making a method call on a local object, even though it crosses process boundaries.

```java
1 AlarmManager alarmManager = (AlarmManager) getSystemService(ALARM_SERVICE);
2 Intent intent = new Intent(getApplicationContext(), MainActivity.class);
3 PendingIntent pendingIntent = PendingIntent.getActivity(getApplicationContext(), 0,
      intent, 0);
4 alarmManager.setRepeating(AlarmManager.RTC_WAKEUP, 1000, 60000, pendingIntent);
```

Listing 2.1: Example Java source code for using system server functionality.

The system server process is a single point of failure since its termination causes other critical system processes to be terminated. In addition, since system server provides an extensive attack surface accessible to local apps, it makes it an attractive target for DoS attacks. A DoS attack against the `system_server` process indirectly controls the availability of the device since a crash of this process results in a user space system crash. During the boot process, the `init` process starts the `zygote` process. The `system_server` process is started by the `zygote` process. All apps are spawned from the `zygote` process which gives them a speedy setup time as the resources are already pre-loaded. Various DoS approaches that I developed against the `system_server` process, as well as the attack mechanics, are provided in Chapters 7 and 8. The base attack methods covered in these chapters do not require the attacking app to have any permissions; therefore, they can be accomplished to the lowest-privileged app (i.e., an app that has no granted permissions).

2.2 Android Apps

Apps provide extensible functionality beyond that of the device's pre-installed software (e.g., Android OS, pre-installed apps, etc.). Pre-installed apps are present when the user first powers on the device. In addition to the apps that come native to the device, apps reach the device through official and unofficial app distribution channels. On many Android devices, the default app distribution platform is Google Play. There are notable exceptions where Android vendors do not include the Google Play app or any other Google apps beyond what is present in AOSP. Some notable exceptions that don't use Google Play are Amazon Fire devices and certain vendors that provide their own app distribution platforms to download and install apps, such as Xiaomi. Moreover, some vendors include both Google Play and their own app store such as Samsung's Galaxy Apps.

During the initial installation of an Android app, the Android OS assigns an app a specific user ID (UID) that remains constant, even during app updates, until the app is uninstalled. The assigned Linux UID protects the app's *private directory*, subdirectories, and constituent files. Each installed app is provided with a private directory that is only accessible to the app itself, by default, due to restrictive file permissions.[16] This private directory currently, as of Android 9, has a path of `/data/data/<package name>` where this directory is owned by the app's UID and also has a Group ID (GID) value that is identical to that of the UID. Moreover, each app, be default, executes in its own process which safeguards its memory. These two features *sandbox* the app's memory and files by isolating it from other processes.

[16] An app can intentionally share its UID and thus files and permissions with other apps by being signed with the same key and using the same values for the `android:sharedUserId` attribute in their respective app manifest files.

2.2.1 App File Structure

Android Package (APK) is the name of the Android app file format that has a file extension of `apk`. The APK file format is a compressed archive that encapsulates all of an app's files. Some files and directories that are generally contained within a typical APK file are the following:

- `AndroidManifest.xml` - configuration data file
- `classes.dex` - Dalvik bytecode file
- `lib` - directory containing native code libraries
- `assets` - directory containing arbitrary files
- `res` - resource directory containing images, layout files, string resources, etc.
- `resources.arsc` - compiled resource file
- `META-INF` - directory containing version and signature data

2.2.2 App Manifest File

The app's manifest file, `AndroidManifest.xml`, serves as a repository for settings and configuration data. Listing 2.2 displays a terse, valid manifest file, illustrating some of the minimal data that is required by the Android platform. Notably, the manifest contains the *package name* of the app which is a string that serves as one of the primary identifiers of an app. On

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:sharedUserId="android.uid.system" package="edu.gmu.example" platformBuildVersionCode="24" platformBuildVersionName="7.0">
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.BLUETOOTH" />

    <application android:allowBackup="false" android:icon="@mipmap/ic_launcher" android:label="@string/app_name" android:name="edu.gmu.example.MyApp">
        <activity
            android:name=".GenMainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name=".GenService"></service>
        <meta-data android:name="KEY" android:value="VALUE"/>
    </application>
</manifest>
```

Listing 2.2: Concrete example of a valid `AndroidManifest.xml file.`

an Android device, only one app with a specific package name can be installed at a particular time. Google Play allows its users to search for an app using its package name. The package name in the manifest is declared in the `manifest` element as `edu.gmu.example` in the `package` attribute. For version information, the manifest file includes the `android:versionCode` and `android:versionName` attributes to the `manifest` element.[17]

The manifest file contains various information to identify the app and its constituent app components. Each app component, with the exception of broadcast receivers that are dynamically registered, must be statically registered in the manifest file. An *app component* is a named executable unit of an app or a repository for structured data. For example, in Listing 2.2, the app statically registers an activity app component named `GenMainActivity` and a service app component named `GenService`. The `uses-feature` element in the manifest allows an app to enumerate the required hardware and software features that the device must have for the app to work properly. In the declaration for each feature, the `android:required` attribute allows the app to indicate if the app cannot function without the listed feature. Some example features are `android.hardware.fingerprint` and `android.software.sip`. The `uses-feature` element informs app distribution channels, such as Google Play, the required features and thus determines which apps are available to download to the current device.

The Android OS requires that an app contain a valid `AndroidManifest.xml` file for it to be installed on the device. Without a manifest file, the Android OS would not be able to identify the app, determine its desired capabilities, and locate it entry points with regard to app components. Each time the device is turned on and boots up, a service thread (i.e., `PackageManagerService`) within the `system_server` process examines the properly installed APKs and parses their manifest files. This allows the system to know which apps are installed and the app components that they contain, allowing it to properly deliver Intents to their destinations. Moreover, it gives the Android OS awareness of the apps in general and their capabilities, especially with regard to knowing which apps have been granted which permissions. The permissions that the app requests can be an important consideration

[17]Some of the available elements in the `AndroidManifest.xml` can be found here: `https://developer.android.com/guide/topics/manifest/manifest-intro`.

for the user to inform them of the scope of the app's behavior. In the manifest, an app requests a permission with the `uses-permission` element. In Listing 2.2, the app requests the `android.permission.INTERNET` and `android.permission.BLUETOOTH` permissions which allow the app to directly access the internet and use Bluetooth functionality. In addition, an app can declare its own permissions and set the access requirements for external apps to acquire them.

2.2.3 App Permission Model

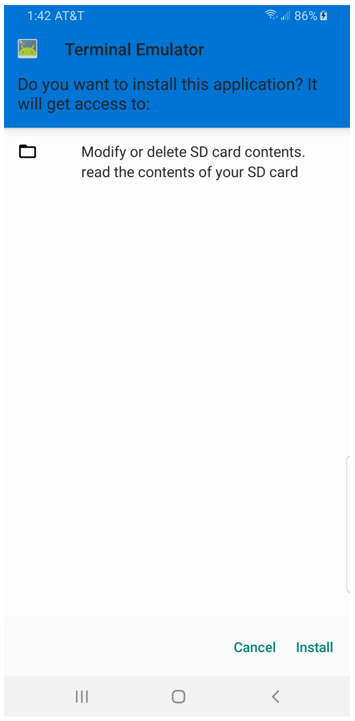
The Android platform provides permissions that apps can request to obtain access to protected functionality and resources on the device.[18] A pre-installed core app with a package name of `android` declares the platform permissions and sets the requirements for all other apps to obtain them. Each permission is simply a string that app developers request in their app's `AndroidManifest.xml` file using a specific syntax. The platform-defined permissions in the AOSP master branch are provided in the core manifest file for the platform [69]. Once an app has obtained the permission(s) it requests, then it can perform the behavior associated with the permission(s). In practical terms, this will allow the app to pass the permission checks associated with permission-protected API calls. For example, once an app has been granted the permission named `android.permission.CALL_PHONE`, the app can programmatically initiate phone calls to non-emergency numbers without user interaction.

A runtime permission granting mechanism was first introduced in Android 6.0 (API level 23) [86]. This mechanism provides the user with more granular control over app behavior since permissions are no longer granted on *all or nothing* basis, as they were in all previous Android versions (API level 22 and lower). By an *all or nothing* basis, I mean that the app could only be installed with all the permissions it requests or the user could deny the installation but not install the app and selectively grant it permissions. With the runtime permission granting, the app will prompt the user to grant or deny it some of the permissions

[18]The AOSP platform-defined permissions are provided here: `https://developer.android.com/reference/android/Manifest.permission`.

it requests. Apps that declare a target API level of 23 or higher are supposed to still be able to operate even without being granted all of the permissions they request. Certainly, this can inhibit some of the app's functionality, but it provides the user the discretion to deny permissions to an app that seem unnecessary or risky.

The runtime permission granting mechanism is active for an app if two conditions are met: (1) the Android device running the app has an API level of 23 or higher and (2) the app's `targetSdkVersion` attribute for the `uses-sdk` element is set to 23 or higher in the app's manifest file [87]. If either of the two conditions do not hold for a particular device and app pair, then all of the permissions will be granted to a third-party app (e.g., `normal` and `dangerous`) when the user installs the app. When apps are installed outside of the standard app distribution channel (e.g., Google Play), then a list of the permissions that the app requests is presented to the user as shown in Figure 2.2.



Do you want to install this application? It will get access to:

Modify or delete SD card contents.
read the contents of your SD card

Figure 2.2: Android app installation dialog.

The installation dialog for the app shown in Figure 2.2 has a `targetSdkVersion` attribute value of 22, so all `dangerous` permissions that app requests are shown to the user. If an app is subject to runtime permission granting, then the installation dialog may not list the `dangerous` permissions in this dialog and present them to the user to grant or deny when the app requests them at runtime. Installing an app outside of an app store requires that the user allow an option that is found in the Settings app: installation from unknown sources [88]. A popular shooting game named *Fortnite*, provided their own app installer that had to be downloaded outside of Google Play. An early version of the Fortnite app installer did not perform any validation of the app that it installs which could allow an attacker with an app co-located on the device to install an app other than the intended Fortnite app [89].

One of the major assumptions underlying the Android permission model is that users are informed enough to understand the functionality corresponding to each permission. Felt et al. [90] found that only 17% of users examined the permissions at install time and that only 3% of respondents in a survey could successfully answer all three permission comprehension questions. This seminal study examining user behavior and comprehension of Android permissions was published in 2012. More recent studies show that even when the permissions are understood, there may be little user comprehension in what they are used for [91, 92].

2.2.4 App Components

Android provides four different app components from which an app can be built: **activity**, **service**, **broadcast receiver**, and **content provider** [93]. An **activity** provides a GUI for direct user interaction via GUI elements. A **service** runs in the background and tends to process long running or persistent tasks. A **broadcast receiver** serves essentially as an event listener where it is registers for certain types of events and responds to them as they occur. A **content provider** provides access to structured data, which is usually in the form of an SQLite database. Each app component must be statically registered in the app's `AndroidManifest.xml` file, except for broadcast receiver components which can be dynamically created and registered at runtime. The compartmentalization of an Android

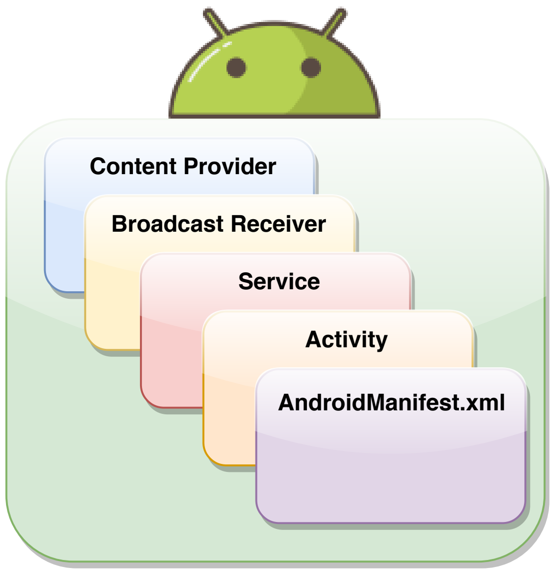


Figure 2.3: Abstract composition of an Android app.

app divides the app into components so that it is not a monolithic program with a single entry point. An abstract version of an Android app and its constituent components are provided in Figure 2.3. Both the activity and service app components have a lifecycle that is specific to its app component type. By *lifecycle*, I mean that the app component starts in a specific state and makes predictable transitions to other states in response to user and system events. The broadcast receiver and content provider have different use cases and thus do not have a rigid lifecycle states in which they progress.

2.3 Android Inter-Component Communication

Much of the communication between and within Android apps occurs via *Intent* objects: a message-like abstraction provided by the Android framework. Intents are a fundamental communication mechanism such that all but the most primitive apps will utilize Intent messages for communication and data exchange. Figure 2.4 conceptually demonstrates the *information flow* between and within apps using Intent objects.

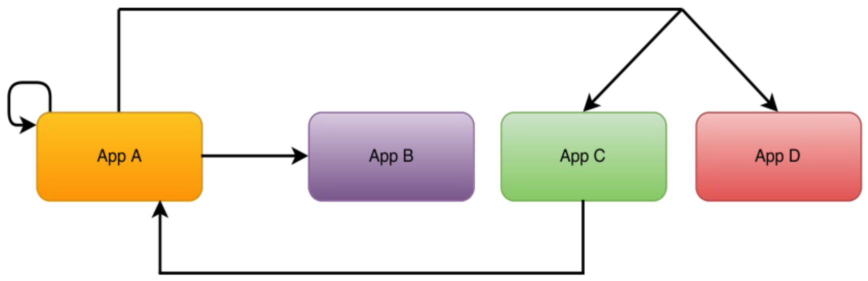


Figure 2.4: Intent usage in Android.

An Intent object requires one necessary item to be of any functional use: *a destination*. An explicit destination can be provided by specifying a recipient app component name in the intent. An explicit app component destination *address* consists of the package name of an app and a fully-qualified class name within the app. These two pieces of data provide a unique recipient location for the delivery of the Intent. When the Intent is delivered, the recipient app component is started if it is not already executing. The Intent and its embedded data that is sent to it will be accessible to the destination app component for processing. An Intent that contains a concrete and unique destination app component is also known as an *explicit intent*. An example of an explicit intent is provided in Listing 2.3, which on vulnerable Essential Android smartphones will cause a programmatic removal of user data (i.e., a *factory reset*) without the corresponding access permission.[19] Additional vulnerabilities stemming from insecure access control exhibited by app components of pre-installed apps is

```
Intent i = new Intent();
i.setClassName("com.ts.android.hiddenmenu", "com.ts.android.hiddenmenu.rtn.
    RTNResetActivity");
startActivity(i);
```

Listing 2.3: Example source code for sending an explicit Intent.

[19]This vulnerability was assigned CVE-2018-14994 and is available here: `https://nvd.nist.gov/vuln/detail/CVE-2018-14994`.

provided in Section 8.3.3.

```
Intent intent = new Intent();
intent.setAction("com.tmobile.oem.RESET");
sendBroadcast(intent);
```

Listing 2.4: Example source code for sending an implicit Intent.

In addition to specifying a particular app component, as in Listing 2.3, Android offers decoupling from using concrete names for target app components, effectively binding them to a particular app, by specifying an action string. This decoupling allows apps to focus on an *action* instead of unique endpoint app components, providing a level of indirection. The Intent shown in Listing 2.4 contains only a string that denotes the general action of the Intent (i.e., `com.tmobile.oem.RESET`), moving to a higher-level of abstraction, as various components on the device may be able to handle a particular action. Android has provided various hard-coded string constants that serve as generic actions that apps can choose to handle by having one or more components in their app register to receive them. There are many action strings that are reserved for specific known actions on the device. For example, the `android.content.Intent` class declares numerous platform-defined actions where the constant name generally has a prefix of `ACTION_` (e.g., `ACTION_CALL` for programmatically initiating an outgoing phone call). The platform-defined actions can generally be segmented into those are intended for activity app components and broadcast receiver app components, as is done on the Intent API on the *Android Developers* website.

The actions for activity components can generally be sent and also received by a third-party app. For example, an app could send an Intent to start an activity app component with a specific action (e.g. `ACTION_DIAL`) and that same app could have a different activity that handles the action that is sends. When the Intent contains an action but lacks a definite destination component, then the Intent can potentially be delivered to multiple activity app components on the device that can handle the action. When there is more than one activity app component that can handle the implicit Intent, the Android OS provides the user with a *chooser* that allows them to select the specific app that they would like to handle the Intent.

For example, an app may send an Intent with an action of `ACTION_VIEW` which is very general. The appropriate destination components that can handle the Intent can also be narrowed down and resolved via Intent filters, where a component can specify various Multipurpose Internet Mail Extension (MIME) types and protocols that it can handle in addition to just an action. *Intent filters* are used to provide more granular detail about the types of intents that an app component can handle by specifying an action, category, and data type.[20]

The platform-defined actions for broadcast receiver app components are generally meant to be sent by the Android OS and received by apps. Broadcast intents can be delivered to multiple recipient apps, unlike implicit Intents with platform-defined actions for activities.[21] Therefore, any app that has registered for an action can receive it *assuming* it does not require a permission to receive. Listing 2.4 provides a Java source code snippet to send an implicit broadcast Intent with an action string of `com.tmobile.oem.RESET`. This broadcast Intent will be delivered to any app on the device that statically or dynamically registers for this particular action string. When the broadcast Intent, shown in Listing 2.4, is sent on certain versions of the T-Mobile Revvl or the Cooldpad Defiant devices sold by T-Mobile, the broadcast intent will be received by an app with a package name of app `com.qualcomm .qti.telephony.extcarrierpack` which initiates a programmatic *factory reset*, resulting in the loss of user data [21].[22] The corresponding declaration of a broadcast receiver app component named `UiccReceiver`, shown in Listing 2.5, to receive broadcasts with an action

```xml
<receiver android:name="UiccReceiver">
    <intent-filter>
        <action android:name="android.intent.action.SIM_STATE_CHANGED"/>
        <action android:name="android.intent.action.SHOW_CAP_NO_SUPPORT"/>
        <action android:name="com.tmobile.oem.RESET"/>
        <action android:name="org.codeaurora.intent.action.
            ACTION_SIMLOCK_TEMP_UNLOCK_EXPIRED"/>
    </intent-filter>
</receiver>
```

Listing 2.5: UiccReceiver broadcast receiver declaration in its manifest file.

[20]Additional data about intents and intent-filters can be found here: `https://developer.android.com/guide/components/intents-filters`.

[21]The most commonly sent implicit Intents in a study of 2,644 apps is provided in [94].

[22]The T-Mobile Revvl Plus is a carrier-branded device that was manufactured by Coolpad.

string of `com.tmobile.oem.RESET`.[23] The sender of an Intent can explicitly require that a process possess a permission to successfully receive the intent. For example, when a text message is received by the phone process, it will send out a broadcast Intent with an action string of `SMS_RECEIVED_ACTION`. Any app that registers to receive this broadcast action must also possess the `RECEIVE_SMS` permission to actually receive the Intent that has the data from the received text message.

An app can statically register to receive a specific action in its `AndroidManifest.xml` or dynamically register to receive broadcast Intents with specific actions at runtime. Certain broadcast actions may be of interest to numerous apps such as those with an action of `ACTION_AIRPLANE_MODE_CHANGED` that is sent by the system when the device enters or exits airplane mode. *Airplane mode* is a configuration on the device that prevents the device from emitting and receiving any signals using wireless communication. To apps, it may be of great interest to know when network connectivity is not available.

Unfortunately, developers can unintentionally expose interfaces in their Android apps, making them accessible to other apps co-located on a mobile device. An exposed interface offers a potential entry point into an app that can be abused. This can be caused by improper handling of received Intents, for instance, making assumptions about the presence of certain data in a received Intent. In addition, inadequate exception handling of inter-app messages can enable an app to *force-crash* other apps and services or even the Android Operating System (OS) itself. System crashes can be intentionally and repeatedly caused by a malicious app on the device to create local DoS attacks or perhaps a crypto-less ransomware. Although the Android OS offers mechanisms to force-remove third-party apps, these removal methods may not be available on all devices or may require wiping all user data on an infected device [32], as detailed in Section 7.3.3.

[23]This component declaration is from the `AndroidManifest.xml` file of app with a package name of `com.qualcomm.qti.telephony.extcarrierpack` from the T-Mobile Revvl Plus device.

2.3.1 Android Availability Model

Android strives to be a stable and secure mobile OS. The Android OS takes numerous steps to create security boundaries around apps themselves and their interaction with the OS. Notably, each app is provided with its own Linux UID and GID at install-time that is used to protect each app's memory and files. This *app sandbox* increases security, and it also increases the availability of an app by reducing the attack surface for external interference. The *Android Developers* website states: "By default, apps can't interact with each other and have limited access to the operating system." with regard to the Android app sandbox [95]. The previous statement is not accurate in all cases and circumstances since the IPC mechanism allowing communication between apps can be abused to send crafted Intent messages to another app in order to make it encounter an uncaught exception. Each app can open external interfaces into its app which can be made available to other apps. When an Android app encounters an uncaught exception, this generally causes the entire app to crash.[24]

The default uncaught thread handler for the app's main thread will terminate the process when an uncaught exception occurs. This event shows up in the system log in a defined format as a FATAL exception with a corresponding stack trace. Listing 2.6 display a fatal crash in the phone process, `com.android.phone`, and a fatal system crash occurring in an Android system process is provided in Appendix D. Inspecting the stack trace shown in Listing 2.6 reveals that the root cause for the crash was a `NullPointerException` due to the phone process derefencing a `null` object. Line 3 in Listing 2.6 provides the cause of the fault, highlighted in red and also displayed on line 15, which is that the `com.android.phone` process *assumes* that every Intent it receives will contain a non-`null` `Bundle` object without checking at runtime to see if the `Bundle` object is `null`. A `Bundle` object contains key-value pairs and can be embedded within an Intent object for data sharing. Since the process does not perform a `null` check on the `Bundle` object prior to derefencing it, it encounters a `NullPointerException`, resulting in process termination. I systematically studied the ability

[24]The crashing of an app is also informally known as a *force close*.

```
FATAL EXCEPTION: main
Process: com.android.phone, PID: 8584
java.lang.RuntimeException: Unable to start activity ComponentInfo{com.android.phone/
    com.android.phone.NetworkSelectSettingActivity}: java.lang.NullPointerException:
    Attempt to invoke virtual method 'int android.os.Bundle.getInt(java.lang.String)'
     on a null object reference  at android.app.ActivityThread.performLaunchActivity(
    ActivityThread.java:3108)
  at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:3251)
  at android.app.servertransaction.LaunchActivityItem.execute(LaunchActivityItem.java
      :78)
  at android.app.servertransaction.TransactionExecutor.executeCallbacks(
      TransactionExecutor.java:108)
  at android.app.servertransaction.TransactionExecutor.execute(TransactionExecutor.
      java:68)
  at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1948)
  at android.os.Handler.dispatchMessage(Handler.java:106)
  at android.os.Looper.loop(Looper.java:214)
  at android.app.ActivityThread.main(ActivityThread.java:7045)
  at java.lang.reflect.Method.invoke(Native Method)
  at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.java
      :493)
  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:964)
Caused by: java.lang.NullPointerException: Attempt to invoke virtual method 'int
    android.os.Bundle.getInt(java.lang.String)' on a null object reference
  at com.android.phone.NetworkSelectSettingActivity.onCreate(
      NetworkSelectSettingActivity.java:49)
  at android.app.Activity.performCreate(Activity.java:7327)
  at android.app.Activity.performCreate(Activity.java:7318)
  at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1271)
  at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:3088)
  ... 11 more
```

Listing 2.6: Fatal exception in the phone process.

of an app to crash other apps and the Android system using Intents in Chapter 8.

Chapter 3: Exploring States of a Program

Various methods have been developed by researchers in an attempt to execute as many conditional branches of a program as possible to identify software faults. These methods include symbolic execution [96, 97], concolic execution [98–102], coverage-oriented genetic algorithms [103, 104], model-based GUI testing [42, 105], mutation testing [106, 107], and fuzz testing [39, 108]. These approaches aim to increase the code coverage of a program under test to enter execution branches that may remain elusive using other testing methods such as random testing [109] and primitive ad-hoc testing [110] which are not informed by the logic of the underlying program under test. Exploring program states can uncover faults that may only manifest when very specific conditions are met (see Chapter 4). To that end, Sections 3.1 and 3.2 conceptually describe symbolic execution and concolic execution, respectively. Section 3.3 introduces *FPE* and formally defines it in Section 3.3.1. Section 3.3.2 describes the differences between *FPE* and symbolic execution and concolic execution and provides motivating examples.

3.1 Symbolic Execution

Symbolic execution was created in the mid-1970s by King [96, 111] and Boyer et al. [97] as a method to perform program verification that went beyond random testing to allow for dynamic test generation. *Random testing* involves randomly selecting input values from the input domains and using them as input in test cases, which can result in redundant exploration of execution paths and limited code coverage due to the extensive size of the input domains. For example, the input domain for the integer primitive data type contains 2^{32} different values. Symbolic execution is informed by the constants and derived values from within a program. Symbolic execution only considers paths that are truly executable for a set of concrete inputs during execution and excels at reaching code used in boundary cases. Symbolic execution can be used to achieve complete branch coverage in smaller programs, although this can quickly become infeasible with larger programs. Generally, an interpreter needs to be created to process the target programming language so that the source code or bytecode can be executed symbolically. The interpreter understands the syntax and semantics of the program code, which are modified to operate on symbolic values. Executing a program symbolically can be used to identify a concrete set of inputs that, when executed, will reach a particular state of a program or program unit [112]. For each feasible path, a concrete set of inputs is generated as a test case to be executed concretely to identify bugs during program testing. The generated test cases increase code coverage and evaluate program correctness during testing.

Using symbolic execution, the program code is executed symbolically where input values are treated as symbolic values instead of concrete values. A *symbolic value* can represent any arbitrary value within an input class domain and operations are performed on the symbolic value. A *path condition* is a set of constraints encountered in conditional statements during symbolic execution that must be satisfied to take a particular execution path through the program during execution. When a symbolic value is used in a conditional statement, the constraint (e.g., $x < 5$) is added to the set of constraints comprising the path condition. In addition, the symbolic execution is *forked* so that the alternate path from an encountered conditional statement can be explored concurrently or subsequently by adding the negation of the current constraint (e.g., $x \geq 5$) to the path condition. Once the termination of the program or an error is reached, a constraint solver is used to check if the path condition can be solved. A constraint solver will determine if the path condition is satisfiable. If the path condition is *satisfiable*, a concrete set of inputs will be generated that, when executed concretely, will take the same execution path that was traversed during symbolic execution. For example, when the constraint solver is provided with a path condition of $x \geq 5 \;\wedge\; x \neq 15 \;\wedge\; x < 20$, it may determine that a possible value of x that satisfies each of the three predicate clauses is $x = 19$. Alternatively, the constraint solver may determine that a path condition is not satisfiable since it contains conflicting constraints (e.g., $x = 10 \ \wedge \ x \neq 10 \wedge \ x < 18$) or is beyond the solver's capabilities to solve. The constraint solver is limited by time and computational resources, its implementation, and the undecidability of Boolean constraint satisfiability in certain cases [113]. Generally, as additional constraints are added to the path condition, the complexity of solving the path condition increases. Various optimizations for symbolic execution have been introduced to increase performance such as expression rewriting, implied value concretization, constraint independence, counter-example cache, and coverage-optimized search [114].

A major difficulty of using symbolic execution is the exponential growth of potential execution paths that occur as the number of conditional statements increase in a program. General code complexity and loops can lead to a phenomenon known as the path explosion problem, although this is not exclusive to symbolic execution. The *path explosion* problem occurs when the number of paths to explore becomes prohibitively expensive in practice with regard to time or resources. Due to this phenomenon, the time and memory required to visit each execution path and use the constraint solver to determine satisfiability can become infeasible due to potentially infinite paths through a program. In practice, only a subset of all execution paths is explored using symbolic execution in non-trivial programs. If the loop condition contains a symbolic value, this can potentially lead to an infinite number of execution paths. This can also occur for unbounded loops and recursion, which can result in a symbolic execution tree of theoretically infinite size. Library calls can also be an issue since it is possible that only the program binary is available and the library binary is dynamically linked at runtime. The determinism of the code can also complicate symbolic execution for reproducibility and test generation. Non-determinism can lead to bugs that are false positives since it may cause a different execution path than the intended one at runtime. In addition, artifacts of the execution environment affect code coverage due to how it is handled and modeled by the symbolic execution engine. Symbolic execution has a difficult time reasoning about arrays [115], pointers [116], and cryptographic and hash functions [117]. Given these strengths and weaknesses, symbolic execution has been used for

Android in detecting privacy leaks [118], systematic test case generation [119], and simply as a proof-of-concept protoype [120].

3.2 Concolic Execution

Concolic execution is a slight refinement to the precursor research by Korel [101, 102] and Offutt et al. [98] on dynamic symbolic execution and dynamic domain reduction, respectively, to address some of the shortcomings of a pure static symbolic execution approach. Later works in the mid-2000s [99, 121, 122] further refined dynamic symbolic execution and coined the term *concolic* as a merger and contraction of the terms *concrete* and *symbolic*. *Concolic execution* utilizes both symbolic execution and concrete execution to explore the execution paths of a program or program unit. As with symbolic execution, its general purpose is to maximize the code coverage of a program under test in order to verify program correctness. Concolic execution instruments the source code or bytecode to perform symbolic execution, which alleviates the need of using an interpreter to process the source code or bytecode of a program. Concolic execution uses symbolic execution to generate concrete sets of inputs that will cause the execution of the program to reach different states in the program under test. The bugs that are encountered using concolic execution are true positives (i.e., real and verified bugs) since they are encountered at runtime with concrete execution. Concolic execution maintains a concrete state and symbolic state of the program. Some approaches utilize symbolic execution to solve the path condition as concrete execution occurs and others wait until the end of an execution path to utilize the constraint solver.

The approach used by CUTE [99], a concolic engine for C programs, traverses paths iteratively instead of forking at conditional statements. During the initial concrete execution run, user-selected or random values for the inputs are provided to the program or program unit under test. A depth-first search approach executes until the program terminates, a bug is found, a timeout occurs, or a user-selected limit to the depth of the analysis is reached. During concrete execution, the instrumented code produces an execution trace. A path condition is created based on the constraints present in the trace. Then a constraint in the path condition (e.g., the last constraint encountered in the path condition) is negated so the next concrete execution will take a different path through the program.[1] A constraint solver is used to try to generate a set of inputs that solves the modified path condition. If the path condition is solvable, then the new input values from the constraint solver are used in the next concrete execution. The new input values will take a new execution path through the program and create a new trace from which a new path condition will be generated and modified. This process continues in an iterative fashion until all execution paths have been taken or a user-selected timeout value is reached.

Concolic execution reduces some of the cost of solely using symbolic execution to solve the path conditions. There are instances where the constraint solver is not able to generate concrete values for data and memory addresses for a path condition. In these cases, some of the symbolic constraints may be replaced by concrete values, which satisfy the path condition as an approximation. Concolic execution can support the concrete execution of libraries so that they are treated as black boxes and are not subjected to symbolic execution. The values used during the initial concrete execution have an effect on the subsequent paths that will be explored, so the initial values should be carefully selected. Some errors that symbolic execution and concolic execution can detect are uncaught exceptions, division by 0, null pointer deference, incorrect computations, and segmentation faults. Concolic execution is subject to the path explosion problem due to the complexity of non-trivial software, which can reduce the code coverage of a program. The building of the path condition becomes more expensive as execution proceeds in terms of storage and complexity for the solver. In addition, an adversarial program may make an attempt to identify an instrumented analysis environment and withhold behavior. Concolic execution has been used for Android apps to explore the GUI [100] and for dynamic analysis [123].

[1]The constraint being negated is sometimes called the *target constraint*.

3.3 Forced-Path Execution

The initial concept of *FPE* was first introduced in 2005 by Lu et al. [124] in a technical report as an approach to detect software bugs. This approach has been used to identify software failures [125, 126] and kernel rootkits [127], examine binaries from a security perspective [128], and by myself to profile the behavior of Android apps [59, 60]. The *FPE* framework I developed aims to elicit and enumerate the potential behavior of a program under test as it programmatically interacts with the GUI of the program and simulates different environments and runtime conditions. *FPE* can force execution into code branches independent of how the predicate of a conditional statement evaluates to explore both branches. The exact conditions that dictate when the *forcing* into a conditional branch depends on the specific execution strategy that is being used (see Section 3.3.3).

Some program branches within an app may require *very* specific conditions to be satisfied for execution. For example, the behavior of an app can depend on its execution environment. For example, its behavior may depend on system properties, versions of software, presence of other apps, system clock, network response, location, and user input. During concrete execution of the program, the execution environment may not contain the necessary conditions to make the app exhibit certain behaviors. Apps can contain *triggers* that, when met, will make the program exhibit exceptional behavior [22, 129]. A *trigger* is a specific condition or set of conditions that must be met for a specific event to occur. For example, there are programs that only display certain behavior after a certain period of time has elapsed, which is unlikely to be met during a normal analysis time frame. The *FPE* framework will enter the time-based branches independent of the amount of time that has passed to explore its behavior. Chapter 4 provides some theoretical and real-world motivating uses cases in Android apps that demonstrate the utility of forcing the execution path in certain circumstances.

3.3.1 Formal Definition of Forced-Path Execution

This section introduces some formal definitions to describe a model of the default execution strategy, named *exploring the execution environment* (see Section 3.3.3), in which the *FPE* framework uses taint analysis to determine when a branching decision should be forced during predicate evaluation. *Taint analysis* taints and tracks data throughout a program when introduced at marked code locations, known as *sources*, and raises an alarm when any tainted data enters another set of marked code locations, known as *sinks*. The selection of sources and sinks for use in taint analysis depend on its purpose. Section 5.3 contains a more detailed explanation for how *FPE* utilizes taint analysis. This formal definition focuses on *FPE* using the default execution strategy and it should not be conceived as all-encompassing, as there are additional execution strategies, as detailed in Section 3.3.3. The primitive unit of input to be processed during execution using *FPE* are single instructions from an Intermediate Representation (IR) of a Dalvik bytecode. An *intermediate representation* is a representation of the code where it has been translated to facilitate a task. In this instance, smali, the IR, is a more human readable version of Dalvik bytecode that is easier to process. Appendix A contains a sample smali file for reference. On Android devices that use the Dalvik VM (i.e., Android devices that have an API level of 19 or lower), an instruction is processed by an interpreter that maps the opcode and operand registers to a native code routine to execute. The *FPE* framework acts as an interpreter and controls the execution of the app using the Dalvik bytecode IR representation in the form of smali [130] files and app resources files (e.g., string tables, layout files, `AndroidManifest.xml`, assets, and arbitrary embedded files). A Dalvik instruction can correspond to a single statement in Java source code, although a single Java source code statement may translate to multiple Dalvik instructions depending on the complexity of its semantics.

An executable Android app is composed of one or more app components. Let *App* denote the set of app components contained within an Android app where $App \neq \emptyset$.[2] Each app

[2]There are Android apps that contain only resource files and lack executable code, but they are omitted from consideration.

component is an executable unit of the app with a predefined entry point and possibly an app component lifecycle. An app component can be viewed as a sequence of Dalvik instructions where some instructions can alter the control flow of the execution according to inputs, events, values from the environment, and non-deterministic functions. Let Comp represent a finite sequence of executable instructions comprising the programming logic of an app component. Let Insn represent the set of instructions of an app component with IC being the instruction counter denoting the current position within the sequence of instructions. A Dalvik instruction necessarily contains an opcode and zero or more operand registers. Let Opcode represent the set of all 226 Dalvik opcodes and Reg represent the set of active registers. A register can contain a primitive data type or an object reference with a possible value of `null`. Android apps are primarily written in Java, which does not allow direct memory access. This mostly abstracts memory addresses from the user due to automatic memory management. Branching instructions contain a label that represents a location within a smali file. Let Label represent the set of all labels in the app. Each instruction is uniquely identified by a smali file name and line number offset within the file. Let Opcode_c represent the set of 15 branching opcodes. Each branching instruction contains a label in the code to branch to using a conditional or unconditional jump. Instructions that invoke a method call contain the fully-qualified method name and zero or more operand registers to be passed as arguments to the method call. There are 10 different opcodes for invoking a method call, which are represented by Opcode_i. Method calls are of particular interest, especially those that invoke sensitive Android APIs, for observing the functionality of an app and performing taint analysis. Let Meth represent the set of all methods that an app can call based on its program logic. The *FPE* framework contains a list of 652 *targeted* fully-qualified methods calls from the Android API. The list of targeted methods is user-configurable to allow the user of the framework to focus on the most relevant behaviors. Let Meth_i represent the set of relevant method calls that will be recorded. Let Meth_i along with their calling object state (i.e., the receiver object for non-static method calls), parameters and return value, if any, captured by zero or more register values (r_i^*), a timestamp (time), and instruction counter *IC* form a 4-tuple representing a logging instance. *State* represents a set of 4-tuple logging instances(s) encountered during execution.

To allow the forcing of conditional statements only when an artifact of the execution environment is being considered in predicate evaluation, the *FPE* framework uses taint analysis at the register level. Let $Meth_s$ represent the set of methods that act as a taint source wherein the register for the return value will be tainted. For example, the `java.lang.System.currentTimeMillis()` API is a taint source as it obtains a current timestamp from the system. For a method to be a taint source, it *must have* a non-void return type. The taint source methods obtain data from the external environment and these externalities can be used in branching decisions. Let $Meth_r$ represent the set of methods with a non-void return type. When the *FPE* framework encounters instructions that invoke a method call, there are certain Android API method calls that will cause a single execution run to terminate and restart to explore a new path through the app component if there are still additional valid execution paths remaining. Let $Meth_e$ represent the set of method calls that will terminate a single execution run. This occurs when a single execution run encounters a known API call that will terminate either the process or the component being executed, such as `java.lang.System.exit(int)`, `java.lang.Runtime.halt(int)`, `android.os.Process.killProcess(int)`, `android.app.Activity.finish()`, etc. These sets and definitions are formally provided below using set notation:

$App := \{Comp_1, Comp_2, ..., Comp_n\}, n \in \mathbb{N}$	Application and constituent components
$Comp := [insn_i^+]$	Component and its sequence of instructions
$Insn := \{insn_1, insn_2, ..., insn_n\}, n \in \mathbb{N}$	All possible Dalvik instructions
$IC := \{IC \in \mathbb{N}\}$	Instruction Counter

$Opcode := \{o_1, o_2, ..., o_n\}, n \in \mathbb{N}$	Dalvik opcodes
$Opcode_c := \{o_{c_1}, o_{c_2}, ..., o_{c_n} \mid Opcode_c \subset Opcode\}, n \in \mathbb{N}$	Opcodes that alter control flow
$Label := \{l_1, l_2, ..., l_n\}, n \in \mathbb{N}$	Labels in code for control flow
$Opcode_i := \{o_{i_1}, o_{i_2}, ..., o_{i_n} \mid Opcode_i \subset Opcode\}, n \in \mathbb{N}$	Opcodes that invoke a method call
$Reg := \{r_1, r_2, ..., r_n\}, n \in \mathbb{N}$	All Registers
$TaintReg := \{t_1, t_2, ..., t_n \mid TaintReg \subseteq Reg\}, n \in \mathbb{N}$	Tainted registers
$Meth := \{m_1, m_2, ..., m_n\}, n \in \mathbb{N}$	All possible methods
$Meth_r := \{m_{r_1}, m_{r_2}, ..., m_{r_n} \mid Meth_r \subseteq Meth\}, n \in \mathbb{N}$	Methods with a non-void return type
$Meth_s := \{m_{s_1}, m_{s_2}, ..., m_{s_n} \mid Meth_s \subseteq Meth_r\}, n \in \mathbb{N}$	Taint source methods
$Meth_e := \{m_{e_1}, m_{e_2}, ..., m_{e_n} \mid Meth_e \subseteq Meth\}, n \in \mathbb{N}$	Methods that will restart an execution run
$Meth_i := \{m_{i_1}, m_{i_2}, ..., m_{i_n} \mid Meth_i \subset Meth\}, n \in \mathbb{N}$	Methods of interest to log
$State := \{s_1, s_2, ..., s_n\}, n \in \mathbb{N}$	Instances of logged methods
$s := <m_i, r_i^*, time, IC>, m_i \subseteq Meth, r_i^* \subseteq Reg, i \in \mathbb{N}$	4-tuple representing a logged method

These sets and definitions provide the basis for the algorithm that performs forced-path execution of an Android app using the default execution strategy (e.g., only forcing when artifacts from the execution environment influence a branching instruction). In addition, I have created functions that operate on sets to make the algorithm more concise and understandable. Some functions take an instruction and parse out some relevant piece of the instruction (e.g., the second argument register) so it can be operated on independently. The *getMeth(*$insn_i$*)* function takes an instruction and returns the fully qualified method from the set *Meth* if it is an instruction that invokes a method call and returns an empty set ($\emptyset$) if it is not an instruction that invokes a method call. The *getReg(*$insn_i$*)* function takes an instruction as an input and returns a set containing the operand registers used in the instruction. If the instruction does not use any operand registers (e.g., a static method call with no arguments), then an empty set will be returned. If the instruction has at least one register operand, it returns the set of register(s) used in the instruction. The *getOpcode(*$insn_i$*)* function takes an instruction as a parameter and returns the opcode contained within the instruction.

The *exec(*$insn_i$*)* function takes an instruction, executes it, and returns a return register for executed method calls that have a non-void return type. The underlying implementation for this function is complex as it acts an interpreter for all the Dalvik instructions. During interpretation, each Dalvik instruction has a switch case for the opcode where the values from operand registers, if any, are used as inputs to execute the instruction. Invoking method calls adds additional processing by using a call stack to store the calling context and passing any argument registers when invoking the method call. This operates in the conventional fashion by storing the state of the current method and adding a stack frame with the relevant data to execute the called method. These implementation details have been omitted from the algorithm to focus on the forcing of constraints related to the execution environment and the taint analysis. The *timestamp* function returns the current timestamp, which is used for logging purposes. The *timeup* function has been left undefined, although it returns *true* if the time spent analyzing an app component has surpassed the user-set threshold and returns *false* otherwise. A timer can be used to constrain the amount of time executing an app component and to bound the execution time in case an infinite loop is encountered. The *applyTaintRules* function is an integral part of the algorithm as it applies the taint propagation rules. The taint analysis is applied on the register level and is described in Section 5.3. The preceding functions are formally defined below using set notation:

$getMeth(insn_{\mathrm{i}}) = \begin{cases} m_{\mathrm{i}} & \text{if } getOpcode(insn_{\mathrm{i}}) \in Opcode_i \\ \emptyset & \text{if } getOpcode(insn_{\mathrm{i}}) \notin Opcode_i \end{cases}$	Obtains the method from an instruction
$getReg(insn_{\mathrm{i}}) = \{r_i^* \mid r_i^* \subseteq Reg\},\ \lvert r_i^*\rvert \geq 0$	Obtain register(s) from the instruction
$getOpcode(insn_{\mathrm{i}}) = \{o_i \mid o_i \subseteq Opcode\},\ \lvert o\rvert = 1$	Obtain the opcode from the instruction
$timestamp() = \{t\}, t \in \mathbb{Z}^+$	Returns the current timestamp
$exec(insn_{\mathrm{i}}) = \begin{cases} r_{\mathrm{i}} & \text{if } getOpcode(insn_{\mathrm{i}}) \in Opcode_i \\ & \wedge\ getMeth(insn_{\mathrm{i}}) \in Meth_r \\ \emptyset & \text{if } getOpcode(insn_{\mathrm{i}}) \notin Opcode_i \\ & \vee\ getMeth(insn_{\mathrm{i}}) \notin Meth_r \end{cases}$	Executes the instruction

Algorithm 1 displays the high-level behavior of the *FPE* framework using taint analysis on the registers to track inputs from the execution environment which informs the decision of whether or not to force the outcome of predicate evaluation or use the actual runtime values during predicate evaluation. Algorithm 1 uses the previously-defined set definitions and functions to display the logic for processing an Android app and its constituent app components in order to profile its behavior with regard to invoking Android API calls. Each app component is processed individually and any data transmitted from one component to another via Intent objects will be stored and delivered to the app component when it is

Algorithm 1 General Forced-Path Execution algorithm for an Android app.

```
Input   : Disassembled Android App (App)
Output : The set of targeted Android API calls and their values (State)
State ← ∅

foreach $Comp_i \in App$ do
    pathAvailable ← true
    IC = 0
    insn = $Comp_i$[IC]
    TaintReg ← ∅
    while $pathAvailable = true$ do
        if $getOpcode(insn) \in Opcode_c \wedge \exists getReg(insn) \in TaintReg$ then
            IC ← findopenpath(insn)
            if $IC = -1$ then
                pathAvailable ← false
            else
                insn ← $Comp_i$[IC]
                continue
            end
        else
            r ← exec(insn)
            if $\exists getReg(insn) \in TaintReg$ then
                applyTaintRules(insn, r)
            end
            if $getOpcode(insn) \in Opcode_i$ then
                if $getMeth(insn) \in Meth_s$ then
                    TaintReg ← TaintReg ∪ r
                end
                if $getMeth(insn) \in Meth_i$ then
                    State ← State ∪ <getMeth(insn), getReg(insn) ∪ r, timestamp(), IC>
                end
            end
        end
        if $timeup()$ then
            pathAvailable ← false
        else
            if $(getOpcode(insn) \in Opcode_i \wedge getMeth(insn) \in Meth_e)$ then
                IC = 0
            else
                IC ← IC + 1
            end
            insn ← $Comp_i$[IC]

        end
    end
end
return State
```

executed. Each app component $Comp_i$ is represented by a finite sequence of instructions. The IC variable is used an index into $Comp_i$ to obtain the current instruction. Each app component is initialized to execute the first instruction in $Comp_i$ and IC increases sequentially until a branching instruction alters sequential control flow. The *pathAvailable* variable is a Boolean variable acting as the loop control variable in the *while* loop. The *pathAvailable* variable initially has a Boolean value of *true*, but will become *false* if the user-set timer, as determined by the *timeup* function, for an app component expires. In addition, the *pathAvailable* variable will be set to false if enough time is given for all paths through an app component to be taken.

The *findopenpath* function is only called when the current instruction has an opcode that can alter the control flow of the execution and at least one of the operand register(s) for the instruction have been tainted. The *findopenpath* function will examine the binary tree that models the execution paths through the app component and the current node for the execution module and determine which path the execution should take depending on which paths are available. There are different execution modules that concurrently execute different paths and share the binary tree structure by using concurrency controls. The *findopenpath* function determines which branch to take and will return the index of the next instruction to execute within the current app component ($Comp_i$) and use it to update the IC value. Depending on the branching instruction, execution may branch to an instruction corresponding to a particular label or continue linear execution. Since there can be numerous execution modules operating concurrently, an execution module may take the last available path while the other execution modules are executing along the same path since there will only be one path left through the app component. This information will be encoded in the binary tree, which is accessed via the *findopenpath* function. If all the paths for an app component have been traversed, then the *findopenpath* function will return a value of *-1* indicating that the binary tree is complete and the subsequent app component should be executed. When this occurs, the *pathAvailable* variable gets set to *false* to break out of the *while* loop.

If the instruction has at least one operand register that is tainted and the opcode for the instruction is not a branching opcode (i.e., $getOpcode(insn_{\mathrm{n}}) \notin Opcode_{\mathrm{c}}$), then the taint rules will be applied via the *applyTaintRules* function after the instruction has been executed. This function will apply rules to propagate the *taint* and *untaint* registers depending on the specific instruction and the placement of the operand registers. If the current instruction has an opcode that will invoke a method call, then the fully qualified method will be examined to check if it is a taint source method, taint sink method, or a method that should be logged. If the method call is a taint source method, then the return value from the method will be tainted and added to the set of tainted registers *TaintReg*.

If the current instruction is invoking a method of interest, then the registers containing its context, location in code, and a timestamp will be added to the set *State*, which contains the 4-tuple logging instances. These 4-tuple instances contain the functionality of the app with regard to its behavior and interaction with the Android framework. For all instructions encountered, except the branching instructions that contain a tainted operand register, the algorithm checks to see if it should stop executing an app component. If the conditions are not met (i.e., time expiration or all paths completed), then the *IC* will be incremented by 1 and used as an index to obtain the next instruction for the app component from the sequence of instructions for the app component from $Comp_i$. If the execution has returned from its initial entry point or a halting API call was encountered, the module will be reinitialized and begin with the initial instruction of the app component and take a different path. Eventually one of the conditions that will stop the execution of the current app component will arise, and then the next app component will be executed. The app components in the app will be iterated over until they have all been processed and the algorithm returning *State* (a set of 4-tuples containing the targeted methods and their context) and the analysis of the app will conclude.

3.3.2 Comparison to Prior Work

Symbolic execution was initially developed to work on non-interactive programs where the program's entry point and its required inputs were well defined. Non-determinism, concurrency, asynchronous behaviors, and the ordering of the events create challenges for symbolic and concolic execution [118, 131]. When analyzing an app, a symbolic execution engine will likely have difficulty reasoning about control flow obfuscation techniques such as reflective method calls and dynamic code loading at runtime [119, 132, 133]. Android apps that intend to hide their behavior from symbolic execution and static analysis platforms, can use these obfuscation techniques. Obfuscation of constants using runtime transformations such as encryption or other reversible transformation (e.g., data compression) can hinder symbolic execution [133–135]. A major difference between *FPE* and symbolic execution is that *FPE* can obtains the runtime constructs and handles these obfuscation techniques that can hinder static analysis techniques and static symbolic execution. Illustrating some of these obfuscation techniques, Listing 3.1 provides a concrete Java source code example containing indirect flows (lines 2 & 15), dynamic code loading (line 7), basic string obfuscation (line 11), reflective method call (line 15), and a constructed runtime value (line 15). The *FPE* framework models the calls to dynamically load a `dex` file and converts the Dalvik bytecode to smali files to make them accessible to the framework.

```
 1  // the getString method obtains a String from the app's strings.xml file
 2  String name = getString(R.string.apk_name);
 3  // helper method to unpack embedded apk file in app's assets directory
 4  String apkPath = unpackAndCopyAsset(name);
 5  File odexDir = getDir("odex", Context.MODE_PRIVATE);
 6  // dynamically loads the classes of the embedded apk file
 7  DexClassLoader cl = new DexClassLoader(apkPath, odexDir.getAbsolutePath(), null,
        getClass().getClassLoader());
 8  // loads a class from the embedded apk
 9  Class sm = cl.loadClass("ob.fus.cat.e");
10  // avoids using string literal
11  String methName = new String(new char[]{'h','a','j','i','m','e','r','u'});
12  // obtains a method reference from the dynamically loaded code
13  Method methGetService = sm.getMethod(methName, new Class[]{String.class});
14  // reflectively makes a method call where the name is constructed at runtime
15  Object what = methGetService.invoke(null, new String[]{name.substring(0, name.length
        ()-(6^2)) + "_entry"});
```

Listing 3.1: Challenging Java source code for static analysis and static symbolic execution.

Concolic execution uses concrete execution of programs, but it also maintains a symbolic state of the program, in addition to the concrete state. Concolic execution uses instrumentation to maintain and update the symbolic state of the program. This can be performed by directly instrumenting the Android app bytecode via *cloning* the app and inserting the instrumentation code. An *app clone* is an unauthorized copy of an app where the app logic has been modified (see Section 6.4 for details). Cloning an app can be used for analysis purposes by injecting code into the original app [123, 136]. Cloning an app breaks the original signature on the APK files and can be detected at runtime by the app using an integrity check. An *integrity check* is a defensive technique by a program to check to see if the program code or resources have been modified from their original authorized state. For example, an app may hash the public-key certificate and compare it to an expected value. It may be difficult to anticipate *and* handle all of the approaches in which an app tries to verify that is has not been modified, including performing the integrity check in native code. *FPE* marks external data, including the signatures of the apps files, as subject to *forcing* for exploration, so this in-app integrity checks will not prevent *FPE* from analyzing an app. In addition, *FPE* does not modify the APK file to instrument the app code. Listing 3.2 shows a concrete Java source code example where an app examines the signature of its embedded files and throws an uncaught exception which will terminate the process if the app detects that it has been modified. When an app is modified, the app's file signatures will reflect this, and can be detected by the app.

```java
PackageInfo packageInfo = getPackageManager().getPackageInfo(getPackageName(),
    PackageManager.GET_SIGNATURES);
MessageDigest messageDigest = MessageDigest.getInstance("SHA-256");
for (Signature signature : packageInfo.signatures) {
    messageDigest.update(signature.toByteArray());
}
String encoded = Base64.encodeToString(messageDigest.digest(), Base64.NO_WRAP);
if (!"HuBASw6thnX2OGtXfiDjYASnwtmop9K7pzSyzxnHyFXA=".equals(encoded)) {
    throw new RuntimeException("App has undergone unauthorized modification");
}
```

Listing 3.2: In-app integrity check that detects unauthorized modification of app contents.

Both symbolic and concolic execution have difficulty analyzing the complex structure of Intent objects and their embedded data [123, 136]. An Android app can have multiple entry points due to an app containing one or more app components. These app components begin execution via an Intent message, which is accessible to the start app component. An Intent object has some known fields, but it can also contain a `Bundle` object that can contain arbitrary data structures including those that implement the `Parcelable` and `Serializable` interfaces. When executing from an app entry point, beyond the standard launcher activity app component, app components may rigidly expect Intent objects of arbitrary complexity and nested data. If the Intent is not present with the expected data, then the application may crash if it performs operations with the missing objects it expects to be present. Chapter 8 focuses on exposing these design flaws within apps and the Android OS and shows that these faults are common. For example, in Chapter 8 in a single study, the system discovered 10,862 fatal `NullPointerExceptions` due to operating on missing Intent data in a set of 18,583 Android apps. Listing 3.3 provides a notional example where concolic execution may have difficulty to solve the complex relationships that exist within the various embedded fields in the Intent, as prior research has noted [123, 136]. For the code in Listing 3.3, dynamic analysis platforms would also have to provide these Intent inputs correctly, otherwise the app will crash due to lack null checking in the code. Chapter 4 provides some additional source code examples that will present difficulty for dynamic analysis and concolic executions platforms that do not actively try to modify the environment by changing the output of Android API calls. Even if the app does not crash by using appropriate conditional logic, it may not make progress due to incorrectly structured Intent contents.

```java
protected void onCreate(Bundle savedInstanceState) {
    Intent intent = getIntent();
    Bundle firstBundle = intent.getExtras();
    Bundle secondBundle = firstBundle.getBundle("another_one");
    CharSequence[] charSequences = secondBundle.getCharSequenceArray(firstBundle.
        getString(firstBundle.getString("indirect")));
    Bundle thirdBundle = secondBundle.getBundle("one_more");
    int[] indexes = thirdBundle.getIntArray(thirdBundle.getString("indexes"));
    String str1 = (String) charSequences[indexes[secondBundle.getInt("a")]];
    String str2 = (String) charSequences[indexes[firstBundle.getInt("b")]];
    String str3 = (String) charSequences[indexes[thirdBundle.getInt("c")]];
    ...
```

Listing 3.3: Challenging Java source code for dynamic analysis and concolic execution.

Therefore, to test app components that are not started directly by any other app components in the app (such as an event listener for a received text message or a service only accessible to the system), then the concolic execution engine will have to attempt to reason about the internal data contents of the Intent objects these app components are expecting to receive. In certain cases, mentioned in Chapter 8, an app will encounter an uncaught fatal exception that terminates the app when there is missing data in the Intent due to a `NullPointerException`. This can be difficult to embed data beyond simply providing the default constructor for an object in an automated way, although the well-known Intent fields can be populated. *FPE* does not try to populate the Intent when executing *exported and accessible* app components. Exported and accessible components can be started from any app co-located on the device and have arbitrary contents in the Intent. If there is an Intent sent from another app component, then it will be delivered to the destination app component. If the app component to be analyzed by *FPE* does not have an Intent waiting for it to be consumed, then the data it is looking for from the Intent will be created at runtime when it can be inferred. Even if the data cannot be exactly inferred, *FPE* controls the interpreter and deals with missing data by reducing the precision of the analysis to get code coverage instead of immediately crashing. Since *FPE* uses its own interpreter, it can control when the app crashes and can continue when needed.

FPE simulates different environmental data that would be difficult to recreate or to be forced to occur at runtime for a concolic execution engine or dynamic analysis. Manually creating these events may be impractical to cover all cases. Concolic execution identifies the initial inputs to programs and can identify event input data that is present in path conditions. For example, the path condition may require that a notional system property named `sys.debug.modem` have a value of `1`. This system property is from the environment is not strictly part of the initial inputs to the program. For concolic execution, these events would need to occur at runtime to be considered by the constraint solver after a concrete execution of the program. The constraint solver will correctly be able to identify the appropriate input to satisfy the predicate, but it may not attempt to satisfy it automatically at runtime since this would require some form of interposition where the output of framework API calls are also controlled by the concolic execution engine. While Anand et al. [100] and CATE [137] do not control the outcome of framework API calls, ConDroid [123] does by means of overwriting calls with modeled values in a recreated instrumented Android app. Using this approach, the app will need to be recreated for each difference in the Intent being sent to a component and for each different output from an overwritten API call. This approach can encounter difficulty when there is a requirement for a more complex Intent message (e.g., Intent for a received text message) or for an in-app integrity check that can detect that the original app file has been tampered with. An in-app integrity check in native presents great difficulty, as modifying app binaries is more difficult that modifying managed code. In addition, symbolic execution and concolic execution, to some degree, will be limited by the effectiveness of the constraint solver to solve path conditions.

While constraint solvers are powerful and have undergone many improvements over the years, analyzing large apps can result in complex path conditions that may require significant resources or stress the solver's practical or theoretical limitations [115–117, 138–140]. Underlying some of the limitations of the constraint solver is that some types of constraints present decision problems that are undecidable [113, 139]. Xu et al. [138] tested some concolic execution engines and identified issues that they encountered even on small binaries. This study identified scalability issues, although it is not known if the results generalize beyond the tested binaries. A well-known problem with both symbolic and concolic execution is the path explosion problem [112, 139, 141]. The path explosion problem is due to programming constructs such as loops (e.g., loop predicate contains symbolic value) and general code complexity with regard to the number of conditional statements which can cause an exponential increase in the number of paths to explore. Issues of scalability arise as the size of the program increases. An adversary that is aware of these limitations can include code at numerous locations that intentionally create the conditions leading to path explosion, wherein these operations to cause path explosion have no real underlying effect on the core functionality of the program. This can seriously inhibit symbolic and concolic execution using diversionary tactics. *FPE* uses execution strategies, detailed in Section 3.3.3, that have varying levels of path exploration. *FPE* uses a configurable threshold to limit loop iterations and the allowed depth of recursive calls. While artificially limiting the execution in this way can result in failure to explore interesting paths resulting from this approximation, it also partially limits the impact of path explosion. If a conditional statement is selected to be explored, as determined by the specific execution strategy, then the branch to be taken at a conditional statement is selected at random if both the branches will lead to a complete execution path that has not been previously taken.

FPE does not provide an absolute guarantee that a path from an app entry point to another code point (e.g., sensitive API call) is indeed feasible in all cases. This can result in infeasible paths that cannot be realized at runtime. This can result from traversing a path where there are two conflicting constraints on the same path. This is a trade-off in terms of precision for practicality with regard to path exploration and removing the overhead in validating the satisfiability of each execution path with a constraint solver. The constraint solver itself can also become a point of attack by apps employing anti-analysis techniques. Banescu et al. [140] discovered some attacks against both symbolic and concolic execution. Most of these obfuscation transformations resulted in at least a 100% slowdown versus symbolically executing the same non-obfuscated code. As *FPE* does not utilize a constraint solver and limits loop iterations even to embedded loops, it should not be significantly affected. Path divergence is known issue in concolic execution [142, 143], including for Android [100]. *Path divergence* occurs when a different execution path is taken at runtime instead of the intended execution path, resulting from a divergence of the symbolic state from the concrete state of the program due to implicit control flow.

A current practical limitation of *FPE* is that is does not handle the execution of native code. While this is not uncommon in analysis platforms that focus on managed code and omit native code from the analysis [123, 132, 136], it still limits the precision of the analysis for an app that uses native code. Not handling and executing native code can result in incorrect states, potentially affecting subsequent states. Nonetheless, not handling native code, other than logging the parameters to native method calls, can prevent anti-analysis techniques from occurring in native code which an app may not be able to prevent. In addition, the *FPE* framework automatically enumerates and interacts with identified GUI elements and GUI widgets, which symbolic execution and concolic execution may or may not handle. Malware may attempt to withhold its behavior [22] unless the app detects that its is executing on a known-vulnerable device from a particular OEM [23] or it does not detect that it is executing in an emulator [144]. *FPE* does use modeling for certain functionality and this manual emulating certain behavior can be time consuming to create and maintain.

3.3.3 Execution Strategies

The *FPE* framework is modular and uses different execution strategies for different high-level goals. An *execution strategy* is a mode of operation where the amount of forcing through conditional statements and concrete execution of conditional statements is moderated depending on some criteria to achieve a specific goal. The higher the amount of concrete execution of conditional statements according to the actual runtime conditions should allow for higher confidence that the path taken through an app is indeed feasible. On the other hand, exploring all branches of a conditional statement, including all switch cases, when data from a particular origin (e.g., data from the execution environment such as user input, network responses, timestamps, etc.) allows the exploration of paths using a wide range of valid data according to its type for a fuller sample of behavior based on the logic of the app being analyzed. Some contrived and real-world use cases are provided in Chapter 4, displaying the value of exploring the different paths that can be taken at runtime based on values extracted from the execution environment. The list below provides some execution strategies:

- **Complete exploration:** A strategy providing a *what-if* analysis that *attempts* to explore every path through an app without any consideration for path feasibility. Exploring every path in a non-trivial program is generally infeasible.

- **Exploring the execution environment:** A strategy that primarily uses concrete evaluation of conditional statements *except* when a value is, directly or indirectly, derived from the execution environment. In this case, all branches of a conditional statement will be taken and explored. The values obtained from the input environment are tracked using taint analysis (see Section 5.3). Values from the environment are obtained using known APIs allowing for this data to be tracked through the program. Any control dependencies on this data are explored when environment data flows into conditional statements to account for the variability in different environments.

- **Capability leak identification:** A strategy that helps to discover concerning data flows where externally-controlled input (e.g., user, system, other processes, etc.) flows from the entry point of an exported and accessible app component to a concerning API call. This strategy focuses on the range of possible external input values and forces and explores conditional statements when these values, indirectly or directly, are evaluated in an expression for a conditional statement. The input data is generally in the form of Intent messages with arbitrary contents or exported methods of a service interface.

- **Additional techniques:** Additional execution strategies can be created based on a particular use case where the forcing of constraints is limited to certain classes of inputs and outputs from selected APIs. Moreover, additional program analysis techniques can be integrated for a more thorough analysis.

In the future, I plan to bridge the two different research areas of this dissertation, by modifying the *FPE* framework to automatically discover faults stemming from incomplete handling of user input that can result in abnormal process termination. Chapter 8 provides various methods in which common developer errors can cause app crashes, as well as system crashes. This specific testing logic could be added as a module to the *FPE* framework to check if both adequate checks are made on user input prior to their use (e.g., checking if an input value is *null* prior to using it) and determining if adequate exception handling is present.

3.3.4 Execution Environment

Values from the environment generally affect the behavior of non-trivial Android apps. Data from the *execution environment* is obtained from sources outside of the program and is introduced into the program code. Some of these data values from the environment will naturally vary due to inputs and runtime conditions external to the app. Some artifacts of the execution environment particular to an Android device are the following: system properties, presence of other apps, presence of a root user management utility, Android OS version, language setting, device vendor, device model, presence of a Subscriber Identity Module (SIM) card, physical location as determined by GPS, system clock, and user input (e.g., entered data, screen touches, and gestures). If an app accesses the Internet or local network, network responses can affect the behavior of an app. In certain instances, the network may not be available, which generally constrains a program's behavior (e.g., no dynamic content, online collaboration, and presence of advertisements). Non-determinism such as using the output of a pseudo-random number generator will also affect an execution path when used in a conditional statement. Chapter 4 provides various contrived and real-world use cases where the environment affects the branching decisions of Android apps. There are many additional ways in which the environment can affect the behavior of an app when there is an adversarial presence on the device or the network and various examples are provided below in which unexpected app behavior can occur.

Intercepting and Modifying Network Responses

Some of these externalities may be able to be leveraged by an attacker either locally from an app co-located on the device or remotely by using a wireless communication mechanism. If an Hypertext Transport Protocol (HTTP) network connection does not use Transport Layer Security (TLS), then an attacker on the local network or upstream may be able to perform a Man-In-The-Middle (MITM) attack on the connection to maliciously modify data to perform a specific action or inject a software fault. This allows the attacker to control the server response that will be consumed by the app and potentially affect its runtime behavior. For example, vulnerable versions of Adups software that came pre-installed on a range of Android vendor devices had an active Command and Control (C&C) channel that solely used HTTP, allowing external attackers to perform MITM attacks to inject shell commands to be executed as the `system` user on the device. Section 4.1.2 contains additional details about the behavior of the Adups software.

Insecure File Storage

External access to an app's files can be obtained in certain instances on both external and internal storage. Early Android devices supported an external Secure Digital (SD) card for additional storage. Some current devices do not provide an interface for a *physical* SD card, but do provide an *emulated* SD card that is located on internal storage for backwards compatibility. Both the emulated and non-emulated SD card storage area is also referred to as *external storage*. *Internal storage* is an integrated solid-state storage device, generally an embedded Multi-Media Controller (eMMC) device. It is recommended that apps do not store sensitive files on external storage [145]. If an app stores a file on external storage, then the contents of files can be modified by any app on the device that has been granted the `WRITE_EXTERNAL_STORAGE` permission. If an app reads an externally modified file from external storage, it is possible that an external app can modify the file to inject a fault that causes a fatal crash or affects the behavior of the app. There have been cases where pre-installed apps have installed apps residing on external storage and this process has been subverted by another party to install a different app than the one that was intended by the party that initiated the app install process [89, 146].

During the installation of an Android app, the Android platform assigns the app a static Linux UID. This UID is used to *sandbox* the app and enforce access control by creating a security boundary for the app's files and memory. Each file created by the app on internal storage has an owner and a group that is set to the UID of the app. Each app has a *private* directory on internal storage to store their files with a path of `/data/data/<package name>`. By default, the files created by the app on internal storage are not readable, executable, or writable by other users (e.g., apps) on the device. Nonetheless, the developer can still create world-readable and world-writable files on internal storage by using specific flags to the `android.content.Context.openFileOutput(String, int)` API method call when creating a file. The second parameter is an integer, which controls the mode. Two integer constants, `MODE_WORLD_READABLE` and `MODE_WORLD_WRITEABLE`, have been deprecated, but are still usable as the mode for compatibility and can expose an app's files to external apps. In addition, the app can obtain references to its own files and directly call the `java.io.File.setWritable (boolean, boolean)` and `java.io.File.setReabadble(boolean, boolean)` API methods to modify the permissions of a file or directory. This can result in data disclosure if a file is world-readable, which would allow any other process aware of its existence to read its contents. In addition, if a file is world-writable, this can lead to data modification, since other processes can modify the contents of the file in certain circumstances. If the device is *rooted*, then an app cannot depend on the integrity of the files that it stores locally on the device. An adversary with root access would be able to modify any files in the app's private directory and even the app binary itself.

Providing Data to Exported Components

In an adversarial model, an attacker that has an app co-located on the device may try to affect the app's runtime behavior or corrupt the app's data by sending Intent objects embedded with crafted data. The Android OS can export app components within an app *by default*, even when the app does not explicitly declare that the app components should be exported. An *exported app component* is an app component that can be receive Intent messages or data from an external app (i.e., any app other than itself). This is a conscious design decision Google made that favors openness and data sharing, sometimes at the expense of security [21,147]. The Android OS will automatically export an app component if it has *at least* one `intent-filter` and the app developer has not explicitly set the `android:exported` attribute to *false* for the component in the app's manifest file. For example, the broadcast receiver shown in Listing 3.4 will be automatically exported by Android even though its declaration does not explicitly indicate that it should be exported. An `intent-filter` is a specification of the set of action values that an app component can handle. An *action* is a string that denotes a specific event that an app component can handle. For example, an Android app may want to be alerted when the user is actively using the device, so they would register to receive the action with a name of `android.intent.action.SCREEN_ON` to know when the user is actively using the device. The Android OS sends a broadcast intent with an action string of `android.intent.action.SCREEN_ON` when the device screen is turned on to all apps that have registered to receive this particular action. Listing 3.4 displays a broadcast receiver that statically declares the actions that it can receive in the app's `AndroidManifest.xml` file. In addition to platform-declared action strings, an app can create, use, and register for its own action strings. An `intent-filter` must be associated with an activity, service, or broadcast receiver app component. Intentionally or unintentionally exported components can lead to privilege escalation [21, 147, 148]. Apps can restrict access to exported app components by using signature-level custom permissions.

```
<receiver android:enabled="true" android:name="Adventurer">
 <intent-filter android:priority="1000">
  <action android:name="android.intent.action.USER_PRESENT"/>
  <action android:name="android.intent.action.SCREEN_ON"/>
  <action android:name="android.intent.action.PHONE_STATE"/>
 </intent-filter>
</receiver>
```

Listing 3.4: Declaring actions in a broadcast receiver.

A content provider app component provides access to structured data. Generally, the content provider utilizes an SQLite database. If the content provider is exported, then an external app can read, write, and modify the underlying data contained in the SQLite database [149]. This can result in data disclosure and data modification if the app developer did not intend for the content provider to be accessible to other apps. This may allow an external app to possibly affect the control flow of an app by modifying values in an accessible content provider. Up to Android version 4.1, content providers app components were exported *by default* if the developer did not explicitly set whether it should be exported or not [150]. This allows an external app to read, write, and modify underlying data using the content provider [151, 152], which may alter its branching decisions or result in fatal errors if the data has an unexpected type or format.

Interacting with IPC Channels

Android apps can create UNIX domain sockets to perform IPC. If the domain socket does not properly check credentials and perform authentication when accepting connections, then this may allow an attacker to inject data into the program and alter its state. Previously, the Android OS was vulnerable to a DoS attack by having a domain socket used by the `zygote` process open to any app on the device in versions of Android up to 4.0.3 [153]. In addition, a pre-installed Adups app allowed command injection via a domain socket that could be used by third-party apps to execute commands with `system` user level privileges [154].

Apps can collude with other apps using IPC or covert channels [155, 156]. In certain instances, apps can aggregate their permissions over two or more apps and share the same UID. Android enables apps to share their resources and permissions with another app if they are signed with the same private asymmetric key and use the same value in the `android:sharedUserId` attribute in each of their `AndroidManifest.xml` files [157]. The appropriate behavior may not be performed unless the accompanying app is installed on the device.

An ordered broadcast can be modified by an app that receives it before being subsequently delivered to other apps. This was a design choice by the developers of the Android OS that can be abused. An *ordered broadcast* is delivered to apps eligible to receive it in an order that depends on the `android:priority` attribute integer value that they have set for the specific `intent-filter` element of a broadcast receiver in their `AndroidManifest.xml` file. The `android:priority` can range from -999 to 999 using a linear scale where -999 represents the lowest priority and 999 represents the highest priority. Apps with a higher priority get access to the broadcast Intent prior to apps with a lower priority, so they can modify the data contained in the Intent before it is passed on to broadcast receivers with a lower priority or simply abort the broadcast so that it will not be received by broadcast receivers with a lower priority [158].

Android apps can dynamically load Dalvik Executable (`dex`) files, containing Dalvik bytecode, to extend their inherent functionality. Dynamic code loading is necessary for apps that have exceeded the 65,536 method or field limit in a single `dex` file. This generally occurs with complex apps such as Facebook and Google Play Services. Android introduced MultiDex [159] to facilitate dynamic class loading in large and complex apps. In addition, apps can perform dynamic class loading by using the facilities provided by the `dalvik.system .DexClassLoader` class. If an attacker can interpose on this file either due to an MITM attack or the `dex` file being stored on insecure storage (e.g., external storage), then the attacker can achieve code execution in the context of the process with its complete set of permissions and corresponding capabilities. Samsung [160] made this mistake by retrieving an Android app over HTTP and then using the app to update a pre-installed app. Since the network connection used HTTP, it was vulnerable to MITM attacks, resulting in code execution in a privileged process. Section 6.2 contains details for 3 apps that *FPE* detected leaking the capability to inject arbitrary commands into pre-installed apps to be executed as the `system` user.

Chapter 4: Motivating Use Cases

This chapter provides motivating use cases from both real-world Android software and theoretical examples. Section 4.1 illustrates the utility of the *FPE* framework to simulate different environments to observe behaviors from apps that may not be readily apparent, depending on the particular analysis technique and runtime environment. Section 4.2 details some of the challenges that can manifest when analyzing Android apps that employ anti-analysis techniques and how the *FPE* framework overcomes them. Section 4.3 provides concrete examples of DoS attacks that illustrate the potential risks of executing a third-party app, that can have devastating effects on the system. These behaviors can be particularly dangerous because they may be outside the user's mental model of what an unprivileged app can accomplish such as permanently disabling an embedded Android device in the worst case.

4.1 Exposing Concealed Functionality

This section provides examples where concerning app behavior is not immediately exhibited by an Android app, although the behavior is predicated on certain conditions that may not be easily or immediately satisfied in a single or limited number of analysis environments. The specificity of the conditions may be by design to make the behavior more difficult to detect.

4.1.1 Fine-grained Targeting

Using a discriminating set of conditions can allow an app to *target* a constrained set of users that share common attributes. Listing 4.1 provides a theoretical Java source code example where the body of the `if` statement, containing a single method call on line 4, only executes when three specific conditions are simultaneously true: (1) the user has their locale set to the country of Kyrgyzstan; (2) their locale language is set to Kyrgyz (as opposed to other languages spoken in Kyrgyzstan); and (3) the user has a SIM card for the Kyrgyz carrier named MegaCom.[1] According to the *Android Developers* webpage for the `Locale` class [161], a *locale* is used to represent “a specific geographical, political, or cultural region.” The constant value of `43705` on line 3 in Listing 4.1 is composed of the Mobile Country Code (MCC), `437` for Kyrgyzstan, followed by the Mobile Network Code (MNC) with a value of `05` denoting the carrier MegaCom [162].

```
1 TelephonyManager telephonyManager = (TelephonyManager)
     getSystemService(TELEPHONY_SERVICE);
2 Locale locale = getResources().getConfiguration().locale;
3 if (locale.getLanguage().equals("ky") && locale.getCountry().equals(
     "KG") && telephonyManager.getSimOperator().equals("43705")) {
4         showtime();
5 }
```

Listing 4.1: Targeting users living in Kyrgyzstan that primarily use the Kyrgyz language and also use MegaCom as their carrier.

Targeting users according to some criteria can have multiple applications, including malicious purposes where the intended target is a subgroup of a population. There are many attributes that software can use to differentiate users such as Android’s GeoFencing API.[2] The GeoFencing API allows developers to easily constrain functionality to a certain geographical area by creating a *geofence* using a set of GPS coordinates and a selected radius in meters. While the example in Listing 4.1 is a contrived example meant to illustrate intentional targeting, this behavior is not limited to the theoretical domain. Section 4.1.2 discusses how certain versions of pre-installed Android apps developed by a company named Adups could remotely regulate which text messages to exfiltrate via a network response. Specifically, Adups could remotely instruct devices to only select and exfiltrate text messages that contained a specified search term or involved a specific phone number as a sender or receiver. In addition, another concrete real-world example is provided in Section 4.1.3 where

[1]MegaCom’s website is located here: `https://www.megacom.kg/`.

[2]Android GeoFencing documentation is provided here: `https://developer.android.com/training/location/geofencing`.

a pre-installed app on a Panasonic Android device would only send a text message containing user PII if the device has a SIM with an MCC corresponding to the country of India.

4.1.2 Adups: Guarding Malicious Behavior

Adups is a Chinese company that provides Firmware Over the Air (FOTA) services to Android device vendors [163]. In this context, the *FOTA* process involves the remote administration, transmission, and execution of updating an Android device's firmware to update the Android OS. Android vendors that do not have the sophistication or desire to directly manage the FOTA process of their devices can outsource this process to a company such as Adups. To manage the FOTA process, the managing entity requires a presence on the device in terms of software, generally manifesting as at least one pre-installed app. The pre-installed software must be privileged in order to initiate a firmware update. The privileged position of pre-installed apps on the device, which is not exclusive to FOTA app(s), can have significant consequences if the pre-installed app is malicious or contains vulnerabilities that allow local or remote privilege escalation. Various examples of pre-installed apps that are insecure in that they allow local apps co-located on the device to use their capabilities without the corresponding access permissions [21, 22, 147, 160].

In 2016, I discovered that certain versions of pre-installed apps from Adups were secretly obtaining the user's text messages, call log, unique device identifiers, and browser history without the user's consent or awareness and sending the PII to a server located in China [22, 164]. Google listed this discovery as one of the four vulnerabilities in the *Noteworthy Vulnerabilities* section of their official *Android Security 2016 Year in Review* document [165]. On the affected devices, the PII exfiltration was accomplished by a set of two interacting apps with package names of `com.adups.fota` (`versionCode = 22`, `versionName = 5.1.0.0.0`) and `com.adups.fota.sysoper` (`versionCode = 505`, `versionName = 5.0.5`).[3] The `com.adups.fota.sysoper` app was a pre-installed platform app that executed as the `system` user.[4] The pre-installed `com.adups.fota` app scheduled and performed the network communication and leveraged the open app components of the `com.adups.fota.sysoper` app to perform actions it was not privileged enough to carry out itself (e.g., obtaining text messages, command execution as `system` user, obtaining browser history, etc.). The `com.adups.fota.sysoper` app exported an app component which allowed local apps to also inject arbitrary commands to be executed as the `system` user. The Adups apps that contained spyware did not immediately display their data collection and exfiltration behavior, which can confound some analysis platforms since they likely analyze apps for a brief and pre-determined time period. *Spyware* is software that secretly obtains data about the user without their knowledge.

The two distinct capabilities, data exfiltration and an active C&C channel, have different time-based requirements that ensure their functionality is not immediately performed and thus visible to dynamic analysis platforms. A *C&C channel* uses communication infrastructure to allow a remote entity to exert control over an entity by instructing it to perform certain actions. The data collection and exfiltration functionality in the `com.adups.fota` app has an entry point that is the `com.msg.analytics.AnalyticsReceiver` broadcast receiver app component which statically registers for the `CONNECTIVITY_CHANGE` and `ACTION_POWER_CONNECTED` broadcast actions. The former is sent by the system when the device joins or leaves a known network, while the latter is sent by the system when the user plugs in the device to charge. The initial condition that needs to be fulfilled is that the device needs to be powered-on for at least 10 minutes the first time the app is run. Listing 4.2 contains Java source code for the `isOverActive(Context)` static method from the `com.msg.analytics.AnalyticsService` class which regulates whether the `AnalyticsService` service app component should be started or not.[5] The `AnalyticsService` app component is

[3]Adups used alternate versions of these apps and also has different package names for their apps as well: `com.data.acquisition`, `com.fw.upgrade`, and `com.fw.upgrade.sysoper`.

[4]To be a platform app, an app must be signed with the platform key and also set the `android:sharedUserId` attribute to `android.uid.system` in its manifest file.

[5]This Java source code example and the others in this dissertation are the product of examining apps' disassembled Dalvik bytecode and manually creating the source code with the same logic.

```java
private static boolean isOverActive(Context con) {
    try {
        // SAVE_SETTING_NAME has a value of 'analytics'
        SharedPreferences sp = con.getSharedPreferences(Const.
            SAVE_SETTING_NAME, 0);
        boolean ft = sp.getBoolean("ft", 0);
        if (ft == true)
            return false;
        long time = android.os.SystemClock.elapsedRealtime();
        // ANALYTICS_FIRST_TIME has a value of 600000
        if (time < Const.ANALYTICS_FIRST_TIME) {
            return false;
        } else {
            SharedPreferences$Editor editor = sp.edit();
            editor.putBoolean("ft", true);
            editor.commit();
            return true;
        }
    } catch (Exception e) {
        return true;
    }
}
```

Listing 4.2: Adups time-based trigger for app's first execution.

the entry point that performs the exfiltration of PII and determines when it should occur. Therefore, the device will need to have an uptime of at least 10 minutes the first time it is run prior to even starting the app component that performs the exfiltration, unless it is coaxed to do so with some manipulation via a program analysis technique such as *FPE*.

Once the uptime condition has been met, the `AnalyticsService` class calls the `com.msg.analytics.AnalyticsReport.report()` method which manages the collection and exfiltration of PII. There are actually two different time checks that occur serially, one for PII collection (i.e., `isOverDCTime()`) and the second for uploading the PII to a server in China (i.e., `isOverDCUploadTime()`). Listing 4.3 shows a method named `isOverDCUploadTime()` from the `AnalyticsReport` class where the Boolean return value determines if the uploading of user PII, after it has been collected, should proceed. Both of the methods, one for PII collection and the one for PII exfiltration, use the constant named `com.msg.analytics.Const.ANALYTICS_SCHEDULE_DEF` that has a long value of 259,200,000. In the context of these methods, the value of 259,200,000 corresponds to the number of milliseconds in a 3 day period, wherein the method checks to see if at least 3 days (72 hours) have passed since the previous exfiltration or since the `AnalyticsService` component first executed. When the `isOverDCUploadTime()` method, displayed in Listing 4.3, returns a Boolean value of true, the PII that has been collected will be uploaded to the following URL: `https://bigdata.adups.com/fota5/mobileupload.action`.

```
private boolean isOverDCUploadTime() {
    long currentTimeMillis = System.currentTimeMillis();
    // PREF_DATA_DC_UPLOAD_TIME has a value of 'dupt'
    long j = this.prefs.getLong(Const.PREF_DATA_DC_UPLOAD_TIME, -1);
    if (j >= 0) {
        // ANALYTICS_SCHEDULE_DEF has a value of 259200000
        return currentTimeMillis - j >= Const.ANALYTICS_SCHEDULE_DEF
            ;
    } else {
        saveLastTime(Const.PREF_DATA_DC_UPLOAD_TIME,
            currentTimeMillis);
        return false;
    }
}
```

Listing 4.3: Adups time-based trigger that controls PII exfiltration.

The Adups app with a package name of `com.adups.fota.sysoper` contained a C&C channel that becomes active after the device has been used for *at least* 20 days. An analysis time duration of 20 days seems unlikely to be met in most Android analysis systems. The `TaskReceiver` broadcast receiver app component, in the `com.adups.fota.sysoper` app, is started by the system whenever the user unplugs their device from charging (i.e., `ACTION_POWER_DISCONNECTED` broadcast intent was sent) and this component would start the `TaskService` service app component. The `TaskService` used the `com.adups.fota.sysoper.k` class to check to see if the app has executed on at least 20 different, not necessarily consecutive, days. I manually recreated the Java source code for the `com.adups.fota.sysoper.k.c(Context)` method, shown in Listing 4.4, that checks to see if the conditions have been met to make a connection to the C&C server to poll for commands. The C&C channel uses the following URL to check to see if any commands are available to be retrieved and executed: `http://rebootv5.adsunflower.com/ps/fetch.do`. If any commands are pulled from the server, they are faithfully executed by the `com.adups.fota.sysoper` platform app which executes them with `system` user privileges.

```
public static boolean c(Context con) {
    String package_name = con.getPackageName();
    SharedPreferences sp = con.getSharedPreferences(package_name, 0)
        ;
    long time = java.lang.System.currentTimeMillis();
    long lt = sp.getLong("LastTime", 0);
    int count = sp.getInt("count", 0);
    StringBuilder sb = new StringBuilder();
    sb.append("isOverSumTime count = ");
    sb.append(count);
    com.adups.fota.sysoper.n.b("task", sb.toString());
    if (count >= 20) {
        return true;
    }
    else {
        SimpleDateFormat sdf = SimpleDateFormat("yyyy-MM-dd");
        Date date = new Date(time);
        String ds = sdf.format(date);
        Date date2 = new Date(lt);
        String ds2 = sdf.format(date2);
        sb = new StringBuilder();
        sb.append("isOverSumTime nowDate = ");
        sb.append(ds);
        sb.append(" preDate = ");
        sb.append(ds2);
        com.adups.fota.sysoper.n.b("task", sb.toString());
        if (ds.equals(ds2)) {
            SharedPreferences$Editor editor = sp.edit();
            editor.putLong("LastTime", time);
            int newCount = count + 1;
            editor.putInt("count", newCount);
            editor.commit();
        }
        return false;
    }
}
```

Listing 4.4: Adups time-based trigger the C&C channel.

4.1.3 SalesTracker: Time and SIM-based Conditions

Panasonic sells Android smartphones and a key market of theirs appears to be India [166].[6] I processed a firmware image using *FPE* for the Panasonic Eluga X1 device running Android 8.1.[7] This particular build for the Eluga X1 device contains a pre-installed app with a package name of `com.staqu.panasalestracker` (`versionCode = 8`, `versionName = 1.3.5`).

[6]The server that hosts Panasonic's webpage featuring their mobile devices redirects requests for the `https://mobile.panasonic.com` URL to `https://mobile.panasonic.com/in`.

[7]`Panasonic/ELUGA_X1/ELUGA_X1:8.1.0/O11019/1536216693:user/release-keys` is the build fingerprint for the Panasonic Eluga X1 device.

This app is not a platform app; therefore, it generally will not be automatically granted the permissions declared by the Android platform with a `android:protectionLevel` of `dangerous` by default [167]. According to the manifest of the `com.staqu.panasalestracker` app, it has its `android:targetSdkVersion` attribute set to a value of `23`. Therefore, the app will not be granted all its requested permissions by virtue of it not being developed prior to Android 6.

I have not examined a Panasonic Eluga X1 device to test the app's behavior at runtime, but the `com.staqu.panasalestracker` app has been granted three permissions by white-listing them in the device's `/system/etc/permissions/privapp-permissions-platform.xml` file by default, as shown in Listing 4.5. White-listing privileged permissions to apps without the user explicitly granting them was introduced in Android 8.0 (Oreo) and is explained in [168]. The permissions provided in Listing 4.5 have a protection level of `signature` or `privileged` and will be granted to the app [169]. I am unsure if the app will be granted the permissions of `GET_ACCOUNTS` and `ACCESS_COARSE_LOCATION` permissions to obtain the user's email address and cell tower ID that the device is using. Independent of whether these two permissions are granted to the app or not, the code where the PII is obtained corresponding to these two permissions is contained with `try-catch` blocks, so the app will not crash if it encounters a `SecurityException` due to insufficient permissions.

```xml
<privapp-permissions package="com.staqu.panasalestracker">
    <permission name="android.permission.READ_PRIVILEGED_PHONE_STATE" />
    <permission name="android.permission.SEND_SMS_NO_CONFIRMATION" />
    <permission name="android.permission.WRITE_SECURE_SETTINGS" />
</privapp-permissions>
```

Listing 4.5: Modified `platform.xml` file to grant permissions by default.

I discovered a special exception for the `com.staqu.panasalestracker` app where it bypasses the standard permission check when sending a text message as shown in Listing 4.6.[8] The Java source code in Listing 4.6 was recreated by manually examining the disassembled `/system/framework/arm/boot-telephony-common.vdex` file from the device firmware. The `com.staqu.panasalestracker` app contains *triggers* that are dependent on time and the MCC

[8]The exemption for the app appears to be superfluous, and potentially a vestige from previous versions, since the app has been white-listed for the `SEND_SMS_NO_CONFIRMATION` permission as displayed in Listing 4.6.

```java
public void sendText(String callingPackage, String destAddr, String scAddr,
    String text, PendingIntent sendIntent, PendingIntent deliveryIntent,
    boolean persistMessageForNonDefaultSmsApp) {
     if (callingPackage == null || (callingPackage != null && !callingPackage
        .equals("com.staqu.salestracker"))) {
         com.android.internal.telephony.Phone phone = IccSmsInterfaceManager.
            mPhone;
         Context context = phone.getContext();
         context.enforceCallingPermission(
         "android.permission.SEND_SMS", "Sending SMS message");
     }
     IccSmsInterfaceManager.sendTextInternal(callingPackage, destAddr, scAddr
        , text, sendIntent, deliveryIntent,
        persistMessageForNonDefaultSmsApp);
}
```

Listing 4.6: Source code for the `sendText` method of the `com.android.internal.telephony.IccSmsInterfaceManager` class.

present on any SIM cards inserted in the device.[9] The app initially obtains user PII, and then attempts to send it over a network connection via Hypertext Transfer Protocol Secure (HTTPS) to the following URL: `https://stapp.panasonicarbo.com`.[10] If the app fails to send the user's PII 13 times over HTTPS, it will attempt to send a text message with a lesser amount of the user's PII *only if* the user has a SIM card with an MCC corresponding to the country of India. The app only sends out the PII once and the app logic internally refers to it as an *activation*. The value controlling whether the device is activated is the `device_activated` key in the global settings that are accessed via the `android.provider.Settings$Global` class. It is possible that another app with the permission to modify the global settings can make it so that the activation message is sent each time the device is turned on by modifying the value of the `device_activated` key, but I have not found any evidence to indicate that it is occurring.

The `com.staqu.panasalestracker` app contains a broadcast receiver app component named `com.staqu.salestracker.SalesTrackerReceiver`. Listing 4.7 shows the declaration of

[9]The device supports dual SIM cards.

[10]A slightly different version of the app with a package name of `com.staqu.salestracker` is present in the Panasonic Eluga Ray 700 device and sends the data over HTTP to the following URL: `http://p.salestrackr.info` (query string omitted).

```
<receiver android:name="com.staqu.salestracker.SalesTrackerReceiver">
  <intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED"/>
  </intent-filter>
  <intent-filter>
    <action android:name="ACTION_SMS_SENT"/>
    <action android:name="ACTION_SMS_SENT_SIM2"/>
    <action android:name="ACTION_SMS_DELIVERED"/>
    <action android:name="ACTION_INTERNET_THRESHOLD_CROSSED"/>
  </intent-filter>
  <intent-filter android:priority="1000">
    <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
  </intent-filter>
</receiver>
```

Listing 4.7: Declaration of the `SalesTrackerReceiver` broadcast receiver app component.

the `SalesTrackerReceiver` app component in the app's manifest file. When the `SalesTrackerReceiver` broadcast receiver app component first receives the `BOOT_COMPLETED` broadcast intent and if the device has not been activated, it creates a `PendingIntent` object to send an `Intent` to itself with an action string of `action_activation_received` with a repeating time interval that is dependent on how many times it has tried to activate the device previously. The *FPE* framework will execute each app component in the app, independent of whether it is externally accessible. Therefore, any app component that statically registers for broadcast actions that are only sent by the system will be executed by the *FPE* framework, simulating the range of possible system events.

If the `com.staqu.salestracker` app has tried to register the device using HTTPS less than 13 times, then it will set an alarm that will expire in 6 hours. When the alarm activates, the app will then try to send the user's PII over HTTPS. If the app has tried to register the device at least 13 times using HTTPS, then it will set the alarm for 30 minutes and also try to activate the device using Short Message Service (SMS) *if* the user has at least one inserted SIM card that has an MCC corresponding to India. *SMS* is responsible for the sending and receiving of text messages on many mobile devices. The app checks if the device has a SIM that is registered to any of the following country codes: 404, 405, and 406. The MCC codes of 404 and 405 both are assigned to India, while 406 is unassigned [170].

```java
public static boolean isOperatorSupported(String mcc) {
    if (mcc.equals("404") || mcc.equals("405") || mcc.equals("406"))
    {
        return true;
    }
    return false;
}
```

Listing 4.8: Filtering the MCC for Indian carriers.

The recreated Java source code for the `com.staqu.salestracker.Util.isOperatorSupported(String)` method that determines if the device is applicable for device activation using SMS instead of HTTPS is provided in Listing 4.8. This preceding method takes a string parameter that contains the attempt to query both SIM card slots for the MCC of either SIM. The MCC is obtained by using the `TelephonyManager.getNetworkOperatorForPhone(int)` method call for both SIM cards. If the device has a SIM card that is registered to an Indian carrier, then it will register the device over SMS if registration over HTTPS fails at least 13 times. This occurs in the `com.staqu.salestracker.SalesTrackerService.activateViaServer()` method. The PII that can be sent out by the `com.staqu.panasalestracker` app, using both HTTPS and SMS, is shown in Table 4.1 where the gray rows indicate certain PII that the app attempts to obtain, but may be denied due to insufficient access permissions.

Table 4.1: PII transmitted by the `com.staqu.panasalestracker` app using HTTPS and SMS.

PII	**HTTPS**	**SMS**
IMEI 1	✓	✓
IMEI 2	✓	✓
Email Address	✓	
Cell Tower ID	✓	
Phone Number		✓
Android ID	✓	✓
Local Area Code	✓	✓
MCC 1	✓	✓
MCC 2	✓	✓
MNC 1	✓	✓
MNC 2	✓	✓
Country Code	✓	✓

```
{
  "call": "java.net.URL.openConnection()",
  "component": "com.staqu.salestracker.
  SalesTrackerInternetConnectivityReceiver",
  "arguments": ["URL:https://stapp.panasonicarbo.com?brand=&model=
  STARHSTARTSTARCSTAR+STARDSTAReSTARsSTARiSTARrSTAReSTAR&
  imei1=1234567890123456&imei2=1234567890123456&android_id=
  9b1587bf3b1d8fc&mcc1=310&mnc1=660&mcc2=310&mnc2=660&date=
  12-03-2019&time=12%3A35%3A27&key=&typ=&ver=&ram=1MB&
  country_code=in&app_version=&manufacturer=HTC&lac=454&
  cid=454&eml=feelsgoodman%40gmail.com&android_version=0
  &build=""],
  "lineNumber": "716",
  "taint": "|136|",
  "category": "network_events",
  "smaliFile": "SalesTracker/smali/com/staqu/salestracker/
  SalesTrackerAsyncTask.smali"
}
```

Listing 4.9: Raw JSON event for sending PII in the querystring using HTTPS (PII highlighted in red text).

Listing 4.9 provides the raw output from the *FPE* framework showing a network request to the `https://stapp.panasonicarbo.com` URL where the PII contained in the querystring is highlighted in red text. A *querystring* is part of a URL that provides information by using key-value pairs. The raw output includes the fully-qualified API method call, its arguments, component, file name, category, taint values, and line number on which the API method occurs. In addition to making the network call in Listing 4.9, the entire URL, query string included, is written to the logcat log and the raw JavaScript Object Notation (JSON) output from the framework is provided in Listing B.1 of Appendix B. *JSON* is a format for data storage and transmission that supports arrays and can represent objects with key-value pairs. While leaking PII to the logcat log may seem like a minor issue due to third-party apps not being able to access the system-wide logcat log, I discovered that various vendors contained certain pre-installed apps that could be induced to leak the contents of the system-wide logcat log (see Section 6.1.2) [21]. Listing B.2 of Appendix B provides the raw JSON output from the framework when the app sends user PII over text message. The *FPE* framework will enter the conditional branch that simulates the device having a SIM card with an MCC that corresponds to India and also defeat the time-based requirement to obtain the destination number of the text message and its message body containing PII since they both come from the environment.

4.2 Detecting an Analysis Environment

Using the Android emulator provides an economical method for analyzing apps at scale by increasing automation and reducing hardware costs. The Android SDK provides a free emulator that can emulate different devices, platforms (e.g., Android TV, Wear OS, etc.), and API levels. Using the Android emulator is a common approach to perform dynamic analysis of Android apps [35, 37–39]. The Android emulator is amenable to modifications such as Virtual Machine Introspection (VMI) [37, 38] and modifying ART [171, 172]. Despite the utility of the Android emulator, it also has idiosyncrasies that are detectable by apps. Petsas et al. [45] provided three methods in which an Android app can detect that its executing within an emulator and subsequently withhold behavior. The methods they listed were checking for (1) static system properties that indicate an emulator, (2) the inaccurate modeling of the sensor data by the emulator, and (3) peculiarities of the VM itself with binary translation and cache coherence. They found that all the available online dynamic analysis platforms were vulnerable to some combination of these VM-detection heuristics. Malicious apps can detect the emulator and then use evasion techniques to thwart external attempts to be analyzed. Vidas et al. [46] presented some approaches that can be used by an Android app to detect that it is executing within an emulator. They focused on the emulation of the Android API, the network, performance differences, and differences in software and hardware components. Maier et al. [173] developed a tool named Sand-Finger that was able to develop a *fingerprint* composed of almost 100 different attributes for 10 different online analysis environments. By *fingerprint*, I mean a set of attributes that can be used to uniquely identify the analysis environments due to aggregating their distinguishing features. They examined artifacts of the environment such as uptime, Wi-Fi connectivity, mobile connectivity, CPU architecture, ability to send Internet Control Message Protocol (ICMP) packets, and the correctness of the system timestamp, to name a few. These approaches can be used by malware to identify popular online dynamic analysis platforms and withhold their behavior when being analyzed.

The *FPE* framework can expose dichotomous behavior by forcing into execution branches containing malicious code that an app dissembles when it is being analyzed. Listing 4.10 shows a contrived Java source code example that contains various known techniques to probe if an Android app is executing within an emulator. The count of emulator artifacts is aggregated to increase confidence in the assessment. The detection methods in Listing 4.10 are tailored to the emulator from the Android SDK running API level 28 (Android 9.0). For a systematic approach in emulator detection, each emulator image can be examined for artifacts that entail or are indicative of an Android emulator to create *fingerprints*, so apps can identify a virtualized environment at runtime.

```java
TelephonyManager telephonyManager = (TelephonyManager)
   getSystemService(TELEPHONY_SERVICE);
int indicators=0
if (new File("/system/xbin/su").exists())
    indicators++;
if (telephonyManager.getImei().equals("000000000000000"))
    indicators++;
if (android.os.Build.getSerial().equals("unknown"))
    indicators++;
...
if (android.os.Build.DEVICE.equals("generic_x86"))
    indicators++;
if (android.os.Build.TAGS.equals("dev-keys"))
    indicators++;
if (indicators > EMULATOR_THRESHOLD_CONSTANT)
    return "emulated";
else
    return "not-emulated"
```

Listing 4.10: Android emulator detection for API level 28 (9.0).

An artifact common to both the Android SDK emulator and some analysis environments is the presence of root access or a root management utility on the device. The Android SDK provides root access by default on the emulator via Android Debug Bridge (ADB). ADB is a facility in the Android SDK that allows a computer to perform actions on the Android device such as installing apps, transferring files, injecting events, backing up data, and more. Using a rooted device is generally required for some instrumentation techniques. A *rooted* *device* is one that provides the user with the discretion to allow processes to execute as the *root* user, bypassing Android's security model. The *root* user is a special user in Unix-like OSs with a UID of 0 that is generally used for the administration of the system. Executing a process as the root user allows a process to access all files, remount the read-only `system` partition as readable and writable to make persistent changes, execute binaries requiring elevated privileges, make administrative changes, and use sensitive capabilities (e.g., reading the raw device file containing the coordinates of the touches and gestures on the screen). Apps that deal with financial data generally perform root detection to determine if the device is compromised. If they detect that a device is compromised, then program may terminate in an attempt to proactively prevent data disclosure and modification. Listing 4.11 shows some methods for gathering evidence that the device is *rooted*. For readability, some of the method calls in Listing 4.11 have been named to indicate their functionality. Malware apps may or may not be discouraged due to the presence of a rooted device depending on their goal. Sun et al. [49] noted that the battle between the processes who hide root and those who wish to detect it is asymmetric since the processes trying to hide root have root access and can interpose on a *sandboxed* process that is trying to detect it. Sun et al. suggested that Google provide a trusted API for root detection that is built into the OS, which they later provided with the SafetyNet Attestation API [174]. Hooking frameworks that intercept method calls require root access to modify Android OS files to perform the interposition.

```
int indicators = 0;
if (new File("/system/xbin/su").exists())
    indicators++;
if (ro_debuggable_prop_is_true())
    indicators++;
if (busy_box_binaries_present())
    indicators++;
...
if (su_in_path_which_command())
    indicators++;
if (indicators > ROOT_THRESHOLD_CONSTANT)
    return "root";
else
    return "noroot"
```

Listing 4.11: Examining for indicators of root access.

In addition, Bionic Lib C interposition using `LD_PRELOAD` generally requires root access to modify an environment variable to enable the analysis.

4.3 Local Denial of Service Attacks

Researchers have previously shown that the Android OS is vulnerable to local DoS attacks [30–33, 175, 176]. As the Android codebase becomes larger and more complex over time, additional local DoS attacks will likely be discovered. The Android framework provides a large attack surface through its rich set of APIs available to third-party apps. Third-party apps can provide input via the APIs that are processed by the system. In addition, a system process, named `system_server`, exposes app components directly to other apps co-located on the device. When the system does not perform adequate input validation, this can result in a fatal fault in a system process, leading to a system crash. A system crash directly affects the user by reducing the availability of the device. The state of all current apps is lost and the device will be unavailable while it restarts. Listing 4.12 provides an example of a fatal exception occurring in a system process that leads to a system crash. A local app

```
Shutting down VM
*** FATAL EXCEPTION IN SYSTEM PROCESS: main
java.lang.RuntimeException: Error receiving broadcast Intent { act=com.sec.android.
    intent.action.SSRM_MDNIE_CHANGED flg=0x10 bqHint=4 } in com.samsung.android.mdnie
    .AdaptiveDisplayColorService$ScreenWatchingReceiver@e5dbce5
  at android.app.LoadedApk$ReceiverDispatcher$Args.run(LoadedApk.java:1003)
  at android.os.Handler.handleCallback(Handler.java:739)
  at android.os.Handler.dispatchMessage(Handler.java:95)
  at android.os.Looper.loop(Looper.java:158)
  at com.android.server.SystemServer.run(SystemServer.java:508)
  at com.android.server.SystemServer.main(SystemServer.java:363)
  at java.lang.reflect.Method.invoke(Native Method)
  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:1230)
  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:1120)
 Caused by: java.lang.NullPointerException: Attempt to invoke virtual method 'int
     android.os.Bundle.getInt(java.lang.String)' on a null object reference
  at com.samsung.android.mdnie.AdaptiveDisplayColorService$ScreenWatchingReceiver
  .onReceive(AdaptiveDisplayColorService.java:419)
  at android.app.LoadedApk$ReceiverDispatcher$Args.run(LoadedApk.java:993)
  ... 8 more
...............
THIS IS SYSTEM_SERVER.. store dumpState!!
Sending signal. PID: 3514 SIG: 9
```

Listing 4.12: Logcat log containing a fatal exception in a system process.

can repeatedly cause a system crash on the device to make the local DoS attack persistent. Therefore, the device will continually crash, denying access to the user, until the attacking third-party app is uninstalled from the device.

Android has a diverse ecosystem and each device provides a specific set of mechanisms to remove a third-party app that interferes with normal usage. Android has been ported to various embedded devices and they do not all offer the same recovery mechanisms that are provided by Android smartphones. These limitations may be due to physical attributes of the device or a lack of certain system software components. For example, some Android devices do not contain any physical buttons which prevents them from booting into an alternate mode from a powered-off state. Certain alternate modes provide the user with different options to remove a misbehaving third-party app. When recovery facilities are absent or incomplete on embedded Android devices, this leaves the user with fewer options to remove the app. Depending on the Android device, there may be cases where the user is unable to remove the app and the device becomes functionally useless (see Section 7.4). In other cases, the user may have to endure complete data loss to regain proper functionality of the device.

Chapter 5: Forced-Path Execution Framework for Android

The *FPE* framework addresses two problems simultaneously: programmatically interacting with an Android app and enumerating its possible behaviors. Section 5.1 provides high-level implementation details about the *FPE* framework and its workflow. Section 5.2 discusses discovering and interacting with the app's GUI and simulating different environments. Section 5.3 describes how taint analysis is used to discover when to explore all branches of a conditional statement.

5.1 Framework Implementation

The *FPE* framework can be generalized and ported to analyze the binaries of programming languages instead of being limited to modeling the various aspects of the Android OS. I have implemented the forced-path execution framework for Android [59, 60] as an exemplar. A generic approach can be used to explore the different states of a binary developed in an arbitrary programming language. The *FPE* framework code should be developed in the same programming language of the binaries that it analyzes in order to obtain access to the core library functions, specific language features, and the ability to easily integrate third-party libraries. A parser needs to be created for the disassembly of the target language if one is not readily available. Each instruction in the target language requires an implementation for the *FPE* framework. A set of targeted functions specific from core libraries and relevant third-party libraries need to be identified that are of interest so that they can be intercepted and recorded. For example, if *FPE* were being developed for Python, various methods from the `subprocess` module [177] would be strong candidates for logging due to their ability to execute commands on the host system and also for the possibility of command injection. Some abstraction is required for the functions to obtain user input and exercise the app under test. To accomplish this, special attention is required to identify and model GUI usage and asynchronous events that do not have direct links in the code such as threads, callbacks, handlers, GUI elements, etc. Identifying and modeling these behaviors is a manual process and can be time-intensive to implement.

The *FPE* framework takes an Android app as an input, programmatically executes it, and provides the behavior of the app as an output. The primary output is the app's usage of the Android API, although the framework can also report interesting data flows with regard to capability leakages and PII leakages. The *FPE* framework is primarily implemented in the Java programming language. This allows the framework to easily access Java language features as well as the Java API that is composed of the Java Class Libraries (JCL). Android apps are primarily developed using Java. Recently, support for Kotlin development has been added. The Java bytecode is converted to Dalvik bytecode, a `dex` file, using the `dx` binary belonging to the Android SDK.

A standalone Android app in encapsulated in an APK file should contain a valid `classes.dex` file, which contains the Dalvik byte code of the app.[1] The framework converts the `classes.dex` file contained within the APK file using a program named baksmali to obtain an IR called smali [130], which is more human-readable than the Dalvik bytecode and is similar to assembly for other languages that use bytecode. Smali is also the name of an assembler to the accompanying baksmali disassembler. These two tools, baksmali and smali, are open-source and can disassemble and reassemble the bytecode in the `classes.dex` file. Together they can be used to repackage Android apps after the smali files have been modified to alter the native functionality of an app. App repackaging, also known as app cloning, is a popular tactic employed by malware authors and countermeasures to detect and prevent app repacking are actively being researched [72, 178, 179]. The *FPE* framework was successfully used to generate app traces in a system to detect repackaged apps, which is detailed in Section 6.4.[2]

The framework directly utilizes the smali files resulting from disassembling an Android

[1]Android APK files simply containing resource files with no executable code are ignored by *FPE*.
[2]Google refers to app repackaging as app impersonation [180].

app. The output of the baksmali tool is a directory containing the smali files where each file is arranged hierarchically corresponding to its package. Each smali file roughly corresponds to a Java source code file except that nested classes and anonymous classes each have their own smali file. This behavior is similar to Java bytecode where nested and anonymous class files can be identified due to them containing a $ character in their class file name (e.g., `PhoneService$3`). Listing 5.1 provides a small method in smali format. The framework has an interpreter that parses and correspondingly executes the assembly in the smali files. There are 226 instructions in the Dalvik bytecode, which have the same format [181] as they do in smali.

```
.method public onCreate(Landroid/os/Bundle;)V
    invoke-super {p0, p1}, Lcom/phonegap/DroidGap;->onCreate(
       Landroid/os/Bundle;)V
    const-string v0, "file:///android_asset/www/index.html"
    invoke-super {p0, v0}, Lcom/phonegap/DroidGap;->loadUrl(Ljava/
       lang/String;)V
    return-void
.end method
```

Listing 5.1: Smali Method Snippet.

The interpreter in the *FPE* framework contains a Java implementation for each Dalvik instruction. The instruction can contain operands, which are referenced by a register number. Android devices primarily use the ARM CPU architecture, which is register-based and uses a Reduced Instruction Set Computing (RISC) CPU. The register-based architecture, as opposed to a stack-based architecture, is an optimization for the mobile platform. As an instruction is encountered, the operand(s) are obtained and the instruction is executed using the Java-based implementation in the *FPE* framework. If the instruction does not initiate a control transfer or invoke a method call, then linear execution continues and the next instruction is fetched and executed.

The linear execution of a basic block continues until an instruction is encountered that alters the control flow. A *basic block* is a sequence of code instructions that does not contain any instructions that could cause a control transfer. Android apps tend to be event-driven with user interaction playing a prominent role in determining the functionality that an app executes. In addition, there are asynchronous behaviors, that are modeled in the *FPE* framework. I used the dataset provided by Cao et al. [182] from their EdgeMiner tool that uncovered indirect control transfers in Android app code. Additional behavior of the execution of an Android app has to be modeled, such as loading the static variables of a class (i.e., executing a class static initialization block) the first time the class is referenced in the code. The semantics of the language should be investigated and the framework should mimic the same runtime behavior.

Inter-Component Communication (ICC) is the process of an app component communicating with another component where the two communicating app components may belong to the same app or different apps. ICC is performed by sending an Intent message from one app component to at least one destination app component. Performing ICC with a broadcast Intent message has a single source app component and potentially multiple destination app components. The sending of Intent messages from an app is recorded by the *FPE* framework since this data is relevant to understand what messages an app sends to itself, other apps, and the system. In certain circumstances an app can send PII via broadcast Intent messages that can be received any app on the device that has registered to receive the action string that is being used in broadcast Intents (see Section 6.3). The *FPE* framework models ICC communication by obtaining the Intent message from the sender app component and delivering it to the destination app component, so the destination app component can obtain data from the Intent. Currently, the *FPE* framework only handles ICC when the sender and receiver app components reside in the same app, although inter-app ICC will be logged. This current limitation is due to the *FPE* framework only analyzing a single app at a time. With some engineering effort, two or more apps could be analyzed concurrently and an Intent message sent from one app to another app could be properly delivered and processed.

The arguments to instructions and the return value, if any, are stored in a custom Java data type representing a register, it contents, and additional attributes for processing. The custom Java data type for registers contains the actual value for primitive data types, a reference to the actual runtime object for objects, or a modeled representation of the object for certain object references. The register custom object data type also contains the runtime type of an object, the register number, fields and elements for modeled data types, taint information, variable name, and the source of the data. The data types defined in the app code (e.g., not part of the Android API) are modeled using the custom Java register data type. Most of the objects in the Android API and Java API use an actual runtime object instead of a modeled representation of it. The actual runtime object from the custom register data type is used to perform method calls via the Java Reflection API.

Smali has various branching instructions to jump to a separate code location within the current method. Unconditional jumps are represented by the `goto_<label>` instruction, which are often used to construct loops. The *label* portion of a `goto` instruction is followed by a hexadecimal number. Smali has two instructions for switch statements, `sparse-switch` and `packed-switch`, which will branch to a particular switch case, including the default case, depending on the input value being evaluated. Another set of conditional jumps is represented by 12 different instructions that evaluate a predicate. These twelve conditional jumps all have an `if-` prefix. For example, the `if-nez v1, :cond_8` smali instruction will branch to the `:cond_8` label if the value contained in `v1` register is not equal to zero. If the value contained in `v1` register has a value of zero (numerical value of *0* or a `null` Java object reference), then linear execution continues.

Figure 5.1 shows a small execution tree for a broadcast receiver app component named `TaskReceiver` that contains three conditional statements. The accompanying smali file for `TaskReceiver` is provided in Appendix A. The path with the red arrows represents the only path through the `TaskReceiver` app component that starts the `TaskService` app component to initiate communication with a C&C channel within a pre-installed app (see Section 4.1.2 for details). Each node contains the type of node, file name (which has been shortened in the figure for readability), line number, and node ID. The nodes that have a type starting with `finished` represent returning from the `onReceive` entry point method for broadcast receivers (i.e., a completed path through the app component) or a program exit (e.g., encountering

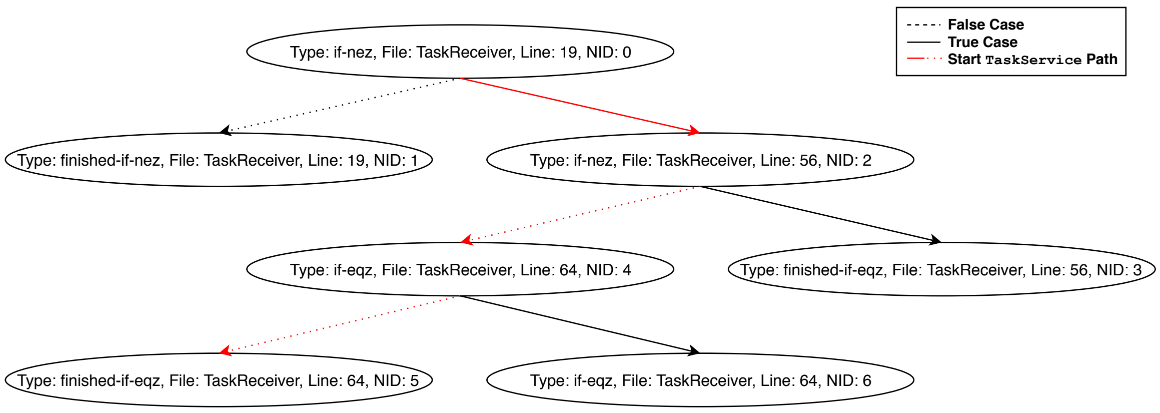

False Case
True Case
Start TaskService Path
Type: if-nez, File: TaskReceiver, Line: 19, NID: 0
Type: finished-if-nez, File: TaskReceiver, Line: 19, NID: 1
Type: if-nez, File: TaskReceiver, Line: 56, NID: 2
Type: if-eqz, File: TaskReceiver, Line: 64, NID: 4
Type: finished-if-eqz, File: TaskReceiver, Line: 56, NID: 3
Type: finished-if-eqz, File: TaskReceiver, Line: 64, NID: 5
Type: if-eqz, File: TaskReceiver, Line: 64, NID: 6

Figure 5.1: Execution tree for the `TaskReceiver` app component.

`java.lang.System.exit(int)`).

The execution paths taken are modeled using a binary tree. The root node is the first conditional statement encountered during execution and its two children represent the two Boolean outcomes of predicate evaluation (i.e., *true* and *false*). When a switch statement is first encountered during the traversal of an execution path, the possible switch cases are enumerated and inserted into the binary tree where each switch case has a switch case as its left child node except for the default switch case, which has no left child node. When a switch case is executed, the next conditional statement encountered will be the right child node of the chosen node representing the switch case. The enumeration of switch cases that belong to a switch statement only occurs the first time it is encountered during a particular execution path. During a subsequent execution along the same path, thus far, the switch cases will be in place and execution will proceed to any switch case that has not had all execution paths stemming from the switch case already completed.

The final node in the execution path will have either its left or right child node, a leaf node, be marked as a completed path (e.g., a *finished* node in Figure 5.1). The Java data type for each node in the binary tree has Boolean values indicating if there is an untaken execution path stemming from the node in general and also for both of its direct child nodes. The *FPE* framework uses a depth-first search exploration method to take execution paths through an app component. A user-selected number of execution modules operate concurrently as they execute through the code and share a binary tree structure modeling the visited execution paths. In the general case, a node in the tree represents an *if statement*, *switch case*, *GUI element*, or a *catch block*.

After each execution run, an in-order traversal of the tree will examine each node and update its references to determine if all execution paths stemming from its child nodes have been traversed. The propagation of this information will initially start with leaf nodes and work its way up towards the root of the binary tree as execution proceeds. The in-order traversal of the binary tree continues iteratively until no references are updated (i.e., all references of a node being finished in the tree have been propagated and updated). This is to inform the execution modules of which paths are *open* and have not been taken so no execution is duplicated. The purpose of the binary tree is to model the execution paths that have been executed by the framework and also to indicate paths that have not been fully traversed. Depending on the execution strategy, the framework may *attempt* to execute all execution paths through the app without regard to their feasibility. In practice, visiting all execution paths is generally infeasible in a complex app due to time and resource constraints.

The *FPE* framework uses certain approximations during analysis for practical reasons that reduce the precision of the analysis. Specifically, the number of iterations through a loop is bounded to a user-configurable amount of loop iterations. Without a loop iteration limit, the execution of a single path through an unbounded loop could theoretically continue indefinitely. Therefore, to prevent indefinite iteration through an unbounded loop, *FPE* limits loop iterations to a maximum limit. Additionally, the depth of recursive calls is also limited to avoid theoretically unbounded recursion. Practically speaking, the recursion depth will be limited by the memory of the computer running the analysis. Since *FPE* is primarily implemented in Java, unbounded recursion results in the following error: `java.lang.StackOverflowError`. These two approximations, bounding loop iterations and recursion, are not an uncommon method to bound the analysis in symbolic execution [183–185], static analysis [186, 187], and model checking [188, 189].

Android apps use the Java API, which is used heavily by the Android API. Since the *FPE* framework operates within a Java VM, it has access to the entire Java API using the Java Reflection API [190]. For coverage of the Android API that is not contained in the Java API, the framework utilizes a Java `jar` file from Robolectric project [191] that is accessed at runtime using the Java Reflection API. Robolectric is an open-source project that facilitates quick testing of Android apps within the Java VM instead of using the Android emulator from the Android SDK.[3] The *FPE* framework relies on modeling of some classes and methods instead of relying on the implementation in the Robolectric `jar` file. In addition, I created a blacklist for certain Android API calls so they *are not* concretely invoked on the host machine for security purposes, although the attempted behavior is logged.

When an instruction to invoke a method call is encountered (i.e., `invoke-` family of instructions), the fully-qualified method call is parsed from the smali file and the parameter(s), if any, are obtained by register number. This allows the framework to build and execute the reflective call at runtime. The *FPE* framework can pinpoint the exact location (i.e., line number and smali file) of a concerning API call or an insecure programming practice in the source code an app. The smali format generally supports and preserves the `.line <number>` directives in the Dalvik bytecode, which is used to populate the line numbers in stack traces when an exception occurs at runtime to facilitate debugging. The line in the smali file generally can be converted to the `.line` directive value to reflect the actual line in the source code file to allow a developer to address an issue. There are cases when the `.line` directive is stripped and the corresponding line in the Java source code file cannot be identified.

Android apps can be composed of four different app components: activity, broadcast receiver, service, and content provider and the *FPE* framework can process each of them. Android apps are compartmentalized into app components for easy code reuse and loose code-coupling of the components. This design allows external apps to utilize app components

[3]The Robolectric source code is available here: `https://github.com/robolectric/robolectric`.

of separate apps, in a sense sharing a segment of code. When an app is analyzed by the *FPE* framework, each app component within an app is executed. The launcher component, if present, is executed first. The *launcher component* is an activity app component that is executed when the user clicks on the app's icon in the launcher. For most apps, the launcher component *must be* exported and have an `intent-filter` that has an *action* string of `android.intent.action.MAIN` and a *category* of `android.intent.category.LAUNCHER`. There are some apps that do not have a launcher component and in this case, the first app component listed in the manifest is executed.[4] An `intent-filter` describes the type of actions that an app component can handle and a `category` allows an app component to be classified with a particular purpose. The analysis focuses on a single app component at a time and launches a user-selected number of execution modules to explore the app component. A binary tree modeling the execution paths is created on a per-component basis to limit the scope of the exploration to a manageable size. Any data sent from one app component to another app component in an Intent is saved and is provided to the destination app component when it is processed.

When the *FPE* framework uses an execution strategy that is biased towards exploration, it will enter each `catch` block that is associated with a `try` block. This is done to explore the app's code for code segments that may intentionally throw an exception. Each `try` block and associated `catch` block(s) are modeled in the binary tree. An app can contain malicious code that is contained within a `catch` block, which can be triggered at a point chosen by an attacker or a timer to influence variable selection. Method calls for file and network Input/Output (I/O) generally utilize checked exceptions since it is distinctly possible that an exception can occur during an I/O operation. If the app connects to a server that the attacker controls, the server can send a response to the app that can remotely cause an exception and cause a control transfer to a `catch` statement that will perform malicious actions. Graa et al. [192] provide a notional example where a `try/catch` block is used to leak sensitive data. Artz et al. [193] introduced a benchmark named *DroidBench* consisting

[4]Apps without a launcher component tend to be pre-installed system apps that execute without user awareness on Android devices.

of test cases for Android taint analysis systems. In this benchmark, there are four test cases where the sink of a data leak occurs in a `catch` block [194].

5.2 GUI and App Component Exploration

Of the four Android app components, only an activity app component has a GUI that allows the user to directly interact with the app. Each app component that is declared in the app's `AndroidManifest.xml` file is executed by the *FPE* framework. A description of each app component type is provided in Section 2.2.4.

5.2.1 Simulating System Events

The user does not interact directly with broadcast receiver app components. Their primary behavior is to "listen" for events and the system will start the app components once one of the events occur. The broadcast receiver decides what events they will respond to by registering to receive them by name using action strings. For example, a broadcast receiver app component may only become activated when external system events occur such as joining a network, receiving a text message, placing an outgoing phone call, system reboot, etc. These external actions can occur normally during regular usage, although in an analysis environment, they must either occur naturally or be intentionally created to activate the broadcast receiver. All *FPE* framework execution strategies will explore the various possible execution paths where branching decisions are based on input from the broadcast Intent since it can be externally controlled (e.g., by an attacker for example). This stress tests the component by testing the different code paths that the app component can handle from external events it expects to occur based on the received broadcast Intent and its embedded data.

5.2.2 App Component Lifecycles

Both the activity and service app components have a defined lifecycle. An app component moves through its lifecycle callback methods as the app component executes and can also move into different lifecycle methods in response to user actions and system events. The lifecycle is imposed on these two app component types due to the parent classes that they extend. The lifecycle contains an initial method that is the entry point for the component after its class constructor and an instance constructor that extends the particular app component has been executed. For activities and services, the framework forces execution into the callback methods corresponding to known transitions in the app component's lifecycle. The broadcast receiver has a known entry point method, `onReceive(Context, Intent)`, which executes with the received Intent as a parameter. The content provider has various abstract methods from its parent class that must implemented, and these methods are known and called by the *FPE* framework. Each app component also contains callbacks that it can receive in regard to specific events. The *FPE* framework identifies the callbacks in the code and executes them.

When an activity is created, the GUI elements are generally loaded from an eXtensible Markup Language (XML) file that dictates the layout of the GUI elements. When the activity is loaded, the *FPE* framework examines the layout XML file to determine which GUI elements can be exercised. Certain GUI elements (e.g., buttons, images, lists, etc.) can statically declare the code to execute when clicked using the `android:onClick` attribute in a layout XML file. When this occurs, the *FPE* framework will programmatically interact with each GUI element to simulate a user interacting with the app. In addition to being declared statically in a layout XML file, GUI elements can also be created at runtime and dynamically added to the layout of the activity. When a clickable GUI element is dynamically registered, it also registers a handler containing the code to execute. When this occurs, the *FPE* framework immediately locates and executes the callback handler code to explore the actions it performs.

5.2.3 Activity GUI Layout

Activity app components are necessary for directly interacting with the user. Once an activity is started, it will progress through a series of callbacks mediated by the system

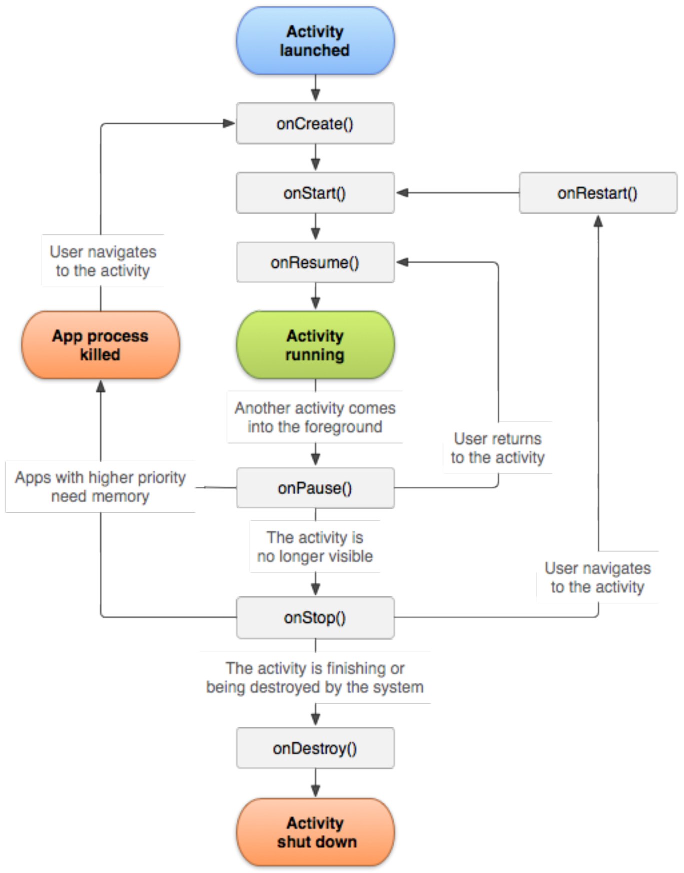

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 5.2: Android activity lifecycle diagram.[5]

and influenced by the user's interaction with the app and system events. Figure 5.2 shows Google's depiction of the activity lifecycle for the activity app component. The first lifecycle callback method to be executed for an app is the `onCreate(Bundle)` method when it is first created and the final callback in the lifecycle is `onDestroy()` when the system removes the activity and frees resources that were allocated to it. Generally, developers override the `onCreate(Bundle)` method and provide their own initialization routines for the app, create the layout, obtain dynamic content, and populate its GUI elements.

The `Activity.setContentView(int)` method will inflate the layout that is identified by

[5]This image is borrowed from Google's Android Developers website: `https://developer.android.com/guide/components/activities/activity-lifecycle`.

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/
    android"
     android:layout_width="match_parent"
     android:layout_height="match_parent"
      android:id="@+id/display_results_scrollable_layout"
     android:orientation="vertical" >

     <LinearLayout
     android:layout_width="match_parent"
     android:layout_height="match_parent"
     android:id="@+id/display_result_layout"
     android:orientation="vertical" >
     <Button
         android:id="@+id/buttonActionString"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:onClick="sendIntent"
         android:text="" />
     <TextView
         android:id="@+id/logResultsTextView"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:maxLines="400"
         android:scrollbars="vertical"
         android:text="" />
     </LinearLayout>
</ScrollView>
```

Listing 5.2: Example Activity Layout XML File.

the integer parameter to the API call which is a resource identifier corresponding to the name of a layout XML file. The layout file dictates the structure and dimensions of GUI elements that the user interacts with. The layout file itself must contain a root element. This element generally acts as a container for the other elements and dictates the placing of its constituent elements (e.g., `LinearLayout`, `ScrollView`, `ListView`, etc.). Listing 5.2 provides a layout file named `activity_displayresult.xml` belonging to an activity in my open-source tool named, *Daze* (see Chapter 8). This layout file contains a `Button` tag that registers the code to executed when it is clicked. Specifically, when the button with an id of `buttonActionString` is clicked, it will execute the `sendIntent(View)` method of the activity that loads the layout file. In the source code for *Daze* project, this layout file is loaded in the `ITA_DisplayResult` activity.[6]

[6]Source code is available here: `https://github.com/Kryptowire/daze/blob/master/app/src/main/java/com/kryptowire/daze/activity/ITA_DisplayResult.java`.

5.3 Taint Analysis

Taint analysis is a technique to track data flows within a program. The code point in which the data of interest is introduced into the program is called a *source*. Data originating from a source is marked as *tainted* and is tracked throughout its lifetime in the program. When tainted data from a source reaches a *sink*, an alarm is raised as this represents a complete data flow originating from a source flowing to a sink. The determination of the sources and sinks in a program is context-dependent and depends on the overall purpose of taint analysis. There are three different methods in which taint analysis is used in conjunction with *FPE* framework:

- Tracking data from the execution environment to determine when it is used in a branching decision. Depending on the executing strategy used, once data from the environment is evaluated as part of a predicate, then all branches of a conditional statement will be explored.
- Discovering data flows where an app obtains user PII and subsequently sends the PII outside of the app boundary where it may be obtained or observed by an entity external to the app.
- Finding input from app component boundaries that reach concerning API calls, resulting in a capability leak. Primarily, this is used to detect an external app sending data, generally via an Intent to another app, where the external app controls one or more parameters to a sensitive API call.

The underlying mechanisms for performing the taint analysis is common among these three approaches except for the respective sources and sinks. The taint propagation rules operate on the register level. As the *FPE* framework operates on smali, a register is used to represent primitive data types and object references. The taint of an object is modeled as a bit-vector where each bit represents a different taint type. The two primary mechanisms in which a taint is introduced into the program is the return value from Android API calls and external input that enters the app boundary and is available to it as an object (e.g., Intents, `Parcel` objects for bound services, etc.).

5.3.1 Detecting PII Leaks

The *FPE* framework uses taint analysis is to identify PII leaks within Android apps. A *PII leakage* occurs when user PII is obtained by an app and is used or transmitted in a way that makes the PII data directly or indirectly available to a local entity on the device or a remote entity via means of wired or wireless network communication. The selection of sources and sinks for PII are based on SuSi [195] and is augmented based on my knowledge of the Android API. A common PII leakage involves leaking the unique identifiers of the device to the network for advertising purposes.

Detecting privacy leaks in Android apps is an active research are where both static [193, 196] and dynamic [197] approaches have been developed. Enck et al. [197] developed a dynamic taint analysis engine, named *TaintDroid*, that detects PII leaks in Android apps. This solution, by design, needs to be built into Android OS to be able to track the data flows through an app, JNI interfaces, and the file system. A group of researchers updated the TaintDroid open-source codebase to support Android 4.3, but it is currently unsupported beyond that version [198]. Artz et al. [193] developed an open-source static taint analysis solution named FlowDroid for detecting PII leaks in Android apps.

Discovering PII leakage involves tainting the output of various API calls that obtain PII from the device (sources) and then tracking their usage and reporting if tainted data from the sources reach an API call that will make the PII be sent over a communication channel or be exposed to other local processes on the device (sinks). Listing 5.3 provides a Java source code example of a simple PII leak where the user's phone number is obtained (i.e., source) and is *leaked* to the logcat log (i.e., sink). This local PII leak enables another entity (e.g., pre-installed app or third-party app using an exploit to obtain the system-wide logcat log) to obtain this PII. Alternatively, the sink on line 4 could have been the tainted data being written to an external host over a network socket.

```java
private String getPhoneNumber() {
    TelephonyManager tm = (TelephonyManager) getSystemService(
       TELEPHONY_SERVICE);
    String number = tm.getLine1Number(); // source
    Log.i(TAG, "phone num=" + number); // sink
    return number;
}
```

Listing 5.3: Contrived source code example of a PII leak.

5.3.2 Detecting Capability Leaks

FPE also uses taint analysis to discover capability leaks involving concerning data flows within an app. A *capability leak* occurs when an app can interact with another app to perform some action on its behalf, effectively leveraging an open interface in another app to obtain a capability. Primarily, these will likely manifest as data flows where an external app provides an Intent message and certain embedded data from the Intent enters into an app component and ultimately flows into a sensitive API call as parameter(s).

When *FPE* uses the *exploring the execution environment* execution strategy, it uses taint analysis to determine when environment data affects branching decisions. The sources are data that comes from the execution environment as explained in Section 3.3.4. The sinks are conditional statements. This allows controlling the outcome of branch evaluation on a selective basis that allows the exploration of branches that can be reached depending on external inputs to the app.

A severe vulnerability manifests when an app can provide arbitrary commands to be executed by a pre-installed app that has `system` user privileges. Listing 5.4 provides a concrete example of a privileged pre-installed app on an Oppo F5 Android device exposing the capability to execute commands as the `system` user [21].[7] In Listing 5.4, the *source* is located on line 2 where externally-provided data residing within an Intent enters the program and the *sink* is on line 9 where a string originating from the Intent is executed. The command string to be executed can be obtained from the Intent on either line 9 or line 11. *FPE* will explore both branches of the conditional statements on lines 5, 10, and 14

[7]I discovered this vulnerability and Mitre assigned it CVE-2018-14996 [199].

```
 1 @Override
 2 public int onStartCommand(final Intent intent, int flags, int
       startId) {
 3   new Thread() {
 4     public void run() {
 5       if (intent == null) {
 6         stopSelf();
 7         return;
 8       }
 9       String action = intent.getStringExtra("action");
10       if (action.isEmpty()) {
11         action = intent.getAction();
12       }
13       Log.i("DropboxChmodService", "action = [" + action + "]");
14       if (action.isEmpty()) {
15         stopSelf();
16         return;
17       }
18       try {
19         Process process = Runtime.getRuntime().exec(action);
20         Log.i("DropboxChmodService", "wait begin");
21         process.waitFor();
22         Log.i("DropboxChmodService", "wait end");
23       } catch (Exception e) {
24         e.printStackTrace();
25       }
26     }
27   }.start();
28   return super.onStartCommand(intent, flags, startId);
29 }
```

Listing 5.4: Concrete example of leaking arbitrary command execution as the `system` user.

since they directly operate on the Intent or its embedded data. Since the outcome of the evaluation of the conditional statements are all influenced externally, then they will all be explored including paths that contain line 19 where externally supplied data from the Intent object reaches the `Runtime.exec(String)` API call as a parameter, resulting in command execution in a privileged process.

5.3.3 Taint Propagation Rules

Within the *FPE* framework, the primitive execution unit is a smali instruction.[8] Each instruction takes zero or more registers as operands. The taint is applied to the custom register data type that can represent a primitive data type or object reference (see Section 5.1). For

[8]Google provides the Dalvik bytecode instructions here: `https://source.android.com/devices/tech/dalvik/dalvik-bytecode`.

modeled data types, the fields are represented as named embedded register data types within the register data type for the base object. For modeled arrays, each element is represented as an indexed register data type allowing for individual tainting of elements. When the array is not modeled and the concrete object is present, then the taint is conservatively applied to the entire array if a tainted object is inserted into the array, which can introduce imprecision.

The primary mechanism for introducing taint information is via the return value for selected Android API calls. The custom register data type contains a bit-vector to support different types of taint simultaneously. When a selected API call returns a value into the program, the bit-vector will contain taint information depending on its source. For example, it may contain a specific bit marking it as PII in the form of unique device identifiers. In addition, certain fields can also introduce taint into the program. The complete set of instructions that can introduce taint are the following: `return`, `return-wide`, `return-object`, `sget-wide`, `sget-object`, `sget-boolean`, `sget-byte`, `sget-char`, `sget-short`, `iget-wide`, `iget-object`, `iget-boolean`, `iget-byte`, `iget-char`, and `iget-short`. For the `return` family of instructions, whether or not taint is applied to a return value depends on the specific API call that was just invoked. For tracking capability leaks, certain objects that are external inputs to an app entry point are marked as tainted.

Whenever a register data type in an instruction is copied or moved, it maintains it taint information. With regard to the `invoke` family of instructions, some of the API calls are modeled where the behavior is known. Since I have not modeled the semantics of each API call, I apply a partially-conservative approach to propagating the taint for those API calls that are not modeled. In this circumstance, the return value, if any, will be tainted with each applicable taint type, if any of the parameters to the method call are tainted or the receiver object for the method call is tainted. Therefore, the taint is primarily propagated through inputs and outputs to the `invoke` family of instructions for invoking method calls. Beyond method calls and the storing and retrieving of member and static fields, arithmetic operations propagate taint to the resulting value if at least one of the operands is tainted.

In Listing 5.4, an externally controlled tainted object, the `Intent`, has its taint propagated through the return objects of `invoke` instructions that obtain embedded data until it reaches the sensitive `java.lang.Runtime.exec(String)` API call.

While *FPE* primarily focuses on introducing taints and tracking them throughout a program, there are some circumstances where it removes taint from certain types of objects. Specifically, certain types of container classes can have their taint removed. For example, if a non-modeled data structure, such as `java.util.ArrayList`, has its `clear()` method called, then the taint from the entire object will be removed. This logic has been applied for all objects that implement the `java.util.Collection` interface or one of its subinterfaces.

5.3.4 Exploring the Input Domain

When *FPE* uses the *exploring the execution environment* execution strategy, taints are primarily introduced through the return values of API calls that obtain data from the execution environment. This execution strategy utilizes taint analysis to have a sounder approach due to primarily relying on the concrete runtime values during conditional statement evaluation, as opposed to exploring both the *true* and *false* branches of every conditional statement.

```java
private void checkDevice() {
    String model = Build.MODEL; // source
    int sdk_version = Build.VERSION.SDK_INT; // source
    String country = Locale.getDefault().getCountry(); // source
    if (model.equals("SM-G955U") && sdk_version == 28 && country.
       equals("US")) { // sink
        samsungSpecificExploit();
    }
}
```

Listing 5.5: Source Code Example to Check Device Attributes.

Listing 5.5 provides an example where a specific method, `samsungSpecificExploit()`, contains control dependencies on values from the execution environment. While the example provided in Listing 5.5 is specifically created for illustrative purposes, I have discovered Samsung specific exploits in the past (see Section 8.3.4 and [23, 200, 201]). In Listing 5.5,

Table 5.1: Sink statements for the *Exploring the Execution Environment* execution strategy.

Sink Instructions	
if statements (unary)	`if-eqz, if-nez, if-ltz, if-gez, if-gtz, if-lez`
if statements (binary)	`if-eq, if-ne, if-lt, if-ge, if-gt, if-le`
switch statements	`packed-switch, sparse-switch`

there are multiple sources that obtain data from the environment (lines 2-4) which flow into a single sink on line 5. Since the values from the API calls, serving as sources, on line 2-4, will vary from device to device, the `if` conditional statement on line 5 will be forced to evaluate to true, to execute the code in the *true* branch and also the *false* branch. Although sink conditional statement residing on line 5 occupies a single line in Java source code, it will be decomposed into multiple smali instructions. In this context, the sinks are conditional statements. Specifically, the conditional statements are sources from the `if` family of instructions and the two `switch` instructions as shown in Table 5.1.

Chapter 6: Applications of *FPE* Framework

This chapter highlights areas where the *FPE* framework has shown utility. Section 6.1 provides instances where the *FPE* framework discovered instances of insecure programming practices within Android apps. Section 6.2 provides various examples of capability leaks it detected in pre-installed Android apps. Section 6.3 contains examples of detected PII leaks. Section 6.4 contains details of *FPE* being used to generate execution traces for a system named *Dexsim* to detect cloned Android apps.

6.1 Exposing App Network, File Storage, and Logging

Free development software (e.g., Integrated Development Environments (IDEs), SDKs, emulators, etc.) and the Google Play infrastructure have democratized mobile app development by facilitating the creation and distribution of Android apps [202, 203]. While this has resulted in a greater diversity of apps available to the user, this has also resulted in mistakes being made by inexperienced developers that are not familiar with generally-accepted secure programming practices for Android [204, 205]. Google provides security tips for app developers on their *Android Developers* website which provides guidance on network usage, file storage practices, and logging [145]. These are common functionalities that many Android apps tend to use. This section examines each of these behaviors and provides real-world examples of insecure usage of these facilities.

6.1.1 Insecure HTTP Usage

Despite its known shortcomings, HTTP is a commonly used network protocol among mobile apps [206]. Networking APIs offered by the Android platform allow a developer to use an HTTP client to transfer data. HTTP is not a secure method to use when transferring sensitive information, and its usage is considered be a software weakness by Mitre [207]. HTTP does not natively provide any encryption, so the data is transferred in plaintext by default. This makes it incumbent on the developer to encrypt sensitive data prior to its transmission using HTTP. While this may be an appropriate approach for some developers, others may not have the sophistication in cryptography, potentially resulting in operational mistakes (e.g., hard-coding symmetric keys, using keying material with insufficient entropy, exercising improper key management, choosing weak cryptographic algorithms, etc.). The *Android Developers* website does provide security tips covering HTTPS usage and other topics [208].

HTTPS uses TLS, the successor to the Secure Sockets Layer (SSL) protocol, to provide authentication and encryption. In June 2015, the Internet Engineering Task Force (IETF) deprecated SSL v3.0 in Request for Comments (RFC) 7568 [209]. The primary benefits of using TLS is that it can protect against eavesdropping and provide authentication of the server.[1] TLS is commonly used for eCommerce, personal finance, messaging, and to protect login credentials and cookies. TLS relies on public-key certificates and certificate chains to establish chains of trust to authenticate the server.

The app named *OnCourse - boating & sailing* [210] with a version of 1.1 was available on Google Play with a package name of `com.marinetraffic.iais`.[2] The results of the *FPE* framework showed that the app was sending the user's login and password credentials over HTTP. This is an insecure programming practice since the login credentials are not encrypted and can be intercepted on the network [207]. The *FPE* framework showed that the app made a HTTP GET request to `http://mob0.marinetraffic.com/ais/loginxml.aspx?email=USER_INPUT&password=USER_INPUT`. The *FPE* framework uses the text `USER_INPUT` to indicate that the input values were obtained from the user. I confirmed this behavior at runtime by running the app on an Android device and used a proxy to capture the HTTP traffic. The app visited the `http://mob0.marinetraffic.com/ais/loginxml.`

[1]TLS can be used to authenticate both the client and the server by means of public-key infrastructure, although this configuration is not the standard mode of operation in practice.

[2]This app has since been updated and is on version 2.2.0 which uses HTTPS for the login.

`aspx?email=test@email.com&password=e8dhr78wh` URL and the username and password were sent in plaintext as part of the URL querystring.

6.1.2 The Significance of Log Leak

Android provides a system-wide log for development, diagnostic, and debugging purposes. This log is generally accessed, for reading, using a binary called `logcat` that is present on Android devices, so I refer to this log as the logcat log.[3] The logcat log contains five different log buffers where each contains different types of data: *system*, *main*, *radio*, *events*, and *crash*. When reading from the logcat log, any single log buffer or a combination of the log buffers can be read using the appropriate command-line parameters to the `logcat` command. The Android framework provides various classes in the `android.util` package that allow an app to write to the logcat log. The primary class used for writing to the logcat log is simply called `Log`, but there additional classes to write to the logcat log (e.g., `EventLog`, `LogPrinter`, etc.).[4]

When writing a message to the logcat log, a log message is composed of a tag and a message. The *tag* is a short string that can be used to group certain types of messages based on a shared attribute (e.g., messages from a certain class, module, or circumstance), and the *message* is the information that the developer intends to communicate via the log message. In addition, processes written in native code can link against the logging library provided with the Android NDK to write to the logcat log. Both Android apps and system processes commonly use the logcat log to write messages. Logging is exceptionally helpful in discovering the cause of faults or unexpected behavior since it contains a trace of log messages and also a stack trace for uncaught exceptions. The logcat log provides the following log levels: *verbose*, *debug*, *info*, *warn*, *error*, and *assert*. Each of these log levels has a priority level with *verbose* being the lowest and *assert* being the highest. When using the `logcat` command, specific log levels and log tags can be passed in as parameters to

[3]In this dissertation, I refer to this log as the *logcat log* to differentiate it from other logs (e.g., bluetooth *snoop* log, modem log, etc.) on the device.

[4]Interestingly, the commonly used `print` and `println` methods of `java.lang.System.out` static field will write the output to the logcat log with a log tag of `System.out`.

filter the output. If the user is only interested in logcat messages with a specific tag, these can be filtered by silencing the log messages from all other log tags.[5] When a log level is provided, either with the wildcard operator for all tags (e.g., `logcat *:E`) or a single tag (e.g. `logcat Zygote:E -s`), all messages at that log level or higher will be present in the log. So if a value of *verbose* is used in conjunction with a log tag of `generic_tag`, then all log messages with a log level of *verbose* (the lowest level) and higher (all other levels) will appear in the output, effectively providing all levels for log messages with a tag of `generic_tag`.

The capability to read from the logcat log has evolved over different versions of Android. Since AOSP code is open source, I determined the exact version in which Google changed the access requirements for apps to obtain the `READ_LOGS` permission from the Android OS. As of Android version 4.0.4, the `READ_LOGS` permission had a `android:protectionLevel` of `dangerous`, which allowed any app, including third-party apps, to read the system-wide logcat log if it was granted the `READ_LOGS` permission [211]. This behavior changed in Android 4.1, so that the `READ_LOGS` permission had a `android:protectionLevel` of `signature|system|development` so that third-party apps could no longer read from the system-wide logcat log [212]. On Android devices running Android 4.1 and higher, an app can obtain its own log messages that they write to the logcat log even if they have not been granted the `READ_LOGS` permission by the Android OS. Apps can simply execute the `logcat` command to capture the log messages that the app itself writes and log messages from other processes will be filtered out. This capability is useful for software testing and maintenance since a log trace may contain helpful data with regard to identifying the cause of a bug or fatal app crash. App developers can easily integrate a library that will capture log messages and crash reports. This enables the developer to investigate and address issues experienced by users of their app, potentially stemming from Android version compatibility issues or faulty app logic.

Although third-party apps installed on recent versions of Android cannot directly access the system-wide logcat log, there have been various vulnerabilities discovered in which a pre-installed app leaks the logcat log to a location accessible to third-party apps. This is not

[5]Silencing all other log tags other than those explicitly provided is done with the `-s` command line argument.

simply a theoretical vulnerability, I found numerous instances where this vulnerability has been introduced by different Android OEMs (e.g., Samsung, LG, ZTE, Asus). I discovered a method to obtain the logcat logs from a local third-party app executing in the background without awareness of the user on Samsung Android devices [23]. Moreover, certain of these Samsung Android devices were also writing the text of the notifications that the user receives to the logcat log. Therefore, an adversary using this technique would be able to secretly monitor the system-wide logcat log containing the content of each notification received such as text messages, Facebook messages, WhatsApp messages, partial emails, and all other notifications. In 2018, I identified 11 different Android devices spanning 5 vendors that allowed a third-party app on the device to cause a pre-installed app to leak the logcat log to a location on the file system that it can access [21]. Table 6.1 provides the Android device models and carrier, if any, of the Android devices that were found to leak the logcat log. For the vendors LG and ZTE, the vulnerability that leaks the logcat log was present on all the models, 4 for each vendor, I examined.

Table 6.1: Android devices that exposed the logcat log.

Device	**Carrier**
ZTE Blade Spark	AT&T
ZTE Blade Vantage	Verizon
ZTE ZMAX Pro	Total Wireless
ZTE ZMAX Champ	Multiple Carriers
LG G6	AT&T
LG Phoenix 2	Unlocked
LG X Power	Unlocked
LG Q6	Unlocked
Vivo V7	Unlocked
Asus ZenFone 3 Max	Unlocked
Orbic Wonder	Unlocked

The *FPE* frameworks discovered that an app with a package name of `com.mmt.salesTracker` (`versionCode = 1`, `versionName = 1.5`) obtains the destination phone number from outgoing phone calls and writes them to the logcat log. Specifically, the `CallReceiver` broadcast receiver component registers for the `android.intent.action.NEW_OUTGOING_CALL` action string and extracts the `android.intent.extra.PHONE_NUMBER` value from the broadcast Intent it receives and then writes the following message to the logcat log: `Dialled Number is: <dialed number>` with a log tag of `CallReceiver`.[6] In addition, the *FPE* framework detected that an app with a package name of `gn.com.android.salestrackgi` (`versionCode = 1`, `versionName = 1.0`) executes with `system` privileges and writes the body of received SMS messages to the logcat log. The app has a broadcast receiver app component named `gn.com.android.salestrackgi.SMSServerReceiver` that registers for the `android.provider.Telephony.SMS_RECEIVED` action string. Whenever the device receives an incoming SMS message, the Android OS sends a broadcast intent to all apps that meet the following two conditions: (1) the app has been granted the `RECEIVE_SMS` permission and (2) the app has a broadcast receiver app component that registers for the `SMS_RECEIVED` action. The `gn.com.android.salestrackgi` app meets both of these conditions, and the `SMSServerReceiver` obtains the SMS message body and then write it to the logcat log for each received SMS message.

6.1.3 Leaking Sensitive Data to External Storage

External storage is a location on the file system serving as a shared resource that is accessible to all apps on the device that have the corresponding access permissions. Storing sensitive data on Android is generally considered an insecure programming practice [213]. Access to external storage is controlled by a read permission (`READ_EXTERNAL_STORAGE`) and a write permission (`WRITE_EXTERNAL_STORAGE`). In the `AndroidManifest.xml` file of the Android framework, both of these permissions have a value of `dangerous` for their `android:protectionLevel` attribute in their declarations [69]. If the app is subject to runtime permission granting (explained in Section 2.2.3), which is dependent on both the app version and the Android OS version, then the app will not be granted these two permissions, `READ_EXTERNAL_STORAGE` and `WRITE_EXTERNAL_STORAGE` by default. Due to these permissions

[6]The word *Dialled* is misspelled in the app code.

having a `android:protectionLevel` of `dangerous`, the app will have to request the permissions from the user at runtime. Based on my research, I found the following data being leaked to external storage, making them accessible to other apps on the device that have the `READ_EXTERNAL_STORAGE` permission:

- **Logcat log** - Various LG devices (CVE-2018-14982), Orbic Wonder (CVE-2018-6599), Coolpad Canvas (CVE-2018-15004), Vivo V7 (CVE-2018-15001), Asus ZenFone 3 Max (CVE-2018-14979), & numerous devices with a MediaTek chipset (CVE-2016-10135)
- **Screenshot** - Leagoo P1 (CVE-2018-14997), Sony Xperia L1 (CVE-2018-14983), various ZTE devices (CVE-2018-14995), & Asus ZenFone 3 Max (CVE-2018-14980)
- **Modem log** - Various ZTE devices (CVE-2018-14995) & numerous devices with a MediaTek chipset (CVE-2016-10135)

This is concerning since users might not be aware that allowing an app to access external storage may also allow an app to exploit the PII leaks shown above. Andriotis et al. [214] performed a study measuring users' adaption to the new permission runtime granting model, and they found that the majority of users in their study granted the storage permissions to the apps that requested them. External storage contains potentially sensitive personal data such as the user's photos, downloads, and screenshots. It seems that apps write files to external storage so that they can easily be retrieved when using the ADB program that comes with the Android SDK.

Based on examining a report from the *FPE* framework, it showed that a pre-installed app in various Xiaomi Android devices was leaking a file containing the logcat log to external storage. This pre-installed app has a package name of `org.codeaurora.gps.gpslogsave` (`versionCode = 27`, `versionName = 8.1.0`). The name indicates that the Code Aurora project developed the app, but Code Aurora was not able to directly answer the question if they were responsible for developing the app when asked on two separate occasions. Nonetheless, I have only ever encountered this app on the Xiaomi devices. In the report, it showed two following two commands being executed: `/system/bin/sh -c logcat -v threadtime -f /data/data/org.codeaurora.gps.gpslogsave/files/log.txt` and `mv /data/data/org.codeaurora.gps.gpslogsave/files/log.txt /sdcard/GPSLogKit/08-28-12-12-16-887_log.txt`. The first command initiates the recording of the system-wide logcat log to the app's private directory on internal storage. The second command, `mv`, moves this log file to external storage on the file system, making it accessible to any app that has the `READ_EXTERNAL_STORAGE` permission. Upon manual inspection of the app on a live Xiaomi device, I verified this behavior and also that is can be controlled by an external app to obtain the system-wide logcat log.

6.2 Detecting Capability Leaks

Taint analysis is used as the primary mechanism for detecting the capability leaks by *tainting* input at the app component boundaries and recording the event when the *tainted* data reaches a sensitive API call. Table 6.2 displays instances where *FPE* framework detected a capability leak where an external app can influence a sensitive behavior.[7] The app with a package name of `com.lovelyfont.defcontainer` (`versionCode = 5`, `versionName = 5.0.1`) that executes as the `system` user has an exported and accessible service app component name `com.lovelyfont.manager.FontCoverService` that allows external apps to execute arbitrary commands with elevated privileges.

Table 6.2: Capability leaks detected in pre-installed Android apps.

Package Name	**vCode**	**Behavior**
com.lovelyfont.defcontainer	5	Arbitrary Command Execution
com.asus.splendidcommandagent	1510200090	Arbitrary Command Execution
com.fw.upgrade.sysoper	3	Arbitrary Command Execution
org.codeaurora.gps.gpslogsave	27	Leaks logcat log to external storage

This `com.lovelyfont.defcontainer` app is actually present as a pre-installed app on

[7] In Table 6.2, the `vCode` column represents app version code.

multiple vendors such as Tecno and Infinix. An external app simply needs to send an Intent to this service with (1) an action string that contains the string '`form`' and (2) contains the command to be executed stored in the Intent with key name of '`form`'. The capabilities that can be achieved by executing commands as the `system` user vary depending on the Android version of the device, but they generally provide the following capabilities: read/write SMS messages, read/write call log, wipe the device (i.e., factory reset), change the keyboard to a malicious version with keylogging functionality, call emergency numbers such as 911, record a video of the screen, obtain the logcat log, take screenshots, and more. The `system` user is the most powerful user on the device except for the `root` user.

Service app components on Android can optionally export an interface to external apps. A service that passes back an interface and allows binding to it is called a *bound service*. The *FPE* framework explores bound services by discovering the interface they return in their `onBind(Intent)` method. If a service returns a non-null `IBinder` object, then the *FPE* framework will call each of the accessible methods in the interface. The *FPE* framework detected that a pre-installed app with a package name of `com.asus.splendidcommandagent` (`versionCode = 1510200090`, `versionName = 1.2.0.18_160928`) in various Asus devices allowed other apps on the device to execute commands as the `system` user. This app contains a bound service named `SplendidCommandAgentService` that exposes a method named `doCommand (String)` that executes the `String` parameter that is passed to it from external processes.

The Adups apps are present as the FOTA solution for various low-end Android devices. The *FPE* framework discovered that an old version of Adups software allowed external app to execute arbitrary commands as the `system` user. Assuming the timestamp from the device build information is accurate, then a vulnerable version of Adups software was shipping as a pre-installed app as early as July 2$^{\text{nd}}$, 2015. The pre-installed app was present on an Android 4.4.2 device has a package name of `com.fw.upgrade.sysoper` (`versionCode = 3`, `versionName = 1.3`).[8] This app has a broadcast receiver app component named `WriteCommandReceiver` that expects the commands to be present in an Intent string extra named `cmd`. The receiver

[8]This is the pre-cursor app to the `com.adups.fota.sysoper` app referenced in Section 4.1.2.

obtains this string value and executes it using a `java.lang.ProcessBuilder` object.

6.3 Detecting PII Leaks

PII leaks that the *FPE* framework detected are provided in Table 6.3. By a *PII leak*, I mean that PII was obtained by an app and was subsequently left the app boundary where it can be obtained by another entity or process. The app with a package name of `com.aeon.salesstatistics` (`versionCode = 3`, `versionName = 1.3`) uses the `AlarmManager` to set a time for 6 hours later to execute the `SalesStatistics` service app component after the device completes the boot process. This service sends an SMS containing the device IMEI to the following number: `008801795942968`. Whether or not to send the SMS message is controlled by a Non-volatile Random Access Memory (NVRAM) setting which the *FPE* framework detects as external, so it explores both cases.

Table 6.3: PII leaks detected in Android apps.

Package Name	**vCode**	**Behavior**
com.mmt.salesTracker	1	Writes outgoing calls to the logcat log
com.reverie.phonebook	5	Sends SMS messages via implicit broadcast
org.codeaurora.gps.gpslogsave	27	Leaks logcat log to external storage
com.aeon.salesstatistics	3	Sends IMEI over SMS
com.staqu.panasalestracker	8	PII leakage over SMS and HTTPS
com.marinetraffic.iais	3	Sends login credentials over HTTP
gn.com.android.salestrackgi	1	Leaks incoming SMS messages to logcat log
com.sts	8	Leaks device identifiers over SMS

The app with a package name of `com.reverie.phonebook` (`versionCode = 5`, `versionName = 2.1.2`) contains a broadcast receiver app component named `in.co.Reverie.Helpers.SmsReceiver`.[9] When this component receives an SMS message, it will leak the sending number of the received SMS to the logcat log with a log tag of `Sender` and then leaks the body of the SMS message to the logcat log with a log tag of `Message`. Just after leaking the

[9]This app is no longer available on Google Play.

contents of the received SMS message to the logcat log, it takes these two values, sender and text message body from the SMS message, and then puts them into an implicit Intent and then broadcasts it. Therefore, any app on the device that registers for the action of `SmsMessage.intent.MAIN` can receive the broadcast Intents its sends which contain the received SMS message contents. The SMS body and the sender can be obtained from the Intent using the following key names: `get_msg` and `number`, respectively.

The app with a package name of `com.sts` (`versionCode = 8`, `versionName = 2.0`), as shown in Table 6.3, obtains user PII (i.e., IMSI, SIM serial, SIM operator, and the cell tower ID) and sends it in an SMS message to a phone number in India: `00919870932094`. This initially only occurs once and then will subsequently occur any time a new SIM is inserted in the device. In addition, the app will sleep 150 seconds prior to doing anything, and it will check to ensure that a SIM is inserted into the device. This occurs programmatically by the app without any user intervention.

6.4 Generating Execution Traces for App Clone Detection

The *FPE* framework was used in a novel way to aid in the detection of Android app clones. An *app clone* is an unauthorized copy of an app where the app logic has been modified. The cloned app may have modified code or resources. Some potential reasons to modify an app are the following: obviate Digital Rights Management (DRM), introduce malware, replace advertisement IDs for financial gain, introduction of ad libraries, etc. App cloning is a known problem in the Android ecosystem and various research approaches to identify cloned apps have been proposed [73, 179, 215–221]. The existing approaches did not perform well at detecting apps clones after obfuscation techniques were used to transform the app.

The Android app format, APK, lends itself to easy modification and also various open source tools exist to facilitate app cloning. Notably, the open-source software named apktool [222] provides facilities to easily disassemble and reassemble Android apps. Internally, apktool uses the facilities provided by the smali project started by JesusFreke [130]. The smali project is so widely used, that is also being mirrored by Google's git repositories for Android [223]. The Android OS necessitates that installed apps must be signed with an asymmetric private key. Android apps contain a X.509 public key certificate that the OS allows to be self-signed. As the Android APK file format is easily malleable, modifications can be made in a well-understood, straightforward process.

6.4.1 Android App Clone Detection Techniques

App cloning is a known-problem in the Android ecosystem, and researchers have developed various approaches to detect app clones among the various app marketplaces. Some research approaches focus on comparing GUI elements [215, 216], examining resource files [217–219], inserting app watermarks [220,221], fuzzy hashing Dalvik bytecode segments [179], comparing Program Dependence Graphs (PDGs) [73], inspecting semantic information and resource features [224], and screenshot analysis [225]. Although not a strict repackaging approach, Aresu et al. [226] used HTTP traffic to cluster Android malware into families. Scalability is a limitation of certain approaches that make them impractical for real-world use. Specifically, pair-wise comparison becomes prohibitively expensive as the app repository grows in size.

6.4.2 *FPE* Trace Collection

A few modifications were made to the way the *FPE* framework generally operates. *FPE* generates the execution traces that are used to create depth-bounded LZ78 compression trees. The general output of *FPE* is a list of behaviors and data flows. *FPE* was modified to output the stream of opcodes that it encounters while processing an Android app. Dalvik bytecode encodes each instruction as a single byte, allowing for 255 different Dalvik instructions [181]. The *FPE* framework examines the `AndroidManifest.xml` of the app to be processed to extract the statically-declared app components from the app. A time limit is assigned to bound the execution of the *FPE* framework. I found 30 seconds to provide a good trade-off between accuracy and execution time. The allotted 30 second time interval for an app was split evenly among its app components to ensure some coverage among all components.

From each app component entry point, *FPE* first executes the static initialization methods, constructor method for the component's class, and then the app component specific lifecycle methods. The taken execution paths are recorded in a tree structure on a per-component basis. Recursion depth and loop iterations are limited to a single iteration.

6.4.3 DEXSIM Workflow

The system, *Dexsim*, begins an initial indexing phase where it processes apps to generate traces and build a corpus of LZ78 compression trees [227]. During app processing, the app has its `classes.dex` file converted into an IR named smali and its `AndroidManifest.xml` file is parsed to obtain the app's entry points. After the app has been converted to an appropriate format, *FPE* will begin and start generating the opcode traces through all available app components. Each execution trace is a concatenation of each encountered opcode, encoded as integer, from an execution path through the app component. The set of traces from an app $T := \{t_1, t_2, t_3, ...t_n\}, n \in \mathbb{N}$ are divided into non-overlapping segments of constant size w. Using the traces in T, an LZ78 prediction tree, also known as a dictionary, is generated and stored for the app along with its identifying material. The LZ78 compression trees can be used for making predictions [228]. The workflow for *Dexsim* is provided in Figure 6.1.

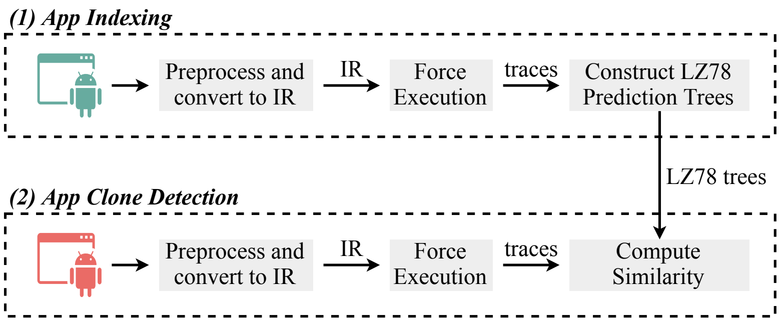


Figure 6.1: An overview of *Dexsim* which works in two phases. First, an indexing phase to build a database of LZ78 trees of opcode traces. Second, a detection phase that finds the nearest trees in the database to an incoming app.

The complete approach to determine an app's similarity to itself and other apps to detect clones can be found in [62].

6.4.4 App Clone Detection Evaluation

Dexsim outperformed all other current research approaches when detecting app clones that had undergone obfuscation transformations. The obfuscations of apps in the sample were performed using SandMark [229]. The first experiment used a set of 3,000 benign apps. From these apps, 18,000 clones were generated where each clone had undergone one obfuscation algorithm using SandMark. Using the 3,000 app benign dataset (BEN), *Dexsim* was run in indexing mode to build a dictionary for each app. Then *Dexsim* was run in detection mode to find the apps that had the closest similarity score to each app in the 18,000 obfuscated app clone dataset (BEN-O). For each clone in the BEN-O dataset, I considered *Dexsim* to properly identify an app clone if it matched its pre-obfuscated app in the BEN dataset using various values of k for its k-nearest neighbors. In the ideal case, *Dexsim* would provide the pre-obfuscated app from the BEN dataset as the nearest neighbor to its post-obfuscated app clone from the BEN-O dataset when k is set to 1. Using k-nearest neighbors with varying levels of k, I provide the detection accuracy of *Dexsim* on the BEN-O dataset when detecting their pre-obfuscated versions as displayed in Figure 6.2.

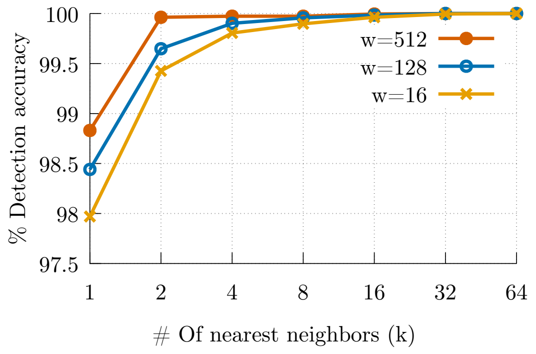


Figure 6.2: Detection accuracy for clones in the BEN-O dataset against their source apps in the BEN dataset. w is the window size.

The next experiment contained a dataset consisting of 20 apps that had highest rate of successful obfuscations among the various algorithms. This dataset (CTR) was used to generate a dataset of 510 app clones (CTR-OS) where each app underwent two or three successful serial obfuscations using SandMark. The initial CTR dataset was curated since certain obfuscation algorithms failed on apps. The CTR-OS dataset contains 150 app clones that underwent two different obfuscation algorithms and 300 app clones that were subject to three serial obfuscations. *Dexsim* provided the original app from the CTR dataset as the nearest neighbor, with an accuracy of 90.1% for three serial obfuscations, 92.5% for two serial obfuscations, and 96.8% for a single obfuscation as show in Figure 6.3. The rate of successful detection to a strict criterion, k=1, goes down as the number of obfuscators applied serially increases, but *Dexsim* displays resilience to obfuscation algorithms even when applied serially.

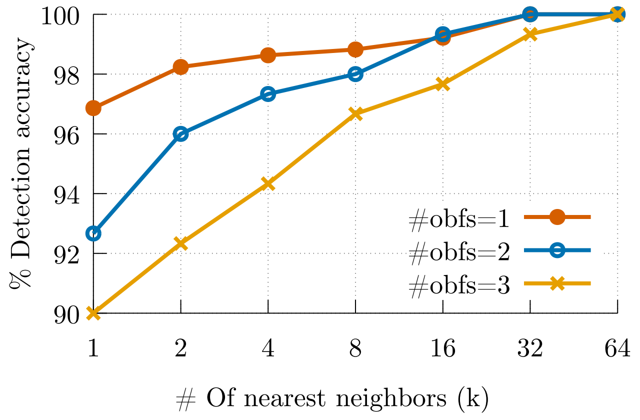


Figure 6.3: Detection accuracy of all clones in the CTR-OS dataset against their source counterparts in the CTR dataset.

Lastly, *Dexsim* evaluated a dataset of 7,000 malware samples (MAL). For each app in the MAL sample, I uploaded the sample to VirusTotal [230] and recorded the malware family name that the anti-virus engines converged (i.e., majority) on, assuming it was not generic. The malware families contain iterations of a sample that has undergone iterative

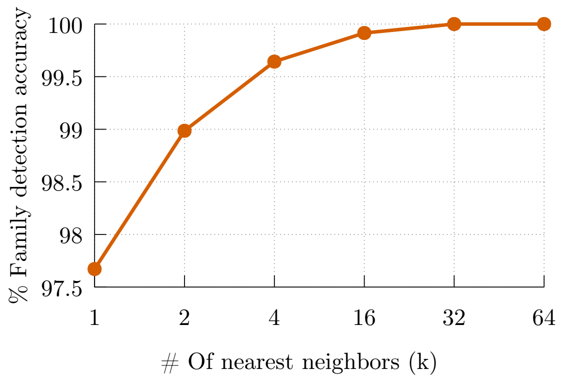


Figure 6.4: Family detection accuracy of all malware samples in the MAL dataset.

development and possibly repackaging. For the MAL dataset, *Dexsim* was considered to be accurate for a malware app if the list of nearest neighbors contained the same family classification as the app under test. Figure 6.4 shows the detection accuracy with regard to the malware family for various levels of k-nearest neighbors. Even with the direct nearest neighbor, *Dexsim* accurately detected the malware family for 97.5% of the samples.

Chapter 7: Intent-Based Denial of Service Attacks on Android

In this chapter, I examine a common communication mechanism in Android, the Intent messaging object [54], and discuss how it can be abused to affect the security and availability of an Android device. I discovered various local DoS attacks that can be launched from a local app that has not been granted any permissions: making the Android OS unresponsive (see Section 7.3.1), selectively targeting and killing other apps co-located on the device (see Section 7.3.2), and causing the Android OS to crash and reboot (see Section 7.3.3). I explain the mechanics of the DoS attacks with regard to an app launching DoS attacks and its interaction with the Android framework in Section 7.3.4. I made the attack more aggressive by programmatically disabling wireless communication mechanisms (i.e., Wi-Fi and Bluetooth) that are used to communicate with certain embedded Android devices. I tested the aggressive DoS attack on a range of Android devices and the results are provided in Section 7.4. I discovered that the DoS attack was particularly severe on certain Android embedded systems that lacked many of the traditional recovery mechanisms that are present on Android smartphones. In certain cases, these DoS attacks can be leveraged to perform a ransomware attack that does not involve cryptography. To counter these attacks, I developed and tested open-source solutions that successfully defended against the local DoS attacks described in Section 7.5.2.

7.1 Threat Model

The threat model is that I assume a low privilege Android app is installed on the device from which it will launch local DoS attacks. This app can reach the device through app repacking, social engineering, unpatched vulnerability, remote exploit, etc. A local presence on the

Android device allows the app to execute code to perform the attacks. Specifically, the attacks I discovered all use the sending of Intent objects as the attack vector. An *Intent* is an abstraction for a IPC message that apps can send between or within apps. Alternatively, an Intent can be sent by an app to communicate between different app components within the same app. Intent objects are available to developers via a class in the Android API [54]. Sending an Intent does not require any permission defined by the Android platform. This enables the least privileged Android app, a third-party app that requests no permissions, to use Intents to perform most of the DoS attacks introduced in this chapter. These DoS attacks were discovered and tested on devices running Android 5.0. After the DoS attacks were responsibly disclosed to Google, they have mostly fixed the issues in their AOSP code base.[1]

7.2 Related Work

Researchers have discovered various local DoS attacks affecting the Android platform. For instance, Armando et al. [30] discovered that the UNIX domain socket to fork processes from `zygote` was world-readable, so a local app could cause excessive process forking from the `zygote` process to overload the device so it becomes unresponsive and eventually crashes and reboots. Arzt et al. [175] found a method to prevent the installation of another app package that relied on a vulnerability introduced by the `dexopt` tool.[2] However, to be useful to an attacker, their attack depended on user interaction, root access, or a separate vulnerability to work surreptitiously since third-party apps cannot programmatically install an app. Ratazzi et al. [176] discovered various DoS attacks that prevent the installation of Android apps by leveraging Android OS limitations. One attack exploited a limitation in the upper limit for the maximum number of app UIDs on a Nexus 10 by installing so many apps that no new apps could be installed. They also noted that Android has a uniqueness requirement for each package name such that no two apps can reside on the device with the same package name.

[1]For testing the attacks in this chapter, please use an appropriate Android version ($\leq$ 5.0) that lacks Google's fixes for the issues.

[2]The `dexopt` tool verifies Dalvik bytecode during app installation.

Chin et al. [158] presented various DoS attacks against apps by intercepting Intents bound for other activity, service, or broadcast receiver app components. This research focuses on the security of app components and the ability to intercept Intents bound for another app. Brand et al. [231] made various modifications to the Android OS source code and environment to automate the sending of Near Field Communication (NFC) messages for fuzz testing and DoS attacks. The NFC DoS attacks focus on battery exhaustion, storage exhaustion, and forcing the NFC subsystem to crash. Huang et al. [31] discovered that invoking computationally heavy API calls in a system process can result in the system process freezing and eventual system reboot. The authors developed a tool that identifies risky method calls to a system process using heuristics to examine the AOSP source code. Pink et al. [232] recognized that sending Intents at a rapid rate can turn into a DoS against the device, but they did not go into more detail beyond that. I explain how the usage of Intents can be used to reboot the device by causing a system crash, make the Android OS unresponsive, and selectively kill other running apps on the device.

7.3 Intent-based DoS Attacks

Sending intents *very rapidly* from a local app can have adverse effects on the Android OS. If intents are sent as quickly as possible, this will cause a user space system crash and subsequent reboot. If the intents are sent at a slightly slower pace, then the device will become unresponsive to user interaction. If the intents are sent rapidly for a controlled period of time, the Android OS will start killing off running apps due to the large amount of Random Access Memory (RAM) that is required to process and service the Intents. The most reliable method to perform Intent-based attacks is from a service app component that returns the `START_STICKY` constant in its `onStartCommand` method so it will be restarted automatically if it crashes or is terminated. For all of the Intent-based attacks, an app component must be selected as the destination to where the intents will be sent. This app component can be external to the attacking app, so as to partially obfuscate the app that is actively performing the attack. Many Android devices will contain a set of pre-installed Google apps that are resident on the device. Therefore, an app such as Gmail or Chrome can be targeted since they have exported activities app components that can be launched externally. The most reliable method is to use an app component within the attacking app. The attack to kill other running apps and the attack to reboot the device require that the Intents being sent use the following two flags from the `android.content.Intent` class: `FLAG_ACTIVITY_MULTIPLE_TASK` and `FLAG_ACTIVITY_NEW_TASK`. These intent flags, when used together, make it so that a new task is allocated and started in the target app even if a matching task already exists [233]. The following flags can be used to prevent an activity app component within an app from showing up in the recent apps list: `FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS` and `FLAG_ACTIVITY_NO_HISTORY`. The app component that will receive the influx of intents needs to have a value of `standard` for its `android:launchMode` attribute which is the default launch mode if one is not provided explicitly. The standard launch mode creates a new instance of the activity and pushes it on top of the current task stack [234]. In addition to the standard launch mode, an activity can have one of the following launch modes: `singleTop`, `singleInstance`, and `singleTask`. The `singleTop` launch mode reuses an activity if it is at the top of the task stack, rather than creating a new instance. The `singleInstance` and `singleTask` launch modes do not allow multiple instances of an activity. The attacks to reboot the device and kill other running apps will work significantly faster if the device's screen is on at the time of the attack. A service app component can turn the screen on from the background by acquiring a wake lock that was created with the `ACQUIRE_CAUSES_WAKEUP` and `SCREEN_BRIGHT_WAKE_LOCK` flags. The Android OS sends broadcast intents when the screen is turned on and turned off, so an app can determine whether or not the screen is on or not by creating a broadcast receiver app component that receivers these broadcast Intents.

7.3.1 Making the Android OS Unresponsive

The Android OS can be made unresponsive to the user if an app sends Intent objects rapidly and continuously. The source code to make the Android OS functionally unresponsive is simple and straightforward. First, an Intent is created for an activity app component that is contained within the attacking app or an external app. Then an `android.app.PendingIntent` object is created with the app's context, the request code, the Intent, and the `FLAG_CANCEL_CURRENT` constant. The `FLAG_CANCEL_CURRENT` flag makes it so that if any `PendingIntent` objects of this type already exist, then they should be canceled and the newest `PendingIntent` object will be used. A `PendingIntent` takes an Intent object and allows another app to send the Intent, potentially at a later point in time. The `android.app.AlarmManager` is retrieved and used to make the `PendingIntent` be launched every 100 milliseconds starting 1 millisecond after the `android.app.AlarmManager.setInexactRepeating(...)` API method call. The `AlarmManager` class interacts with the system's alarm manager which allows an app to send Intents at scheduled times and intervals.

```java
Intent i = new Intent(this, DisplayText.class);
PendingIntent pi = PendingIntent.getActivity(getApplicationContext(), 0, i,
    PendingIntent.FLAG_CANCEL_CURRENT);
AlarmManager am = (AlarmManager) this.getSystemService(Context.ALARM_SERVICE);
am.setInexactRepeating(AlarmManager.ELAPSED_REALTIME, 1, 100, pi);
```

Listing 7.1: Java source code for the DoS attack on device responsiveness.

The Java source code provided in Listing 7.1 will leave the GUI mostly unresponsive. The GUI may be able to switch between screens in the launcher, but it will not be able to launch any apps or perform any meaningful tasks. Therefore, the Settings app cannot be launched to uninstall the attacking app. The device will usually stay in a persistently unusable state until the device is rebooted or the attacking app is uninstalled via ADB. The logcat log will contain numerous instances of the activity being launched and the UID of the app that launched the activity on Android 5.0 builds. The UID can be used to identify the attacking app through the use of the Android API or ADB.

7.3.2 Killing External Apps

I discovered a method to kill other running apps on an Android device from a locally installed app. By the word *kill*, I mean that the process for an app will be terminated. This is accomplished by sending a large number of intents with specific intent flags to an app on the device. Receiving and handling incessant intents requires a large amount of RAM and causes the Low Memory Killer (LMK) [235] mechanism to start killing running processes to free up RAM. This is a blunt approach that generally kills many other processes on the device in addition to the targeted app. The approach to kill another running app is to first use the `android.app.AlarmManager.setRepeating(...)` API call to launch a `PendingIntent` every 10 milliseconds. The value of 10 milliseconds may need to be increased for faster Android devices. The processing of these intents will require a large amount of RAM. When the device crosses a threshold for the minimum amount of free RAM available, this activates LMK which will start killing processes based on their priority groups [236]. Processes with the lowest priority are killed first, then those with the next lowest priority, and so on, until sufficient memory is available for the system. The following are the main priority groups, in order of importance:

1. **Foreground**. This is a process that the user is currently interacting with through some foreground component. It has the highest priority, and therefore is killed last.

2. **Visible**. A process moves to the visible group if it does not have any foreground components, but is still running. For example, calling the `onPause()` method of an activity moves the process from the foreground to the visible group.

3. **Service**. A process that is running a service with no foreground components, e.g., a background music player.

4. **Background**. Here, the process `onStop()` method has been called. The process' foreground components are not visible to the user.

5. **Empty**. A cached terminated process. LMK eliminates empty processes first.

To narrow the attack down on a single app, I leverage the priority groups by forcing the victim process to be moved to the *Background* group if it is in the foreground. I achieve this by launching a foreground activity in front of the victim app, which obscures all foreground components of the victim process, resulting in the system moving it from the *Foreground* (or *Visible*) to the *Background* group. Being in the *Background* group, the process now will be killed by the LMK before any other visible app on the system. To monitor when the process gets killed, I run the process status (`ps`) command in a loop from a foreground service until the target process no longer exists in the process status output. Once the target process is terminated, the attacking app ceases the sending of the Intents using the `android.app.AlarmManager.cancel(PendingIntent)` API method call and breaks out of the loop. When LMK kills processes, there is no system message alerting the user that the targeted app has been terminated. In addition, the activities of the killed process are not recreated. The killed services will be scheduled to restart as long as they do not return the `android.app.Service.START_NO_STICKY` constant in their `onStartCommand` method. Even if the service returns `START_STICKY` or `START_REDELIVER_INTENT`, it is common for the service to be restarted after some delay due to the low memory condition that activated LMK. Therefore, the attack can take place at certain intervals to ensure that the service is killed repeatedly.

7.3.3 System Crash DoS Attack

The same mechanism, sending intents rapidly, that is used to make the device unresponsive and kill external apps can be made more aggressive to overload the Android OS and force the system to crash and reboot. Sending a large amount of intents rapidly generally leads to an uncaught exception in one of the service threads in the `system_server` process. Once this occurs, all the services residing in the `system_server` process start to die. Once the `system_server` process itself terminates, `zygote` will terminate and the system will reboot. Listing 7.2 shows the source code to reboot an Android device. Additional information on the `system_server` process is provided in Section 2.1.3.

The `system_server` process can be terminated, causing the system reboot, in a number of different ways. Prior to the device rebooting, uncaught exceptions tend to occur in the `com.android.server.am.ActivityManagerService` class since it is responsible for starting activities. A fatal runtime exception can occur when the `ActivityManagerService` fails

```java
Intent i = new Intent(this, DisplayText.class);
i.setFlags(Intent.FLAG_ACTIVITY_MULTIPLE_TASK | Intent.FLAG_ACTIVITY_NEW_TASK);
PendingIntent pi = PendingIntent.getActivity(getApplicationContext(), 0, i,
    PendingIntent.FLAG_CANCEL_CURRENT);
AlarmManager am = (AlarmManager) this.getSystemService(Context.ALARM_SERVICE);
am.setInexactRepeating(AlarmManager.ELAPSED_REALTIME, 1, 1, pi);
TaskStackBuilder tsb = TaskStackBuilder.create(this);
for (int a = 0; a < 400; a++)
  tsb.addNextIntent(i);
tsb.startActivities();
```

Listing 7.2: Java source code for the system crash DoS attack

to generate a pair of `android.view.InputChannel` objects since the OS runs out of file descriptors for available files, sockets, and pipes due to limits on open file descriptors imposed by the kernel. A fatal exception can also occur when an object of type `android.view.DisplayEventReceiver` fails to initialize. In addition, the `system_server` process can be killed by the `watchdog` daemon process due to unresponsiveness or a deadlock in the `com.android.server.am.ActivityManagerService` class. An out of memory error can occur in the `com.android.server.AlarmManagerService` class when it fails to allocate a contiguous buffer for additional alarms. Using `AlarmManager` to launch an Intent every millisecond requires that it create an alarm for each of these events.

Table 7.1: System crash DoS Attack on various Android devices and versions.

Device	OS Version	Build Number	Sec. to reboot
Nexus 9	5.0.1	LRX22C	3.4
Samsung Galaxy S4	4.4.4	KTU84P.I337UCUFNJ4	12.2
Samsung Galaxy S6 Edge	5.0.2	LRX22G.G925AUCU1AOE2	10.9
Nexus 5	4.4.2	KOT49H	9.4
Motorola Moto 360	5.0.2	LWX49L	74.1
Fire TV	4.2.2	51.1.5.3_user_515040320	10.5
Mini-PC	4.0.4	RKM-emg4.0.4IMM76D	9.8
Emulator (Nexus 5)	5.0.2	LSY66D-1797986	8.3

This attack works on various Android platforms and versions. Table 7.1 shows the time to reboot different Android devices using this attack. According to the core manifest file for Android 5.0 (`/frameworks/base/core/res/AndroidManifest.xml`) [237], the `android.permission.REBOOT` permission has a value of `signature|system` for the `android:protectionLevel` attribute [69]. Being granted this permission on Android 5.0 requires that an app reside on the `system` partition (i.e., a pre-installed app) or be signed with the device platform key. Therefore, a third-party app should not be able to directly obtain this permission. An zero-permission third-party app by using this DoS attack can regain the functional equivalent of the `REBOOT` permission.

7.3.4 Most Common Underlying Cause for the System Crash

Various potential causes of the system crash were mentioned in Section 7.3.3. Figure 7.1 provides the most common cause of the system crash I witnessed during testing. During the DoS attack, the Intents sent by the attacking app must have the `FLAG_ACTIVITY_MULTIPLE_TASK` and `FLAG_ACTIVITY_NEW_TASK` flags set, so that a new starting window with a new task stack will be required for each activity. In this subsection, the classes that end with "`Service`" are contained within the `system_server` process. The `com.android.server.wm.WindowManagerService` class [238] creates a window for the activity and each window requires a pair of `android.view.InputChannel` objects to be created so that the input events from the input device files can be delivered to the activity window. Third-party apps cannot directly read from the input device files which are contained in the `/dev/input` directory, but `system_server` has permission to read from them since it belongs to the `input` group. Therefore, `WindowManagerService` creates a pair of sockets using the `socketpair()` system call, registers the input channel with the window via the `com.android.server.input.InputManagerService` class, and transfers the output channel to the app. This allows the app to consume and process input events from the user via `system_server`.

A socket pair requires a file descriptor for each end of the socket pair. Each created activity will initially make `system_server` use two file descriptors. It will then transfer one socket to the attacking app, although during the attack `system_server` is processing a deluge

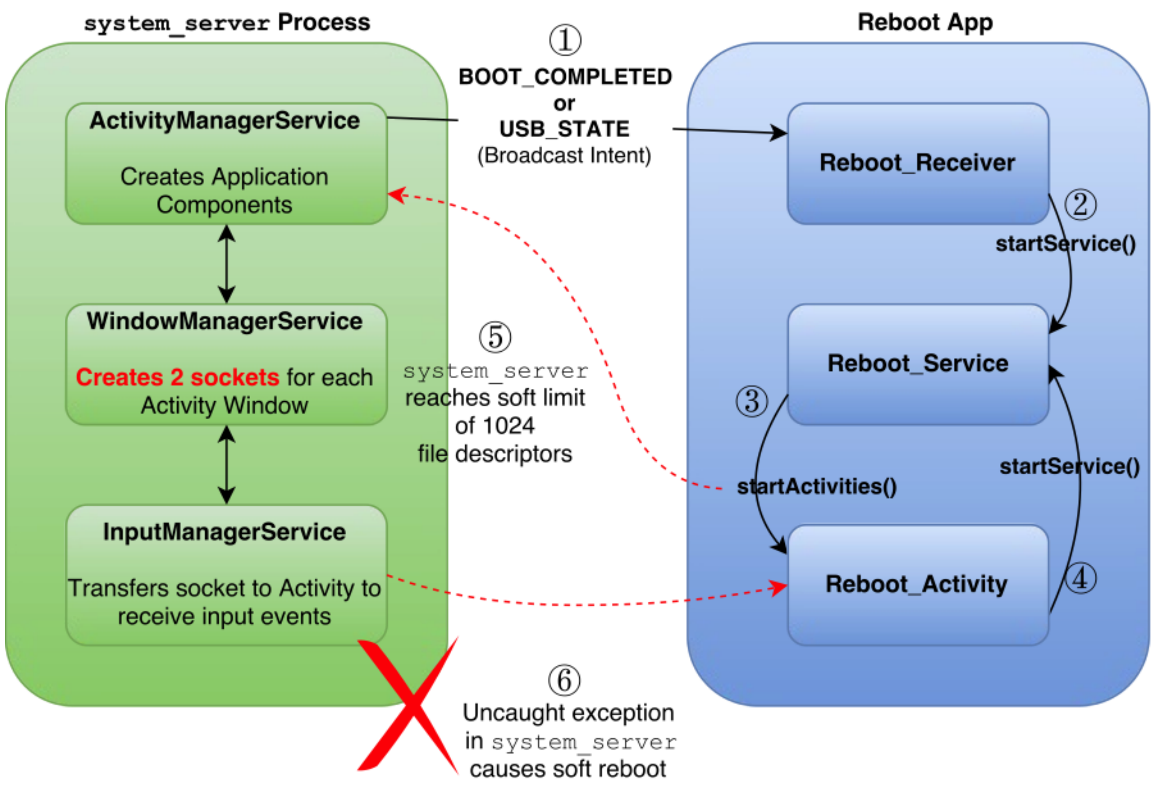

system_server Process
ActivityManagerService
Creates Application Components
WindowManagerService
Creates 2 sockets for each Activity Window
InputManagerService
Transfers socket to Activity to receive input events
①
BOOT_COMPLETED or USB_STATE (Broadcast Intent)
Reboot App
Reboot_Receiver
② startService()
Reboot_Service
③ startActivities()
④ startService()
Reboot_Activity
⑤ system_server reaches soft limit of 1024 file descriptors
⑥ Uncaught exception in system_server causes soft reboot

Figure 7.1: Interaction between a third-party app and the Android OS to cause a system crash.

of Intents and does not get a chance to transfer the socket. This results in `system_server` using two file descriptors per activity created which makes `system_server` get closer to approaching the soft limit of 1,024 per-process file descriptors set by the kernel.[3] Once the soft limit is reached, `system_server` cannot open or create any new files, pipes, or sockets, and `WindowManagerService` will fail to create the starting window for each activity.

The attacking app will encounter an uncaught exception once its activities cannot be created. The attacking app uses an `android.view.InputChannel` object received from the `WindowManagerService` as a parameter to the `android.view.InputEventReceiver` constructor. The `InputEventReceiver` object is used to queue the received user events so that they can

[3]This was the limit on Android 5.0 and has since increased in subsequent Android versions.

be stored while waiting to be consumed by the app. The `InputChannel` object that the app received will be `null`. So an exception will be thrown by the `InputEventReceiver.nativeInit()` native method in the attacking app which goes uncaught and causes it to terminate.

When the attacking app crashes, `ActivityManagerService` tries to display an `android.app.Dialog` object indicating that the attacking app has crashed. A socket will be required to deliver the user input to the window of the `Dialog` system message. `system_server` will not be able to create the socket, and an uncaught exception occurs. The `zygote` daemon process contains pre-loaded classes and resources and forks itself to create other apps quickly. `zygote` [239] starts `system_server` with the `--runtime-args` flag which provides the threads of `system_server` with an `UncaughtExceptionHandler` interface object of the type `com.android.internal.os.RuntimeInit.UncaughtHandler` [240]. It receives uncaught exceptions occurring within the threads of `system_server`. It only has one method and all of its code is within `try-catch-finally` blocks. The `finally` block calls the `android.os.Process.killProcess(int)` API call with an integer parameter that is the result of the `Process.myPid()` API call. Since the thread that has the uncaught exception occurs within `system_server`, this results in `system_server` both sending and receiving the `SIGKILL` signal, which results in its termination.

`zygote` is the parent process of `system_server`, so it will receive a `SIGCHLD` signal when `system_server` terminates. For each `SIGCHLD` signal that `zygote` receives, it will specifically check if the terminated child process is `system_server`. If `system_server` terminates, then `zygote` will send the `SIGKILL` signal to itself [241] which results in a system crash. The `init` process will then restart `zygote` since it is declared as a service in the `init.rc` file [242]. `zygote` will then restart `system_server`.

7.4 System Crash DoS Attack Evaluation

The local system crash DoS attack can be performed repeatedly to persistently deny the user meaningful usage of their Android device. When the DoS attack recurs each time the system boots, I refer to it as the system crash cycle DoS attack. I tested the system crash cycle DoS attack on various Android devices. Some of the embedded Android platforms tend not have safe mode and some do not have easy access to recovery mode, so I focused on these devices. *Safe mode* is an alternate mode of operation that prevents the execution of third-party apps. Safe mode is useful when a third-party app is interfering with normal usage of the device and allows the user to more easily uninstall apps. *Recovery mode* is a special mode of operation that the user can enter by pressing a sequence of buttons during the boot sequence. Recovery mode allows the user to wipe the cache partition, apply an update, and perform a factory reset which will remove all apps that the user has installed. All of the test devices were running a non-rooted stock version of the Android OS that came pre-installed on the device. All of these devices had ADB over USB disabled by default. Table 7.2 aggregates the results of the experimental data.

Table 7.2: Results summary for the tested devices.

Device	Build No.	Android Version	Vulnerable	Recoverable
Sony Bravia XBR-43X830C TV	LMY48E.S63	5.1.1	Yes	No
Moto 360 1st Gen. Smartwatch	LDZ22O	5.1.1	Yes	Yes≀
Amazon Fire TV Stick 1st Gen.	JDQ39	4.2.2	Yes	No†
Xiaomi Mi Mini TV Box	KOT49H	4.4.2	No	Yes
Nvidia Shield Android TV	LMY47D	5.1	Yes	Yes‡
Amazon Fire 7″ Tablet	LMY47O	5.1.1	Yes	Yes‡
Devices prior to Android 4.1	-	<4.1	Yes	Yes‡

≀ Recovering requires crafting a special USB cable and flashing firmware images.
† Recovering requires ADB over USB, which is disabled by default, to be enabled prior to the attack.
‡ Recovering requires a *full* factory reset in recovery mode or flashing firmware images.

7.4.1 Sony Bravia XBR-43X830C Android TV

The Sony Bravia XBR-43X830C Android TV is vulnerable to the system crash cycle DoS attack, and there is no known way to recover. During testing, the device was running Android 5.1.1 with a build fingerprint of `Sony/SVP4KDTV15_UC/SVP-DTV15:5.1.1/LMY48E.S63/2.473:user/release-keys`. The only way to perform a factory reset of the device is through the Settings app [243]. During the attack, the GUI becomes unresponsive to the infrared remote which prevents the user from reaching the Settings app to perform a factory reset. The device does have ADB over Wi-Fi, but this can be subverted since the attacking app programmatically disables Wi-Fi. This device does not have the ADB over USB capability. The device also does not have safe mode, recovery mode, or fastboot mode. Therefore, the user is unable to uninstall the attacking app, perform a factory reset, or flash firmware images. Booting to fastboot mode via ADB over Wi-Fi will show a black screen, but it will also *soft brick* the device as it will not boot properly after that.[4] The device comes pre-installed with Google Play so the user can download apps, and they can also be installed via ADB over Wi-Fi for the attacking app to reach the device.

7.4.2 Moto 360 1st Generation Smartwatch

The Moto 360 1st generation smartwatch is vulnerable to the system crash cycle DoS attack, although there is a way to recover via a modified USB cable that can be used to unlock the bootloader and flash firmware images to the device [244].[5] During testing, the device was running Android 5.1.1 with a build fingerprint of `motorola/metallica/minnow:5.1.1/LDZ22O/2006643:user/release-keys`. The device allows the user to directly install or uninstall apps using ADB over Bluetooth. When a user installs or uninstalls an app on an Android smartphone or tablet, which is paired with an Android Wear device, the accompanying Android Wear app, if present, will also be installed or uninstalled from the Android Wear device. The Moto 360 does not have a direct way to uninstall a particular app through its

[4]This occurred when I was testing the device, and I am unsure if it will occur on all versions of the device.

[5]The modified USB cable involves stripping the cable and creating an adapter. Instructions are provided here: `https://www.rootjunky.com/moto-360-adapter-usb-cable/`.

GUI. Based on my testing, the user has about 8 seconds to perform some action on the device before the GUI becomes unresponsive. The user can initiate a factory reset through the GUI, but it will not have enough time to complete and be successful before the device encounters a system crash due to the DoS attack. The Moto 360 lacks a standard USB interface, so only ADB over Bluetooth is available. The attack app will disable Bluetooth to prevent communication with paired devices.

7.4.3 Amazon Fire TV Stick 1st Generation

The Amazon Fire TV Stick 1st generation is vulnerable to the system crash cycle DoS attack and can leave the device in an unusable state if ADB over USB is not enabled prior to the attack. The device runs Amazon Fire OS 3.0, which is a modified version of Android 4.2.2. The device I tested had a build fingerprint of: `BRCM/montoya:4.2.2/JDQ39/54.1.2.2_user_122066120:user/release-keys`. If ADB over USB is enabled prior the attack, the user can list the installed third-party apps and uninstall them as the device is booting. The malicious app programmatically disables both Bluetooth and Wi-Fi. This renders any paired remotes ineffective and precludes ADB over Wi-Fi. There are no hardware buttons to force the device to boot into recovery mode or bootloader mode from a powered-off or booting state. This will effectively preclude the user from removing the app if ADB over USB is not enabled prior to the attack, making the device effectively useless.

7.4.4 Xiaomi Mi TV Box Mini

The Xiaomi Mi TV Box Mini is not vulnerable to the system crash cycle DoS attack. The device I tested was running Android 4.4.2 and had a build fingerprint of `Xiaomi/forrestgump/forrestgump:4.4.2/KOT49H/566:user/release-keys`. Apps can be installed through the browser or a network-connected device. Communication with the device is performed via a Bluetooth remote, and it contains no USB interfaces. The device does not send the `BOOT_COMPLETED` broadcast Intent to third-party apps, so the app is unable to system crash the device after the devices completes the boot process.

7.4.5 Amazon Fire 7″ Tablet

The Amazon Fire 7″ Tablet is vulnerable to the system crash cycle DoS attack if ADB over USB is not enabled prior to the attack. If ADB over USB is not enabled prior to the attack, then the user must perform a factory reset of the device or flash firmware images to the device to recover it. The device I tested was running Amazon Fire OS 5.0, which is a modified version of Android 5.1.1 and had a build fingerprint of `Amazon/full_ford/ford:5.1.1/LMY47O/37.5.4.1_user_541112720:user/release-keys`. The attacking app receives the `android.hardware.usb.action.USB_STATE` broadcast Intent because it is sent prior to the `BOOT_COMPLETED` broadcast Intent and does not require any permissions to be able to receive it. This broadcast Intent is received by the attacking app prior to the Amazon launcher being displayed, so the user is precluded from uninstalling the app via the GUI. The device provides easy access to recovery mode from a powered-off state by holding the volume down and power buttons during boot. In recovery mode, the user can perform a factory reset of the device.

7.4.6 Nvidia Shield Android TV

The Nvidia Shield Android TV device is vulnerable to the system crash cycle DoS attack if ADB over USB is not enabled prior to the attack. The device I tested was running Android 5.1.1 and had a build fingerprint of `NVIDIA/foster_e/foster:5.1/LMY47D/35739_609.6420:user/release-keys`. The device does not have safe mode and ADB over Wi-Fi can be programmatically disabled. The only way to recover is by performing a factory reset or flashing firmware images to the device. There is a method to perform a factory reset that is not published on Nvidia's website [245]. Alternatively, the user can access the fastboot menu and flash firmware images.

7.4.7 General Android mini PC Devices

Android mini PC devices are *somewhat* vulnerable to the system crash cycle DoS attack since they generally lack safe mode. Some devices allow the user to push a button during boot to enter recovery mode. In addition, some devices can utilize the SD card to flash firmware images to the device. Whether the attack is effective or not depends on the specific device and the mechanisms for recovery it provides.

7.4.8 Android Devices Prior to Android 4.1

Safe mode was introduced in Android 4.1. Prior to Android 4.1, the user was forced to perform a factory reset via recovery mode or flash firmware images to remove an app that persistently causes a system crash on the device. According to the *Android Dashboard* webpage, devices running a version of Android prior to Android 4.1 made up 5.0% of all Android devices as of March 7, 2016 [246].[6]

7.5 Defending Against the Attack

7.5.1 Attack Mitigation App

I developed an anti-reboot app (source code available at [55]) that passively monitors Intents sent by third-party apps on the system, and disables or uninstalls apps that attempt to flood the system with Intents. The anti-reboot app observes Intents by reading the system log buffer using `logcat` on the device, and parsing the log messages searching for Intents. The app filters log messages using relevant log tags to reduce the amount of log messages it processes. For every observed Intent, the sender's package name is logged and its total outbound Intents count n is incremented. The anti-reboot app only considers Intents that create new tasks, i.e., the `FLAG_ACTIVITY_NEW_TASK` and `FLAG_ACTIVITY_MULTIPLE_TASK` Intent flags are set. It also ignores Intents sent by system apps by filtering on the process UID since system apps are assigned UIDs that are less than 10,000. Anti-reboot uses a one-level decay, where the Intent count n is decreased by a constant c every second. This is intended to simulate the time a user would interact with a new activity before dismissing it. In other words, the value of c controls the tolerable persistence level of an offending app. For a period

[6]As an update, as of July 2019, only 0.6% of Android devices are running a version of Android prior to 4.1.

of t seconds, this results in an effective Intent count $n' = n - ct$, and an effective sending rate $\rho = \frac{n'}{t} = \frac{n}{t} - c$. Finally, a monitored app is disabled or uninstalled if its corresponding n' exceeds a preset threshold (θ), which indicates that the monitored app has more than θ *active* task stacks.

Parameters Selection

There are two parameters that control the detection performance of the anti-reboot app: the Intent decay c, and the cutoff threshold θ at which an app is disabled or uninstalled. The value of c controls the tolerance level of the defense to apps that persistently send multiple Intents over time. While benign apps may create new tasks, such behavior typically lasts for only a very short period of time (i.e., short bursts) compared to attacking apps which need to be highly persistent in order to adversely affect the system. Therefore, the higher the value of c, the higher the tolerance and the more likely an attack may go undetected. A reasonable value of c would mimic the time it takes a user to click the recent tasks button and dismiss an activity off the screen, which takes about 2 seconds. Therefore, I set c to one Intent every 2 seconds, i.e., $c = 0.5$.

Avoiding False Positives The cutoff threshold θ controls when an attack is detected, based on the number of active task stacks s the attack app has created. Note that $s \leq n'$, since each task stack would hold at least one activity. Since an attack is detected if $n' \geq \theta$, setting θ to a very small value may result in faster detection at the expense of false positives (i.e., false alarms). Conversely, a very large value of θ results in lower detection rate. I can pick a reasonable value of θ by estimating an upper bound on n' for *benign* apps. Studies (e.g., [42, 247]) have shown that the total number of activities declared in an app's manifest is less than 110 for the top 30 apps in the market, with a total of 60 foreground activities created on the device *per day* from the top 800 apps on the market. Therefore, I set $\theta = 200$, which allows 200 task stacks to be created at any point in time. This is more than three times the number (60) of task stacks that would be created, in the worst case, by benign apps if I assume each of the benign 60 activities was created in a new task stack and was never terminated.

In versions of Android earlier than 6.0, where `AlarmManager` does not have a minimum recurrence interval of 60 seconds, attacking apps can flood the system with activities using pending Intents with short repeat intervals. To mitigate this, and in addition to observing Intents, the anti-reboot app monitors the count and repeat interval of active pending Intents being processed by the `AlarmManager`. It periodically retrieves a snapshot of the `AlarmManager` state by executing the `dumpsys alarm` command. Note that excessively running `dumpsys` can harm the overall system performance, while very long query periods can cause the attacks to go undetected. I empirically found that executing `dumpsys` every 500 milliseconds is suitable on the test devices used in this study. For each pending Intent record, the anti-reboot app extracts the package name of the source app and the repeat interval. If the interval is less than a predefined threshold (set to 60 seconds as in Android 6.0), or the number of active pending Intents of a source app is more than θ, the source app is flagged and is either disabled or uninstalled.

Detection Results

The anti-reboot app detected the system crash DoS attack and identified the source of the attack 100% of the time during testing, even when the attack was in its most aggressive form. In many cases, I observed that the device reboots before the anti-reboot app gets a chance to disable or uninstall the attacking app. This is mainly due to the fact that the attacking app can request to start up to $5,500$ new tasks in a single transaction using `Service.startActivities(Intent[])` API call. This quickly depletes the file descriptors of `system_server` which inhibits its capabilities and renders `system_server` unresponsive to any requests to disable or uninstall the offending app. To mitigate this, the anti-reboot app records the package name of offending apps along with a time stamp of when the attack was detected in persistent memory. It then checks when the system last encountered a system crash, and if an offending app was detected within a 60 second period before the system crash, it disables the offending app after the system crash and informs the user. In addition, I confirm a system crash by checking to see if the Process ID of `system_server` has changed, which occurs during a system crash. The user can re-enable disabled apps through the GUI of the anti-reboot app.

I emphasize that it is not possible to rate-limit the Intents sent by processes, without changes to the OS itself. Even then, a balance has to be struck between usability and security. If the system sets overly strict limits on the sending rate of Intents, apps may become unresponsive or sluggish, resulting in an overall degradation of the system performance and user experience. In addition, it is not straightforward to implement rate-limiting in a system that is heavily event-driven such as Android. If the system decides to silently drop Intents, apps are likely to malfunction as a result of lost Intents. Notifying apps that they are exceeding the rate-limit would require a back channel from `system_server` to the app, besides requiring the app to anticipate and handle the notification, which further complicates the design of both the OS and the apps. I am unaware if this attack has been used in *the wild*. After informing Amazon of the DoS attack, they created a detection mechanism for it in the Amazon AppStore. Google did not respond to my question whether or not the attack app would make it through their vetting process to be available on Google Play.

Performance Evaluation

I tested the overhead introduced by the anti-reboot defense app by using the following two benchmarks: AnTuTu Benchmark v6.0.1 and BenchmarkPI v1.1. AnTuTu Benchmark provides an aggregate score that combines both multitasking, user experience, CPU and memory speeds, and 3D rendering performance. BenchmarkPI is a CPU time benchmark that computes π to the n^{th} digit. I tested the defense app on the following devices: Nexus 5 running AOSP Android 6.0.1, Nvidia Shield Android TV running Android 5.1.1, Amazon Fire TV 1^{st} generation running Android 4.2.2, and Amazon Fire $7''$ tablet running Android 5.1.1. Under each scenario, I performed 20 runs and took the average of the resulting benchmark scores. I report the overhead as the percentage degradation in the aggregated

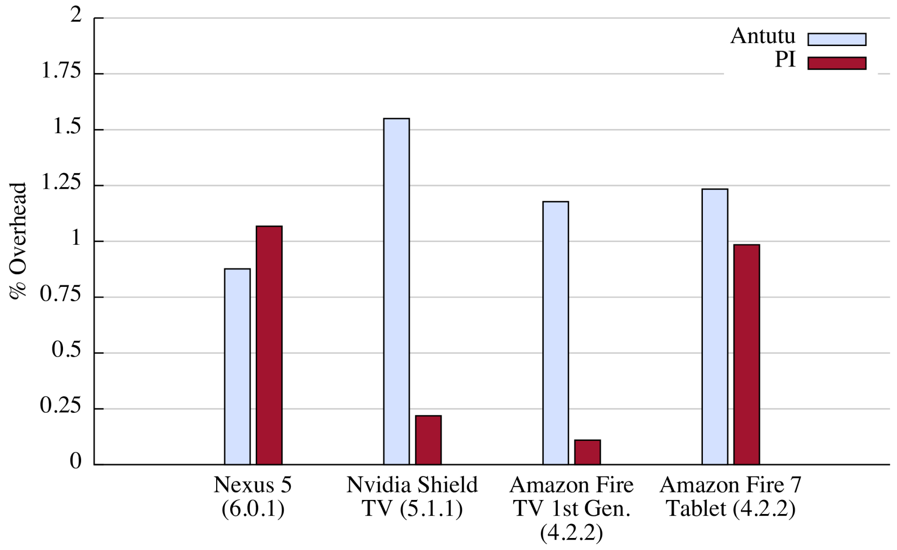


Figure 7.2: Performance overhead based on AnTuTu Benchmark and BenchmarkPI scores.

average of the benchmark scores.

Figure 7.2 shows the overhead in the benchmark scores of AnTuTu Benchmark and BenchmarkPI. The overhead ranged from 0.8% to 1.51% for AnTuTu Benchmark and 0.14% to 1.15% for BenchmarkPI. The overhead from the defense app is mainly due to the threads it spawns to continuously monitor the logcat log and process the output of the `dumpsys alarm` command to record Intent usage and attribute them to the app that sent them. Overall, the defense app introduced a small amount of overhead (less than 1.6%) which I believe is acceptable for the service it provides.

7.5.2 Proposed Platform Defenses

I suggest changes be made to the `ActivityManagerService` class in the Android framework to prevent a single app from starting an arbitrarily large amount of activities. Currently, the amount of Intents that can be sent to be processed by `ActivityManagerService` is only limited by the Android Binder transaction buffer size. On Android 6, this enables an app to send around 5,500 Intents to be processed by `ActivityManagerService` in a single transaction using the `Service.startActivities(Intent[])` API call. I believe that a limit of less than 400 concurrent activities should be imposed on each app to preclude it from causing a system crash. Alternatively, a proper rate for rate-limiting of Intents can be established from empirical analysis of Intent usage among third-party apps. I recommend that once the user selects to perform a factory reset of an Android Wear device, that all third-party apps should be terminated so they cannot attempt to interfere with the factory reset process. In addition, introducing some delay before sending the `BOOT_COMPLETED` broadcast Intent and similar Intents to third-party apps can provide the user additional time to perform a factory reset through the Settings app without having to race against a misbehaving app.

7.6 Summary

By introducing a novel system crash cycle DoS attack, I show that installing a third-party app, even with a limited set of permissions, can render certain Android devices unusable. In other cases, the user needed to perform a factory reset or flash firmware images to recover the victim device. Furthermore, I provide a detailed explanation as to the underlying cause of the system crash that occurs in the Android framework. To support my claims, I reference the actual Android 6 source code and describe the mechanics of the attack strategy. To mitigate the attack, I leverage the existing Android framework to suggest changes that would either significantly reduce or eliminate the effects of the attacks. As a proof-of-concept, I implemented an open-source Android app that provides concrete countermeasures to prevent the attack and can be utilized by device manufacturers without modifying the device or the Android framework. As a final note, to ensure that my research is not misused, I informed Google and all of the affected device manufacturers so that Android devices can be made more secure.

Chapter 8: Large Scale Testing for Android Inter-App Vulnerabilities

This chapter expands upon the previous chapter in continuing the examination of the security and availability implications involved in the usage of the Intents. I developed an open-source system, named *Daze*, that identifies common developer errors in stand-alone apps and entire Android devices. The workflow for *Daze* is explained in Section 8.2. I investigate the prevalence of these errors on 32 different devices in Section 8.3. As part of device testing, I tested all the available AOSP builds of two different devices: Nexus 5 and Nexus Player. Section 8.4 describes a longitudinal study I performed to obtain insights into how long inter-app vulnerabilities tend to exist in different versions of an app with regard to the time duration in which it existed and the number of versions in which it persisted. Based on AOSP testing, I discovered vulnerable app components in AOSP base code that have been propagated to Android vendors. Section 8.5 discusses the root causes of the vulnerabilities *Daze* discovered in AOSP code. Lastly, Section 8.5.3 describes a generic system crash DoS that works on all Android devices.

8.1 Errors of Omission

Once an Intent is sent to an app component (see Section 2.2), various errors of omission can occur. If the destination app component is declared but not actually implemented, the receiving app will crash due to an uncaught `ClassNotFoundException`. An undefined app component is a component that has a valid entry in the app's manifest file, but the component itself is not implemented in the app's code. If the app component receiving an Intent is implemented in code, there are various exceptions that can be encountered by further errors of omission resulting in an uncaught exception and process termination if appropriate error handling is not present. Even before an app processes the received Intent, an app component may encounter an `UnsatisfiedLinkError` if a required native library is missing. When an Intent is received by an app component, one or more callback methods specific to the app component type are executed. During the processing of a received Intent, the most common error of omission manifests as a `NullPointerException` due to accessing null objects. Certain apps do not gracefully handle the absence of expected data, resulting in an unexpected crash. Most of the exceptions can be addressed through proper input validation and thoughtful exception handling at runtime.

```java
public void onReceive(Context context, Intent intent) {
    String action = intent.getAction();
    Bundle bundle = intent.getExtras();
    ...
    if (action.equals("com.sec.android.intent.action.APP_HQM_SEND_REQ")) {
        int type = bundle.getInt("Type", 0);
        // NullPointerException happens above when the ‘bundle’ object is
           null
        ...
    }
}
```

Listing 8.1: Recreated source code of a vulnerable broadcast receiver.

Internally, most embedded data within an Intent is stored in a `Bundle` object, which is a map data structure that allows the storage and retrieval of key-value pairs. A recipient app component may also extract the action string or embedded Uniform Resource Identifier (URI) of an Intent. When an app component with inadequate error handling and null-checking *assumes* these values *will not* be null, a `NullPointerException` can occur. The Android OS itself declares app components that are accessible to third-party apps.[1] An uncaught exception occurring in a component *within the Android OS* can lead to the crashing of critical system processes, which triggers the OS to reboot in an attempt to recover. I provide a motivating example in Listing 8.1 that shows a source code snippet from a broadcast receiver app component within the Android OS that will encounter a system crash if the received Intent does not carry a `Bundle` object. The snippet was recreated from disassembled

[1]The app components declared by the Android framework can be viewed in [69].

bytecode from a Samsung Galaxy S8+ running Android 8.0 with a build number of `R16NW.G955USQS5CRL1`. Appendix D provides the corresponding stack trace produced when this system crash occurs.

In addition to inadequate error handling, certain app components will themselves throw a `RuntimeException` if an unexpected data item is not present in the Intent. Various other exceptions can occur, such as errors in handling the app lifecycle (`IllegalStateException`), forgetting to call a base method (`SuperNotCalledException`), and referencing classes that are not defined (`NoClassDefFoundError`). The full list of exceptions encountered in my experiments are provided in Tables 8.5 and 8.7.

8.2 *Daze* Overview

Figure 8.1 illustrates the workflow of *Daze*. I developed *Daze* to automatically determine if certain classes of concrete failures or unexpected behaviors exist in Android apps and the Android OS on a given device.[2] *Daze* tests all four types of app components and provides the user with a list of faulty processes, stack traces, discovered behaviors, and an exploit composed of a trace of events and API calls that can be replayed to trigger a discovered vulnerability.

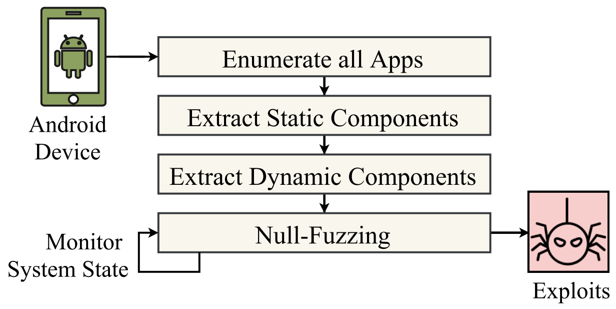


Figure 8.1: Workflow of *Daze*.

[2]*Daze* is open-source and available at: `https://github.com/Kryptowire/daze`.

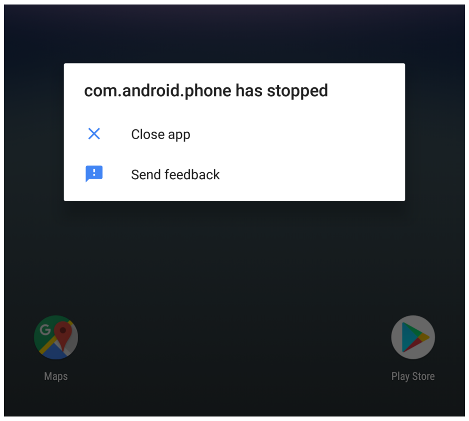

com.android.phone has stopped

Figure 8.2: Dialog box for an app crash, taking away the input focus from underlying GUI elements.

Furthermore, going beyond discovery, *Daze* automatically generates zero-permission exploits that give direct control over the victim device availability and usability. This can be exploited to craft ransomware via crypto-less attack vectors. Having the ability to crash an app may seem low-risk, but it can enable a malicious app to set itself as the gatekeeper to a vulnerable app, determining when and if a vulnerable app gets to execute. Even worse, persistently exploiting a known fault in an app enables a **controlled crash-loop** DoS attack on Android devices where the Android OS recurrently pops up the app crash dialog (see Figure 8.2) and restarts the crashing app in the background, and then the attacker crashes it again. This allows the attacker to control overall device usability since the recurring OS app crash dialog box takes the input focus away from other GUI elements, hindering the user from productively interacting with the device.

8.2.1 Identifying App Components

Identifying Statically Registered Components

Daze extracts statically registered components from an app by querying the OS package manager for all installed packages, then iterating through each package information looking for components declared with the activities, services, receivers and providers tags. The package manager fills in this information from the app's manifest file, which cannot be modified once the app has been installed. *Daze* ignores components that are not exported or that require a permission to access. The focus is for zero-permission reproducible test cases to crash apps or the system so only open components with no permission requirements are considered. In addition, users may be more willing to download an app with no permissions. Since *Daze* is open-source, it can be easily modified to request all available third-party permissions and test components that are protected with permissions declared by the Android OS.

Identifying Dynamically Registered Broadcast Receivers

Broadcast receivers are the only app components that can be both statically registered and dynamically registered. At runtime, an app can create and register a broadcast receiver to be eligible to receive one or more action strings. Dynamically registered broadcast receivers can be addressed only by using an action string. *Daze* enumerates dynamically registered broadcast receivers by executing the command `dumpsys activity broadcasts`. For security reasons, third-party apps are not allowed to obtain this list of dynamically registered broadcast receivers or read from the system-wide logcat log. To work around this limitation, *Daze* is granted the system development permissions using the ADB command: `adb pm grant <package> <permission>`.[3] The `READ_LOGS` and `DUMP` permissions can be externally granted only via ADB for development or testing. After these two development permissions are granted to *Daze*, it can obtain a listing of the active broadcast receivers, including those that are dynamically registered, and access the system-wide logcat log. *Daze* parses the logcat log to detect app crashes, native crashes, and system crashes.

8.2.2 Testing App Components

Daze identifies statically and dynamically registered components residing within all apps on a device and sends intents to all discovered components that are exported and not

[3]ADB is an Android SDK tool that allows a computer to interact with Android devices.

permission-protected. For all components except content providers, the system sends up to four intents. First, it sends an intent containing the minimum data required to be delivered to the target component — namely, the package name and class for an activity, broadcast receiver, or service, or the action string for dynamically registered broadcast receivers. If a crash is not encountered due to this intent, then *Daze* adds an empty `Bundle` object to the intent and sends it again. If an error is still not encountered for components that are not dynamically registered, it adds an empty action string and sends the intent again. Lastly, an Intent with a schemeless URI is sent. I chose to focus on the action string, Bundle, and URI since, based on my experience, these are commonly used data items in Intents. To cover all code sites reachable by an Intent, the system sends Intents with the `FLAG_ACTIVITY_SINGLE_TOP` flag to also force the delivery of the Intent to the `onNewIntent` method of activity components. *Daze* also calls the `stopService` method to trigger cleanup routines that may access the Intent that started the service.

Daze monitors the logcat log to record the system-wide effects of issued intents. Once it finishes sending Intents to all exported components, it examines the log file recorded for each sent Intent. To avoid side-effects and to isolate different runs, *Daze* separately replays each Intent that resulted in a fatal exception or a system crash to verify that it indeed triggers a failure condition to provide accurate attribution to individual Intents.

Testing Content Providers

Unlike other app components, content providers are not directly accessed via Intents. Any content provider has to implement a set of methods from the abstract `ContentProvider` class provided by the platform. Apps can read and write data to a content provider using a platform-managed content resolver that has the most safeguards with regard to handling null object references and invalid input. In addition, content providers are not exported by default and tend to be protected by permissions since they act as data repositories. Content providers are generally backed by an SQLite database and must implement the following operations: Delete, Insert, Query, and Update. *Daze* tests content providers by null-fuzzing all callable methods in their classes.

Testing content providers leads to some difficulty as a crash in a content provider causes any app connected to the crashing provider to be killed. Specifically, `ActivityManagerService` within the Android OS will terminate any process with an ongoing connection to a crashing content provider. *Daze* will stop testing a content provider after this occurs three (3) times and note that the content provider encounters a fatal error during testing.

8.2.3 Monitoring System State

Prior to the sending of each Intent, *Daze* enumerates all the files on external storage (i.e., SD card) to obtain a snapshot of the current state. After sending Intents to a component, it will again take a snapshot of the files and compare. If a file has been removed or added, the change will be detected and the file path and file size will be recorded. External storage can be read by any app that requests the `READ_EXTERNAL_STORAGE` permission, so sensitive data should not be written to it. At the end of the analysis, the changes are presented to the user, who can view the newly added files to examine their contents. This capability can detect the taking of screen snapshots and dumping of log files to external storage, which was observed during the testing of devices (see Section 8.3.3 for details). Screen snapshots are flagged since they are stored in a known set of directories on external storage with a known file extension.

Changes to device settings are also recorded before and after testing each component to determine if a privileged process modifies them during its execution. This is accomplished by querying system properties and the secure, global, and system settings.[4] The capability shows whether the tested component has made changes, such as enabling or disabling various communication capabilities. For example, the enabling and disabling of Wi-Fi is automatically detected by monitoring changes to the value of `wifi_on` key in global settings.[5] In addition, a component may extract an expected field from an incoming intent and write

[4]Secure settings are present only on devices running API level 17 and higher.

[5]Additional items in global settings are found in `https://developer.android.com/reference/android/provider/Settings.Global.html`.

its value to device settings. This may be observed when a value of null or an empty string is written to system settings instead of a concrete value.

Once an app encounters a fatal exception, the Android OS displays a dialog box to the user indicating that an app has crashed. Once this dialog box is present, no additional components can be launched within the crashed app until the dialog box is dismissed or until a crashed service in the app restarts. To be able to quickly launch additional components in a crashed process, *Daze* obtains the current window handle by using the `dumpsys` command and uses the `input` command via ADB to inject key events to programmatically dismiss the dialog box. This allows for the crashed process to be restarted when testing a different app component, without requiring user intervention.

8.3 Device Evaluation

I tested *Daze* on a representative set of 32 low-end to flagship Android devices from 21 vendors covering Android 4.4 to Android 8.0. In this section, I discuss the findings for the number of concrete process crashes, system crashes on specific Android devices, instances of privilege escalation, and data disclosure on the tested devices.[6] Seventy two percent (72%) of the tested devices contained at least one system crash vulnerability. I tested all of the pre-installed apps present on the 32 Android devices. Table 8.3 provides the total number of process and system crashes on the tested devices. *Daze* triggered 4,972 unique app crashes and 64 unique system crashes across all devices, taking about three hours on average to scan an entire device.

I attributed each of the vulnerabilities discovered in the tested devices to either vendor apps or AOSP apps (see Table 8.3; ratios are plotted in Figure 8.3).[7] I identified AOSP apps by recording the package names of the apps present in AOSP builds for smartphones, tablets, and Android TV. I attributed a fault to an AOSP app if it occurred within an app in the AOSP apps list. For attributing system crashes, I manually examined the stack trace

[6]All findings have been responsibly disclosed to Google and affected vendors prior to the publication of this dissertation.

[7]I made no distinction between GApps (i.e., Google apps) and AOSP apps in this study.

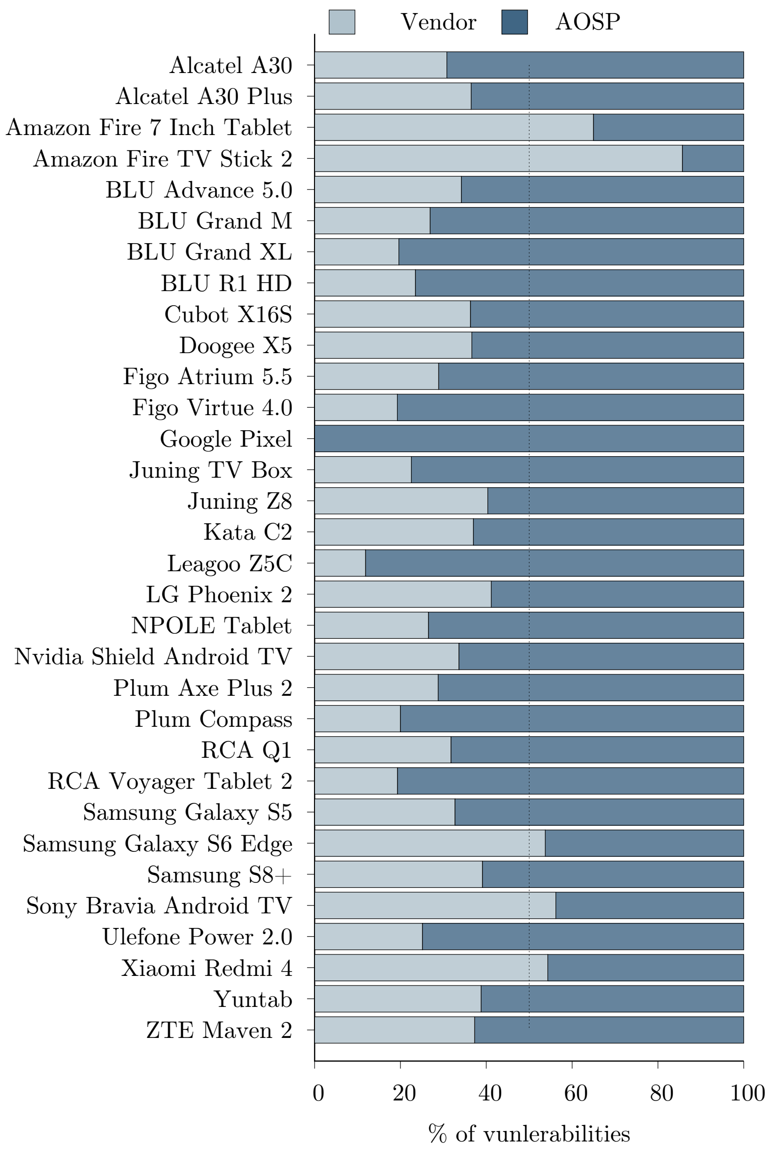


Figure 8.3: Attribution of device vulnerabilities to AOSP or vendor customization.

and checked whether it occurred in AOSP or vendor code by examining AOSP source code.

8.3.1 App Crashes

I discovered that more than 50% of fatal exceptions were present in AOSP apps. Amazon devices were an exception since Amazon maintains its own Android version, called Fire OS,

Table 8.1: Processes with the highest number of crashes during testing.

Process Name	**# of Crashes**
com.google.android.gms.ui	1011
com.android.settings	611
com.android.phone	572
com.google.android.setupwizard	228
com.android.contacts	111
com.google.android.gms	104
com.google.android.gms.persistent	94
com.android.mms	84
com.android.cts.priv.ctsshim	84
com.google.android.gm	81
com.android.bluetooth	67
android.process.media	66

which primarily uses Amazon's own apps instead of GApps [248]. For the Google Pixel device, I consider all apps on the device to be AOSP/GApps. Table 8.1 shows the top 12 crashed processes across all devices. These were either Google or AOSP processes, with Google `com.google.android.gms.ui` (a process within the Google Play Services app) topping the list with a total of 1,011 crashes. A particularly important process, `com.android.phone`, is surprisingly vulnerable to being crashed by an external app with 572 crashes. A crash of the `com.android.phone` process can deny telephony functionalities, including the ability to receive or make calls, which can have dire consequences in times of emergency. There were, on average, 17.87 vulnerable components on each device that crashed the `com.android.phone` process. These ranged from the Google Pixel running Android 8.0 with two vulnerable components to the Yuntab running Android 4.4.2 with a total of 54 distinct components or broadcast actions to crash the `com.android.phone` process. The `com.android.bluetooth` process has the 11th most crashes, with 67. Launching continual DoS attacks using intents can restrict the user's access to wireless communication capabilities on the device, or hinder the usability of the device by causing an app crash-loop.

8.3.2 System Crashes

For each system crash that *Daze* uncovered, I investigated the cause to attribute it to either AOSP code or vendor code. All of the devices were running the most recent Android versions available to them at the time of testing. The majority of system crashes were caused by vendor modifications (62.5%). If the system crashes from Samsung were excluded, AOSP code would be responsible for 85% of the system crashes. Vendor modification was responsible for system crashes in three component types: broadcast receiver (34 crashes), service (1 crash), and content provider (1 crash). AOSP code was responsible for crashes in two component types: activity (22 crashes) and broadcast receiver (6 crashes). This breakdown is provided in Table 8.2. All of the reported system crashes were triggered by a zero-permission third-party app.

Table 8.2: Discovered components by type that will trigger a system crash DoS vulnerability in common Android devices.

Type	System Crash Instances	Cause
Receiver	34	Vendor
Activity	22	AOSP
Receiver	6	AOSP
Service	1	Vendor
Provider	1	Vendor

Implementation errors in AOSP code are particularly based on their ubiquitous nature due to code inheritance. I tested all 27 factory builds for Nexus 5 and discovered two app components that did not properly perform null-checking before operating on data. I discovered that all AOSP 5.1 to 6.0.1 builds contain a vulnerable activity, named `com.android.internal.app.IntentForwarderActivity`, in the `android` package that crashes the system when the Intent contains a null action string. In addition, I discovered that all Nexus 5 AOSP 6.0.1 builds contain a vulnerable broadcast receiver that will crash the system when receiving a broadcast intent with an action of `android.net.conn.CONNECTIVITY_CHANGE_SUPL` and an empty body. These two components are explained in more detail in Section 8.5.

Table 8.3: Unique app and system crashes per device. 📱 indicates the vulnerability was introduced by the vendor. 🤖 indicates it was introduced by AOSP.

Device	OS Version	App Crashes		Sys Crashes	
		📱	🤖	📱	🤖
Alcatel A30	7.0	66	146	0	0
Alcatel A30 Plus	7.0	77	135	0	0
Amazon Fire 7.0 Inch Tablet	5.1.1	66	34	0	1
Amazon Fire TV Stick 2	5.1	18	2	0	1
BLU Advance 5.0	5.1	41	76	0	1
BLU Grand M	6.0	36	94	0	1
BLU Grand XL	7.0	34	135	0	0
BLU R1 HD	6.0	43	139	1	1
Cubot X16S	6.0	66	113	0	1
Doogee X5	6.0	59	102	0	1
Figo Atrium 5.5	5.1	37	90	1	1
Figo Virtue 4.0	6.0	33	133	0	1
Google Pixel	8.0	0	137	0	0
Juning TV Box	5.1.1	22	78	2	1
Juning Z8	5.1.1	69	102	0	1
Kata C2	6.0	75	125	0	1
Leagoo Z5C	6.0	13	88	0	1
LG Phoenix 2	6.0	112	160	1	0
NPOLE Tablet	5.1.1	14	35	0	1
Nvidia Shield Android TV	7.0	39	74	0	1
Plum Axe Plus 2	6.0	51	125	0	1
Plum Compass	6.0	21	84	0	1
RCA Q1	6.0	55	117	0	1
RCA Voyager Tablet 2	5.0	28	121	1	0
Samsung Galaxy S5	6.0.1	57	145	14	1
Samsung Galaxy S6 Edge	6.0.1	98	98	17	1
Samsung S8+	7.0	45	70	0	0
Sony Bravia Android TV	6.0.1	86	65	0	2
Ulefone Power 2.0	7.0	43	128	0	0
Xiaomi Redmi 4	6.0.1	100	82	0	2
Yuntab	4.4.2	92	145	0	0
ZTE Maven 2	6.0.1	74	124	1	2
TOTAL		1,670	3,302	26	38

These vulnerabilities have been propagated to the vendors' implementations of Android as shown in Table 8.3.

Of all the devices I tested, Samsung contained the most exposed interfaces that can be used to make the device encounter a failure state via a system crash. I initially reported to Samsung that the Samsung Galaxy S6 Edge (AT&T) running Android 6.0.1 with a build number of `MMB29K.G925AUCS5DPK5` contained 18 different vulnerable components. I received 6 Samsung Vulnerabilities and Exposures (SVEs) for vulnerabilities discovered using *Daze*. I sent another disclosure to Samsung for the Samsung Galaxy S8+ running Android 7.0 with a build number of `NRD90M.G955USQU1AQD9` containing 7 vulnerable broadcast receivers. Samsung fixed all the vulnerable components in their current Android devices. This is particularly relevant since Samsung Android devices were also disclosing user data during a system crash as discussed in Section 8.3.4 and also held the greatest global market share of smartphones in Q3 2017 [249]. Interestingly, despite null-checks present in content providers, *Daze* crashed Juning TV running Android 5.1.1 with a null pointer exception in the `HdmiControlService$SettingsObserver` class of the `com.android.server.hdmi` package.

I discovered that various components on the tested devices could be used for privilege escalation in the form of a confused deputy attack [250]. This occurs when a process uses the exposed interface of a privileged process to perform an action on its behalf. The confused deputy attack is well-known on Android and research has been conducted to mitigate its impact [148, 251, 252]. My findings show that this issue still persists in Android on a range of devices.

The Android OS will export an app component in certain circumstances even if this is not what the developer has intended. Correspondingly, even if an app component does not have the `android:exported` attribute set to true, the OS will *still export the component* if it contains at least one `Intent-filter`. An `intent-filter` is used by an app component to register for action(s) that it expects to receive. An exported component will be accessible to all external apps if the component does not use the `android:permission` attribute in its manifest file. The `android:permission` attribute creates an access requirement that only allows processes with the specified permission to interact with the app component. Android app developers have a tendency to unintentionally export app components, which makes them accessible to third-party apps [158]. Exported components can lead to privilege escalation and local DoS attacks [148, 253].

8.3.3 Privilege Escalation

Of particular concern are devices that can be "factory reset" simply by sending an Intent, a capability that is supposed to be reserved for system apps and enabled Mobile Device Management (MDM) apps. A factory reset will wipe all user data. The most severe privilege escalation I noticed occurred in the MXQ TV Box, which has an exported broadcast receiver named `SystemRestoreReceiver` that when called will "brick" the device, making it nonfunctional even after a factory reset. This component modifies the system partition so that the device will not boot properly. Table 8.4 displays my findings showing the device, Android version, and the capability obtained by sending an Intent.

Table 8.4: Discovered privilege escalation vulnerabilities in common Android devices and Android OS versions.

Device	OS	Build ID	Privilege Escalation Action
Alcatel A30	7.0	NRD90M	Take screenshot
Alcatel A30 Plus	7.0	NRD90M	Take screenshot
Amazon Fire TV Stick 2nd Gen.	5.1	LMY47O	Enable/disable Wi-Fi
BLU Grand XL	7.0	NRD90M	Device shutdown
Doogee X5	6.0	MRA58K	Video record screen
Juning TV Box	5.1.1	LMY49F	Take screenshot
Leagoo Z5C	6.0	MRA58K	Factory reset
LG Phoenix 2	6.0	MRA58K	Device shutdown
MXQ TV Box	4.4.2	KOT49H	Factory reset; brick the device
Plum Compass	6.0	MRA58K	Factory reset
Samsung S6 Edge (AT&T)	6.0.1	MMB29K	Forget Wi-Fi networks; device shutdown
Ulefone Power 2	7.0	NRD90M	Device shutdown; kill foreground app
Xiaomi Redmi 4	6.0.1	MMB29M	Take screenshot; leak bug report

Certain components can cause data leakage when receiving an Intent. For example, via an Intent with only an action string, the Xiaomi Redmi 4 device will dump the text of active notifications and logcat log into a bug report on external storage. The Xiaomi Redmi 4 device and three other devices contained an open interface to a privileged process that will take a screenshot and write it to external storage when it receives an Intent with a specific action string. Obtaining the contents of the screen is regarded as sensitive and not granted to third-party apps. Using this vulnerability, a malicious app can, for example, send an Intent to open a messaging app or an email app, then take a screenshot and dismiss the app by sending an Intent requesting the home screen. If needed, the malicious app can cause a system reboot to remove any notifications that a screenshot was taken.

8.3.4 Data Disclosure

A system crash is an exceptional event since a fatal error occurs within a critical Android OS process. Vendors may be interested in recording the cause so it can be identified and fixed in future releases. Certain vendors record the logcat log and write it to a file during or after a system crash. Information such as unique device identifiers, the user's email address, phone number, Global Positioning System (GPS) coordinates, the body of text messages, and sensitive log messages from other apps can be present in the logcat log [23]. A system crash can result in an information leak of sensitive data if the log file is not adequately protected. Therefore, any app on a vulnerable device can deliberately cause a system crash to obtain and process the log file for sensitive user data.

I discovered that Samsung devices running Android 5.0 to 7.0 create a world-readable file that contains the kernel log and logcat log whenever a system crash occurs.[8] Samsung introduced a *special* system process called `bootchecker` to ease the collection of needed debugging information after a system crash. However, `bootchecker` failed at setting the proper file permissions of the file in which it collects the logs, leaving it world-readable to any app on the device. Some Android devices with a MediaTek chipset have a modified `debuggerd` binary and non-AOSP system binaries such as `aee_dumpstate` and `aee_archive`. The `debuggerd` process sets the signal handlers for each process and will obtain debugging information by attaching to the process before it terminates. When a system crash occurs, it will attach to the `system_server` process. The `system_server` process is a critical system

[8]More information about the vulnerability can be found here: `https://nvd.nist.gov/vuln/detail/CVE-2017-7978`.

process that provides services to apps. On certain devices with a MediaTek chipset, it will write a world-readable archive file containing the logcat log, the kernel log, and various other logs to the `/data/aee_exp` directory or to the `/sdcard/mtklog/aee_exp` directory. The generated archive file is password protected, but the password was hard-coded in the `debuggerd` binary as `X4rLa8f3`. Examples of vendors that exhibited this behavior are BLU, RCA, Kata, Yuntab, Ulefone, and Figo. The two information disclosure vulnerabilities I discovered and reported have been fixed by Samsung and MediaTek.

8.4 Google Play Testing

Google Play is the official app distribution channel for the Android platform, facilitating the installation of apps. To determine the prevalence of inadequate exception handling during inter-app communication within Android apps, I tested a representative sample of 18,583 free Android apps from Google Play. These apps were the most popular apps from each app category on Google Play that were downloaded once every four weeks between March 2016 and April 2017. The 18,583 apps comprised 4,972 unique package names, each of which had four different versions on average (rounded up). Figure 8.4 shows the distribution of

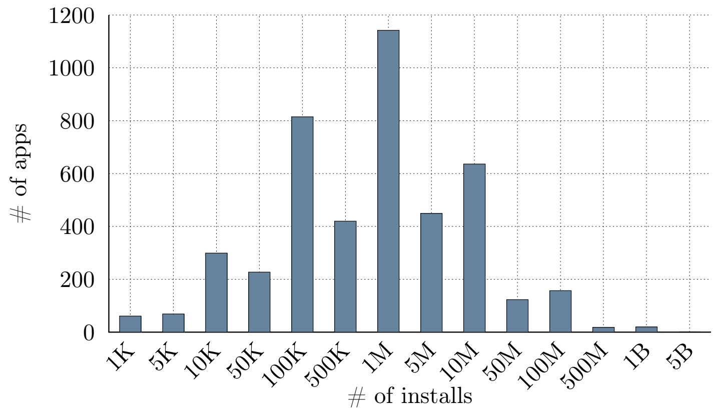


Figure 8.4: Distribution of apps in the dataset. (Apps with multiple versions were counted only once.)

apps in the dataset grouped by unique package name (apps with multiple versions were counted once). I also separately tested the top 300 most popular free apps on Google Play to determine the exposure of the apps that have the largest number of users. The top 300 apps were downloaded on November 27, 2017.

Daze tested 18,583 apps and discovered that 34.7% of the apps (6,463) in the sample could be crashed externally by a zero-permission Android app co-located on the device. *Daze* found a total of 14,413 fatal exceptions covering 53 types of exceptions. Table 8.5 presents a count of all fatal exceptions by type. The most common exception encountered during testing was `NullPointerException` with 10,862 instances, accounting for 75.3% of all fatal exceptions. During testing, *Daze* leaves various intent fields set to null, causing a receiving process to crash if it does not perform proper null-checking. The next most common reason for fatal exceptions (7.5%) was the failure of the developer to implement a particular class (`ClassNotFoundException` and `NoClassDefFoundError`), which causes an uncaught exception when the class loader fails to find the class. This is generally caused by an app component that was registered in the app's manifest, but it was not implemented in the app's code. The third most common reason is that the developer failed to call the appropriate superclass method when executing an app component life-cycle method, resulting in a `SuperNotCalledException`, occurring 650 times (4.5%). The 126 instances (0.8%) of `SecurityException` were due to apps performing permission-protected functionality without the corresponding permission.

Components form the skeleton of an Android app where the developers implement components to perform specific functions. Table 8.6 presents the aggregate number of crashes by component type and the corresponding ratio of externally crashable components to the total number of exported components. Activity components were the most numerous and also the most vulnerable (12.0%) to fatal exceptions. Content providers were the least vulnerable (2.2%). The 18,583 apps had a total of 143,790 total exported components with 14,413 app components being vulnerable (10.0%) to having an external app crash the process containing the vulnerable component.

Table 8.5: Exceptions in the 18K apps dataset.

Exception Name	**Freq.**
java.lang.NullPointerException	10,862
android.util.SuperNotCalledException	650
java.lang.ClassNotFoundException	583
java.lang.NoClassDefFoundError	492
java.lang.IllegalArgumentException	322
java.lang.RuntimeException	252
java.lang.IllegalStateException	211
java.lang.IndexOutOfBoundsException	161
java.lang.UnsatisfiedLinkError	140
java.lang.SecurityException	126
java.lang.ClassCastException	83
content provider crash	82
android.content.res.Resources$NotFoundException	63
java.lang.NumberFormatException	52
java.lang.InternalError	46
java.lang.UnsupportedOperationException	42
android.content.ActivityNotFoundException	35
android.view.WindowManager$BadTokenException	29
java.lang.ArrayIndexOutOfBoundsException	24
android.database.CursorIndexOutOfBoundsException	24
android.database.sqlite.SQLiteException	16
java.lang.InstantiationException	15
java.lang.NoSuchFieldError	14
java.lang.StringIndexOutOfBoundsException	12
android.util.AndroidRuntimeException	10
java.lang.AssertionError	9
android.content.ReceiverCallNotAllowedException	8
java.security.InvalidParameterException	7
android.view.InflateException	5
java.lang.NoSuchMethodError	4
java.lang.IllegalAccessException	4
java.util.MissingFormatArgumentException	3
java.io.FileNotFoundException	3
android.view.ViewRootImpl$CalledFromWrongThreadException	3
signal 6 (SIGABRT)	2
java.lang.AbstractMethodError	2
android.runtime.JavaProxyThrowable	2
java.lang.ExceptionInInitializerError	2
signal 11 (SIGSEGV)	1
org.json.JSONException	1
java.lang.VerifyError	1
java.lang.NoSuchFieldException	1
java.lang.IncompatibleClassChangeError	1
java.lang.IllegalAccessError	1
java.lang.Exception	1
java.lang.ArithmeticException	1
android.support.v4.app.SuperNotCalledException	1
android.database.sqlite.SQLiteCantOpenDatabaseException	1
android.content.pm.PackageManager$NameNotFoundException	1
android.app.RemoteServiceException	1
android.app.Fragment$InstantiationException	1
TOTAL	14,413

Table 8.6: Breakdown of the number of vulnerable components for the Google Play study.

Type	**#Exported**	**#Vulnerable**	**%Vulnerable**
Activity	62328	7483	12.0
Static Receiver	50453	4625	9.2
Dynamic Receiver	16041	1449	9.0
Provider	3749	82	2.2
Service	11219	774	6.9
TOTAL	143790	14413	10.0

To my knowledge, *Daze* is the only system that tests dynamically registered broadcast receivers. Of the 14,413 fatal exceptions *Daze* identified, 1,449 were due to a dynamically registered broadcast receivers registered by a component. Certain dynamically registered broadcast receivers register for actions that can only be sent by the Android OS itself, which *Daze* is not able to send.

8.4.1 Apps Dataset Fidelity

The reported exposure measurements for the 18k dataset might be conservative (under-approximations) in some cases due to potentially missing versions of apps that were updated outside the market sampling interval. Compared to the version updates history on AppBrain, I found that 119 apps had two to three missing updates between the last vulnerable version and the fixed version in the dataset.[9] This may lead to under-approximation of the number of consecutively vulnerable versions and the exposure time window if any of these missing versions are still vulnerable. Though I was unable to find download links for these missing versions, these 119 apps comprised about 30% of fixed vulnerabilities that persisted in a single app version in the dataset (total 50% of all fixed vulnerabilities existed in a single app version; see Figure 8.6). If I assume an equal probability that one of the missing versions were still vulnerable, it would drop the percentage of fixed vulnerabilities existing in only one app version from 56% to around 26%, and the difference would be redistributed over vulnerabilities that existed in two and three consecutively vulnerable versions.

[9]AppBrain can be accessed at: `https://www.appbrain.com`.

8.4.2 Longitudinal Vulnerability Analysis

I examined all apps that had between three and eight (inclusive) different versions with the same package name — a total of 1,451 package names comprising 5,491 app versions. I determined, in each version of an app, whether identified exceptions were introduced in that version or inherited from the previous version of the app. I consider all exceptions found in the first version (lowest version code) of an app to be introduced in this version. Note that results reported in this section are conservative (see Section 8.4.1).

I considered a recurring exception to be a specific exception introduced in a particular version of an app that propagates to a subsequent version. Within the 5,491 apps subset, *Daze* discovered 6,427 fatal exceptions. Of these exceptions, 2,706 were unique and the remaining were recurrences of the same exception in different versions of the same app. I found that the majority of the exceptions were inherited from previous app versions instead of being newly introduced as shown in Figure 8.5. Of all apps with at least three versions in the sample, 2,085 apps (37.9%) contained at least one recurring exception and 1,008 apps (18.3%) contained at least one fatal exception that was non-recurring. There were 1,580 apps (28.7%) in the sample that contained the same exception recurring through all versions

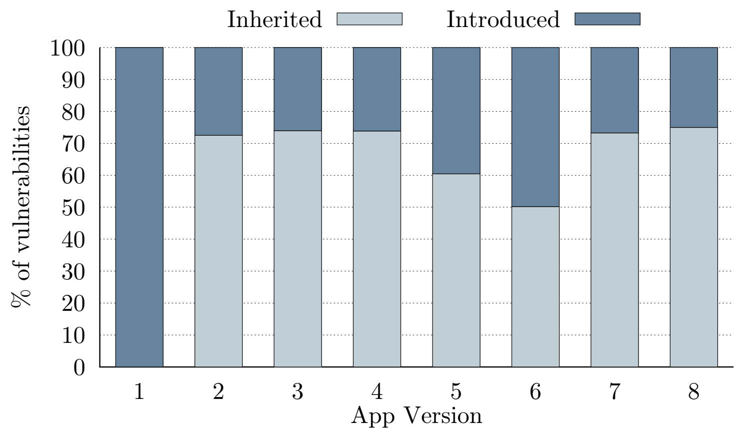


Figure 8.5: Vulnerabilities evolution as the relative percentage of inherited and introduced exceptions in apps across eight versions.

(covering 419 apps with 1,580 different versions).

I categorized the 2,706 exceptions into those that had been fixed (1,241) and those that were still vulnerable (1,465 as of the last app versions available in the sample). Figure 8.6 illustrates the number of consecutive versions a vulnerability persisted through in the dataset. Around 40% of the vulnerabilities were present in a single version and then fixed in the subsequent version of an app. An open vulnerability is a vulnerability that is still present in the last version (most recent) of an app contained in the sample. More than 50% of the open vulnerabilities persisted in at least the latest two app versions in the sample and about 10% of vulnerabilities persisted without patching through at least the latest four versions of the same app. Interestingly, 16% of the vulnerabilities were introduced in the last version of apps in the sample (about 30% of open vulnerabilities).

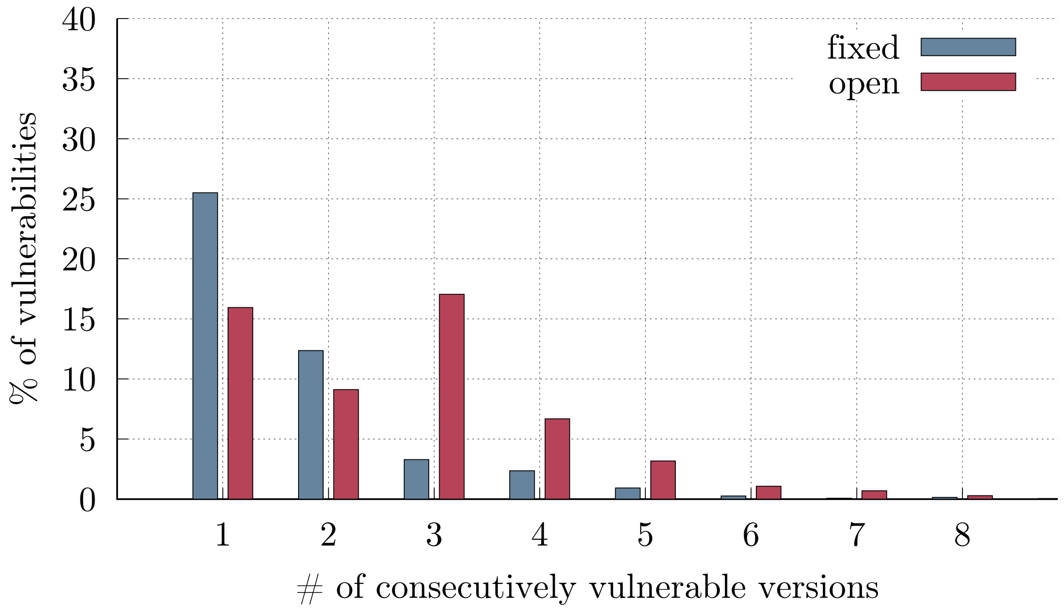


Figure 8.6: Distribution of the exposure window in terms of the number of consecutively vulnerable versions to fixed and open vulnerabilities (till April 2017).

I also examined the exposure window of vulnerabilities with respect to time. I used AppBrain to determine when an app version was updated on Google Play. AppBrain did not contain data for all versions, so in those cases I relied on the date I downloaded the app. The exposure window started when a vulnerable app version was uploaded to Google Play. Certain apps that were infrequently updated increased the size of the exposure window since the available app version at the time of downloading may have been uploaded prior to the beginning of the collection period. For fixed vulnerabilities, the time window ended when the vulnerability was first fixed in a subsequent app version contained within the sample. For open vulnerabilities, I conservatively assumed that the vulnerability would be fixed in the version released after the last version in the sample. If the last vulnerable version in the sample was the current version on Google Play, I set the end date of the exposure window to December 10, 2017.

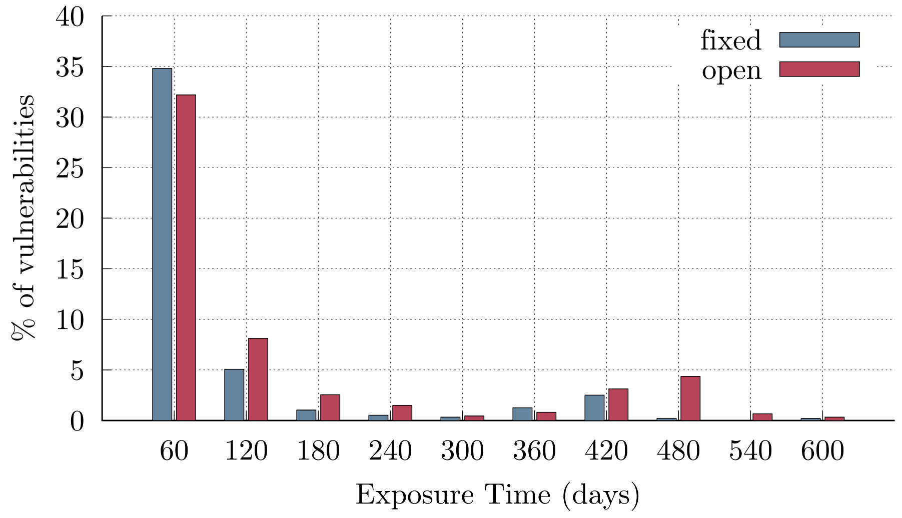


Figure 8.7: Distribution of the exposure time of fixed and open vulnerabilities (till April 2017).

Figure 8.7 provides the exposure window of fixed and open vulnerabilities in days. About 35% of the vulnerabilities were fixed with a subsequent update occurring within 60 days, and about 15% of the vulnerabilities persisted for 60 to 600 days before being fixed. More than 30% of the vulnerabilities remained unpatched for 60 or more days in the sample, about

30% of these have been unpatched for more than 100 days, and 5% remained unpatched for more than 360 days (about 15% of the open vulnerabilities).

8.4.3 Top 300 Most Popular Android Apps

The top 300 free apps on Google Play are the most widely used apps available to Android devices with some apps having billions of installations. One might presume that the top 300 apps would be more carefully coded to be resilient to inter-app vulnerabilities due to their popularity and user base from which to obtain feedback. *Daze* tested the top 300 free apps and found that 149 of the 300 apps (49.6%) contained at least one vulnerable component that could be crashed externally. There were a total of 310 fatal exceptions in the 300 most popular apps on Google Play. Table 8.7 provides all the fatal exceptions in the 300-app sample ranked by the number of occurrences. The top 300 apps have a higher ratio of apps contain at least on vulnerable component than the 18,583 app sample (34.7%). This is likely because the top 300 free apps have a higher average number of exported app components (14.84) than the 18,583 app sample (7.75). The larger attack surface due to a higher average

Table 8.7: Exceptions in the top 300 Google Play apps.

Exception Name	**Frequency**
java.lang.NullPointerException	237
java.lang.ClassNotFoundException	44
java.lang.IllegalArgumentException	5
java.lang.ClassCastException	5
java.lang.IllegalStateException	4
java.lang.RuntimeException	4
android.util.SuperNotCalledException	3
java.lang.SecurityException	2
org.xmlpull.v1.XmlPullParserException	1
Signal 11 (SIGSEGV)	1
java.lang.UnsatisfiedLinkError	1
java.io.FileNotFoundException	1
android.view.InflateException	1
android.os.FileUriExposedException	1
TOTAL	310

number of exported components in the top 300 free apps may result in additional chances for developer error. The larger number of components to be tested also affected the time to test each app for the two samples as discussed in Section 8.4.4.

The most popular app (as of November 30, 2017) named *Rules Of Survival* (`com.netease.chiji`, version code 221929) had five fatal exceptions. Two components (`PushServiceReceiver` and `PushService`) encountered a `SecurityException` when creating a shared preferences file with a mode of `MODE_WORLD_READABLE`. This is an issue with Android version compatibility since the API call that threw the exception has behavior dependent on the Android version of the device in which it runs. It is also a security issue since the shared preferences file the app tries to create is world-readable and may contain sensitive data. *Google Photos* (`com.google.android.apps.photos`, version code 1992480) contained four fatal `NullPointerException` exceptions. In the top 300 apps, there were 43 instances of `ClassNotFoundException` due to not implementing a particular class. This occurred in 32 apps (10.7% of the sample) with *AdVenture Capitalist* (`com.kongregate.mobile.adventurecapitalist`, version code 2040016345) encountering the `ClassNotFoundException` five times.

8.4.4 Performance Overhead

The primary factor influencing the amount of time *Daze* takes to test an app is the number of exported components that it contains. Android apps have a wide variance with regard to complexity and the number of interfaces it exposes externally. Basic apps can contain a single exported activity component, whereas more complex apps can contain hundreds of components. The number of dynamically registered broadcast components also increases overhead since each will be tested. The average time to test an individual app for the 18,583 app sample was 76.5 seconds, whereas the average time to test an app in the top 300 free app sample was 126.1 seconds. The apps in the top 300 free apps on Google Play had 14.84 exported components per app on average and the 18,583 had an average of 7.75 exported components per app. Figure 8.8 shows the analysis time for the two samples. The 18,583 app sample contained outliers that were due to apps with numerous components to test and

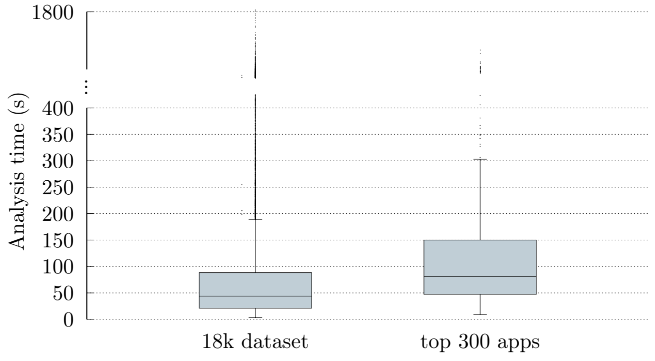


Figure 8.8: Analysis time statistics of the 18K apps and the top 300 apps datasets.

retest. There were more than 101 apps that contained 60 or more exported components. Some of the apps in the both samples trigger the system to kill background processes to free up resources, collaterally terminating *Daze* and causing it to incur delays in retesting the component to attribute this behavior to the responsible component.

I have witnessed some cases where testing an app component will result in the analysis app getting killed. If this happens five times, the component is skipped. Skipping of an app component is fairly uncommon except with content providers that crash during testing. Of 143,790 tested components in the 18,583 apps, only 105 components needed to be skipped due to the analysis app being terminated. I manually examined the logs to determine the cause of the *Daze* being terminated. This was due to crashing content providers, apps killing background processes, and apps creating the conditions necessary to activate the Low Memory Killer module which terminates processes to free memory.

There may be additional faults in the components exported by the `system_server` process that can be found by additional fuzzing of the inputs in intent objects. This analysis can be extended to provide a more complete analysis using static analysis to determine the key names and value types stored in intents to provide aid with the fuzzing of inputs. *Daze* can be configured to propagate content provider faults into calling apps by exporting a content provider that throws a null pointer exception when queried. Therefore, if the SQLite operation on the client app side is not caught (i.e., not within a `try-catch` block), the client app will itself crash.

8.5 Testing the Stability of AOSP Builds

To determine the robustness of AOSP builds and, by extension, the vendors that modify them, I tested all available AOSP factory images (i.e., builds) for the Nexus 5 and Nexus Player devices.[10] The 27 builds for the Nexus 5 ranged from Android 4.4 (KRT16M) to Android 6.0.1 (M4B30Z). I installed each available firmware for Nexus 5, and used *Daze* to determine the exposed interfaces of the core Android package (i.e., `system_server`). According to Google's *Android Dashboards*, as of November 2017, 51.7% of all Android devices (i.e., 5.1 to pre-7.0) contain a core component that allows any app co-located on the device to quickly crash the system by sending a single intent message.[11] Although some of these vulnerabilities have been fixed in later Android releases, a majority of current Android devices are vulnerable due to the Android fragmentation problem [75, 76].

8.5.1 Faults in Android 5.1 to 6.0.1 (Smartphones)

I discovered that the activity `IntentForwarderActivity` in the `com.android.internal.app` package of the `system_server` process can crash the system in all Android versions from 5.1 to versions prior to 7.0 (20 builds in total for the Nexus 5). This occurs since the component blindly operates on the action string from the intent without null checking, which causes an uncaught exception by calling the `equals` method on a null string reference.

I discovered that all 13 AOSP 6.0.1 builds for Nexus 5 were vulnerable to a system crash when an app broadcasts an Intent for the action `android.net.conn.CONNECTIVITY_CHANGE_SUPL` without supplying any embedded data in the Intent. The root issue resides in the

[10]Factory images for Nexus devices can be downloaded from: `https://developers.google.com/android/images`.

[11]Google Dashboards can be accessed at: `https://developer.android.com/about/dashboards/index.html`.

`GpsLocationProvider` class in `system_server`, which dynamically registers a broadcast receiver that assumes every broadcast intent it receives will not contain a null `Bundle` object, resulting in a null pointer exception when it tries to handle an intent with no data. The vulnerable broadcast receiver is dynamically registered in the `GpsLocationProvider` constructor and an object of this type will be created in the `LocationManagerService`.

Table 8.8: Broadcast actions causing a system crash for AOSP Android TV.

Broadcast Action	**Vulnerable TV Version**			
	5.x	**6.x**	**7.0**	**8.0**
android.bluetooth.input.profile.action.HANDSHAKE	●	●		
android.bluetooth.input.profile.action.REPORT	●	●		
android.intent.action.SIM_STATE_CHANGED		●		
android.intent.action.EMERGENCY_CALLBACK_MODE_CHANGED			●	●

8.5.2 Faults in Android 5.0 to 8.0 (Android TV)

Daze tested all 31 AOSP factory images for the Asus Nexus Player device to determine the prevalence of vulnerable system components within the Android TV device. I discovered that each of the 31 factory images from Android 5.0 (LRX21M) to 8.0 (OPR6.170623.021) had at least one vulnerable broadcast receiver. The results are presented in Table 8.8. The primary cause of the system crashes in Android TV were inadequate input validation and error handling for Bluetooth and telephony-related intents. The Bluetooth-related Intents caused uncaught exceptions in the `RemoteControlService` class due to inadequate null-checking and the assumption a `Bundle` will be not be null in the received intent. All the Android TV devices I tested (i.e., Sony Bravia XBR-43X830C, Nexus Player, and Nvidia Shield) would crash when certain Bluetooth or telephony-related broadcast Intents were sent without an embedded `Bundle` object.

I examined the AOSP code and found that `system_server` had registered broadcast receivers to listen for specific telephony-related broadcast Intents, even though the device does not have telephony capabilities. Generally, the phone app (`com.android.phone`) uses the `protected-broadcast` tag in its manifest file so that only the system can send these telephony-related Intents. Since the phone app is not installed on Android TV devices starting with Android 6.0, a third-party app can send these broadcast Intents and cause `system_server`, and thus the device, to crash. Android TV devices usually lack safe mode, so a persistent local DoS attack against `system_server` can result in the user having to perform a factory reset to recover the device if ADB was not enabled prior to the DoS attack, possibly resulting in data loss.

8.5.3 Generic Android DoS Attack

Certain Android devices will not have exposed system components that allow a single intent to crash the system. I discovered a novel approach to trigger a controlled boot loop attack on Android by making the `system_server` process in the Android OS encounter an `OutOfMemoryError` condition, leading to a system restart. This is accomplished by a zero-permission app repeatedly using a specific API method call where a parameter to the method call will end up being stored on the heap of the `system_server` process. When a process is started in Android, including `system_server`, it is allocated a fixed maximum heap size. Once `system_server` allocates all of its heap memory, it will eventually crash if it cannot free any additional memory.

Apps can dynamically register a broadcast receiver by providing an object that inherits from the `BroadcastReceiver` class and an `IntentFilter` object that contains one or more action strings. The `system_server` process manages all app components in the `ActivityManagerService` class. When an action string is provided during broadcast receiver registration, it is stored in a variable that can hold an arbitrary amount of data (a `HashSet` instance variable named `mfilters` in the `IntentResolver` class). Therefore, the app can provide large strings to be stored on the heap of the `system_server` process to exhaust its memory, consequentially causing a system crash. The app registers broadcast receivers that have an `IntentFilter` with a unique action string containing 55,405 characters and a integer value that is incremented and appended to the end of the string to ensure the action is unique. The registration of broadcast receivers is quickly repeated and eventually the `system_server` process throws an `OutOfMemoryError` as it tries to allocate more memory while aggressively performing garbage collection.

Most of the `OutOfMemoryError`s that repeatedly occur will be caught by the underlying Binder implementation. Binder is part of the Android architecture that enables IPC via a kernel module. Once the heap memory of `system_server` is exhausted, the app can wait for an uncaught error to occur or perform additional action(s) to facilitate an uncaught exception, such as starting a large number of activities using the `startActivities(Intent[])` API method call. An app can determine the maximum heap size for apps by obtaining the value for the `dalvik.vm.heapsize` system property. `system_server` will generally have a maximum heap size of either 256 MB or 512 MB. Although the current heap size of `system_server` varies depending on its current workload, the maximum heap size multiplied by a factor of 17.7 generally yields the appropriate number of actions to register to exhaust its heap.

8.5.4 Comparison to Prior Work

It has been shown that vendor customization can introduce vulnerabilities via their pre-installed apps [254], hanging attribute references [255], and device driver customization [256]. Previous studies have found that apps and ad libraries tend to be over-privileged, with more unneeded permissions often added to updates without regard to the increased risk these permissions pose [257–259].

Previous approaches that generated reproducible crash test-cases have relied on two methods to obtain the statically registered app components: parsing the apps' manifest directly [253, 260–262] or relying on the OS package manager [263, 264]. Unfortunately, these approaches inherently suffer from poor inter-app coverage as they overlook dynamically registered components in apps and those declared by the Android OS. As shown in Section 8.3, more than 62% of system crash DoS vulnerabilities detected by *Daze* in the 32 devices resided in dynamically-registered components. Sasnauskas et al. [253] compared null-intent fuzzing to data fuzzing of intents and discovered that data fuzzing only yielded a 1% increase in code coverage in their evaluation. Ye et al. [260] focused exclusively on fuzzing the category, data, and action fields of activity components. Yang et al. [265] created a system to detect privilege escalation events. This approach overcomes key limitations of the other approaches for null-intent testing in regard to completeness (covering all types of app components), automation (e.g., clearing the crash dialogue and restarting the app), and settings and file system monitoring. Maji et al. [263] proposed a semi-manual approach to test exported app activities on three specific versions of Android.

Recently, Garcia et al. [262] created LetterBomb to detect inter-app vulnerabilities. They used complex backward data-flow analysis algorithms to discover DoS vulnerabilities due to missing null-checks when accessing intent fields, then dynamically generated Intents to see if the discovered vulnerabilities were indeed exploitable. They tested 10,000 apps and discovered only 104 exploitable DoS vulnerabilities, taking three minutes per app on average. Similarly, Hay et al. [266] proposed IntentDroid as a full data-fuzzing approach to detect inter-app vulnerabilities. IntentDroid identified 31 intent-related DoS vulnerabilities in 55 apps among the top-rated apps on Google Play in 2014, taking an average of 25 minutes per app. The work in [262] also compared LetterBomb to IntentDroid and found that IntentDroid detected only two thirds of the DoS vulnerabilities detected by LetterBomb in a sample of 40 apps. In contrast, *Daze* detected 219 vulnerabilities for the oldest versions in the dataset for the same 55 vulnerable apps reported in [266] compared to only 108 detected by IntentDroid (presuming IntentDroid successfully detected all Java and native crashes in Table 1 in [266]).

Although I did not measure code coverage during testing (mainly since collecting coverage metrics from stock firmware images and apps requires tampering with their packaged code and resigning them, which could influence the outcomes of the study), I argue that the main difficulty of inter-app data-fuzzing on Android is that Android enforces type-safety on Intent fields by returning a null-like value when a field is accessed using the wrong value type. This type-safe access means that data-fuzzing approaches must use only correct value types in order to achieve any meaningful coverage beyond null-fuzzing. However, the overhead incurred by data-fuzzing prohibits large-scale vetting due to the large input space and complex algorithms involved. *Daze* differs from these solutions in that it approaches the problem directly by null-fuzzing exported components, allowing it to discover significantly more vulnerabilities in significantly less time: 14,413 exploitable vulnerabilities in 18,583 apps, taking two minutes per app on average.

8.5.5 Summary

I presented *Daze*, an automated system to identify fatal exceptions within Android apps and the Android OS. Using *Daze*, I discovered that more than 50% of the current Android devices are vulnerable to a persistent system crash DoS attack, enabled by inadequate exception handling in the Android base code. *Daze* created reproducible test cases for more than 20,000 fatal errors within the three datasets spanning 13 months across 59 versions of AOSP from the smartphone to Android TV distributions. Furthermore, the longitudinal analysis quantified the exposure period of fatal exceptions in consecutive versions of apps. The results showed that the majority of fatal exceptions in an app were inherited from the previous app version, and 20% of unpatched vulnerabilities existed more than 100 days. Moreover, 10% of app components tested were susceptible to attacks, causing fatal crashes by an external app. Beyond DoS attacks, I discovered that system crashes facilitated data disclosure vulnerabilities on certain popular Android devices. Lastly, I presented a novel and universal attack to force any Android device to encounter a system crash by exhausting its heap memory using standard APIs from a zero-permission app.

Chapter 9: Conclusion and Future Work

This dissertation introduced a novel program analysis technique for the Android platform, Forced-Path Execution (*FPE*), and investigated the effects of failure states stemming from various DoS attacks against the Android OS and apps. These research areas aim to make the Android ecosystem more secure by focusing on two primary features of the Android: apps and the OS. Android apps provide a rich user experience, although they can also be malicious or unintentionally insecure. The potential consequences are more severe for pre-installed apps due to their privileged position on the device. Some potential undesirable behaviors range from leaking PII to unintentionally exposing capabilities to other processes. To simultaneously investigate the possible states of the program under numerous execution environments, I developed the *FPE* framework for Android.

Although I used Android as an exemplar, the methodology underlying *FPE* can be applied to different programming languages and contexts. *FPE* provides benefit by exploring program states that may be difficult to reach using other program analysis techniques. Different execution strategies can be used to limit the amount of forcing through constraints, allowing activities ranging from exploration to verification. Pairing *FPE* with taint analysis can discover interesting data flows within apps, especially pre-installed apps. In addition, to simply theoretical and contrived apps, I used *FPE* on a set of real world pre-installed apps from a number of Android firmware images.

I displayed the extensibility of the *FPE* framework by including a module for data flow analysis that allows for different execution modes to limit the exploration of conditional statements and for discovering vulnerabilities where an attacker can provide untrusted input to a third-party app or pre-installed app for privilege escalation. I highlight the risks of insecure pre-installed apps due to their extensive capabilities in the system through my independent research referenced in this dissertation.

I developed various DoS attacks for Android, investigated their consequences, and implemented defenses for the attacks I discovered. Although DoS attacks may appear crude and rudimentary, I refined the DoS attacks so that they resulted in devastating effects on certain embedded Android devices. I studied these attacks and their effects on a multitude of Android devices and platforms. These effects resulting from the attacks ranged from temporarily reducing the availability of the device to making certain Android-based devices permanently unusable. These DoS attacks were reported to Google and resulted in modifications to AOSP source code. In addition, some of the bugs were marked as infeasible to fix.

I discovered an attack where a third-party app indirectly affect separate third-party apps by selectively killing them while solely using IPC behavior that does not require any permissions and is available to all apps. By exploring failure states of the system processes, I discovered two universal DoS attacks that can cause a system crash in Android. I also gained various insights on inter-app vulnerabilities by performing a Google Play longitudinal app study over multiple app versions and discovered thousands of inter-app vulnerabilities and dozens of system crashes among a range of vendors.

In conclusion, my research from these two research branches will ideally provide insights and potential research directions for current and future Android researchers. As Android is currently the most popular OS, additional research should be undertaken to make it more secure for the end-user.

Appendix A: Smali format for the TaskReceiver component.

```
.class public Lcom/adups/fota/sysoper/TaskReceiver;
.super Landroid/content/BroadcastReceiver;

# direct methods
.method public constructor <init>()V
    .registers 1

    invoke-direct {p0}, Landroid/content/BroadcastReceiver;-><init>()V

    return-void
.end method

# virtual methods
.method public onReceive(Landroid/content/Context;Landroid/content/Intent;)V
    .registers 7

    if-nez p2, :cond_3

    :cond_2
    :goto_2
    return-void

    :cond_3
    invoke-virtual {p2}, Landroid/content/Intent;->getAction()Ljava/lang/String;

    move-result-object v0

    const-string v1, "sys"

    new-instance v2, Ljava/lang/StringBuilder;

    invoke-direct {v2}, Ljava/lang/StringBuilder;-><init>()V

    const-string v3, "TaskReceiver action = "

    invoke-virtual {v2, v3}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)
        Ljava/lang/StringBuilder;

    move-result-object v2

    invoke-virtual {v2, v0}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)
        Ljava/lang/StringBuilder;

    move-result-object v2

    invoke-virtual {v2}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;

    move-result-object v2

    invoke-static {v1, v2}, Lcom/adups/fota/sysoper/n;->a(Ljava/lang/String;Ljava/
        lang/String;)V

    invoke-static {v0}, Landroid/text/TextUtils;->isEmpty(Ljava/lang/CharSequence;)Z

    move-result v1

    if-nez v1, :cond_2

    const-string v1, "android.intent.action.ACTION_POWER_DISCONNECTED"

    invoke-virtual {v0, v1}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z

    move-result v1

    if-eqz v1, :cond_2

    new-instance v1, Landroid/content/Intent;

    const-class v2, Lcom/adups/fota/sysoper/TaskService;

    invoke-direct {v1, p1, v2}, Landroid/content/Intent;-><init>(Landroid/content/
        Context;Ljava/lang/Class;)V

    invoke-virtual {v1, v0}, Landroid/content/Intent;->setAction(Ljava/lang/String;)
        Landroid/content/Intent;

    invoke-virtual {p1, v1}, Landroid/content/Context;->startService(Landroid/content
        /Intent;)Landroid/content/ComponentName;

    goto :goto_2
.end method
```

Appendix B: The `com.staqu.panasalestracker` (`versionCode = 8`, `versionName = 1.3.5`) app behavior captured by the *FPE* framework.

```
{
  "call": "android.util.Log.d(java.lang.String, java.lang.String)",
  "component": "com.staqu.salestracker.SalesTrackerService",
  "arguments": ["Tag:STracker", "Message:Activation url ==>
  https://stapp.panasonicarbo.com?brand=&model=
  STARHSTARTSTARCSTAR+STARDSTAReSTARsSTARiSTARrSTAReSTAR&
  imei1=1234567890123456&imei2=1234567890123456&android_id=
  9b1587bf3b1d8fc&mcc1=310&mnc1=660&mcc2=310&mnc2=660&date=
  12-03-2019&time=12%3A35%3A27&key=&typ=&ver=&ram=1MB&
  country_code=in&app_version=&manufacturer=HTC&lac=454&
  cid=454&eml=feelsgoodman%40gmail.com&android_version=0
  &build=""],
  "lineNumber": "54",
  "taint": "|0|136|",
  "category": "logging",
  "smaliFile": "SalesTracker/smali/com/staqu/salestracker/SLog.smali
  "
}
```

Listing B.1: Raw JSON event for writing PII to the logcat log (PII highlighted in red text).

```
{
  "call": "android.telephony.SmsManager.sendTextMessage(
  java.lang.String, java.lang.String, java.lang.String, android.app.
     PendingIntent, android.app.PendingIntent)",
  "component": "com.staqu.salestracker.SalesTrackerReceiver",
  "arguments": ["Destination number:+919220092200",
  "Service center number:0", "Text message body:PTRACR
  ||STARHSTARTSTARCSTARSTARDSTAReSTARsSTARiSTARrSTAReSTAR|
  1234567890123456|1234567890123456|9b1587bf3b1d8fc||||
  310|660|310|660|1GB"],
  "lineNumber": "3244",
  "taint": "|0|0|136|0|0|",
  "category": "telephony_events",
  "smaliFile": "SalesTracker/smali/com/staqu/salestracker/
  Util.smali"
}
```

Listing B.2: Raw JSON event for sending a text message (PII highlighted in red text).

Appendix C: Broadcast Intent actions to crash the Samsung Galaxy S6 Edge and Galaxy Tab A Android devices.

Table C.1: Broadcast actions that will cause a system crash on the Galaxy S6 Edge and Galaxy Tab A.

Broarcast Action	**S6 Edge**	**Tab A**
com.sec.intent.action.SARDEVICE_CP	✓	✓
com.sec.android.intent.action.SSRM_MDNIE_CHANGED	✓	✓
com.sec.android.intent.action.REQUEST_BACKUP_WALLPAPER	✓	
com.samsung.android.net.wifi.SEC_NETWORK_STATE_CHANGED	✓	
com.samsung.action.HOTSPOT_EXEC_COMMAND	✓	✓
android.net.ethernet.STATE_CONNECTOR_REMOVE	✓	✓
com.samsung.action.REDIRECT_STATUS	✓	
android.dirEncryption.DirEncryptionManager.UNMOUNT_POLICY	✓	
action_wfc_switch_profile_broadcast	✓	
com.sec.android.intent.action.REQUEST_RESTORE_WALLPAPER	✓	
android.intent.action.AGPS_UDP_RECEIVED	✓	
android.net.conn.CONNECTIVITY_CHANGE_SUPL	✓	✓
android.intent.WIDGETSSR.WIDGETADD	✓	✓
android.net.wifi.hs20.TEST_START_OSU_PROCESS_NOSIGMA	✓	✓
android.net.ethernet.STATE_CHANGE	✓	✓
com.samsung.android.net.wifi.NETWORK_OXYGEN_STATE_CHANGE	✓	✓
com.sec.location.nsflp.locblacklist		✓
android.net.wifi.hs20.TEST_TRIGGER_INSTALL_FILE	✓	✓
com.sec.android.inputmethod.Subtype	✓	✓
MARS_AUTO_RUN_POLICY_TRAFFIC_STAT_ALARM		✓
MARS_AUTO_RUN_TRAFFIC_STAT_REPEAT_ALARM		✓
android.content.jobscheduler.JOB_DEADLINE_EXPIRED		✓
com.samsung.selfhealing.abusivedetected		✓

Appendix D: System crash from faulty Intent tnput handling.

```
!@*** FATAL EXCEPTION IN SYSTEM PROCESS: main
java.lang.RuntimeException: Error receiving broadcast Intent { act=com.sec.android.
    intent.action.APP_HQM_SEND_REQ flg=0x10 } in com.samsung.android.hqm.
    BigDataModule$1@a9bbc0c
  at android.app.LoadedApk$ReceiverDispatcher$Args.lambda$-
      android_app_LoadedApk$ReceiverDispatcher$Args_52225(LoadedApk.java:1329)
  at android.app.-$Lambda$FilBqgnXJrN9Mgyks1XHeAxzSTk.$m$0(Unknown Source:4)
  at android.app.-$Lambda$FilBqgnXJrN9Mgyks1XHeAxzSTk.run(Unknown Source:0)
  at android.os.Handler.handleCallback(Handler.java:789)
  at android.os.Handler.dispatchMessage(Handler.java:98)
  at android.os.Looper.loop(Looper.java:164)
  at com.android.server.SystemServer.run(SystemServer.java:724)
  at com.android.server.SystemServer.main(SystemServer.java:543)
  at java.lang.reflect.Method.invoke(Native Method)
  at com.android.internal.os.Zygote$MethodAndArgsCaller.run(Zygote.java:327)
  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:1374)
Caused by: java.lang.NullPointerException: Attempt to invoke virtual method 'int
    android.os.Bundle.getInt(java.lang.String, int)' on a null object reference
  at com.samsung.android.hqm.BigDataModule$1.onReceive(BigDataModule.java:606)
  at android.app.LoadedApk$ReceiverDispatcher$Args.lambda$-
      android_app_LoadedApk$ReceiverDispatcher$Args_52225(LoadedApk.java:1319)
  ... 10 more
```

Listing D.1: Example system crash occuring on a Samsung Android device.

Bibliography

[1] K. Taylor and L. Silver, “Smartphone ownership is growing rapidly around the world, but not always equally,” February 2019, retrieved August 21, 2019 from https://www.pewresearch.org/global/2019/02/05/smartphone-ownership-is-growing-rapidly-around-the-world-but-not-always-equally/.

[2] J. Booton, “The rise and fall of the pc in one chart,” April 2016, retrieved August 21, 2019 from https://www.marketwatch.com/story/one-chart-shows-how-mobile-has-crushed-pcs-2016-04-20.

[3] C. Arthur, “Facebook’s mobile journey has only just begun, but already makes money,” February 2014, retrieved August 21, 2019 from https://www.theguardian.com/technology/2014/feb/03/facebook-mobile-desktop-pc-platforms.

[4] B. Carson, “More people now search google on their phone than from their computers,” October 2015, retrieved August 21, 2019 from https://www.businessinsider.com/more-people-now-search-google-on-their-phone-than-from-their-computers-2015-10.

[5] S. Gibbs, “Mobile web browsing overtakes desktop for the first time,” November 2016, retrieved August 21, 2019 from https://www.theguardian.com/technology/2016/nov/02/mobile-web-browsing-desktop-smartphones-tablets.

[6] L. Handley, “Nearly three quarters of the world will use just their smartphones to access the internet by 2025,” January 2019, retrieved August 21, 2019 from https://www.cnbc.com/2019/01/24/smartphones-72percent-of-people-will-use-only-mobile-for-internet-by-2025.html.

[7] W. Lee, “People spend more time on mobile devices than tv, firm says,” June 2019, retrieved August 21, 2019 from https://www.latimes.com/business/la-fi-ct-people-spend-more-time-on-mobile-than-tv-20190605-story.html.

[8] C. Yildirim, E. Sumuer, M. Adnan, and S. Yildirim, “A growing fear: Prevalence of nomophobia among turkish college students,” *Information Development*, vol. 32, no. 5, pp. 1322–1331, 2016.

[9] A. L. S. King, A. M. Valença, A. C. Silva, F. Sancassiani, S. Machado, and A. E. Nardi, “nomophobia: Impact of cell phone use interfering with symptoms and emotions of individuals with panic disorder compared with a control group,” *Clinical Practice & Epidemiology in Mental Health*, vol. 10, no. 1, 2014.

[10] M.-P. Tavolacci, G. Meyrignac, L. Richard, P. Dechelotte, and J. Ladner, “Problematic use of mobile phone and nomophobia among french ccollege students,” *The European Journal of Public Health*, vol. 25, no. suppl 3, 2015.

[11] S. Perez, “China accounted for nearly half of app downloads in 2018, 40% of consumer spend,” January 2019, retrieved August 21, 2019 from https://techcrunch.com/2019/01/16/china-accounted-for-nearly-half-of-app-downloads-in-2018-40-of-consumer-spend/.

[12] “Number of Android apps on Google Play - AppBrain,” retrieved April 6, 2019 from http://www.appbrain.com/stats/number-of-android-apps.

[13] *Operating System Market Share Worldwide*, retrieved February 23, 2019 from http://gs.statcounter.com/os-market-share.

[14] “Android,” retrieved April 9, 2019 from https://www.android.com/.

[15] Z. Doffman, *New Android Warning: 100M+ Users Installed App With Nasty Malware InsideUninstall Now*, August 2019, retrieved August 31, 2019 from https://www.forbes.com/sites/zakdoffman/2019/08/27/android-warning-nasty-malware-hiding-inside-app-installed-by-100m-users/.

[16] ——, *Android Warning: Devious Malware Found Inside 34 Apps Already Installed By 100M+ Users*, August 2019, retrieved August 31, 2019 from https://www.forbes.com/sites/zakdoffman/2019/08/13/android-warning-100m-users-have-installed-dangerous-new-malware-from-google-play.

[17] E. Price, *More Than 500,000 People Downloaded These Malware-Infected Android Apps*, November 2018, retrieved August 31, 2019 from https://fortune.com/2018/11/26/google-play-malware-apps/.

[18] D. Palmer, *This Android malware can take photos and videos and spy on your app history*, July 2019, retrieved August 31, 2019 from https://www.zdnet.com/article/this-android-malware-can-take-photos-and-videos-and-spy-on-your-app-history/.

[19] ——, *New Android malware replaces legitimate apps with ad-infested doppelgangers*, July 2019, retrieved August 31, 2019 from https://www.zdnet.com/article/new-android-malware-replaces-legitimate-apps-with-ad-infested-doppelgangers/.

[20] K. O’Flaherty, *Google Android Warning As Devious Spyware Hits The Play Store*, August 2019, retrieved August 31, 2019 from https://www.forbes.com/sites/kateoflahertyuk/2019/08/22/google-android-spyware-warning-as-devious-app-hits-the-play-store-twice/.

[21] R. Johnson and A. Stavrou, “Vulnerable Out of the Box: An Evaluation of Android Carrier Devices,” 2018, retrieved April 25, 2019 from https://media.defcon.org/DEF%20CON%2026/DEF%20CON%2026%20presentations/Ryan%20Johnson%20and%20Angelos%20Stavrou%20-%20Updated/DEFCON-26-Johnson-and-Stavrou-Vulnerable-Out-of-the-Box-An-Eval-of-Android-Carrier-Devices-WP-Updated.pdf.

[22] R. Johnson, A. Benameur, and A. Stavrou, “All Your SMS & Contacts Belong To Adups & Others,” 2017, retrieved April 25, 2019 from https://www.blackhat.com/docs/us-17/wednesday/us-17-Johnson-All-Your-SMS-&-Contacts-Belong-To-Adups-&-Others.pdf.

[23] R. Johnson and A. Stavrou, "Resurrecting the READ_LOGS Permission on Samsung Devices," 2015, retrieved April 25, 2019 from https://www.blackhat.com/docs/asia-15/materials/asia-15-Johnson-Resurrecting-The-READ-LOGS-Permission-On-Samsung-Devices-wp.pdf.

[24] A. Ng, *Android malware that comes preinstalled is a massive threat*, August 2019, retrieved August 31, 2019 from https://www.cnet.com/news/android-malware-that-comes-preinstalled-are-a-massive-threat/.

[25] C. Cimpanu, *Malware found preinstalled on some Alcatel smartphones*, January 2019, retrieved August 31, 2019 from https://www.zdnet.com/article/malware-found-preinstalled-on-some-alcatel-smartphones/.

[26] S. Dent, *Report finds Android malware pre-installed on hundreds of phones*, May 2018, retrieved August 31, 2019 from https://www.engadget.com/2018/05/24/report-finds-android-malware-pre-installed-on-hundreds-of-phones/.

[27] C. Osborne, "Android spyware campaign spreads across the middle east," June 2019, retrieved September 4, 2019 from https://www.zdnet.com/article/android-spyware-campaign-spreads-across-the-middle-east/.

[28] ——, "This new android ransomware infects you through sms messages," July 2019, retrieved September 4, 2019 from https://www.zdnet.com/article/this-new-android-ransomware-infects-you-through-sms-messages/.

[29] D. Palmer, "This new android ransomware infects you through sms messages," October 2017, retrieved September 4, 2019 from https://www.zdnet.com/article/this-nasty-new-android-ransomware-encrypts-your-phone-and-changes-your-pin/.

[30] A. Armando, A. Merlo, M. Migliardi, and L. Verderame, "Would you mind forking this process? a denial of service attack on android (and some countermeasures)," in *Information Security and Privacy Research*, D. Gritzalis, S. Furnell, and M. Theoharidou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 13–24.

[31] H. Huang, S. Zhu, K. Chen, and P. Liu, "From system services freezing to system server shutdown in android: All you need is a loop in an app," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 1236–1247. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813606

[32] R. Johnson, M. Elsabagh, and A. Stavrou, "Why software dos is hard to fix: Denying access in embedded android platforms," in *Applied Cryptography and Network Security*. Springer International Publishing, 2016, pp. 193–211.

[33] R. Johnson, M. Elsabagh, A. Stavrou, and J. Offutt, "Dazed droids: A longitudinal study of android inter-app vulnerabilities," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ser. ASIACCS '18. New York, NY, USA: ACM, 2018, pp. 777–791. [Online]. Available: http://doi.acm.org/10.1145/3196494.3196549

[34] T. Blsing, L. Batyuk, A. Schmidt, S. A. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," in *2010 5th International Conference on Malicious and Unwanted Software*, Oct 2010, pp. 55–62.

[35] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen, and C. Platzer, "Andrubis: Android malware under the magnifying glass," *Vienna University of Technology, Tech. Rep. TR-ISECLAB-0414-001*, 2014.

[36] M. N. P. Pravin, "Vetdroid: Analysis using permission for vetting undesirable behaviours in android applications," *Int. J. Innov. Emerg. Res. Eng*, vol. 2, pp. 131–136, 2015.

[37] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," in *NDSS*, 2015.

[38] L.-K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *USENIX security symposium*, 2012, pp. 569–584.

[39] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou, "A whitebox approach for automated security testing of android applications on the cloud," in *2012 7th International Workshop on Automation of Software Test (AST)*, June 2012, pp. 22–28.

[40] M. Zheng, M. Sun, and J. C. S. Lui, "Droidtrace: A ptrace based android dynamic analysis system with forward execution capability," in *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, Aug 2014, pp. 128–133.

[41] M. Lindorfer, M. Neugschwandtner, and C. Platzer, "Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 2, July 2015, pp. 422–433.

[42] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 641–660. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509549

[43] "pjlantz/droidbox: Dynamic analysis of Android apps," retrieved May 6, 2019 from https://github.com/pjlantz/droidbox.

[44] W.-C. Wu and S.-H. Hung, "Droiddolphin: A dynamic android malware detection framework using big data and machine learning," in *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*, ser. RACS '14. New York, NY, USA: ACM, 2014, pp. 247–252. [Online]. Available: http://doi.acm.org/10.1145/2663761.2664223

[45] 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*, ser. EuroSec '14. New York, NY, USA: ACM, 2014, pp. 5:1–5:6. [Online]. Available: http://doi.acm.org/10.1145/2592791.2592796

[46] 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*, ser. ASIA CCS '14. New York, NY, USA: ACM, 2014, pp. 447–458. [Online]. Available: http://doi.acm.org/10.1145/2590296.2590325

[47] 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*, ser. ACSAC '14. New York, NY, USA: ACM, 2014, pp. 216–225. [Online]. Available: http://doi.acm.org/10.1145/2664243.2664250

[48] K. Lim, Y. Jeong, S.-j. Cho, M. Park, and S. Han, "An android application protection scheme against dynamic reverse engineering attacks," *JoWUA*, vol. 7, no. 3, pp. 40–52, 2016.

[49] S.-T. Sun, A. Cuadros, and K. Beznosov, "Android rooting: Methods, detection, and evasion," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '15. New York, NY, USA: ACM, 2015, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2808117.2808126

[50] D. Maier, T. Müller, and M. Protsenko, "Divide-and-conquer: Why android malware cannot be stopped," in *2014 Ninth International Conference on Availability, Reliability and Security*. IEEE, 2014, pp. 30–39.

[51] W. Diao, X. Liu, Z. Li, and K. Zhang, "Evading android runtime analysis through detecting programmed interactions," in *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, ser. WiSec '16. New York, NY, USA: ACM, 2016, pp. 159–164. [Online]. Available: http://doi.acm.org/10.1145/2939918.2939926

[52] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. v. d. Veen, and C. Platzer, "Andrubis – 1,000,000 apps later: A view on current android malware behaviors," in *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, Sep. 2014, pp. 3–17.

[53] J. Oberheide and C. Miller, "Dissecting the android bouncer," *SummerCon2012, New York*, 2012.

[54] "Intent |Android Developers," retrieved April 9, 2019 from https://developer.android.com/reference/android/content/Intent.html.

[55] "Android app to disable apps that try to soft reboot the device," retrieved May 6, 2019 from https://github.com/endlessrecursion/antireboot.

[56] "An xposed module to defend against assorted Denial of Service attacks on Android," retrieved May 6, 2019 from https://github.com/endlessrecursion/dosdefense.

[57] "Daze: Null-fuzzing Intent Framework for Android," retrieved May 6, 2019 from https://github.com/Kryptowire/daze.

[58] R. Johnson, Z. Wang, C. Gagnon, and A. Stavrou, "Analysis of android applications' permissions," in *2012 IEEE Sixth International Conference on Software Security and Reliability Companion*, June 2012, pp. 45–46.

[59] R. Johnson, Z. Wang, A. Stavrou, and J. Voas, "Exposing software security and availability risks for commercial mobile devices," in *2013 Proceedings Annual Reliability and Maintainability Symposium (RAMS)*, Jan 2013, pp. 1–7.

[60] R. Johnson and A. Stavrou, "Forced-path execution for android applications on x86 platforms," in *2013 IEEE Seventh International Conference on Software Security and Reliability Companion*, June 2013, pp. 188–197.

[61] R. Johnson, M. Elsabagh, A. Stavrou, and V. Sritapan, "Targeted dos on android: How to disable android in 10 seconds or less," in *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, Oct 2015, pp. 136–143.

[62] M. Elsabagh, R. Johnson, and A. Stavrou, "Resilient and scalable cloned app detection using forced execution and compression trees," in *2018 IEEE Conference on Dependable and Secure Computing (DSC)*, Dec 2018, pp. 1–8.

[63] Z. Wang, R. Johnson, and A. Stavrou, "Attestation & authentication for usb communications," in *2012 IEEE Sixth International Conference on Software Security and Reliability Companion*, June 2012, pp. 43–44.

[64] R. Johnson, N. Kiourtis, A. Stavrou, and V. Sritapan, "Analysis of content copyright infringement in mobile application markets," in *2015 APWG Symposium on Electronic Crime Research (eCrime)*, May 2015, pp. 1–10.

[65] R. Johnson, A. Stavrou, and V. Sritapan, "Improving traditional android mdms with non-traditional means," in *2016 IEEE Symposium on Technologies for Homeland Security (HST)*, May 2016, pp. 1–6.

[66] R. Johnson, R. Murmuria, A. Stavrou, and V. Sritapan, "Pairing continuous authentication with proactive platform hardening," in *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, March 2017, pp. 88–90.

[67] J. Callaham, *Google made its best acquisition 13 years ago: Can you guess what it was?*, July 2018, retrieved March 20, 2019 from https://www.androidauthority.com/google-android-acquisition-884194/.

[68] D. Reisinger, *Google: Android was our best acquisition ever*, October 2010, retrieved February 23, 2019 from https://www.cnet.com/news/google-android-was-our-best-acquisition-ever/.

[69] "core/res/androidmanifest.xml - platform/frameworks/base - git at google," retrieved May 6, 2019 from https://android.googlesource.com/platform/frameworks/base/+/master/core/res/AndroidManifest.xml.

[70] "Androidxref," retrieved May 6, 2019 from http://androidxref.com/.

[71] S. Byford, *How China Rips Off the iPhone and Reinvents Android*, October 2018, retrieved February 23, 2019 from https://www.theverge.com/2018/10/17/17988564/chinese-phone-software-android-iphone-copy-ui.

[72] T. Vidas and N. Christin, "Sweetening android lemon markets: Measuring and combating malware in application marketplaces," in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '13. New York, NY, USA: ACM, 2013, pp. 197–208. [Online]. Available: http://doi.acm.org/10.1145/2435349.2435378

[73] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *Computer Security – ESORICS 2012*, S. Foresti, M. Yung, and F. Martinelli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 37–54.

[74] "Secure-D uncovers pre-installed suspicious application com.tct.weather on Alcatel Android smartphones manufactured by TCL - Upstrea," January 2019, retrieved June 1, 2019 from https://www.upstreamsystems.com/secure-d-uncovers-pre-installed-malware-alcatel-android-smartphones-manufactured-tcl/.

[75] L. Wei, Y. Liu, S. C. Cheung, H. Huang, X. Lu, and X. Liu, "Understanding and detecting fragmentation-induced compatibility issues for android apps," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[76] P. Mutchler, Y. Safaei, A. Doup, and J. Mitchell, "Target fragmentation in android apps," in *2016 IEEE Security and Privacy Workshops (SPW)*, May 2016, pp. 204–213.

[77] "Android certified - partners," retrieved February 23, 2019 from https://www.android.com/certified/partners/.

[78] "Google nexus," retrieved May 6, 2019 from https://en.wikipedia.org/wiki/Google_Nexus.

[79] "Verifying app behavior on the android runtime (art)," retrieved May 6, 2019 from https://developer.android.com/guide/practices/verifying-apps-art.

[80] J. Levin, *Android Internals: a Confectioner's Cookbook: Volume 1: the Power Users's View.* Cambridge, MA, USA: Technologeeks.com, 2015.

[81] K. Yaghmour, *Embedded Android: Porting, Extending, and Customizing.* "O'Reilly Media, Inc.", 2013.

[82] "Moto 360 adapter usb cable | How to Root Android," retrieved May 6, 2019 from https://source.android.com/setup/start/build-numbers.

[83] S. Samat, *A pop of color and more: updates to Androids brand*, August 2019, retrieved August 31, 2019 from https://www.blog.google/products/android/evolving-android-brand/.

[84] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "Cid: Automating the detection of api-related compatibility issues in android apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: ACM, 2018, pp. 153–163. [Online]. Available: http://doi.acm.org/10.1145/3213846.3213857

[85] D. Wu, X. Liu, J. Xu, D. Lo, and D. Gao, "Measuring the declared sdk versions and their consistency with api calls in android apps," in *Wireless Algorithms, Systems, and Applications*, L. Ma, A. Khreishah, Y. Zhang, and M. Yan, Eds. Cham: Springer International Publishing, 2017, pp. 678–690.

[86] "Request App Permissions," retrieved April 9, 2019 from https://developer.android.com/training/permissions/requesting.

[87] "Permissions overview | Android Developers," retrieved April 7, 2019 from https://developer.android.com/guide/topics/permissions/overview.

[88] *Alternative distribution options |Android Developers*, retrieved July 9, 2019 from https://developer.android.com/distribute/marketing-tools/alternative-distribution#unknown-sources.

[89] R. Amadeo, *Fortnites Android vulnerability leads to Google/Epic Games spat*, August 2018, retrieved February 23, 2019 from https://arstechnica.com/gadgets/2018/08/fortnites-android-vulnerability-leads-to-googleepic-games-spat/.

[90] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*. ACM, 2012, p. 3.

[91] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov, "Android permissions remystified: A field study on contextual integrity," in *USENIX Security Symposium*, 2015, pp. 499–514.

[92] M. Van Kleek, I. Liccardi, R. Binns, J. Zhao, D. J. Weitzner, and N. Shadbolt, "Better the devil you know: Exposing the data sharing practices of smartphone apps," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 2017, pp. 5208–5220.

[93] "App components," retrieved May 5, 2019 from https://developer.android.com/guide/components/fundamentals#Components.

[94] 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.

[95] "Application sandbox |android open source project," retrieved May 6, 2019 from https://source.android.com/security/app-sandbox.

[96] J. C. King, "A new approach to program testing," *SIGPLAN Not.*, vol. 10, no. 6, pp. 228–233, Apr. 1975. [Online]. Available: http://doi.acm.org/10.1145/390016.808444

[97] R. S. Boyer, B. Elspas, and K. N. Levitt, "Select&mdash;a formal system for testing and debugging programs by symbolic execution," *SIGPLAN Not.*, vol. 10, no. 6, pp. 234–245, Apr. 1975. [Online]. Available: http://doi.acm.org/10.1145/390016.808445

[98] A. J. Offutt, Z. Jin, and J. Pan, "The dynamic domain reduction procedure for test data generation," *Softw. Pract. Exper.*, vol. 29, no. 2, pp. 167–193, Feb. 1999. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1097-024X(199902)29:2⟨167::AID-SPE225⟩3.3.CO;2-M

[99] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081750

[100] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 59:1–59:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393666

[101] B. Korel, "Dynamic method for software test data generation," *Software Testing, Verification and Reliability*, vol. 2, no. 4, pp. 203–213, 1992. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.4370020405

[102] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, Aug 1990.

[103] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Software Testing Verification and Reliability*, vol. 9, no. 4, pp. 263–282, 1999.

[104] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 599–609. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635896

[105] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *IEEE Software*, vol. 32, no. 5, pp. 53–59, Sep. 2015.

[106] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[107] R. A. DeMilli and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, Sep. 1991.

[108] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, pp. 40–44, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2093548.2093564

[109] J. W. Duran and S. C. Ntafos, “An evaluation of random testing,” *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 438–444, July 1984.

[110] I. Bhatti, J. A. Siddiqi, A. Moiz, and Z. A. Memon, “Towards ad hoc testing technique effectiveness in software testing life cycle,” in *2019 2nd International Conference on Computing, Mathematics and Engineering Technologies (iCoMET)*, Jan 2019, pp. 1–6.

[111] J. C. King, “Symbolic execution and program testing,” *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976. [Online]. Available: http://doi.acm.org/10.1145/360248.360252

[112] C. Cadar and K. Sen, “Symbolic execution for software testing: Three decades later,” *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2408776.2408795

[113] C. A. Tovey, “A simplified np-complete satisfiability problem,” *Discrete Applied Mathematics*, vol. 8, no. 1, pp. 85 – 89, 1984. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0166218X84900817

[114] C. Cadar, D. Dunbar, and D. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[115] R. Ferguson and B. Korel, “The chaining approach for software test data generation,” *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 1, pp. 63–86, Jan. 1996. [Online]. Available: http://doi.acm.org/10.1145/226155.226158

[116] N. Gupta, A. P. Mathur, and M. L. Soffa, “Generating test data for branch coverage,” in *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, Sep. 2000, pp. 219–227.

[117] P. Godefroid, “Compositional dynamic test generation,” in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’07. New York, NY, USA: ACM, 2007, pp. 47–54. [Online]. Available: http://doi.acm.org/10.1145/1190216.1190226

[118] 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.* ACM, 2013, pp. 1043–1054.

[119] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, “Testing android apps through symbolic execution,” *SIGSOFT Softw. Eng. Notes*, vol. 37, no. 6, pp. 1–5, Nov. 2012. [Online]. Available: http://doi.acm.org/10.1145/2382756.2382798

[120] J. Jeon, K. K. Micinski, and J. S. Foster, “Symdroid: Symbolic execution for dalvik bytecode,” University of Maryland, Tech. Rep., 7 2012.

[121] N. Williams, B. Marre, and P. Mouy, “On-the-fly generation of k-path tests for c functions,” in *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, Sep. 2004, pp. 290–297.

[122] C. Cadar and D. Engler, "Execution generated test cases: How to make systems code crash itself," in *Model Checking Software*, P. Godefroid, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 2–23.

[123] J. Schtte, R. Fedler, and D. Titze, "Condroid: Targeted dynamic analysis of android applications," in *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, March 2015, pp. 571–578.

[124] S. Lu, P. Zhou, W. Liu, Y. Zhou, and J. Torrellas, "Spancoverage: Architectural support for increasing the path coverage of dynamic bug detection," University of Illinois at Urbana Champaign, Tech. Rep. UIUC-DCS-R-2005-2618, 2005.

[125] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 272–281. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134324

[126] S. Lu, P. Zhou, W. Liu, Y. Zhou, and J. Torrellas, "Pathexpander: Architectural support for increasing the path coverage of dynamic bug detection," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, Dec 2006, pp. 38–52.

[127] J. Wilhelm and T.-c. Chiueh, "A forced sampled execution approach to kernel rootkit identification," in *Recent Advances in Intrusion Detection*, C. Kruegel, R. Lippmann, and A. Clark, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 219–235.

[128] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 829–844. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671278

[129] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, *Automatically Identifying Trigger-based Behavior in Malware.* Boston, MA: Springer US, 2008, pp. 65–88. [Online]. Available: https://doi.org/10.1007/978-0-387-68768-1_4

[130] "GitHub - JesusFreke/smali: smali/baksmali," retrieved April 25, 2019 from https://github.com/JesusFreke/smali.

[131] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 224–234. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491450

[132] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 377–396.

[133] K. O. Elish, X. Shu, D. D. Yao, B. G. Ryder, and X. Jiang, "Profiling user-trigger dependence for android malware detection," *Computers Security*, vol. 49, pp. 255 – 273, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404814001631

[134] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *NDSS*, 2008.

[135] Z. Wang, J. Ming, C. Jia, and D. Gao, "Linear obfuscation to combat symbolic execution," in *Computer Security – ESORICS 2011*, V. Atluri and C. Diaz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 210–226.

[136] H. van der Merwe, O. Tkachuk, S. Nel, B. van der Merwe, and W. Visser, "Environment modeling using runtime values for jpf-android," *SIGSOFT Softw. Eng. Notes*, vol. 40, no. 6, pp. 1–5, Nov. 2015. [Online]. Available: http://doi.acm.org/10.1145/2830719.2830727

[137] P. McAfee, M. W. Mkaouer, and D. E. Krutz, "Cate: Concolic android testing using java pathfinder for android applications," in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 213–214. [Online]. Available: https://doi.org/10.1109/MOBILESoft.2017.35

[138] H. Xu, Y. Zhou, Y. Kang, and M. R. Lyu, "Concolic execution on small-size binaries: Challenges and empirical study," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2017, pp. 181–188.

[139] R. Baldoni, E. Coppa, D. C. Delia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 50:1–50:39, May 2018. [Online]. Available: http://doi.acm.org/10.1145/3182657

[140] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ser. ACSAC '16. New York, NY, USA: ACM, 2016, pp. 189–200. [Online]. Available: http://doi.acm.org/10.1145/2991079.2991114

[141] P. D. Coward, "Symbolic execution systems-a review," *Software Engineering Journal*, vol. 3, no. 6, pp. 229–239, Nov 1988.

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

[143] T. Chen, X. Lin, J. Huang, A. Bacchus, and X. Zhang, "An empirical investigation into path divergences for concolic execution using crest," *Security and Communication Networks*, vol. 8, no. 18, pp. 3667–3681, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1290

[144] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna, "Baredroid: Large-scale analysis of android apps on real devices," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC 2015. New York, NY, USA: ACM, 2015, pp. 71–80. [Online]. Available: http://doi.acm.org/10.1145/2818000.2818036

[145] "Security Tips |Android Developers," retrieved April 25, 2019 from https://developer.android.com/training/articles/security-tips.html.

[146] "Man-in-the-Disk: A New Attack Surface for Android Apps - Check Point Software," retrieved June 9, 2019 from https://blog.checkpoint.com/2018/08/12/man-in-the-disk-a-new-attack-surface-for-android-apps/.

[147] R. Johnson and A. Stavrou, "Factory resets and obtaining notifications on samsung android devices," 2016, retrieved August 5, 2017 from http://www.kryptowire.com/disclosures/CVE-2016-6910/Factory_Reset_and_Obtaining_Notifications_on_Samsung_Android_Devices.pdf.

[148] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *NDSS*, vol. 14, 2012.

[149] "CWE-926: Improper Export of Android Application Components (2.10)," retrieved May 6, 2019 from https://cwe.mitre.org/data/definitions/926.html.

[150] "<provider>|Android Developers," retrieved April 25, 2019 from https://developer.android.com/guide/topics/manifest/provider-element.html.

[151] "eBay for Android Content Provider Injection Vulnerability - NowSecure," retrieved December 11, 2018 from https://www.nowsecure.com/blog/2013/10/03/ebay-for-android-content-provider-injection-vulnerability/.

[152] "NVD - CVE-2014-1986," retrieved April 25, 2019 from https://nvd.nist.gov/vuln/detail/CVE-2014-1986.

[153] "NVD - CVE-2011-3918," retrieved April 25, 2019 from https://nvd.nist.gov/vuln/detail/CVE-2011-3918.

[154] "disclosures/GetSuperSerial.md at master rednaga/disclosures - GitHub," retrieved April 25, 2019 from https://github.com/rednaga/disclosures/blob/master/GetSuperSerial.md.

[155] W. Gasior and L. Yang, "Exploring covert channel in android platform," in *2012 International Conference on Cyber Security*, Dec 2012, pp. 173–177.

[156] L. Deshotels, "Inaudible sound as a covert channel in mobile devices," in *8th USENIX Workshop on Offensive Technologies (WOOT 14)*. San Diego, CA: USENIX Association, 2014. [Online]. Available: https://www.usenix.org/conference/woot14/workshop-program/presentation/deshotels

[157] D. Barrera, J. Clark, D. McCarney, and P. C. van Oorschot, "Understanding and improving app installation security mechanisms through empirical analysis of android," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '12. New York, NY, USA: ACM, 2012, pp. 81–92. [Online]. Available: http://doi.acm.org/10.1145/2381934.2381949

[158] 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*, ser. MobiSys '11. New York, NY, USA: ACM, 2011, pp. 239–252. [Online]. Available: http://doi.acm.org/10.1145/1999995.2000018

[159] *Enable multidex for apps with over 64K methods |Android Developers*, retrieved June 9, 2019 from https://developer.android.com/studio/build/multidex.

[160] "New exploit turns Samsung Galaxy phones into remote bugging devices |Ars Technica," May 2015, retrieved April 25, 2019 from https://arstechnica.com/security/2015/06/new-exploit-turns-samsung-galaxy-phones-into-remote-bugging-devices/.

[161] *Locale |Android Developers*, retrieved July 9, 2019 from https://developer.android.com/reference/java/util/Locale.

[162] "Mobile Country Codes (MCC) and Mobile Network Codes (MNC)," retrieved May 6, 2019 from http://mcc-mnc.com/.

[163] "adups fota," retrieved May 6, 2019 from https://www.adups.com/.

[164] M. Apuzzo and M. S. Schmidt, "Secret back door in some u.s. phones sent data to china, analysts say," November 2016, retrieved August 21, 2019 from https://www.nytimes.com/2016/11/16/us/politics/china-phones-software-security.html.

[165] *Android Security 2016 Year In Review*, March 2017, retrieved June 26, 2019 from https://source.android.com/security/reports/Google_Android_Security_2016_Report_Final.pdf.

[166] "Panasonic India Smartphones," retrieved May 6, 2019 from https://mobile.panasonic.com/in/.

[167] "Permissions Overview," retrieved May 6, 2019 from https://developer.android.com/guide/topics/permissions/overview.html#normal-dangerous.

[168] "Privileged Permission Whitelisting," retrieved May 6, 2019 from https://source.android.com/devices/tech/config/perms-whitelist.

[169] "core/res/AndroidManifest.xml - platform/frameworks/base - Git at Google," retrieved May 6, 2019 from https://android.googlesource.com/platform/frameworks/base/+/refs/tags/android-8.1.0_r1/core/res/AndroidManifest.xml.

[170] International Telecommunication Union, "Mobile Network Codes (MNC) for the international identification plan for public networks and subscriptions," November 2016, retrieved May 6, 2019 from https://www.itu.int/dms_pub/itu-t/opb/sp/T-SP-E.212B-2016-MSW-E.docx.

[171] V. Costamagna and C. Zheng, "Artdroid: A virtual-method hooking framework on android art runtime." in *IMPS@ ESSoS*, 2016, pp. 20–28.

[172] Z. Ning and F. Zhang, "Dexlego: Reassembleable bytecode extraction for aiding static analysis," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2018, pp. 690–701.

[173] D. Maier, M. Protsenko, and T. Muller, "A game of droid and mouse: the threat of split-personality malware on android," *Computers amp; Security*, vol. 54, pp. 2 – 15, 2015/10/. [Online]. Available: http://dx.doi.org/10.1016/j.cose.2015.05.001

[174] "Checking Device Compatibility with SafetyNet — Android Developers," retrieved April 25, 2019 from https://developer.android.com/training/safetynet/index.html.

[175] S. Arzt, S. Huber, S. Rasthofer, and E. Bodden, "Denial-of-app attack: Inhibiting the installation of android apps on stock phones," in *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones &#38; Mobile Devices*, ser. SPSM '14. New York, NY, USA: ACM, 2014, pp. 21–26. [Online]. Available: http://doi.acm.org/10.1145/2666620.2666621

[176] P. Ratazzi, Y. Aafer, A. Ahlawat, H. Hao, Y. Wang, and W. Du, "A systematic security evaluation of android's multi-user framework," *arXiv preprint arXiv:1410.7752*, 2014.

[177] *subprocess Subprocess management*, retrieved August 23, 2019 form https://docs.python.org/3/library/subprocess.html.

[178] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang, "Plagiarizing smartphone applications: Attack strategies and defense techniques," in *Engineering Secure Software and Systems*, G. Barthe, B. Livshits, and R. Scandariato, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 106–120.

[179] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '12. New York, NY, USA: ACM, 2012, pp. 317–326. [Online]. Available: http://doi.acm.org/10.1145/2133601.2133640

[180] *Potentially Harmful Application (PHAs) Categories |Play Protect*, retrieved August 23, 2019 form https://developers.google.com/android/play-protect/phacategories.

[181] "Dalvik bytecode |Android Open Source Project," retrieved April 25, 2019 from https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html.

[182] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "Edgeminer: Automatically detecting implicit control flow transitions through the android framework." in *NDSS*, 2015.

[183] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 193–204. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254088

[184] E. T. Barr, T. Vo, V. Le, and Z. Su, "Automatic detection of floating-point exceptions," in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '13. New York, NY, USA: ACM, 2013, pp. 549–560. [Online]. Available: http://doi.acm.org/10.1145/2429069.2429133

[185] V. Sharma, M. W. Whalen, S. McCamant, and W. Visser, "Veritesting challenges in symbolic execution of java," *SIGSOFT Softw. Eng. Notes*, vol. 42, no. 4, pp. 1–5, Jan. 2018. [Online]. Available: http://doi.acm.org/10.1145/3149485.3149491

[186] F. Shen, J. D. Vecchio, A. Mohaisen, S. Y. Ko, and L. Ziarek, "Android malware detection using complex-flows," *IEEE Transactions on Mobile Computing*, vol. 18, no. 6, pp. 1231–1245, June 2019.

[187] K. Vorobyov and P. Krishnan, "Combining static analysis and constraint solving for automatic test case generation," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, April 2012, pp. 915–920.

[188] A. Metzner, "Why model checking can improve wcet analysis," in *Computer Aided Verification*, R. Alur and D. A. Peled, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 334–347.

[189] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-c programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, K. Jensen and A. Podelski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 168–176.

[190] *Trail: The Reflection API (The Java® Tutorials)*, retrieved June 9, 2019 from https://docs.oracle.com/javase/tutorial/reflect/index.html.

[191] *Robolectric*, retrieved June 26, 2019 from http://robolectric.org/.

[192] M. Graa, N. C. Boulahia, F. Cuppens, and A. Cavalliy, "Protection against code obfuscation attacks based on control dependencies in android systems," in *2014 IEEE Eighth International Conference on Software Security and Reliability-Companion*, June 2014, pp. 149–157.

[193] 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," *SIGPLAN Not.*, vol. 49, no. 6, pp. 259–269, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2666356.2594299

[194] *secure-software-engineering/DroidBench: A micro-benchmark suite to assess the stability of taint-analysis tools for Android*, retrieved June 12, 2019 from https://github.com/secure-software-engineering/DroidBench.

[195] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks." in *NDSS*, vol. 14. Citeseer, 2014, p. 1125.

[196] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale," in *Trust and Trustworthy Computing*, S. Katzenbeisser, E. Weippl, L. J. Camp, M. Volkamer, M. Reiter, and X. Zhang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 291–307.

[197] 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 Trans. Comput. Syst.*, vol. 32, no. 2, pp. 5:1–5:29, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2619091

[198] "Realtime privacy monitoring on smartphones," retrieved May 6, 2019 from http://www.appanalysis.org/.

[199] "NVD - CVE-2018-14996," retrieved June 26, 2019 from https://nvd.nist.gov/vuln/detail/CVE-2018-14996.

[200] "NVD - CVE-2017-7978," retrieved April 25, 2019 from https://nvd.nist.gov/vuln/detail/CVE-2017-7978.

[201] "NVD - CVE-2017-5217," retrieved August 11, 2019 from https://nvd.nist.gov/vuln/detail/CVE-2017-5217.

[202] F. Cuadrado and J. C. Dueas, "Mobile application stores: Success factors, existing approaches, and future developments," *IEEE Communications Magazine*, vol. 50, no. 11, pp. 160–167, November 2012.

[203] Q. Xu, J. Erman, A. Gerber, Z. Mao, J. Pang, and S. Venkataraman, "Identifying diverse usage behaviors of smartphone apps," in *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, ser. IMC '11. New York, NY, USA: ACM, 2011, pp. 329–344. [Online]. Available: http://doi.acm.org/10.1145/2068816.2068847

[204] C. Weir, A. Rashid, and J. Noble, "Reaching the masses: A new subdiscipline of app programmer education," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 936–939. [Online]. Available: http://doi.acm.org/10.1145/2950290.2983981

[205] R. Balebako, A. Marsh, J. Lin, J. Hong, and L. Cranor, "The privacy and security behaviors of smartphone app developers," in *NDSS*, April 2014.

[206] A. Razaghpanah, A. A. Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill, "Studying tls usage in android apps," in *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '17. New York, NY, USA: ACM, 2017, pp. 350–362. [Online]. Available: http://doi.acm.org/10.1145/3143361.3143400

[207] "CWE-319: Cleartext Transmission of Sensitive Information," retrieved May 6, 2019 from https://cwe.mitre.org/data/definitions/319.html.

[208] "Security tips — Android Developers," retrieved May 6, 2019 from https://developer.android.com/training/articles/security-tips.

[209] "Deprecating Secure Sockets Layer Version 3.0," retrieved May 6, 2019 from https://tools.ietf.org/html/rfc7568.

[210] "OnCourse - boating & sailing - Android Apps on Google Play," retrieved April 9, 2019 from https://play.google.com/store/apps/details?id=com.marinetraffic.iais.

[211] "core/res/androidmanifest.xml - platform/frameworks/base - git at google," retrieved May 6, 2019 from https://android.googlesource.com/platform/frameworks/base/+/android-4.0.4_r2.1/core/res/AndroidManifest.xml.

[212] "core/res/androidmanifest.xml - platform/frameworks/base - git at google," retrieved May 6, 2019 from https://android.googlesource.com/platform/frameworks/base/+/android-4.1.1_r1/core/res/AndroidManifest.xml.

[213] "CWE-312: Cleartext Storage of Sensitive Information," retrieved June 12, 2019 from https://cwe.mitre.org/data/definitions/312.html.

[214] P. Andriotis, S. Li, T. Spyridopoulos, and G. Stringhini, "A comparative study of android users' privacy preferences under the runtime permission model," in *Human Aspects of Information Security, Privacy and Trust*, T. Tryfonas, Ed. Cham: Springer International Publishing, 2017, pp. 604–622.

[215] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "Viewdroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless &#38; Mobile Networks*, ser. WiSec '14. New York, NY, USA: ACM, 2014, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/2627393.2627395

[216] 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 *USENIX Security Symposium*, vol. 15, 2015.

[217] O. Gadyatskaya, A.-L. Lezza, and Y. Zhauniarovich, "Evaluation of resource-based app repackaging detection in android," in *Secure IT Systems*, B. B. Brumley and J. Röning, Eds. Cham: Springer International Publishing, 2016, pp. 135–151.

[218] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. La Spina, and E. Moser, "Fsquadra: Fast detection of repackaged applications," in *Data and Applications Security and Privacy XXVIII*, V. Atluri and G. Pernul, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014.

[219] M. Sun, M. Li, and J. C. S. Lui, "Droideagle: Seamless detection of visually similar android apps," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, ser. WiSec '15. New York, NY, USA: ACM, 2015, pp. 9:1–9:12. [Online]. Available: http://doi.acm.org/10.1145/2766498.2766508

[220] W. Zhou, X. Zhang, and X. Jiang, "Appink: Watermarking android apps for repackaging deterrence," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '13. New York, NY, USA: ACM, 2013, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2484313.2484315

[221] C. Ren, K. Chen, and P. Liu, "Droidmarking: Resilient software watermarking for impeding android application repackaging," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 635–646. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642977

[222] "ibotpeaches/apktool: A tool for reverse engineering android apk files," retrieved May 5, 2019 from https://github.com/iBotPeaches/Apktool.

[223] "master - platform/external/smali - Git at Google," retrieved April 25, 2019 from https://android.googlesource.com/platform/external/smali/+/master.

[224] L. Li, T. F. Bissyand, and J. Klein, "Simidroid: Identifying and explaining similarities in android apps," in *2017 IEEE Trustcom/BigDataSE/ICESS*, Aug 2017, pp. 136–143.

[225] L. Malisa, K. Kostiainen, and S. Capkun, "Detecting mobile application spoofing attacks by leveraging user visual similarity perception," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ser. CODASPY '17. New York, NY, USA: ACM, 2017, pp. 289–300. [Online]. Available: http://doi.acm.org/10.1145/3029806.3029819

[226] M. Aresu, D. Ariu, M. Ahmadi, D. Maiorca, and G. Giacinto, "Clustering android malware families by http traffic," in *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, Oct 2015, pp. 128–135.

[227] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[228] J. Rissanen, "A universal data compression system," *IEEE Transactions on Information Theory*, vol. 29, no. 5, pp. 656–664, Sep. 1983.

[229] C. Collberg, G. R. Myles, and A. Huntwork, "Sandmark-a tool for software protection research," *IEEE Security Privacy*, vol. 99, no. 4, pp. 40–49, July 2003.

[230] "VirusTotal," retrieved August 10, 2018 from https://www.virustotal.com.

[231] N. B. Brandt and M. Stamp, "Automating nfc message sending for good and evil," *Journal of Computer Virology and Hacking Techniques*, vol. 10, no. 4, pp. 273–297, 2014.

[232] G. Pink, S. Gerber, M. Fry, J. Kay, B. Kummerfeld, and R. Wasinger, "Safe execution of dynamically loaded code on mobile phones," in *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Springer, 2010, pp. 1–12.

[233] "Intent," retrieved May 6, 2019 from http://developer.android.com/reference/android/content/Intent.html#FLAG_ACTIVITY_MULTIPLE_TASK.

[234] "Activity," retrieved May 6, 2019 from http://developer.android.com/guide/topics/manifest/activity-element.html#lmode.

[235] "Low ram |android open source project," retrieved May 6, 2019 from https://source.android.com/devices/tech/ram/low-ram.html#lowmem.

[236] "Processes and threads," retrieved May 5, 2019 from http://developer.android.com/guide/components/processes-and-threads.html.

[237] "core/res/AndroidManifest.xml - platform/frameworks/base - Git at Google," retrieved June 12, 2019 from https://android.googlesource.com/platform/frameworks/base/+/refs/tags/android-5.0.0_r1/core/res/AndroidManifest.xml.

[238] "Android Core Window Manager Service," retrieved September 2, 2019 from https://android.googlesource.com/platform/frameworks/base/+/android-6.0.0_r1/services/core/java/com/android/server/wm/WindowManagerService.java.

[239] "Android Core Zygote Init," retrieved September 2, 2019 from https://android.googlesource.com/platform/frameworks/base/+/android-6.0.0_r1/core/java/com/android/internal/os/ZygoteInit.java.

[240] "Android Core Runtime Init," retrieved September 2, 2019 from https://android.googlesource.com/platform/frameworks/base/+/android-6.0.0_r1/core/java/com/android/internal/os/RuntimeInit.java.

[241] "Android Core Zygote," retrieved September 2, 2019 from https://android.googlesource.com/platform/frameworks/base/+/android-6.0.0_r1/core/jni/com_android_internal_os_Zygote.cpp.

[242] "Android Core Initialization Script," retrieved September 2, 2019 from https://android.googlesource.com/platform/system/core/+/android-6.0.0_r1/rootdir/init.rc.

[243] "SONY | eSupport - How to reset the Android TV to factory settings," retrieved August 16, 2015 from https://us.en.kb.sony.com/app/answers/detail/a_id/60594.

[244] "Moto 360 adapter usb cable | How to Root Android," retrieved May 6, 2019 from http://www.rootjunky.com/moto-360-adapter-usb-cable/.

[245] "Accessing SATV stock Recovery |nVidia Shield Android TV," retrieved May 6, 2019 from http://forum.xda-developers.com/shield-tv/general/accessing-satv-stock-recovery-t3300211.

[246] "Dashboards | Android Developers," retrieved May 6, 2019 from http://developer.android.com/about/dashboards/index.html.

[247] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby, "Smartphone energy drain in the wild: Analysis and implications," in *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '15. New York, NY, USA: ACM, 2015, pp. 151–164. [Online]. Available: http://doi.acm.org/10.1145/2745844.2745875

[248] *Fire OS Overview |Amazon Fire TV*, retrieved June 26, 2019 from https://developer.amazon.com/docs/fire-tv/fire-os-overview.html.

[249] *Gearing Up for a Flagship-Filled Holiday Quarter, Smartphone Shipments Grew 2.7% Year-Over-Year in the Third Quarter, According to IDC*, retrieved November 7, 2017 from https://www.idc.com/getdoc.jsp?containerId=prUS43193517.

[250] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," *SIGOPS Oper. Syst. Rev.*, vol. 22, no. 4, pp. 36–38, Oct. 1988. [Online]. Available: http://doi.acm.org/10.1145/54289.871709

[251] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on android," in *Network and Distributed System Security Symposium*, vol. 17, 2012, p. 19.

[252] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *USENIX Security Symposium*, vol. 30, 2011.

[253] R. Sasnauskas and J. Regehr, "Intent fuzzer: Crafting intents of death," in *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, ser. WODA+PERTEA 2014. New York, NY, USA: ACM, 2014, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/2632168.2632169

[254] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The impact of vendor customizations on android security," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 623–634. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516728

[255] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace, "Hare hunting in the wild android: A study on the threat of hanging attribute references," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 1248–1259. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813648

[256] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *2014 IEEE Symposium on Security and Privacy*, May 2014, pp. 409–423.

[257] V. F. Taylor and I. Martinovic, "To update or not to update: Insights from a two-year study of android app evolution," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17. New York, NY, USA: ACM, 2017, pp. 45–57. [Online]. Available: http://doi.acm.org/10.1145/3052973.3052990

[258] B. Carbunar and R. Potharaju, "A longitudinal study of the google app market," in *2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, Aug 2015, pp. 242–249.

[259] T. Book, A. Pridgen, and D. S. Wallach, "Longitudinal analysis of android ad library permissions," *arXiv preprint arXiv:1303.0857*, 2013.

[260] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag," in *Proceedings of International Conference on Advances in Mobile Computing &#38; Multimedia*, ser. MoMM '13. New York, NY, USA: ACM, 2013, pp. 68:68–68:74. [Online]. Available: http://doi.acm.org/10.1145/2536853.2536881

[261] A. Zhang, Y. He, and Y. Jiang, "Crashfuzzer: Detecting input processing related crash bugs in android applications," in *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*, Dec 2016, pp. 1–8.

[262] 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*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 661–671. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106286

[263] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer, "An empirical study of the robustness of inter-component communication in android," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, June 2012, pp. 1–12.

[264] "Intent Fuzzer," retrieved May 6, 2019 from https://www.nccgroup.trust/us/about-us/resources/intent-fuzzer/.

[265] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan, "Intentfuzzer: Detecting capability leaks of android applications," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '14. New York, NY, USA: ACM, 2014, pp. 531–536. [Online]. Available: http://doi.acm.org/10.1145/2590296.2590316

[266] R. Hay, O. Tripp, and M. Pistoia, "Dynamic detection of inter-application communication vulnerabilities in android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 118–128. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771800

Curriculum Vitae

Ryan Johnson graduated from Fayetteville-Manlius High School, Manlius, New York, in 2000. He received his Bachelor of Science in Information Technology from George Mason University in 2007. He received his Master of Science in Information Security and Assurance from George Mason University in 2010.