FIRMSCOPE: Automatic Uncovering of Privilege-Escalation Vulnerabilities in Pre-Installed Apps in Android Firmware

Mohamed Elsabagh
Kryptowire
`melsabagh@kryptowire.com`

Ryan Johnson
Kryptowire
`rjohnson@kryptowire.com`

Angelos Stavrou
Kryptowire
`astavrou@kryptowire.com`

Chaoshun Zuo
The Ohio State University
`zuo.118@osu.edu`

Qingchuan Zhao
The Ohio State University
`zhao.2708@osu.edu`

Zhiqiang Lin
The Ohio State University
`lin.3021@osu.edu`

Abstract

Android devices ship with pre-installed privileged apps in their firmware — some of which are essential system components, others deliver a unique user experience — that users cannot disable. These pre-installed apps are assumed to be secure as they are handpicked or developed by the device vendors themselves rather than third parties. Unfortunately, we have identified an alarming number of Android firmware that contain privilege-escalation vulnerabilities in pre-installed apps, allowing attackers to perform unauthorized actions such as executing arbitrary commands, recording the device audio and screen, and accessing personal data to name a few. To uncover these vulnerabilities, we built FIRMSCOPE, a novel static analysis system that analyzes Android firmware to expose unwanted functionality in pre-installed apps using an efficient and practical context-sensitive, flow-sensitive, field-sensitive, and partially object-sensitive taint analysis. Our experimental results demonstrate that FIRMSCOPE significantly outperforms the state-of-the-art Android taint analysis solutions both in terms of detection power and runtime performance. We used FIRMSCOPE to scan 331,342 pre-installed apps in 2,017 Android firmware images from v4.0 to v9.0 from more than 100 Android vendors. Among them, FIRMSCOPE uncovered 850 unique privilege-escalation vulnerabilities, many of which are exploitable and 0-day.

1 Introduction

Ever since its acquisition by Google in 2005, we have witnessed the rapid development and prodigious adoption of the Android platform. Today, it has become the dominant OS in the mobile domain with a market share of 76% [1] as well as the most widely used OS of any platform, surpassing even Windows [2]. Some key factors for the success of Android are its open ecosystem, assortment of available models, inclusion of Google's suite of apps, and a multitude of app marketplaces hosting millions of Android apps. Currently, anyone (including hardware vendors, device manufacturers, cellular service providers, social media companies, and mobile app developers) can develop and introduce apps into an Android mobile device, with the difference that apps introduced by app developers are typically downloaded from app stores by the users whereas the rest are directly introduced in the supply-chain and pre-installed in device firmware by manufacturers.

There are many reasons to introduce pre-installed apps in Android firmware. First, pre-installed apps often provide unique features and special services that distinguish a vendor or device from its competitors. Second, pre-installed apps come with pre-approved sensitive permissions and capabilities that are unavailable to user-level apps downloaded from app stores and often do not require user approval or consent to operate. In most cases, pre-installed apps typically run as the highly-privileged `system` user and cannot be uninstalled by the end user, even if a pre-installed app is found to be vulnerable, malicious, or simply unwanted. When users face these threats, their options are limited: wait for an update that hopefully fixes the vulnerable pre-installed app; or remove the app by rooting the device, potentially voiding its warranty and compromising its security.

Although intuitive from a marketing and ease of distribution perspectives, software distributed via firmware can expose end users to severe security risks unbeknownst to them and in many cases even to device manufacturers. This was partly shown in the past where Android vendor customization introduced vulnerabilities [3–6]. Of course, vulnerabilities can have different causes: software that is not tested, poorly tested, or purposefully designed to be easily exploitable or malicious. Even if the enterprise or an end user is diligent and follows a stringent security guidance, they may still be at risk of malicious or insecure apps that they did not install but were present on the device firmware when it was first delivered.

Therefore, there is a pressing need to address the supply-chain threats that stem from vulnerable or malicious software distributed through firmware images on mobile devices. To this end, this paper presents FIRMSCOPE, a scalable, comprehensive, and automated static taint analysis system to identify the firmware-borne vulnerabilities residing in pre-installed apps, both malicious and (un)intentionally insecure, present in Android firmware. Not all vulnerabilities are of our interest, and instead we particularly focus on detecting privilege-escalation vulnerabilities in pre-installed apps where the sensitive behavior is externally invokable (e.g., by a third-party app or a remote party). For instance, an unprivileged third-party app executing a command as the `system` user by exploiting an insecure interface of a pre-installed app.

While static taint analysis of mobile apps has been well studied (e.g., [7–11]) there are still enormous challenges (due to the complex OOP language constructs and also sophisticated control and data flows in Android APIs and callbacks) to the precision and scalability of the analysis when applied to real-world apps without source code access. For instance, how to precisely and efficiently track data flows through different objects, class fields, the Android framework APIs, and runtime callbacks. We have thus developed several novel techniques in FIRMSCOPE to handle these challenges in an efficient and precise manner suitable for large scale real-world app analysis. FIRMSCOPE achieves unprecedented detection power and performance. It incurs only 7 FPs and 11 FNs on the latest DroidBench 2.0 [12], and is 2X to 24X faster than FlowDroid [9], Amandroid [10], and DroidSafe [11].

We have evaluated FIRMSCOPE on 331,342 pre-installed apps from 2,017 Android firmware images from v4.0 to v9.0 covering more than 100 Android vendors, including the top 20 Android vendors worldwide. FIRMSCOPE has uncovered a total of 850 unique privilege-escalation vulnerabilities (3,483 total) in 1,547 firmware (77% of the analyzed images). These vulnerabilities included code and command injection; obtaining the modem logs and `Logcat` logs; wiping all user data from a device (i.e., factory reset); accessing, sending, and manipulating calls and text messages; (un)installing arbitrary apps; recording the device screen and microphone; among others. Coordinated disclosure of our findings is still ongoing. Thus far, we have disclosed 370 vulnerabilities in Android 7 to 9 to impacted vendors and received 147 CVEs.

In short, we make the following contributions:

- **Novel System**. We present FIRMSCOPE, a scalable, comprehensive, and automated system to identify privilege-escalation vulnerabilities residing in pre-installed apps in Android firmware at a large scale.

- **Efficient Techniques**. We significantly improve the scalability and accuracy of existing static taint analysis with an efficient on-demand custom flow-, context-, field-sensitive, and partially object-sensitive analysis.

- **Large-Scale Evaluation**. We evaluate FIRMSCOPE using hundreds of thousands of pre-installed apps from over two thousand firmware, in which it identified more than three thousand privilege-escalation vulnerabilities.

2 Background and Threat Model

Background. Android apps are composed of app components, which are functional code units that developers use to build an app. App components are implemented by extending certain framework classes containing a platform-managed lifecycle. App components serve as app entry points and can be started by the app itself, the system, and sometimes external apps, effectively permitting the sharing of code and possibly data. Components include Activities (GUI screens), Services, Broadcast Receivers, and Content Providers. Each Android app contains an `AndroidManifest.xml` file listing all the app's components and various configuration data.

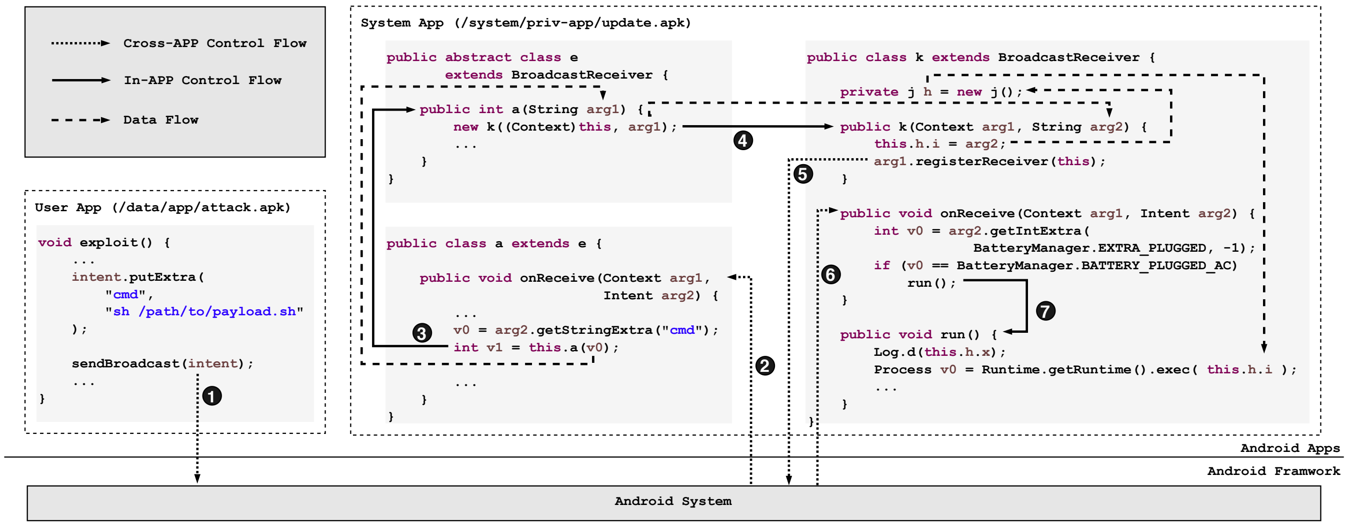
Android apps are sandboxed by the kernel where each app runs in its own isolated process and gets an isolated private storage space on the filesystem. By default, apps are not allowed to execute code in each other's context or access each other's data. There is no system-wide enclave for sensitive data. Instead, each app stores its own private information in its sandboxed storage space.

A pre-installed app is any app that comes pre-loaded with a firmware image. These apps can be non-essential apps that the device vendor decided to ship with the firmware (e.g., bloatware) or internal firmware apps implementing critical system components necessary for the proper functionality of the device (e.g., managing device settings, apps installation, and carrier negotiation). Pre-installed firmware apps are typically installed under `/system/app` and `/system/priv-app` on a read-only device partition whereas apps downloaded from app stores are installed under `/data`.

Pre-installed apps are privileged by design; some can run in the background as the privileged `system` user and cannot be uninstalled by the end-user. Android has four permission protection levels: Normal (lowest), Dangerous, Privileged, and Signature (highest).[1] Pre-installed apps can access highly sensitive device functionalities protected by Privileged- and Signature-level permissions that are not accessible by third-party apps downloaded from app stores. Due to the highly-privileged status of these apps, (un)intentional design or programming mistakes can facilitate confused deputy attacks, allowing unprivileged third-party apps, and perhaps remote entities, to abuse the capabilities of pre-installed apps and cross security boundaries set by the Android OS.

Threat Model. An Android firmware archive typically contains several modules, including a bootloader, the Android Linux kernel, the Android runtime framework, an embedded radio firmware, and pre-installed apps. We exclusively focus on discovering vulnerabilities in pre-installed firmware apps. Our objective is to use static analysis to uncover high-impact weaknesses (often posing as backdoors) in pre-installed apps

[1]There are additional permission levels reserved for the OS or require explicit granting over USB (e.g., Development, Instant, Installer, Verifier, Appop, etc.) that can be found in [13].



```
void exploit() {
    ...
    intent.putExtra(
        "cmd",
        "sh /path/to/payload.sh"
    );

    sendBroadcast(intent);
    ...
}
```

```
public abstract class e
        extends BroadcastReceiver {
    public int a(String arg1) {
        new k((Context)this, arg1);
        ...
    }
}

public class a extends e {

    public void onReceive(Context arg1,
                          Intent arg2) {
        ...
        v0 = arg2.getStringExtra("cmd");
        int v1 = this.a(v0);
        ...
    }
}
```

```
public class k extends BroadcastReceiver {

    private j h = new j();

    public k(Context arg1, String arg2) {
        this.h.i = arg2;
        arg1.registerReceiver(this);
    }

    public void onReceive(Context arg1, Intent arg2) {
        int v0 = arg2.getIntExtra(
                    BatteryManager.EXTRA_PLUGGED, -1);
        if (v0 == BatteryManager.BATTERY_PLUGGED_AC)
            run();
    }

    public void run() {
        Log.d(this.h.x);
        Process v0 = Runtime.getRuntime().exec( this.h.i );
        ...
    }
}
```

Figure 1: A running example simplified from a real-world pre-installed `system` app exhibiting a command injection vulnerability.

stemming from improper access control to privileged capabilities. These weaknesses result in *privilege-escalation vulnerabilities* that can be leveraged by local or remote parties to escalate privileges, bypass security boundaries set by the Android OS, and execute sensitive functionalities. For example, executing an attacker-controlled command in the context of another app's process allows the attacker to — at least — access that app's private code and data, bypassing sandboxing.

In particular, we focus on functionalities exported by firmware apps that can be invoked without user's awareness (i.e., minimal use of the app's GUI components, if any). In other words, we assume that functionalities invoked solely via an app's GUI are trusted. For instance, if the user launches a pre-installed factory-reset app (e.g., the Settings app) and requests to factory reset the device then that behavior should not be flagged as a vulnerability. The GUI is a legitimate interface and the user (the human using the phone) is always trusted. Therefore, anything visible to the user in a pre-installed app is considered trusted.

Again, FIRMSCOPE exclusively focuses on identifying privilege-escalation vulnerabilities in pre-installed apps. While it can also detect privacy leakage (e.g., personally identifiable information) in pre-installed apps, we exclude it from our scope since it is well-studied in prior research [9–11, 14–16]. In addition, analyzing the vulnerabilities in the kernel and the Android runtime framework is also out of scope. Interested readers can consult related works in this area, e.g., [15] for insecure memory management vulnerability discovery inside the Android kernel, PERISCOPE [17] and DR. CHECKER [18] for kernel driver vulnerability discovery, and [6, 19] for insufficient input validation in interfaces exposed by the Android runtime and its components.

3 Challenges and Key Insights

Running Example. We start by giving a running example in Figure 1 to motivate and illustrate some of the key challenges addressed by this work. This example is simplified from a real-world pre-installed system app that can be exploited by unprivileged third-party apps to execute arbitrary commands as the privileged `system` user. We omitted non-essential details for clarity. At a high level, this system app exposes an insecure interface, namely class `a`, which can receive an `Intent` sent from any app co-located on the device (step ❶). Once the `Intent` arrives at the `onReceive` method of class `a` (step ❷), an attacker-controlled string is extracted from the `Intent` and passed to method `a` of the parent class `e` (step ❸). Method `a` then creates a new object of type `k` and passes the incoming string to the constructor of `k` (step ❹). Inside the constructor of `k`, the string is saved to a field `h.i` then the class instance registers itself as a receiver for all battery events (step ❺). When a battery event is dispatched by the system, the `onReceive` callback in class `k` is invoked (step ❻), inside which the `run` method is called if the battery status indicates that the phone is plugged to an AC charger (step ❼). Inside method `run`, the attacker-controlled string in the `h.i` field is finally passed as the first argument to the `Runtime.exec` call which, in turn, executes the contents of the string as a command *with the vulnerable system app's own privileges*, i.e., as the highly-privileged `system` user.

Challenges. While various prior works have used static taint analysis to identify vulnerabilities in Android apps, e.g., FlowDroid [9], Amandroid [10], and DroidSafe [11], there are still enormous challenges that hinder their practicality, especially scaling to large apps and striking a good balance between detection power and runtime performance. At a high level, these challenges mainly stem from (*i*) the semantics of Java (e.g., how to resolve the points-to relations among objects to reason about data dependencies), and (*ii*) the semantics of the Android framework and runtime environment (e.g., how to handle data- and control-flow discontinuities due to calls to the Android APIs and callbacks from the runtime environment to an app). Below we elaborate on the important challenges and how we address them. Our solutions consist of several techniques that allow us to precisely and efficiently track data flows in real-world apps.

C1: Tracking Flows Through Class Fields. Android apps use rich OOP constructs that involve dynamic composite types. It is essential for an analysis to be able to track flows to class fields, otherwise sensitive flows may go undetected. In the example in Figure 1 the attacker-controlled command flows through the nested field `this.h.i` (steps ❹ to ❼) which would flow undetected unless the analysis properly and precisely handles data flowing through fields. In addition, it is important that information tracked *per field* can differentiate between tainted and untainted fields belonging to the same class type or instance. For example, the field `this.h.i` in our running example should be tainted but not `this.h.x`.

Prior works used context-insensitive CFGs and injected context information during taint tracking to track flows through fields [9], did not model nested fields [10], or used flow-insensitive analysis without binding fields to instances (i.e., tainting a field f of any object instance of type c taints all fields f in all object instances of type c) [11]. These approaches are either inherently imprecise or incur excessive runtime overhead rendering them inapplicable at large scale for real-world apps [11, 16, 20, 21]. In contrast, we track field reads and writes in a context-, flow-, and field-sensitive manner by keeping record of field definitions in our Def-Use analysis, using custom composite data-flow nodes representing field references, and encoding parent-child field flows and flows through their corresponding instance registers in the interprocedural dataflow graphs. Our construction allows us to model taint tracking as a direct path-finding problem — including flows through class fields — that can be efficiently solved without sacrificing precision or scalability.

C2: Full vs. Partial Object Sensitivity. Statically determining and tracking the actual types of class references requires full object-sensitivity which can be very prohibitive in terms of computational overhead and memory consumption since it requires identifying all object construction sites in an app and propagating the actual object type information on all forward and backward control-flow paths in the app in a flow- and context-sensitive manner [10, 20, 21].

To allow FirmScope to scale to large apps, we opted for a novel approach that involves only partial object-sensitivity by performing custom *on-demand* context validation by maintaining a per-tainted-path callstack and enforcing parent-child type-compatibility up the callstack between sources and sinks without full-blown type inference and tracking. By enforcing type-compatibility we ensure that the receiver class type at a call site is assignable from the class type of the callee. This allows us to offer object-sensitivity over sibling classes and single-definition virtual methods, but not for multi-definition virtual methods between a child and its ancestors. Our results show that partial object-sensitivity can be sufficient, achieving comparable object-sensitivity precision to prior solutions and scaling to real-world scenarios.

C3: Handling the Android Runtime Framework APIs. Android apps heavily depend on the runtime framework APIs which are not compiled into the apps and resemble a black box for static analysis. Modeling the entirety of the Android runtime can prove very expensive to develop, maintain, or even analyze. Prior solutions approached this challenge by either manually crafting extensive summarized flow rules (e.g., [9]), using approximate blanket policies for all runtime APIs (e.g., [22]), or implementing simplified behaviors of select APIs commonly used by apps (e.g., [11]). These approaches were imprecise and unrealistic to properly implement or maintain at a large scale [11, 16, 20], especially for firmware apps since they can access APIs not available to third-party apps.

In FirmScope, instead of modeling the entire Android runtime or using blanket policies, we modeled information flows only through Android framework methods and classes that can carry data indirectly. We argue that the internal functionalities of most of these framework APIs are not necessary for static information flow analysis. Instead, what matters is indirect data flows through these APIs, which can only happen by a few methods and by classes that flow data between their constructors/setters and member fields/getters. With this key observation, our models totaled less than one thousand lines of simple code (mostly assignment and return statements) and covered Android v4.0 to v9.0. Note that we do not need to model GUI-specific APIs since we consider user inputs as trusted, i.e., if an app executes a sensitive functionality based on a user request via the app's GUI, then that behavior is not a vulnerability, as mentioned in our threat model.

C4: Handling Asynchronous Callbacks. Android apps are multi-threaded by design and utilize asynchronous tasks to perform background operations. These asynchronous tasks often trigger callbacks that can result in indirect data flows depending on the runtime ordering of the callback events. For example, in Figure 1 there is a discontinuation in control- and data-flow at the code level between step ❺ and step ❻. This gap is then bridged by the system itself by invoking the callback in step ❻ when the battery status changes. Correctly accounting for all possible orderings in a flow- and context-sensitive analysis has proven to be a challenging task [9–11]. Not considering all possible event orderings may result in missing sensitive flows. On the other hand, considering all possible event permutations may incur prohibitive analysis overhead. Prior solutions have attempted to either model some of the callback orderings using dummy injected methods

[9, 10] or use flow-insensitive analysis [11], often sacrificing completeness and precision [20, 21, 23].

We overcome this by allowing information flowing through instance fields to cross method boundaries without sacrificing flow-sensitivity, enabling FIRMSCOPE to comprehensively cover all possible callback orderings without needing to inject dummy methods nor opt for a completely flow-insensitive analysis (details in §4.2.2). Such a configuration positions FIRMSCOPE in a unique spot compared to prior solutions. In fact, and to the best of our knowledge, this is the closest static approximation of how information flows through non-local fields at runtime due to callbacks on Android.

C5: Handling Inter-component Communication. Components in Android apps can communicate by sending and receiving messages (called Intents). Not accounting for outbound inter-component communication (ICC) may result in missing sensitive flows [10, 24]. While this is especially true for GUI apps since they depend on ICC for GUI transitions, it is uncommon for sensitive functionalities in firmware apps to span multiple components since these sensitive functionalities are often standalone and non-end-user facing.

A key insight here is that the Intent used in an outbound ICC call is often constructed within close code proximity to the ICC call. Therefore, we recover Intent targets by identifying the arguments at an Intent construction site and extracting the target component name by backward tracking the arguments through their Def-Use chains to their definition points. We then install data flow edges from the ICC call sending the Intent to the incoming ICC entry point in the target component receiving the Intent (e.g., calls to `getIntent(...)`). This approach offers a practical balance between precision and runtime overhead, whereas prior solutions that used involved techniques (e.g., [10, 11]) proved unusable on our data set due to their prohibitive runtime penalty (see §5).

4 Detailed Design

The workflow of FIRMSCOPE is illustrated in Figure 2. There are two phases of analysis: *preprocessing* (§4.1) and *static taint analysis* (§4.2). In this section, we present the detailed design of FIRMSCOPE based on the workflow of the analysis.

4.1 Preprocessing

FIRMSCOPE fundamentally relies on static taint analysis to identify vulnerabilities. To this end, a firmware image has to go through a number of preprocessing steps. In particular, when providing an Android firmware image, FIRMSCOPE unpacks and extracts the individual file-system images contained within the archive (§4.1.1). Then it extracts all system apps contained within an image file, analyzes each extracted app's manifest and metadata, identifies exported components, and disassembles the app's Dalvik Executable (DEX) files (§4.1.2).

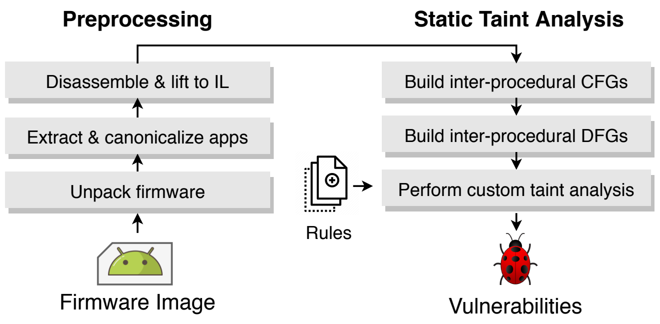

Preprocessing
Static Taint Analysis
Disassemble & lift to IL
Extract & canonicalize apps
Unpack firmware
Firmware Image
Rules
Build inter-procedural CFGs
Build inter-procedural DFGs
Perform custom taint analysis
Vulnerabilities

Figure 2: Workflow of FIRMSCOPE.

4.1.1 Unpacking Firmware Images

An Android firmware image is typically delivered as a compressed archive containing multiple file system images packing the raw contents of device partitions (block devices). Nearly all vendors store these block images using the standard Android Sparse Image (`SIMG`) format [25], which is a compressed `ext4` format that can be uncompressed using `Simg2img` [26] and mounted or traversed using tools such as `e2tools` [27]. However, after decompressing, it is not always a standard process to unpack, since some vendors used custom block image formats that require either vendor-provided or third-party unpacking tools, including: Huawei's `UApp` format, unpackable using `Splituapp` [28]; HTC's `RUU` archives, unpackable using the `HTC RUU DecryptTool` [29]; Sony's `.sin` archives, unpackable using `AnyXperia Dumper` [30]. Some vendors also used Sparse Data (`SDAT`) block images, which can be reconstructed into an `SIMG` using `Sdat2img` [31]. For vendors without available unpacking tools, we used simple heuristics that search for known `SIMG/ext4` headers and try to unpack from there. Often times, the image files were padded with extra headers that, once stripped, revealed standard images. These included images from vendors such as Motorola and vendors using custom signed images. Some vendors also used what is called a "sparse chunk" block image which is basically an `SIMG` file split into multiple files each with its own `SIMG` header. These can be converted to regular `SIMG` files using `Simg2img` then stripping any excess headers. The unpacking process is repeated recursively until all nested archives within a firmware image are extracted. We also search for and extract any `build.prop` and `default.prop` files found in the block images for bookkeeping purposes as these property files contain useful device information such as the build fingerprint, exact make and model, the OS version, and various build configurations.

4.1.2 Extracting and Disassembling Apps

From each unpacked firmware, we extract the Android framework directory which contains device-specific compiled binaries (`ODEX` and `OAT` files) necessary for disassembling pre-installed apps packaged with the firmware. We then extract apps from all block images by searching and extracting files matching any of the following formats DEX, ODEX, VDEX, OAT, JAR, and APK. More information about the different formats of pre-installed apps is available at [32].

Due to the various formats of pre-installed apps, we decided to transform all extracted apps into a stand-alone canonical APK format. These canonical apps typically contain one or more traditional DEX class files, a binary XML manifest file, and binary XML layout and resource files, in a well-structured ZIP archive. To this end, and for devices containing pre-compiled OAT apps, we used a combination of Oat2dex [33] and Baksmali [34], and with reference to the framework files extracted earlier, to extract the raw DEX classes embedded inside the ODEX/VDEX/OAT files. This step outputs assembled DEX classes or a directory of disassembled classes in Smali.[2]

We then disassemble any Dalvik bytecode into Smali and translate the Smali code into a custom three-address code intermediate language (IL) similar to Jasmin [35] and Jimple [36]. Using an IL is a standard step to facilitate analysis [16]. We omit the detail about the IL and the translation step for brevity. We also decompile the app's binary XML manifest file and extract metadata about the app and all its declared components. Specifically, we extract the app package name and version information, used and declared permissions, and the fully qualified names and types of all exported components along with any component-specific access permissions.

4.2 Static Taint Analysis

Next, FIRMSCOPE performs its static taint analysis on the app. In particular, it first builds inter-procedural Control-Flow Graphs (CFGs); reconstructs the class hierarchy and resolves calls (§4.2.1); infers Def-Use chains, and builds inter-procedural Data-Flow Graphs (DFGs) (§4.2.2); and finally performs custom flow-, context-, field-sensitive, and partially object-sensitive, taint analysis to identify vulnerable execution paths (§4.2.3). We present several constructions that allow us to model taint tracking under our threat model as a direct path-finding problem that can be efficiently solved using existing tools without sacrificing precision or scalability.

4.2.1 Building Inter-Procedural CFGs

The next step is to construct an inter-procedural CFGs (ICFGs) for the app. We build ICFGs that have both call-in and call-out edges that represent control-flow transfers to target methods either within the same class as the caller or in other classes in the app. Each node in the ICFG is a basic block consisting of a number of consecutive statements ending in a control-transfer statement (e.g., jump). The entry block to a method ICFG is labeled with the unique method signature. For try-catch blocks, we identify all statements inside a try-catch block that can throw an exception (i.e., statements using any of the following expressions: method call, array access, casting, new instance, and explicit throw statements) and add branch edges from each identified can-throw statement within the try-catch block to the node corresponding to the first statement in the catch block. This per-method construction process is repeated for every method in an app, resulting in a forest of ICFGs. We then build a holistic ICFG by re-pointing call-in and call-out edges to the entry blocks of their respective callee methods.

To install return-to-caller edges and call-out edges to virtual methods defined in parent/child classes, we start by constructing a precise class hierarchy for the whole app by adding is-a edges between child and parent classes and interfaces based on parent classes and interface references in the class definitions in the bytecode. For external classes and methods (referenced but not defined in the app) we generate skeleton classes, fields, and method bodies by inspecting the class hierarchy, field reads and writes, and call-out edges after building all ICFGs and adding any missing edges. Given the class hierarchy information, we then resolve all call and return sites in the ICFGs by building callee-sets based on the Dalvik method resolution semantics (see invoke-kind in [37]) and add ICFG edges to target callee entry points and caller return sites. Note that our resolution is static while the semantics in [37] describe runtime resolution. Therefore, we have to widen the target callee set while resolving non-static/non-direct calls in order to produce a complete ICFG. This widening may result in valid but not necessarily realizable (at runtime) call transfers. We filter out unrealizable call targets by narrowing down the callee set per call-site based on object types on the call stack during taint analysis.

4.2.2 Building Inter-Procedural DFGs

Next, we annotate each instruction in the ICFG forest with its Def-Use information, namely: incoming definition statements (INS), outgoing definition statements (OUTS), and referenced definitions (REFS). For each CFG, we produce an implicit directed acyclic graph (DAG) representing data dependency among the instructions in the CFG. The DAG is implicit in the sense that no actual graph is generated, instead def-use and use-def information are stored per each instruction in the CFG. We use the use-def information to build interprocedural data-flow graphs (IDFGs) atop which we perform taint analysis.

In particular, we developed a custom Def-Use analysis algorithm to correctly track definitions and uses involving class member fields as shown in Algorithm 1. Since Android apps are written in Java and heavily use OOP, it is important to correctly capture the data flow semantics through member fields. For instance, instructions writing to a member field not only define that member field but also modify the definition of the

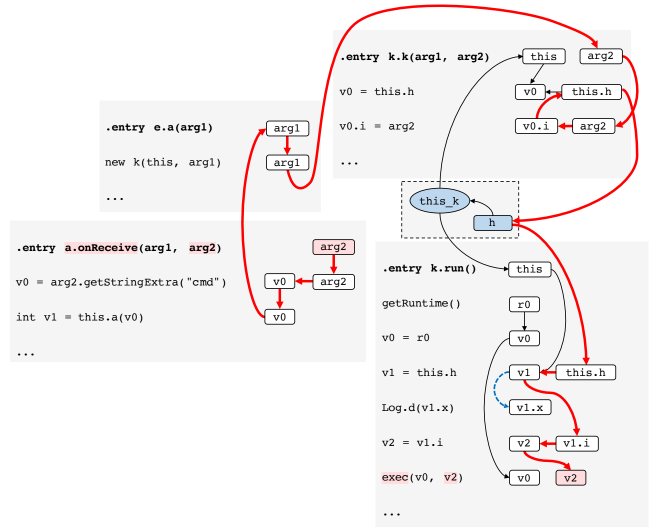
[2]Smali is a DEX assembler and also a Dalvik assembly language. Smali is to Dalvik as Assembly is to Machine code. In the rest of this paper, we use the terms "Dalvik instructions" and "Smali instructions" interchangeably unless explicitly stated otherwise.

Algorithm 1: Field-Aware Def-Use Analysis.

```
input  :CFG
output :CFG annotated with def-use and use-def chains
INS ← {{∅}}
OUTS ← {{∅}}
REFS ← {{∅}}
repeat
    foreach instruction $i \in CFG$ do
        INS[$i$] ← {all OUTS of predecessors of $i$}
        if $i$ reads a member field then
            REFS[$i$][$i$.rhs.reg] ← {$j \in$ INS | $j$ defines $i$.rhs.reg}
            REFS[$i$][$i$.rhs] ← {$j \in$ INS | $j$ defines $i$.rhs}
        else if $i$ writes a member field then
            REFS[$i$][$i$.lhs.reg] ← {$j \in$ INS | $j$ defines $i$.lhs.reg}
            REFS[$i$][$i$.rhs] ← {$j \in$ INS | $j$ defines $i$.rhs} ∪ {$j \in$ INS | $j$ defines a
              (sub)field of $i$.rhs}
        else
            foreach operand $r$ read by $i$ do
                REFS[$i$][$r$] ← {$j \in$ INS | $j$ defines $r$} ∪ {$j \in$ INS | $j$ defines a
                  (sub)field of $r$}
        if $i$ is return then
            KILLS[$i$] ← INS[$i$]
        else
            KILLS[$i$] ← {$j \in$ INS | $j$ defines an operand defined by $i$} ∪ {$j \in$ INS |
              $j$ defines a (sub)field of an operand defined by $i$}
            if $i$ is a move instruction then
                KILLS[$i$] ← KILLS[$i$] ∪ {$j \in$ INS | $j$ defines $r_0$ or $e_0$}
            GENS[$i$] ← {operand $r$ | $i$ writes $r$}
        OUTS[$i$] ← GENS[$i$] ∪ (INS[$i$] - KILLS[$i$])
until OUTS stops changing;
```

field instance register (the register holding the `this` pointer of the instance object) but without killing previous definitions of the instance object. For example, in Figure 1, the assignment to `this.h.i` inside the constructor of `k` kills all previous in-scope definitions of `this.h.i`, creates a new definition of `this.h.i`, and modifies the definition of `this.h` but without killing previous definitions of `this.h`. Similarly, instructions reading only an instance register (and not a member field of the instance) also implicitly read all member fields accessible via that instance register. Additionally, if an instance register is redefined, then that redefinition also kills the definitions of all fields accessible via that instance register. For example, if the statement `this.h = new j()` is added after the assignment to `this.h.i` in the constructor of `k` in Figure 1, that statement would kill previous definitions of `this.h`, `this.h.i` and `this.h.x`. Likewise, if `this.h` by itself flows to the entry of a method `foo`, e.g., via a call `foo(this.h)`, then `this.h.i` and `this.h.x` also flow to the entry of `foo`. Constructing Def-Use chains that correctly span reads and writes through member fields and their instance registers is essential for precise, field- and object-sensitive analysis.

We build an inter-procedural Data-Flow Graph (IDFG) as a multi-graph consisting of data flow nodes, each of which corresponds to one data flow source or destination operand in a corresponding instruction. We route data flow facts through all graph nodes by applying the data propagation semantics associated with each instruction (e.g., an assignment statement propagates data from operands



```
.entry e.a(arg1)
new k(this, arg1)
...
```

```
.entry k.k(arg1, arg2)
v0 = this.h
v0.i = arg2
...
```

```
.entry a.onReceive(arg1, arg2)
v0 = arg2.getStringExtra("cmd")
int v1 = this.a(v0)
...
```

```
.entry k.run()
getRuntime()
v0 = r0
v1 = this.h
Log.d(v1.x)
v2 = v1.i
exec(v0, v2)
...
```

Figure 3: A simplified illustrative figure of the IDFG constructed by FIRMSCOPE for the code in Figure 1. White nodes correspond to operands (registers, field references, and literals). Blue nodes correspond to instance nodes through which we route flows concerning the class instance and its fields. The tainted path from `arg2` (taint source) at the entry of `a.onReceive` to the `v2` argument of the `exec` call (taint sink) is highlighted. The dotted blue edge would result in a cross-field read from `this.h.i` to `this.h.x` which we reject during context validation using the callstack at the `Log.d` call (assuming `Log.d` was a sink).

on its RHS to the written register or field on its LHS). Figure 3 shows a simplified representation of the IDFG constructed by FIRMSCOPE corresponding to Figure 1. Our IDFG construction can be considered a specialized form of IFDS/IDE [38] with several extensions to handle Dalvik-specific semantics which we discuss in the following.

Static and Instance Fields. We route data flows through static fields (global fields bound to a class type rather than to a specific class instance) and instance fields without requiring an encapsulating method (to which the fields are deemed local) to carry the data flow. For static fields, we create special nodes in the IDFGs and route data flowing in and out of static fields to operands based on statements semantics. Note that the field reference used in an assignment statement may not necessarily reference the containing class inside which the field was declared. Therefore, for each referenced static field, we search up the class hierarchy for the node corresponding to the concrete declaration point of the field.

For each class containing instance fields, we add one global node representing the instance pointer of the class object instances, and one node for each field declared by the class (referred to as class field henceforth). Note that in Dalvik, the instance of `this` pointer is passed as the first argument to non-void calls. Then, for assignment statements that read instance fields, we add three flow edges: (*i*) from the local field reference to the LHS register (written); (*ii*) from the instance register of the read field reference to the LHS register; and (*iii*) from the global instance pointer node to the instance register. Likewise, for written instance fields, we flow the RHS register (read) to the local field reference and to the instance register, and flow the instance register to the global instance pointer. We also flow each class field to its corresponding read instance field references, and flow written instance field references that are live at return sites to their corresponding class fields. An example of this is illustrated in Figure 3 for the flows involving the `this.h.i` field in class `k` from Figure 1. These constructions allow us to precisely track all flows to and from instance fields and their aliases.

More importantly, these constructions also allow FIRMSCOPE to efficiently handle indirect flows through callbacks. Prior solutions (e.g., [9]) attempted to handle this by creating dummy encapsulating lifecycle methods to encapsulate some of the known permutations involving callbacks. Creating these lifecycle methods, however, requires correct modeling of the execution semantics involving these flows, including any app-defined callbacks, which cannot be automatically done at scale. Other solutions (e.g., [10]) ignored these field flows altogether, risking higher miss rates of sensitive flows. Using the example in Figure 1, FIRMSCOPE allows the field write `this.h.i = arg2` to cross the method boundary of the constructor `k` and flow back in at the `exec` call inside method `run`, organically handling what would happen at runtime when the system sends the battery status event that would trigger the `onReceive` callback in class `k`, without needing to model all possible callback permutations that may be triggered at runtime inside class `k`.

Synthetic Methods. The Dalvik compiler generates accessor synthetic methods (i.e., methods with a `synthetic` access modifier) for nested classes that declare private fields that are accessed by the enclosing class. These synthetic methods also have the `static` modifier, yet they implicitly take the nested instance pointer as the first argument and only read or write a nested field. Some common names of these accessor methods in Dalvik include `access$`, `-get`, `-set`, and `-wrap`. We identify and handle synthetic accessor methods by also routing flows through the nested instance registers and fields.

Inter-procedural Flows. We route flows across method boundaries by adding edges from argument registers in the caller's call site to parameter registers at the callee entry point on the forward edge, and adding edges from the return value register in the callee return site of non-void methods to the pseudo last-result register (i.e., `r0`) in the call site. This is repeated for every possible caller-callee in the resolved callee set, depending on the call resolution. If a non-resolved call is non-static, we assume arguments can flow to the instance receiver register (implicit first argument). In addition, we implemented models and stubs for common non-GUI Android 4.0 to 9.0 framework classes that can carry data from arguments to return values or fields, including indirect flows through the runtime APIs for threads, handlers, asynchronous tasks, and common native calls such as `java.lang.System.arrayCopy(...)`.[3] These stubs consisted of simple data-flow edges and Smali snippets to carry the data from read arguments to written arguments and return values.

4.2.3 Custom Taint Analysis

Given our IDFG construction, taint tracking is reduced to a graph traversal problem from a taint source to one or more taint sinks where taint sources and sinks are nodes in the multi-graph. During traversal, we apply validation rules to reject and prune paths that invalidate sensitivity goals. These context validation rules are essential for efficiency and precision since the classical constructions used in prior solutions do not necessarily scale in practice (see §3). For example, we cannot provide context-sensitivity using call-site stacks at a large scale since creating these stacks for every call site proved computationally prohibitive, especially when virtual calls are involved where multiple stacks would need to be maintained per call site. In addition, flows through fields must remain flow- and context-sensitive, but we cannot make a copy of all nodes corresponding to a field every time the field is accessed due to the obvious overhead this involves and the complexity of tracking and linking all these copies to where they are read and written throughout the app. Therefore, eventually, our analysis achieves context, flow, and field sensitivity, and partial object sensitivity:

- **Context-Sensitive.** We guarantee context-sensitivity by pairing each call instruction with its own return pseudo register, and maintaining a callstack overlaid atop each tainted path during taint tracking. We use this callstack to enforce control-flow on the backward edge by ensuring that a return node flows back only to its corresponding caller up the callstack. In addition, we prune unrealizable virtual call paths by ensuring that caller and callee types are compatible, i.e., the receiver class type at the call site is assignable from the class type of the callee.[4]

- **Flow- and Field-Sensitive.** Our IDFG construction is flow-sensitive since we take statement order into account and track flow facts per program point. Our construction is also field-sensitive since we track flows per class field. For cross-field flows that may occur when we flow field writes through the field instance node, we enforce field-sensitivity over these flows by recording field flows in the call frames

[3] A list of Android framework APIs can be found at: `https://raw.githubusercontent.com/aosp-mirror/platform_frameworks_base/master/api/current.txt`.

[4] We decided to only enforce type-compatibility rather than strict-typing (possible by tracking type information from object definition sites to object use sites) due to the computational cost incurred by dynamic type resolution which proved prohibitive for many of the apps in our data set without yielding significant improvements (orders of hours per app; less than 0.3% reduction of output space on a random sample of 100 apps).

of the callstack and ensuring that data flows out from the same field it previously flew in up the callstack.

- **Partially Object-Sensitive.** Ensuring type-compatibility makes our analysis object-sensitive over sibling and unrelated classes, object-sensitive over single definition methods, but object-insensitive over virtual methods with definitions both in a child and one of its ancestors.

- **Path-Insensitive** FIRMSCOPE is path-insensitive in the sense that, while it flows information according to the control-flow graph, the information flows irrespective of conditional dependencies that might exist between disjoint conditional branches. Path-sensitivity is a known hard problem with no absolute solution in practice [39].

Detection Rules. We developed a rules engine that takes detection rules as YAML files and invokes the taint engine as needed. Specifically, we implemented rules and plugins to detect the following privilege-escalation vulnerabilities: (*i*) command injection; (*ii*) arbitrary app installation/removal; (*iii*) code injection; (*iv*) factory reset of the device; (*v*) SMS injection, including accessing, sending, and manipulating text messages; (*vi*) device recording, including audio, video, and screen recording; (*vii*) log leakage to external storage or to other apps; (*viii*) `AT` Command injection; (*ix*) wireless settings modification; and (*x*) system settings modification. Some of these detectors involve additional analysis not discussed herewith, such as reachability and string analysis, necessary for capturing some vulnerability semantics. For instance, to detect leakage of Logcat logs to external storage the detection plugin detects cases where a vulnerable app contains bytecode segments that execute the "logcat" system command and writes its output to external storage either directly or perhaps by reading the output then writing it out using an output stream pointing to external storage. Likewise, the factory reset detection plugin needs to identify privileged apps that can be externally influenced into sending out the `MASTER_CLEAR` broadcast intent or writing the string "`recovery --wipe -data`" to `/cache/recovery/command` on the device followed by requesting a device reboot. Nevertheless, taint analysis remains the primary behavior modeling element of all detectors in this study. Finally, for each identified weakness, we report the vulnerable app meta data, the vulnerable components and their permissions, and the relevant inter-procedural traces through the app's bytecode instructions and global fields. Sample rules are shown in Appendix B.

5 Evaluation

We implemented FIRMSCOPE in 37 KSLOC of Cython, Python, and C/C++, in addition to 1.6 KSLOC of Shell Script. We used `graph-tool` [40] for efficient graph storage and manipulation. This section presents our evaluation results.

Table 1: Per-firmware-vendor count of firmware images, the number of apps analyzed per vendor, and the distribution of analyzed firmware Android versions (majors). Only vendors with more than 20 firmware images are shown.

Vendor	# Firmware	# Apps	v4	v5	v6	v7	v8	v9
Alcatel	31	4,390	15	3	9	3	1	0
Alps	48	9,557	15	7	22	3	1	0
ASUS	93	17,944	16	24	21	19	13	0
BLU	132	16,355	32	17	58	20	5	0
Coolpad	29	3,429	12	7	3	1	6	0
Doogee	25	3,310	3	3	10	9	0	0
Elephone	23	2,840	4	10	5	3	1	0
Google	372	54,057	0	1	0	175	142	54
HTC	39	9,361	15	11	11	2	0	0
Huawei	63	9,143	19	21	19	3	1	0
Infinix	29	4,476	0	0	8	8	13	0
Lenovo	82	9,209	52	14	12	3	1	0
Motorola	65	11,101	5	17	13	19	11	0
Panasonic	21	2,963	6	3	5	5	2	0
Samsung	219	61,457	9	1	65	71	55	18
TCL	33	5,309	6	6	16	4	1	0
Tecno	55	8,057	21	6	8	8	12	0
XBO	72	8,264	24	35	13	0	0	0
Xiaomi	102	21,331	11	10	36	14	20	11
ZTE	73	10,557	12	13	24	24	0	0
Other	411	58,232	126	82	83	55	65	0
Total	2,017	331,342	403	291	441	449	350	83
			19%	14%	22%	23%	17%	4%

We describe our primary dataset and experiment setup in §5.1, then present and discuss the uncovered privilege-escalation vulnerabilities by FIRMSCOPE in §5.2, followed by performance benchmarks and comparisons with closely related work in §5.3.

5.1 Dataset and Experiment Setup

We collected 2,017 publicly available (see Appendix A for acquisition details) stock Android firmware images from v4.0 to v9.0 covering more than 100 Android vendors in total, including the top 20 Android vendors worldwide. The firmware images contained 331,342 apps with 15,144 unique package names and 39,541 unique package versions. The details of this corpus are shown in Table 1.[5]

We deployed FIRMSCOPE on three servers each running 64-bit Ubuntu 18.04 on Intel(R) Xeon(R) E5-2630 v4 2.20GHz with 40 logical cores and 150 GiB of RAM. We implemented a pipeline using GNU Parallel [41] to manage jobs and distribute firmware images over the three servers, analyzing as many apps in parallel as possible to maintain a maximum server load of 80% with no memory swapping. We analyzed each firmware image in full, regardless of whether some of its apps might have appeared in other analyzed images.

[5] We use "vendor" to refer to the party responsible for providing (developing, building, and signing) a firmware image rather than the manufacturer of a device. For instance, HTC is the device manufacturer of Nexus 9, but Google is the device firmware vendor.

Table 2: Summary of discovered privilege escalation vulnerabilities and the percentage of vulnerable firmware.

Vulnerability	# Total	# Unique	% Firmware
Command Injection	1,420	211	41%
Wireless Settings Modification	901	212	26%
SMS Injection	232	63	7%
Screen Recording	207	63	6%
Factory Reset	169	48	5%
System Properties Modification	160	54	5%
App (Un)Installation	153	54	5%
Full Logcat Leakage	110	85	4%
Microphone Audio Recording	61	38	2%
AT Command Injection	55	17	2%
Code Injection	15	5	1%
Total	3,483	850	77%

Table 3: Breakdown of discovered unique vulnerabilities per firmware vendor. Only vendors with more than 20 firmware images are shown.

Vendor	# Total Vuln. per firmware	# Unique Vulnerabilities	Command Injection	Wireless Settings Modification	SMS Injection	Screen Recording	Factory Reset	System Properties Modification	App (Un)Installation	Full Logcat Leakage	Microphone Audio Recording	AT Command Injection	Code Injection
Alcatel	1.3	23	15	0	4	1	0	0	0	0	3	0	0
Alps	1.1	21	20	0	0	0	1	0	0	0	0	0	0
ASUS	3.7	132	41	53	5	0	11	17	2	2	0	1	0
BLU	2.2	63	43	5	7	4	0	0	1	0	1	2	0
Coolpad	3	54	22	4	2	3	0	7	6	1	1	7	1
Doogee	3.3	48	26	2	6	9	1	0	2	0	0	2	0
Elephone	2.7	36	26	1	0	2	0	2	3	0	0	1	1
Google	0.6	21	0	3	18	0	0	0	0	0	0	0	0
HTC	1.5	27	4	15	8	0	0	0	0	0	0	0	0
Huawei	1.2	22	2	12	0	0	0	0	0	6	1	1	0
Infinix	0.6	8	2	0	2	1	0	3	0	0	0	0	0
Lenovo	1.2	44	21	4	2	1	1	1	0	11	1	2	0
Motorola	0.6	24	0	7	10	0	0	7	0	0	0	0	0
Panasonic	2.3	34	27	0	0	2	0	1	0	1	0	3	0
Samsung	3.3	178	16	50	1	15	29	0	30	37	0	0	0
TCL	1.4	33	20	1	0	1	0	0	0	0	9	2	0
Tecno	1.2	28	17	0	3	2	0	2	0	0	0	2	2
XBO	2.2	37	29	0	2	3	0	1	1	0	0	1	0
Xiaomi	2.2	118	46	27	4	1	0	5	2	19	13	1	0
ZTE	0.6	23	11	0	3	0	1	6	0	0	0	2	0
Other	1.7	239	82	50	20	23	6	20	15	8	11	2	2

5.2 Privilege Escalation Vulnerabilities

Table 2 shows the summary of our findings. We discovered 850 unique privilege escalation vulnerabilities (3,483 total) spanning 77% of the analyzed firmware. We uniquified the vulnerabilities by arranging identical vulnerability bytecode traces into groups and counting each group only once. Command Injection vulnerabilities came at the top, impacting more than one third of firmware images. We provide the per-vendor breakdown by weakness category in Table 3.

Unsurprisingly, the most "vanilla" Android vendors, namely Google and Motorola, had no discovered weaknesses concerned with command or code execution. Since images from these vendors involve minimal customization over AOSP, the chances of introducing severe weaknesses are minimized. In particular, we inspected the SMS Injection vulnerabilities in Google firmware images and found that they all belonged to two specific versions of one system app on some Android 7.0 and 7.1 images in which the subscriber ID could be spoofed on dual-SIM devices. These weaknesses were fixed in subsequent versions of the app in later image builds. Also, the 3 Wireless Settings Modification vulnerabilities in Google images were all cases where an attacker could modify WiFi and Bluetooth configurations without permission and have been fixed in a recent commit [42, 43].

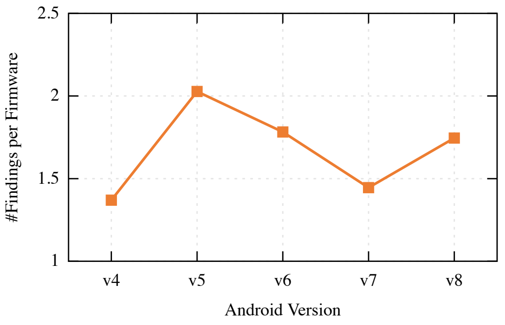


Figure 4: Estimate degree of vulnerability of pre-installed apps in different Android versions in the market measured as the number of findings normalized by the number of analyzed firmware images per Android version.

Figure 4 shows the estimate degree of vulnerability of pre-installed apps in different Android versions in the market measured as the number of findings normalized by the number of analyzed firmware images per Android version. The results suggest that there was a peak in Android weaknesses around versions 5 and 6. (We excluded Android 9 from the figure since its sample size was too small, in regard to the number of vendors, compared to other versions in our dataset.) We inspected the findings and found that a vulnerable version of ADUPS [44] and a number of diagnostic apps that were introduced in some Android 5 and 6 images were among the primary contributors to these weaknesses and that most of these vulnerable apps were retracted or patched in subsequent Android releases.[6] We also investigated the slight peak around version 8 and it appeared that the majority of the weaknesses were due to vulnerable apps and services introduced by chipset manufacturers spanning several device vendors (in some cases, more than 19 different device vendors had the same suite of vulnerable chipset manufacturer apps).

The aggregate numbers of identified vulnerabilities in AOSP vs. Vendor apps are shown in Table 4. About 92% of the vulnerabilities were in apps (375 unique package names) introduced by vendors, while only 8% of the vulnerabilities were in AOSP (18 unique AOSP package names). These results support the long-preached proposition that *AOSP-like images are more secure than vendor-customized images* since vendor modifications often introduce unforeseen weaknesses.

[6]ADUPS is a Shanghai based software provider for firmware over-the-air (FOTA) updating services.

Table 4: Number of total vulnerabilities in AOSP vs Vendor apps across all analyzed images and the number of unique package names of impacted apps.

	# Distinct Package Names	# Total Vulnerabilities Total	v4	v5	v6	v7	v8	v9
AOSP	18	289 (8%)	13	24	40	187	19	6
Vendor	375	3,194 (92%)	539	572	812	507	592	172

In the following, we present two representative case studies showing some of the vulnerabilities discovered by FIRMSCOPE and how they can be exploited. More case studies are presented in Appendix D.[7]

Vulnerability Case Study I: `SplendidCommandAgent`. A range of Asus firmware contained a severe vulnerability in a pre-installed platform app with a package name of `com.asus.splendidcommandagent`. This app exhibited inadequate access control, allowing any app co-located on the device to provide arbitrary commands for it to execute within its own context with `system` privileges. The app's manifest explicitly exported a bound service named `SplendidCommandAgentService` that receives and executes commands. A bound service allows client apps to bind to the service and call exposed methods using an interface returned from its `onBind` method, providing richer communication than unbound services. The command string flows to the `java.lang.Runtime.exec(String)` API where the entire command is externally controlled (i.e., there are no hard-coded components of the command).

The `SplendidCommandAgentService` service exposed an interface named `ISplendidCommandAgentService` containing a single method, `doCommand(String)`, that simply executed the string parameter as a command. To interact with the interface, the client app first binds to the service to obtain an `IBinder` reference. As the client app will likely lack the programming interface for the bound service, it inserts the command string to be executed in a `Parcel` and calls the `IBinder.transact(int, Parcel, Parcel, int)` method with the appropriate function number to initiate IPC, transferring the `Parcel` to the `SplendidCommandAgentService` bound service. The bound service extracts the string from the received `Parcel` object, and provides it to the `doCommand(String)` for execution using the `java.lang.Runtime.exec(String)` API.

Vulnerability Case Study II: `LovelyFont`. The `LovelyFont` apps consist of two related and interacting apps that effectively provide a covert local and remote Command and Control (C&C) channel. These two apps are devils in disguise. On the surface, they offer the functionality of allowing the user to change the default system font (presumably to more "lovely" variants). Interestingly though, the two apps stealthily run in the background and are hidden from the user (they do not appear in the device app launcher).

[7] All case studies detailed herewith have been responsibly disclosed to impacted vendors at least 90 days prior to the time of this writing.

Table 5: Count of vendors firmware containing a version of the `LovelyFont` apps containing at least one vulnerability.

Vendor	# Vulnerable Firmware Total	v5	v6	v7	v8
Tecno	20/55	2	2	3	13
Infinix	7/30	0	3	4	0
Lava	5/8	0	0	0	5
ASUS	3/94	0	0	3	0
Coolpad	3/35	0	0	0	3
Elephone	1/23	0	1	0	0
Haier	1/5	0	0	0	1
SWIPE	1/3	1	0	0	0
Walton	1/1	0	0	0	1

We describe the functionality in terms of the two most popular package names in our dataset for the `LovelyFont` app suite: (*i*) The `com.lovelyfont.defcontainer` system app provides interfaces to execute commands and dynamically execute code by its (*ii*) accompanying app, `com.ekesoo.lovelyhifonts`, which polls a remote server for commands to execute. If the `com.ekesoo.lovelyhifonts` app obtains any commands to execute, it uses an exported and accessible service component, `FontCoverService`, in the `com.lovelyfont.defcontainer` app to execute the commands.[8] The `FontCoverService` app component is exported and not permission-protected, allowing any app on the device to execute commands with elevated privileges. Table 5 provides the number of firmware by vendor that contained a vulnerable version of the `LovelyFont` apps.

The `LovelyFont` apps utilize HTTP communication for the endpoints involving the C&C channel, exposing the user to potential Man-In-The-Middle (MITM) attacks. Curiously, ADUPS also implemented their C&C channel over HTTP [44]. The `com.lovelyfont.defcontainer` also contained an exported app component called `FunctionService` that allowed local and remote execution of arbitrary Dalvik bytecode as the `system` user (a Code Injection vulnerability). The `FunctionService` component allowed a client app to provide the path to a DEX file, the fully-qualified class name, method name, and the type and values of parameters to be executed. This affords great power and flexibility to client apps using these capabilities, allowing them to obtain secret key material, such as the passwords of saved WiFi networks.

5.3 Benchmarking FIRMSCOPE Performance

We benchmarked FIRMSCOPE's taint analysis detection performance on the latest DroidBench 2.0 [12] against the state-of-the-art static taint analysis systems for Android in the literature, namely: FlowDroid [9], Amandroid [10], and DroidSafe [11]. DroidBench 2.0 contains over 100 hand-crafted benchmarks to assess the accuracy and precision of static and

[8] We also identified instances where the two `LovelyFont` apps had alternate package names with equivalent functionality.

Table 6: Summary of DroidBench 2.0 benchmark results. The benchmark consists of 100 real Positives (Ps) and 20 real Negatives (Ns).

Benchmarks	FlowDroid		Amandroid		DroidSafe		FIRMSCOPE	
	FP	FN	FP	FN	FP	FN	FP	FN
Aliasing	0	0	0	0	0	0	0	0
AndroidSpecific	0	2	0	2	0	1	0	1
ArraysAndLists	4	0	4	2	4	0	3	0
Callbacks	2	1	7	9	4	0	1	1
EmulatorDetection	0	0	0	0	0	0	0	0
FieldAndObjectSensitivity	0	0	0	0	2	0	0	0
GeneralJava	4	4	4	5	2	1	3	2
InterComponentComm.	0	8	2	0	0	1	0	4
Lifecycle	0	9	3	10	11	8	0	3
Reflection	0	0	0	2	0	0	0	0
Threading	0	0	0	0	5	4	0	0
Total (lower is better)	10	24	20	30	28	15	7	11

dynamic Android taint analysis tools, covering various analysis aspects and common Java and Android constructs.[9] We used FlowDroid v2.0 (without the IccTA extension), Amandroid v3.1.1, and the latest version of DroidSafe that was available as of June 2016. The benchmark summaries are shown in Table 6 (details are given in Appendix C). Overall, FIRMSCOPE had the highest detection power of all benchmarked solutions, incurring only 7 FPs and 11 FNs.[10] We emphasize that FIRMSCOPE's low number of FPs is paramount in practice, especially given the large number of firmware images and apps in the market.

We used the FPs incurred by FIRMSCOPE on DroidBench as sample reference cases to arrive at a rough estimate of the number of FPs in our real-world findings. We computed the worst-case scenario false discovery rate (FDR) in our findings by assuming all identified vulnerabilities containing any of the constructs appeared in these 7 cases were FPs, coming to a total of 451 findings out of 3,483 (12.95%). This accounts to less than 0.22360 FP per firmware and less than 0.00136 FP per app. We also manually inspected a sample of 400 identified vulnerabilities in Android 7 to 9 (by validating the semantics of their bytecode traces and checking for any incorrect tainted flows) and found less than 9% false discoveries.

5.3.1 Runtime Performance

The total start-to-finish runtime of FIRMSCOPE on the 2017 images was approximately 37 d (note that we had only three servers in our farm; this start-to-finish time is inverse proportional to the number of servers). In terms of per firmware and per app runtime, it took 4,901 s (81.7 min) per firmware on average with 50% of firmware images finishing in less than 3,342 s (55.7 min) and 95% finishing in less than 7,785 s (129.8 min). Apps took about 424 s (7.1 min) on average for static analysis (from analyzing metadata all the way to finishing the custom taint analysis) with 50% and 95% of the apps finishing in less than 53 s and 327 s (5.5 min), respectively.

On DroidBench, FlowDroid took 10 s on average, while Amandroid and DroidSafe took 2 min on average despite DroidBench minimalist apps. (DroidSafe took more than 10 min to finish on some of these benchmarks.) FIRMSCOPE consumed the least amount of time among the comparants, requiring less than 5 s per DroidBench app (2X faster than FlowDroid, 24X faster than Amandroid and DroidSafe).

Table 7: Summary of DIALDroid-Bench benchmark results with a 30 min timeout limit. The Runtime column gives the min., avg., and max. runtime in minutes.

Tool	# Analyzed	# Timeout	Runtime (min)
FlowDroid	18	12	0.42, 2, 30+
Amandroid	21	9	3.00, 8, 30+
DroidSafe	5	25	2.00, 5, 30+
FIRMSCOPE	30	0	0.05, 2, 12

We also measured the runtime performance on DIALDRoid-Bench, a representative sample of 30 real-world apps from Google Play [45] that appeared in related studies [20, 21, 46]. We used the same configuration as [20] by setting a maximum execution time of 30 min per app. The results of this benchmark are shown in Table 7. Neither FlowDroid, Amandroid, nor DroidSafe were able to process each of the apps within the allotted time: FlowDroid timed out on 12 apps, Amandroid on 9 apps, and DroidSafe on 25 apps. In contrary, FIRMSCOPE processed the 30 apps without exceeding the time limit, taking only 2 min on average and 12 min at a maximum. We observed similar results on the standard Android 9 Settings app (one of the largest apps that come pre-installed on every device) where all comparants but FIRMSCOPE timed out after 30 min while FIRMSCOPE analyzed it in less than 19 min. In summary, our results show that FIRMSCOPE outperforms prior work in terms of both detection power and scalability to large apps.

6 Discussion and Future Work

Extra Semantics. We currently handle calls invoked via reflection in a manner similar to [47]. Although this captures the majority of cases we have seen in practice, there are certain constructs that are not handled (e.g., nested reflection and reflection through native code). We do not model the internals of containers except for primitive array reads and writes. Writing a tainted value to a container eventually results in tainting the entire container (e.g., once the container crosses a method boundary) and vice versa. We also do not model all the semantics of throwing exceptions. For instance, apps can register a special global exception handler for all uncaught exceptions (e.g., using `java.lang.Thread .setDefaultUncaughtExceptionHandler(...)`) creating potential information flows from uncaught exceptions and

[9]DroidBench bundles an extensive set of benchmarks implemented as "miniature apps" with an established ground truth in terms of true positives and true negatives.

[10]We avoid discussing the overall precision and recall of the comparants since the reported DroidBench metrics are only useful for limited-scale comparisons and might not be commensurate with the overall performance of the measured tools on all categories of real-world apps.

their runtime contexts to the handler. We plan on supporting more of these semantics in future work.

Exploit Generation. While responsibly disclosing the findings, some vendors were only interested in exploit proof-of-concepts (PoCs) rather than the bytecode path traces discovered by FIRMSCOPE. Manually developing PoCs proved a rather laborious process. We are working on a novel system to help automatically synthesize PoC exploits to trigger vulnerabilities identified by FIRMSCOPE by means of selective symbolic execution and path condition analysis which we plan on presenting in a future work.

7 Related Work

Clearly, we are not the first to study the security of pre-installed apps within Android firmware. For instance, Woodpecker [48] made a first step in analyzing the security of Android firmware, in particular the permission models in pre-installed apps. By analyzing 8 popular Android phones, it discovered 11 out of 13 privileged permissions can be leaked. SEFA [3] studied the impact of vendor customizations, performing a permission and vulnerability analysis of pre-installed apps. With a provenance analysis of 10 popular firmware images from five major vendors, it discovered 85.78% of all preloaded apps are actually over-privileged.

ADDICTED [4] analyzed the device driver customizations in Android Device, and found such customization can introduce serious security flaws that allow unprivileged app to execute security-sensitive operations such as taking pictures. DroidRay [14] used a control flow signature matching approach to scan the security of 250 Android firmwares and 24,009 pre-installed apps and discovered that 7.6% of firmwares in their dataset contained pre-installed malware.

Vendor customization of the firmware can also introduce new attack surfaces such as hanging attribute references (Hares) [5] and privileged AT commands [49]. Using an automated analysis with 97 firmware images, HareHunter [5] discovered tens of thousands of likely Hares flaws. With a corpus of 2,000 Android firmware, Tian et. al. [49] uncovered 3,500 AT commands, many of which can be exploited via USB to execute dangerous operations such as bypassing the screen lock.

Most recently, Gamba et al. [50] made a comprehensive study, especially of privacy issues and use of "custom" and platform permissions by pre-installed apps in Android devices. They used an outsourcing approach to collect both the pre-installed apps and the network traffic from live devices. They discovered that advertising and data-driven services were among the primary incentives for vendors to include pre-installed apps, and argued that the privileged nature of these apps coupled with their obscurity and lack of transparency could potentially lead to backdoored access, which is exactly what FIRMSCOPE aims to uncover.

8 Conclusion and Final Remarks

Pre-installed apps in Android firmware present a potent attack vector due to their access to privileged permissions, potential widespread presence, and the fact that they often cannot be disabled or removed. We have presented FIRMSCOPE, an efficient and practical analysis system to uncover different types of vulnerabilities in pre-installed apps. By analyzing over 331,342 apps in 2,017 Android version 4 to 9 firmware images from over 100 Android vendors, FIRMSCOPE uncovered 3,483 privilege-escalation vulnerabilities including command injection, app installation, device recording, among others.

Coordinated Disclosure. We are following a coordinated vulnerability disclosure process in which we responsibly disclose our findings to vendors and allow them to test and offer corrective measures before any party releases detailed vulnerability or exploit information to the public. Some challenges in the disclosure process we encountered were: (*i*) Finding the appropriate procedure or contact point within an organization for reporting vulnerabilities. (*ii*) Most vendors requested PoC exploits instead of the bytecode traces produced by FIRMSCOPE, requiring us to manually go through the findings, assess exploitability, develop PoCs, and prepare exploitation reports laying out the technical details of each exploitable vulnerability. (*iii*) Absence of response from certain vendors precluding us from knowing if they confirm a vulnerability and plan to fix it. At the time of this writing, only Android versions 7 to 9 were covered by security updates. While our exploitability assessment and disclosure process is still ongoing, we have verified and reported more than 370 zero-day vulnerabilities in Android 7 to 9 and received 147 CVEs in the CVE-2019-15xxx block thus far, involving 30 vendors, 20 unique package names, and 26 unique package versions.

Acknowledgments

We thank Dimitris Tsiounis and Nick Kiourtis for providing technical assistance and for many constructive conversations. We also thank the anonymous reviewers and our shepherd, Benjamin Andow, for their insightful remarks.

This work was partially supported by the U.S. Department of Homeland Security (DHS) under contract 70RSAT19C00000007. The team at The Ohio State University was partially supported by National Science Foundation (NSF) award 1834215. Opinions expressed in this article are those of the authors and do not necessarily reflect the official policy or position of any agency of the U.S. government.

References

[1] *Mobile Operating System Market Share Worldwide | Statcounter Global Stats*, http://gs.statcounter.com/os-market-share/mobile/worldwide.

[2] *Operating System Market Share Worldwide*, retrieved March 27, 2019 from http://gs.statcounter.com/os-market-share.

[3] 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 & communications security*. ACM, 2013.

[4] 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*. IEEE, 2014.

[5] 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*. ACM, 2015.

[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. [Online]. Available: http://doi.acm.org/10.1145/3196494.3196549

[7] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications." in *NDSS*, 2011.

[8] 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.

[9] 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," *Acm Sigplan Notices*, vol. 49, no. 6, 2014.

[10] F. Wei, S. Roy, X. Ou *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014.

[11] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information Flow Analysis of Android Applications in DroidSafe." in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2015.

[12] *secure-software-engineering/DroidBench: A micro-benchmark suite to assess the stability of taint-analysis tools for Android*, retrieved October 4, 2019 from https://github.com/secure-software-engineering/DroidBench.

[13] *core/res/AndroidManifest.xml - platform/frameworks/base - Git at Google*, retrieved October 4, 2019 from https://android.googlesource.com/platform/frameworks/base/+/master/core/res/AndroidManifest.xml.

[14] M. Zheng, M. Sun, and J. Lui, "Droidray: a security evaluation system for customized android firmwares," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 2014.

[15] H. Zhang, D. She, and Z. Qian, "Android ion hazard: The curse of customizable memory management system," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[16] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, 2017.

[17] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "Periscope: An effective probing and fuzzing framework for the hardware-os boundary," in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2019.

[18] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "Dr. checker: A soundy analysis for linux kernel drivers," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[19] L. Zhang, Z. Yang, Y. He, Z. Zhang, Z. Qian, G. Hong, Y. Zhang, and M. Yang, "Invetter: Locating insecure input validations in android services," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243843

[20] F. Pauck, E. Bodden, and H. Wehrheim, "Do android taint analysis tools keep their promises?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3236024.3236029

[21] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018.

[22] J. Kim, Y. Yoon, K. Yi, and J. Shin, "ScanDal: Static analyzer for detecting privacy leaks in android applications," in *MoST 2012: Mobile Security Technologies 2012*, H. Chen, L. Koved, and D. S. Wallach, Eds. Los Alamitos, CA, USA: IEEE, May 2012.

[23] B. Reaves, J. Bowers, S. A. Gorski III, O. Anise, R. Bobhate, R. Cho, H. Das, S. Hussain, H. Karachiwala, N. Scaife *et al.*, "* droid: Assessment and evaluation of android application analysis tools," *ACM Computing Surveys (CSUR)*, vol. 49, no. 3, 2016.

[24] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android: An essential step towards holistic security analysis," in *22nd USENIX Security Symposium*, 2013.

[25] *Partitions and Images*, retrieved October 4, 2019 from https://source.android.com/devices/bootloader/partitions-images.

[26] *anestisb/android-simg2img: Tool to convert Android sparse images to raw images*, retrieved October 4, 2019 from https://github.com/anestisb/android-simg2img.

[27] *e2tools - utilities to manipulate files in an ext2/ext3 filesystem*, retrieved October 4, 2019 from https://www.unix.com/man-page/all/7/e2tools/.

[28] *superr/splituapp: Unpack UPDATE.APP files*, retrieved October 4, 2019 from https://github.com/superr/splituapp.

[29] *nkk71/HTC-RUU-Decrypt-Tool: Universal HTC RUU/ROM Decryption Tool*, retrieved October 4, 2019 from https://github.com/nkk71/HTC-RUU-Decrypt-Tool.

[30] *munjeni/anyxperia_dumper: Tool for dump any Sony Xperia image*, retrieved October 4, 2019 from https://github.com/munjeni/anyxperia_dumper.

[31] *xpirt/sdat2img: Convert sparse Android data image into filesystem ext4 image*, retrieved October 4, 2019 from https://github.com/xpirt/sdat2img.

[32] *Configuring ART | Android Open Source Project*, retrieved October 4, 2019 from https://source.android.com/devices/tech/dalvik/configure.

[33] *testwhat/SmaliEx: A wrapper to get de-optimized dex from odex/oat/vdex.*, retrieved October 4, 2019 from https://github.com/testwhat/SmaliEx.

[34] *JesusFreke/smali: smali/baksmali*, retrieved October 4, 2019 from https://github.com/JesusFreke/smali.

[35] J. Meyer, *Jasmin Assembler*, 1996, retrieved October 4, 2019 from http://jasmin.sourceforge.net/about.html.

[36] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782008

[37] *Dalvik Bytecode | Android Open Source Project*, retrieved October 4, 2019 from https://source.android.com/devices/tech/dalvik/dalvik-bytecode.

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

[39] S. Gulwani and G. C. Necula, "Path-sensitive analysis for linear arithmetic and uninterpreted functions," in *International Static Analysis Symposium*. Springer, 2004.

[40] T. P. Peixoto, "The graph-tool Python library," *figshare*, 2014. [Online]. Available: http://figshare.com/articles/graph_tool/1164194

[41] O. Tange, "GNU Parallel - The Command-Line Power Tool," *;login: The USENIX Magazine*, vol. 36, no. 1, Feb 2011. [Online]. Available: http://www.gnu.org/s/parallel

[42] *Android Security Bulletin—November 2018 | Android Open Source Project*, retrieved March 19, 2019 from https://source.android.com/security/bulletin/2018-11-01.

[43] *6409cf5c - platform/packages/apps/Settings - Git at Google*, retrieved March 21, 2019 from https://android.googlesource.com/platform/packages/apps/Settings/+/6409cf5c94cc1feb72dc078e84e66362fbecd6d5.

[44] R. Johnson, A. Stavrou, and A. Benameur, "All Your SMS & Contacts Belong To Adups & Others," 2017, blackhat USA. [Online]. Available: https://www.blackhat.com/docs/us-17/wednesday/us-17-Johnson-All-Your-SMS-&-Contacts-Belong-To-Adups-&-Others.pdf

[45] *DIALDroid Benchmark*, retrieved November 7, 2019 from https://github.com/amiangshu/dialdroid-bench.

[46] A. Bosu, F. Liu, D. D. Yao, and G. Wang, "Collusive data leak and more: Large-scale threat analysis of inter-app communications," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017.

[47] B. Livshits, J. Whaley, and M. S. Lam, "Reflection analysis for java," in *Asian Symposium on Programming Languages and Systems*. Springer, 2005.

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

[49] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, C. Raules, P. Traynor, H. Vijayakumar, L. Harrison, A. Rahmati, M. Grace *et al.*, "Attention spanned: Comprehensive vulnerability analysis of {AT} commands within the android ecosystem," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[50] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An analysis of pre-installed android software," in *2020 IEEE Symposium on Security and Privacy*. IEEE, 2020.

[51] *RuntimeException*, retrieved October 4, 2019 from https://docs.oracle.com/javase/8/docs/api/java/lang/RuntimeException.html.

[52] *Java Language Specifications: 12.4. Initialization of Classes and Interfaces*, retrieved October 4, 2019 from https://docs.oracle.com/javase/specs/jls/se7/html/jls-12.html#jls-12.4.

[53] *CVE - CVE-2018-9525*, retrieved March 19, 2019 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-9525.

A Android Firmware Acquisition

We downloaded firmware images from official vendor websites whenever possible. For vendors that did not have an official firmware download center, we downloaded their firmware images from third-party websites. Table A.1 shows the list of URLs from which we crawled firmware images in this study. We automated the firmware collection and downloading process by implementing web-crawlers using Scrapy[11].

[11] `https://scrapy.org/`

Table A.1: List of online resources from which we downloaded Android stock firmware images.

Vendor	URL
ASUS	https://www.asus.com/support/
Google	https://developers.google.com/android/images/
HTC	https://www.htc.com/us/support/rom-downloads.html
Huawei	https://consumer.huawei.com/en/support/
Oppo	https://oppo-au.custhelp.com
ZTE	https://www.ztedevices.com/en/support/
Other	https://androidmtk.com
	https://firmwarecare.com
	https://www.stockrom.net

B Sample Detection Rules

B.1 Sample Command Execution Rules

The following is a snippet of the YAML rules used for Command Execution detection:

```
impact: ...
CWEs: ...
description: ...
sources:
  - entry: onTransact(ILandroid/os/Parcel;Landroid/os/
       Parcel;I)Z
    operands: [2]
  - entry: onReceive(Landroid/content/Context;Landroid/
       content/Intent;)V
    operands: [2]
  - entry: onStartCommand(Landroid/content/Intent;II)I
    operands: [1]
  - ...
sinks:
  - call: Ljava/lang/Runtime;->exec(*)
  - call: Ljava/lang/ProcessBuilder;->command(*)
  - ...
```

B.2 Sample Factory Reset Detection Rules

The Factory Reset detector implements the following steps:

1. Detect a data flow path from a component entry point E to an API call site A_1 that sends a Broadcast Intent.

2. Detect the string `"android.intent.action.MASTER_CLEAR"` flowing to A_1 and falling on the same control-flow path from E to A_1.

3. Detect the string `"--wipe-data"` flowing to an API call site A_2 that writes it to a file output stream.

4. Detect the string `"/cache/recovery/command"` flowing to the construction site of the stream object at A_2.

5. Detect a data flow from the string `"recovery"` to an API call A_3 that reboots the device.

6. Finally, detect a control-flow path from A_1 to A_2 to A_3.

C Benchmarking Details and Discussion

Table C.2 shows the detailed DroidBench 2.0 results. FIRMSCOPE incurred one FP in each of 13.ArrayAccess2, 16.HashMapAccess, and 17.ListAccess1 due to tainted data flowing to a container object then non-tainted data flowing out of the same object. All tools in our comparison triggered FPs on these test cases. Statically tracking data through specific container elements is an NP-Hard problem since it requires full pointer analysis. FIRMSCOPE encountered no FPs in 12.ArrayAccess1 because we modeled reads and writes to primitive arrays and 12.ArrayAccess1 used constant hard-coded offsets to read and write to a primitive array.

In 33.Unregister1, a button click callback that leaks the IMEI via text message is registered then immediately unregistered in a subsequent statement, triggering a FP according to DroidBench under the proposition that the (unregistered) callback will not execute at runtime. Likewise, in 47.Exceptions3 a `catch` block leaks the IMEI when a `RuntimeException` is thrown and caught, but the code in the `try` block is presumed to not throw any `RuntimeExceptions` at runtime. We argue that this case is unrealistic since exceptions of the `RuntimeException` type are unchecked exceptions that can be thrown during the *normal operation* of the JVM [51], hence impossible to eliminate statically.[12] None of our findings involved a thrown exception.

Benchmarks 64.VirtualDispatch2 and 65.VirtualDispatch3 resulted in one FP each. In these two cases, a design pattern is employed in which a method is invoked on an object using a base type reference but the actual object is allocated and returned via a separate call to an allocator method (e.g., `Base b = allocActual(); b.foo();`. Due to the actual object type flowing on the backward edge from the nested call, these two cases would require inferring and propagating runtime type information on both the forward and backward control-flow edges which we currently do not support.

As for False Negatives, FIRMSCOPE missed 8.Parcel1 where a tainted string s is stored in an object O, O is serialized to an Android parcel where O implemented a custom logic that only serializes its $O.s$ field, the parcel is deserialized, one object O' is read from it, then $O'.s$ is sent over a text message. FIRMSCOPE could not track the taint information of $O.s$ through the serialized parcel bytes to $O'.s$. We argue that solutions cannot both detect this case yet maintain field sensitivity since tainting O or the entire parcel instead of specifically $O.s$ and its bytes in the parcel (because of the custom serialization logic implemented by O) will taint other fields in O and the parcel, destroying field sensitivity.

Another controversial case is 57.StaticInitialization3 in which an app has a taint source inside a static class initializer block and the data leakage only manifests in case the JVM happens to invoke the static initializer at a specific call site.

[12] An unchecked exception is an exception that does not need to be declared in a method's `throws` clause.

Both FlowDroid and FirmScope detected no leaks, while Amandroid and DroidSafe detected it. We argue that detecting this leakage is in fact problematic since an engine that can detect this case will inherently trigger FPs on all sources that may happen to reside in a static initializer code block, irrespective of whether the initializer has actually executed at a vulnerable call site (i.e., a leaking flow exists) or not. The reason is that the calls to static initializers (called `clinit` methods in Dalvik) are *implicit*, i.e., done by the runtime according to the runtime Java language specifications [52] and do not appear in the app's bytecode. Therefore, it is impossible for a static engine to correctly determine whether a class is initialized and the order at which that initialization had occurred to judge whether the leaking execution path originating inside `clinit` would actually execute or not.

Finally, FirmScope missed 67.ActivityCommunication1, 72.ActivityCommunication7, 73.ActivityCommunication8, 74.ActivityCommunication9, and 83.UnresolvableIntent1, as they involved contrived ICC situations that require elaborate Intent target resolution and propagation on backward control-flow edges which FirmScope did not perform.

Table C.2: DroidBench 2.0 benchmark details (100 Ps and 20 Ns).

App Name	P	FlowDroid		Amandroid		DroidSafe		FirmScope	
		FP	FN	FP	FN	FP	FN	FP	FN
1.Merge1	0	0	0	0	0	0	0	0	0
2.ApplicationModeling1	1	0	0	0	0	0	0	0	0
3.DirectLeak1	1	0	0	0	0	0	0	0	0
4.InactiveActivity	0	0	0	0	0	0	0	0	0
5.Library2	1	0	0	0	0	0	0	0	0
6.LogNoLeak	0	0	0	0	0	0	0	0	0
7.Obfuscation1	1	0	0	0	0	0	1	0	0
8.Parcel1	1	0	0	0	1	0	0	0	1
9.PrivateDataLeak3	1	0	1	0	1	0	0	0	0
10.PublicAPIField1	1	0	0	0	0	0	0	0	0
11.PublicAPIField2	2	0	1	0	0	0	0	0	0
12.ArrayAccess1	0	1	0	1	0	1	0	0	0
13.ArrayAccess2	0	1	0	1	0	1	0	1	0
14.ArrayCopy1	1	0	0	0	1	0	0	0	0
15.ArrayToString1	1	0	0	0	1	0	0	0	0
16.HashMapAccess1	0	1	0	1	0	1	0	1	0
17.ListAccess1	0	1	0	1	0	1	0	1	0
18.MultidimensiolArray1	1	0	0	0	0	0	0	0	0
19.AnonymousClass1	2	0	0	2	2	0	0	0	0
20.Button1	1	0	0	0	0	0	0	0	0
21.Button2	2	1	0	0	0	1	0	0	0
22.Button3	1	0	0	0	1	0	0	0	0
23.Button4	1	0	0	0	1	0	0	0	0
24.Button5	1	0	1	0	1	0	0	0	1
25.LocationLeak1	2	0	0	3	2	0	0	0	0
26.LocationLeak2	2	0	0	3	2	0	0	0	0
27.LocationLeak3	2	0	0	2	2	0	0	0	0
28.MethodOverride1	1	0	0	0	0	0	0	0	0
29.MultiHandlers1	2	0	0	4	0	0	0	0	0
30.Ordering1	0	0	0	3	0	2	0	0	0
31.RegisterGlobal1	1	0	0	0	1	0	0	0	0
32.RegisterGlobal2	1	0	0	0	1	0	0	0	0
33.Unregister1	0	1	0	1	0	1	0	1	0
34.ContentProvider1	2	0	0	0	0	0	0	0	0
35.IMEI1	0	0	0	0	0	0	0	0	0
36.PlayStore1	2	0	0	0	0	0	0	0	0
37.FieldSensitivity1	0	0	0	0	0	0	0	0	0
38.FieldSensitivity2	0	0	0	0	0	0	0	0	0
39.FieldSensitivity3	1	0	0	0	0	0	0	0	0
40.FieldSensitivity4	0	0	0	0	0	1	0	0	0
41.InheritedObjects1	1	0	0	0	0	0	0	0	0
42.ObjectSensitivity1	0	0	0	0	0	0	0	0	0
43.ObjectSensitivity2	0	0	0	0	0	1	0	0	0
44.Clone1	1	0	0	0	0	0	0	0	0
45.Exceptions1	1	0	0	0	0	0	0	0	0
46.Exceptions2	1	0	0	0	0	0	0	0	0
47.Exceptions3	0	1	0	1	0	1	0	1	0
48.Exceptions4	1	0	0	0	1	0	0	0	0
49.FactoryMethods1	2	0	0	1	2	0	0	0	0
50.Loop1	1	0	0	0	0	0	0	0	0
51.Loop2	1	0	0	0	0	0	0	0	0
52.Serialization1	1	0	1	0	1	0	0	0	0
53.SourceCodeSpecific1	1	0	0	4	0	0	0	0	0
54.StartProcessWithSecret1	1	0	0	0	1	0	0	0	0
55.StaticInitialization1	1	0	1	0	0	0	0	0	0
56.StaticInitialization2	1	0	0	0	0	0	0	0	0
57.StaticInitialization3	1	0	1	0	0	0	0	0	1
58.StringFormatter1	1	0	1	0	1	0	0	0	1
59.StringPatternMatching1	1	0	0	0	0	0	0	0	0
60.StringToCharArray1	1	0	0	0	0	0	0	0	0
61.StringToOutputStream1	1	0	0	0	0	0	0	0	0
62.UnreachableCode	0	0	0	0	0	0	0	0	0
63.VirtualDispatch1	1	1	0	1	0	1	1	0	0
64.VirtualDispatch2	1	1	0	0	0	0	0	1	0
65.VirtualDispatch3	0	1	0	0	0	0	0	1	0
66.ActivityCommunication1	1	0	0	0	0	0	0	0	0
67.ActivityCommunication2	1	0	1	2	0	0	0	0	1
68.ActivityCommunication3	1	0	1	0	0	0	0	0	0
69.ActivityCommunication4	1	0	1	0	0	0	0	0	0
70.ActivityCommunication5	1	0	1	0	0	0	0	0	0
71.ActivityCommunication6	1	0	1	0	0	0	0	0	0
72.ActivityCommunication7	1	0	1	0	0	0	0	0	1
73.ActivityCommunication8	1	0	1	0	0	0	0	0	1
74.BroadcastTaintAndLeak1	1	0	1	1	0	0	1	0	1
75.ComponentNotInManifest1	0	0	0	0	0	0	0	0	0
76.EventOrdering1	1	0	1	0	1	0	0	0	0
77.IntentSink1	1	0	0	1	0	0	0	0	0
78.IntentSink2	1	0	0	1	0	1	0	0	0
79.IntentSource1	0	0	0	3	0	0	0	0	0
80.ServiceCommunication1	1	0	1	0	0	0	0	0	1
81.SharedPreferences1	1	0	1	0	1	0	0	0	0
82.Singletons1	1	0	0	0	1	0	0	0	0
83.UnresolvableIntent1	2	0	0	0	0	0	0	0	2
84.ActivityLifecycle1	1	0	0	0	0	0	0	0	0
85.ActivityLifecycle2	1	0	0	0	0	0	0	0	0
86.ActivityLifecycle3	1	0	0	0	0	0	0	0	0
87.ActivityLifecycle4	1	0	0	0	0	0	0	0	0
88.ActivitySavedState1	1	0	1	0	1	1	1	0	0
89.ApplicationLifecycle1	1	0	0	0	1	1	1	0	0
90.ApplicationLifecycle2	1	0	0	0	1	1	1	0	0
91.ApplicationLifecycle3	1	0	0	0	1	1	1	0	0
92.Asynch*EventOrdering1	1	0	0	0	0	1	1	0	0
93.BroadcastRec*Lifecycle1	1	0	1	0	0	0	0	0	0
94.BroadcastRec*Lifecycle2	1	0	1	0	1	0	0	0	0
95.EventOrdering1	1	0	0	0	0	0	0	0	0
96.FragmentLifecycle1	1	0	0	0	0	1	1	0	0
97.FragmentLifecycle2	1	0	1	0	1	1	0	0	0
98.ServiceLifecycle1	1	0	1	0	0	1	1	0	0
99.ServiceLifecycle2	1	0	0	0	0	1	0	0	0
100.SharedPref*Changed1	1	0	1	0	1	1	1	0	0
101.Reflection1	1	0	0	0	0	0	0	0	0
102.Reflection2	1	0	0	0	1	0	0	0	0
103.Reflection3	1	0	0	0	1	0	0	0	0
104.Reflection4	1	0	0	0	0	0	0	0	0
105.AsyncTask1	1	0	0	0	0	1	1	0	0
106.Executor1	1	0	0	0	0	1	1	0	0
107.JavaThread1	1	0	0	0	0	1	0	0	0
108.JavaThread2	1	0	0	0	0	1	1	0	0
109.Looper1	1	0	0	0	0	1	1	0	0
Total (lower is better)		10	24	20	30	28	15	7	11

D More Vulnerability Case Studies

D.1 Unauthorized Settings Modification in AOSP Settings App

Any software vulnerability introduced by Android vendor code generally limits the scope of affected devices to those manufactured by the vendor. On the other hand, a software vulnerability that occurs in in AOSP code has a more severe impact since the vulnerability is usually inherited by all vendors, thus greatly enhancing the scope of affected devices. FIRMSCOPE discovered a vulnerability in the AOSP Settings app, with a package name of `com.android.settings`, in certain versions of Android 9.0 that allows a local app to toggle (enable/disable) the following options without the appropriate access permissions: Wi-Fi, Wi-Fi calling, Bluetooth, and Zen Mode. These capabilities allow an unprivileged app to mediate access to protected resources and perform a local Denial of Service (DoS) attack.

The Settings app serves as a critical nexus for modifying the device settings. This vulnerability is caused by an unprotected broadcast receiver named `SliceBroadcastReceiver` in the Settings app. This component is exported by default and is not protected by an access permission. When FIRMSCOPE discovered the vulnerability, it was already publicly known and had been assigned CVE-2018-9525 [53] and `A-111330641` by Google with a severity rating of *high* [42]. Google remediated the vulnerability by not exporting the `SliceBroadcastReceiver` app component, making it inaccessible to external apps [43].

D.2 Factory Resetting the Device

A "factory reset" operation will wipe the data and cache partitions. This removes any apps the user has installed and any other user or app data that the user does not have synced externally. An unintentional factory reset can present an inconvenience to the user due to the potential for irrecoverable data loss. Apps need the the protected `MASTER_CLEAR` permission to be able to factory reset a device (a permission that can only granted to pre-installed system apps). In this case study, we use Essential Phone as an example to illustrate this vulnerability. The vulnerability resides within a pre-installed app with a package name of `com.ts.android.hiddenmenu`. This app is a platform app and executes as the `system` user. Moreover, the vulnerable interface exposed to other apps on the same device is the activity app component `RTNResetActivity`. An external app can create an explicit Intent that starts this activity, and the activity will programmatically initiate an immediate factory reset of the device.

D.3 Logcat Leakage in Code Aurora

Some firmware contained an app with a package name of `org.codeaurora.gps.gpslogsave` that can be induced to leak the Logcat logs to external storage. The package name indicates that the app was developed for the Code Aurora project, which is an association of companies developing open source wireless communications projects for mobile devices. The package name of an app is selected by the app developer and can easily be "spoofed" to make it appear as though the app was created by another organization. We inquired directly with Code Aurora to see if one of their members was responsible for its development, but they did not respond. In our dataset, 18 different Xiaomi firmware contained this app which was not present in any other vendor firmware.

Although the naming of the app and its components focus on GPS, the app has no other ostensible relation to GPS. The app captures the entire Logcat log and *does not* use a filter for log tags that are related to the GPS subsystem. This app will not be started by the system in response to common events due to absence of their corresponding intent filters in its manifest. Moreover, the app's icon does not appear in the launcher, so it is unlikely to be started by the user. Due to an exported and accessible component, `GPSLogKitReceiver`, an external app can initiate the logging of the system-wide Logcat log to a location in the `org.codeaurora.gps.gpslogsave` app's private directory with a single intent. An external app can send a different intent message which makes the `org.codeaurora.gps.gpslogsave` app copy the log file from internal storage to external storage, making it accessible to any app with the `READ_EXTERNAL_STORAGE` permission.

Interestingly, FIRMSCOPE also detected a command injection vulnerability stemming from a dataflow from an app component entry, `GPSLogKitReceiver.onReceive(Context, Intent)`, to the `Runtime.exec(String)` API. We manually investigated the data flow and confirmed that the flow is indeed valid. An external app can provide a string in an intent which the `org.codeaurora.gps.gpslogsave` app will encrypt, insert, and execute as a command with the format: "`/system/bin/sh -c echo enc(<timestamp> - <attacker controlled string>)>> /persist/gps/gps-strength`" where the `enc(x)` is encryption of the string with a static key and Initialization Vector (IV) using Advanced Encryption Standard (AES). Since the `sh` command is used in batch mode, the attacker can perform multiple command injections, although it is challenging to exploit because the dynamic timestamp adds to the variability of the ciphertext.