Analysis and Manipulation of Android Apps and Malware in Real-Time

Salahuddin Khan

Information Security Department

Royal Holloway, University of London

A thesis submitted for the degree of

Doctor of Philosophy

2018

Declaration of Authorship

I, Salahuddin Christopher Jules Khan, hereby declare that this thesis and the work presented in it is entirely my own. Where I have consulted the work of others, this is always clearly stated.

Signed: ______________________________

Date: ______________________________

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Lorenzo Cavallaro, for his continuous support of my Ph.D including his patience and motivation through very stressful times.

Besides my supervisor, I would like to thank Dr Laura Christie for her guidance and Dr Rebecca Thomas for her mentorship.

I thank the ISG department, in particular, Professor Carlos Cid and Professor Rainer Dietmann for their sustained support.

Most importantly, I thank my wife, Samira, for her immeasurable support, encouragement and love, without which my PhD would not have been possible.

Thank-you Mom and Dad for believing in me and for making me the person I am today. You are both sorely missed.

Abstract

The number of apps in the Google Play store (∼3 million) necessitates an automated approach towards analysis for security threats. Such analysis relies on the ability to fully comprehend, and potentially modify, the actions being taken by a given app, whether low-level (system call) or high-level (services such as SMS or Location). Therefore, this thesis seeks to determine how accurate and scalable methods for the analysis and manipulation of Android apps/malware can be constructed that transcend the significant changes to the Android system through each release.

First, the author describes the potential of utilising a system call only based approach to reconstructing both low-level and high-level behaviours. A novel method for automatically reconstructing system call information in a version-agnostic manner is presented, as is the robust, scalable and extensible framework that enables real-time reconstruction, analysis and manipulation of low-level and high-level operations using this approach. While prior work does explore utilising a system call based approach it is a primitive implementation supporting a single version of Android and requiring significant manual effort. While this approach permits automatic system call reconstruction it cannot reconstruct Binder ICC and Android objects.

Next, the author explores a novel approach for reconstructing Binder ICC and Android objects through static analysis of the Android framework source code. This approach precisely determines the relationship between Binder interfaces and Android objects, permitting automatic generation of reconstruction code to correctly and efficiently reconstruct Binder ICC in real-time, integrating the automatically generated code into the framework.

The author demonstrates the efficacy of this design by building an information leakage detection plug-in that uses differential analysis to detect leakage of sensitive information. This plug-in is further extended to test anti-evasion techniques.

Finally, the author discusses utilising the approach in other ways, including automatically constructing Berkeley Packet Filter support for on-device analysis.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Chapter 1

Introduction

There are approximately 2.6 billion smartphones worldwide, 88% of the which are running Android [76] [85]. The Google Play Store [38] hosts almost 3 million Android apps and saw first-time app installs of nearly 64 billion last year. Revenue from Google play was approximately $20.1 billion in 2017 [20]. Furthermore, Google removed 700,000 apps from the Google Play store in 2017 for violating its policies [6]. Some of these removals where due to apps mimicking others already on the store (approximately 250,000), while others violated the policies in others ways. An undefined yet significant portion of these removed apps contained malware [6].

Despite safeguards, malware continues to be found on app stores, including the Google Play Store. For instance, the HummingBad and related HummingWhale malware [12] [47] netted the authors' approximately $300,000.00 every month through Ad fraud at its peak. After Google removed infected apps from the Google Play Store it reappeared in other apps. HummingBad/HummingWhale also download and run additional apps and malware on an infected device. They further utilise a plug-in infrastructure developed by the Chinese security vendor Qihoo 360 known as DroidPlugin [61]. This plugin infrastructure allows the malware to download other APKs and run them without having to install them or root the device.

Thus far many millions of users have downloaded malware from the Google Play Store (HummingBad alone affected over 10 million users). The sheer number of applications necessitates an automated approach to analysis [47]. Although the Google Play Store is policed by the Google Bouncer it is clear that malware is still able to bypass this safeguard and make onto the store and eventually infect user's devices [58].

The path to profit for mobile malware is typically shorter than that for desktop malware (where stealing financial information or extorting money through ransomware is more likely). For instance, a mobile device can be used for calling premium phone numbers, sending SMS messages and other tasks. It could also collude with desktop malware to defeat 2-factor authentication over SMS [48]. While users have become more cautious when downloading unknown software on desktop systems, they expect an official store to be safe and the available apps to be trustworthy.

Android apps are generally written in `Java`, although they can include native code written in `C++`. Prior to Android Lollipop (version 5.x), apps `Java` code runs in an instance of the Dalvik Virtual Machine (DVM), however, apps running on Android Lollipop and later are compiled into native code during installation. Apps run within a sandbox environment and are isolated from other apps and the various services that Android provides. These services perform high-level tasks, such as sending SMS messages or providing location and contact information. Apps are only able to utilise these services using an interprocess communication (IPC) mechanism known as Binder.

Current approaches to malware analysis on Android are either static (wherein malware is examined at a binary level but not executed) or dynamic (where malware is executed and its behaviours captured and analysed). Dynamic analysis is able to catch runtime behaviours that can be obfuscated through encryption or using `Java` reflection as well as code that is downloaded and executed, not unlike the DroidPlugin example mentioned earlier. While static analysis has the potential to examine every code path in a given app, it can be thwarted using a variety of techniques [74]. Similarly, dynamic analysis may not execute the pertinent code paths and hence miss important behaviours. Other approaches have attempted to combine the two in order to use the strengths and mitigate their weaknesses, which is known as hybrid analysis.

A significant portion of dynamic application analysis research on Android focuses on the Dalvik VM, by instrumenting, modifying or tracing its instructions [88] [23] [11] [73]. The introduction of the Android Runtime (ART) in Android 5.x (Lollipop) has improved the performance of apps, as they are compiled into native code during installation. However, since the Dalvik VM is no longer present and apps execute as native code, further research on Android needs to look to solutions that are not reliant on the application runtime environment. TaintART [77] is one example that implements dynamic taint tracking on Android Lollipop (5.x) and later by modifying the ahead of time (AOT) compiler. While this provides taint tracking for apps running on Android Lollipop and later, it does not provide a general model that supports research on Android regardless of the runtime environment.

In order to analyse a given app for potentially malicious behaviour, it is necessary to understand all of the components of each operation performed the app. For system calls, this is generally simple, for example a `write` system call will include a buffer of the data to write, a `sendto` system call will similarly contain a buffer of the data to send. For high-level operations such as interprocess communication between apps, knowing which service the app is calling, what it is requesting and the parameters for that request all need to be determined. In the context of this thesis (and other related papers), building the context (including the meaning of parameters in system calls and in IPC) related to a given operation is called reconstruction and it applies to both low-level and high-level operations.

With the ever increasing number of apps on the various Android app stores and the associated risk of malware, it is imperative that the analysis of Android apps remain viable on current versions. As mentioned earlier, analysing low-level and high-level behaviours requires the ability to reconstruct the operations performed by a given app. Approaches to this have either modified the Android image (Dalvik VM or Android Runtime), analysed Dalvik instructions or used a system-call centric approach. Modifying the Dalvik VM in order to support analysis requires significant effort and must be repeated for every version, furthermore, the Dalvik VM is no longer present on Android. For the same reason, tracing Dalvik instructions is no longer viable.

Therefore of the different approaches, only a system call based approach is able to transcend Android versions regardless of the app runtime. However, achieving this requires in-depth details of system calls and their parameters as well as Binder interfaces, call parameters and Android objects. The number of system calls, Binder interfaces and Android objects continues to increase for every new version (detailed later in this chapter). Thus reconstructing system calls and Binder interface calls across versions requires an automated approach to obtaining the information relating to each.

A key challenge in analysing apps on any system is that apps are essentially a black-box and only through determining what actions those apps perform, can one hope to make decisions regarding them. Doing so on Android requires having the requisite understanding of all system calls as well as a method for understanding Binder ICC. System calls are fairly simple to understand as the parameters are fixed in their type. For example, a `read` system call will always have the same number of parameters, including a file descriptor, an output buffer and a buffer length. The contents of the read can always be determined through simply accessing the output buffer after the read has completed. Binder ICC, however, varies depending on the interface being called and the specific method being invoked. Furthermore, the Android objects vary in length and can be one of more than 11,000 objects (depending on the call in question).

The challenge in reconstructing Binder ICC on Android is to determine a mechanism through which any given interface call and associated objects can be restored into a form that can be intelligently analysed (reconstructed in the context of this thesis).

In order to do so, it becomes necessary to determine the precise relationship between Binder interfaces and Android objects. How the author achieves this is described in Chapter 4.

The goal of the research performed for this thesis is to determine a mechanism through which apps can be analysed and to demonstrate the results of the research through the development of a system that enables such analysis.

Chapter 3 and Chapter 4, describe how the author approaches the challenge of automatically constructing an analysis framework for both low-level and high-level operations that is version agnostic. This includes describing how the system is implemented by the author towards this end. It is important to note that the implementation described is only one method through which this can be achieved. A key insight, that only Binder interfaces and their intersection with Android objects is required to reconstruct Binder ICC, can be applied in multiple forms including to construct Berkeley Packet Filters for such analysis. This is discussed in more detail in Chapter 8.

Any software, or portion thereof, has two sources of truth, one is the executable code and the other is the source code. While some research analyses the executable code, such as the Soot Framework, source code provides more context, since field and variable names are fully accessible, in addition to the control flow. It is for this reason that the research undertaken for this thesis focussed on static analysis of the Android framework source as doing so ensures that all context is fully captured.

New versions of Android introduce new system calls, Binder interfaces and Android objects. The number of system calls on Android has grown from 171 in Froyo (version 2.2.3) to 200 in Oreo (8.0.0). Furthermore, some have been replaced with different versions, for instance `dup2` with `dup3`. Table 1.1 describes the system call changes from Android Froyo (2.2.3) through Oreo (8.0.0). Android does not change the kernel for every release and thus the system calls can remain static across multiple releases. Chapter 3 describes the method utilised in reconstructing system calls across Android versions.

Similarly, the number of Binder interfaces and Android objects also varies per version, with interfaces being added and others being deprecated. Table 1.2 illustrates the growth in interfaces and their associated calls across versions. Although interfaces are removed over time, in reality this number remains small <40 from Froyo to Oreo. For example, IUrlRendererCallback is present in Froyo through Gingerbread, but is removed in Jellybean. However, in general, interfaces are not removed but rather documented as deprecated (to ensure compability apps developed for earlier versions).

Version	Total	Added	Removed
Froyo 2.2.3 (baseline)	171	0	0
Gingerbread 2.3.2	174	4	1
Gingerbread 2.3.7	174	0	0
Jellybean 4.1.1	195	21	0
Jellybean 4.2.2	199	4	0
KitKat 4.4.2	205	6	0
Lollipop 5.0.2	200	29	34
Lollipop 5.1.1 → Oreo	200	0	0

Table 1.1: Number of system calls per Android version.

Version	Interfaces	Calls
Froyo 2.2.3	118	1186
Gingerbread 2.3.2	129	1345
Gingerbread 2.3.7	130	1369
Jellybean 4.1.1	165	1867
Jellybean 4.2.2	179	1927
KitKat 4.4.1	233	2438
Lollipop 5.0.2	342	3698
Lollipop 5.1.1	350	3790
Marshmallow 6.0.1	372	4235
Nougat 7.0.0	397	4285
Nougat 7.1.0	429	4847
Nougat 7.1.1	430	4857
Nougat 7.1.2	431	4865
Oreo 8.0.0	506	5149

Table 1.2: Number of interfaces and calls per Android version.

The Android objects and their methods similarly increase for each version, as can be seen in Table 1.3. Reconstructing high-level operations (including Binder IPC) using virtual machine introspection across Android versions requires the ability to reconstruct interfaces and objects based on the contents of the marshaled data. Through observation of the marshaled data along with visual inspection of the Android interface and object code, the author intuited that reconstruction of interfaces and objects may be achieved by leveraging those portions directly involved in interface/object marshaling.

The research undertaken and described in this thesis seeks to determine a mechanism through which system calls, Binder IPC interfaces and associated Android objects can be seamlessly reconstructed. In so doing, it also establishes a clear relationship between Binder interfaces and the Android objects with which they interact. The resultant model is critical it determining only those components of a given object that are pertinent to

Version	Objects	Methods
Froyo 2.2.3	3467	38581
Gingerbread 2.3.2	3476	39351
Gingerbread 2.3.7	3575	40646
Jellybean 4.1.1	5175	59450
Jellybean 4.2.2	5530	63292
KitKat 4.4.1	5506	61840
Lollipop 5.0.2	7455	87838
Lollipop 5.1.1	7501	89117
Marshmallow 6.0.1	8311	98014
Nougat 7.0.0	9859	115252
Nougat 7.1.0	10069	118450
Nougat 7.1.1	10077	118695
Nougat 7.1.2	10083	118906
Oreo 8.0.0	11694	134955

Table 1.3: Number of objects and methods per Android version.

reconstructing Binder IPC and, as a result, high-level behaviours.

The novel approach taken in this thesis utilises static analysis of the Android framework source code to construct a model that is used in the automatic generation of code that enhances dynamic analysis of Android apps and malware. The automatically generated code further makes it possible to manipulate operations carried out by apps/malware in real-time, thereby offering the ability to explore different aspects of apps/malware and potentially discover further properties. The results of this approach can be seen in [78], which was published at the NDSS conference in 2015 and [21], published at IEEE Security & Privacy conference in 2016.

The static analysis of the Android framework `Java` source generates code that integrates directly into the CopperDroid v2 code base and provides real-time reconstruction and manipulation of complex Android objects. As a purely system call based solution, CopperDroid v2 enables reconstruction and manipulation of operations performed by both `Java` and native code. Furthermore, the reconstructed objects are shared with plug-ins that can query and even modify them in real-time.

While the work described in this thesis leverages the abstract syntax trees (ASTs) of the Android framework source code directly to automatically generate reconstruction code (as described in Chapter 4), that is only one possible research outcome. The author also considered designing a new language that explicitly captured the relevant information required to reconstruct Binder IPC interfaces and Android objects (including control flow) so that the resultant model can be applied for different purposes. This is further described in Section 8.3 (in Chapter 8).

The potential this allows for further research is demonstrated through the use of a plug-in infrastructure to build an information leakage detection plug-in that performs multiple sequential executions modifying only one datum each time while controlling critical properties of the system. A second example demonstrates building on the information leakage plug-in to alter the properties of the emulator (for anti-evasion) thereby exposing evasive malware.

The problem of achieving real-time analysis and manipulation of Android apps and malware motivates the central research thesis of this disseration:

> *The Android Source code contains all the information necessary for reconstructing high-level behaviours from low-level system calls with sufficient semantics to perform malware classification, information leakage detection and other research.*

The thesis statement provides the basis for building a model to test its hypothesis. As the source code of any system is a source of truth for that system, it should, intuitively, provide insight into the system. The level of insight varies depending on the depth of analysis performed on the system. However, one can theorize that if a given system is completely understood, then any action performed on that system should similarly be understood and could be broken down into its components. For example, if a given program adds two numbers together and returns the result, then anyone understanding this and seeing the input values can always correctly determine the output value. Similarly, even though there are several hundred system calls and many thousand Binder interfaces and Android objects, if one is able to correctly analyse each of these components and determine precisely how they interact then reconstructing such operations should be no different from determining the output of the addition program mentioned earlier.

The research carried out in this thesis therefore focuses on analysis of the Android source code and in so doing produces the necessary context required to fully reconstruct system calls and Binder interprocess communication. This research is realised in the form of a robust and highly scalable analysis framework known as CopperDroid v2. There are several different components that form this framework, most of which (94%) is automatically generated as a direct result of the analysis. This dissertation describes the design of each component of the framework as well as how they integrate to realise this goal. In addition, this framework is evaluated with regards malware classification, information leakage detection, anti-evasion techniques as well as the amount of effort required to build further research using this framework.

1.1 Motivation

The introduction of the Android Runtime (ART) has significantly hindered dynamic analysis research on the Android platform. The majority of existing dynamic analysis platforms rely on the Dalvik runtime, which is obsolete as of Android Lollipop. The number of Android devices continues to grow, as does the number of apps available for download.

Mobile malware is continuing to be a very lucrative opportunity for unscrupulous individuals and the threat is only likely to increase. Existing solutions are either no longer relevant due to the removal of the Dalvik runtime or are unable to scale sufficiently in both performance and functionality to meet the needs imposed by an ever-increasing application base. Furthermore, many of the solutions currently in existence are not extensible and thus don't provide a base on which further research can be built.

The author therefore seeks to close this gap by overcoming not only any performance issues but also those hindering extensibility and further research.

1.2 Thesis Contributions

To overcome such limitations, the author proposes CopperDroid v2, a VM-based dynamic analysis framework to seamlessly reconstruct Android apps' behaviours. Besides reconstructing generic system call behaviours, the solution is rooted in the key observation that the Android framework contains all the code and data to enable a *correct* and *efficient* reconstruction of ICCs behaviors along with their arguments (interfaces and objects).

To this end, the author relies on static backward and forward slicing of the entire Android framework to automatically build the class hierarchy of ICC-related interfaces and objects and identify all and only the `Java` statements that are responsible for their marshaling and unmarshaling. The author constructs an abstract syntax tree to then automatically generate `C++` code, which is included in the codebase of CopperDroid v2 to reconstruct Android apps' behaviours at runtime with a negligible overhead.

To facilitate the quick development of new analyses, the author further enhances CopperDroid v2 with a `Python` and `C++` plug-in interface along with automatically-generated `Python` and `C++` plug-in boilerplate code for both system calls and inter-component communications. This enables direct access to the automatically reconstructed Android objects, thereby allowing modification of inter-component communications in real-time.

The author's design supports a seamless automatic generation of code to reconstruct Android behaviours across different Android versions (from Froyo to Oreo) with a negligible overhead, reporting micro-benchmarks of 7–12.22% and 35–85μs for the reconstruction of complex ICCs and ICC-related resources. In addition, the static program analysis enables the introspection and manipulation of system calls, including ICC requests as well as replies, de-facto providing a more accurate and diverse set of behavioural reconstruction than the state-of-the-art [78], enabling realistic analyses of Android apps.

Therefore, the author makes the following contributions:

1. *Determines and describes the precise relationship between Binder interfaces and Android objects (including their methods and fields).*

2. *Constructs a model that can be applied to Binder interfaces and Android objects to extract only those portions necessary in Binder IPC.*

3. *Applies this model in order to perform static analysis on the Binder interface and Android framework source code and enable precise slicing of interfaces and objects for reconstruction.*

The author realises these contributions through the design and implementation CopperDroid v2, a VM-based dynamic analysis solution for reconstructing system calls, ICCs, and ICC-related resources of apps through static program analysis of the entire Android framework. Applying the model, as per the contributions, enables automatic generation of version-specific Android `C++` code that is included in the code base of CopperDroid v2 for seamless reconstruction of Android apps' behavioural profiles. In addition to reconstruction, the automatically generated code also enables manipulation of system calls and Binder ICC in real-time.

CopperDroid v2is further extended through the implementation of a plug-in infrastructure that exposes unmarshaled system call and Binder ICC data to plug-ins thus enabling real-time analysis and manipulation of these operations. Additionally, CopperDroid v2 simplifies the construction of new analysis tasks through the automatic generation of `Python` and `C++` plug-in boilerplate code. This in turn supports the quick development of analyses that monitor or manipulate system calls or ICC payloads, de-facto facilitating complex analyses, such as information leakage detection, anti-VM evasion, and ICC-based fuzzing in real-time.

The author further demonstrates that CopperDroid v2 is low-overhead and more accurate than the state-of-the-art unmarshaling Oracle—which is hindered by its design—in recreating behavioural profiles, while keeping manual engineering efforts to a bare minimum through the evaluation on more than 4,500 benign and malicious apps.

An information leakage detection plug-in, utilising differential analysis, is described and demonstrates the ease with which system calls and ICC manipulation can be achieved using the plug-in infrastructure. The plug-in is then extended to enable anti-evasion support for the purposes of detecting evasive malware within an emulated environment.

Chapter 2

Background and Related Work

2.1 Introduction

In traditional operating systems, such as Windows and Linux, operations are generally performed via system calls, although some high-level abstraction such as the Win32 API may be present (in the case of Windows). Some language features also abstract the underlying system call (for example `ofstream` in `C++` or `FileOutputStream` in `Java`). Actions performed using these higher-level API's still result in one or more system calls being invoked. Furthermore, these actions take place in the context of the calling user and process.

Mobile operating systems, such as iOS, Windows Phone and Android, however, run applications within a sandbox environment and provide high-level services for apps such as Location, Contacts and Telephony operations are managed through services hosted in external processes. Programs written on these platforms utilise a Software Development Kit (SDK) in order to access these services. For Windows Mobile they currently use the Universal Windows Platform (UWP) API, on iOS the iOS SDK and on Android, the Android Software Development Kit (SDK) [36]. In the case of Android, the services that manage these features (hosted in external processes) are accessed via an interprocess communication mechanism known as Binder ICC/IPC [62]. Certain operations, such as an app reading from/writing to files, still use system calls directly.

Android applications[1] are written mostly in `Java`, but can include `C++` native code for performance and Intellectual Property Rights (IPR) protection. Android apps are written using the Application Programming Interface (API) provided by the Android Software Development Kit (SDK) [36]. Apps are written using the Android SDK to

[1]Application and app are used interchangeably throughout this thesis

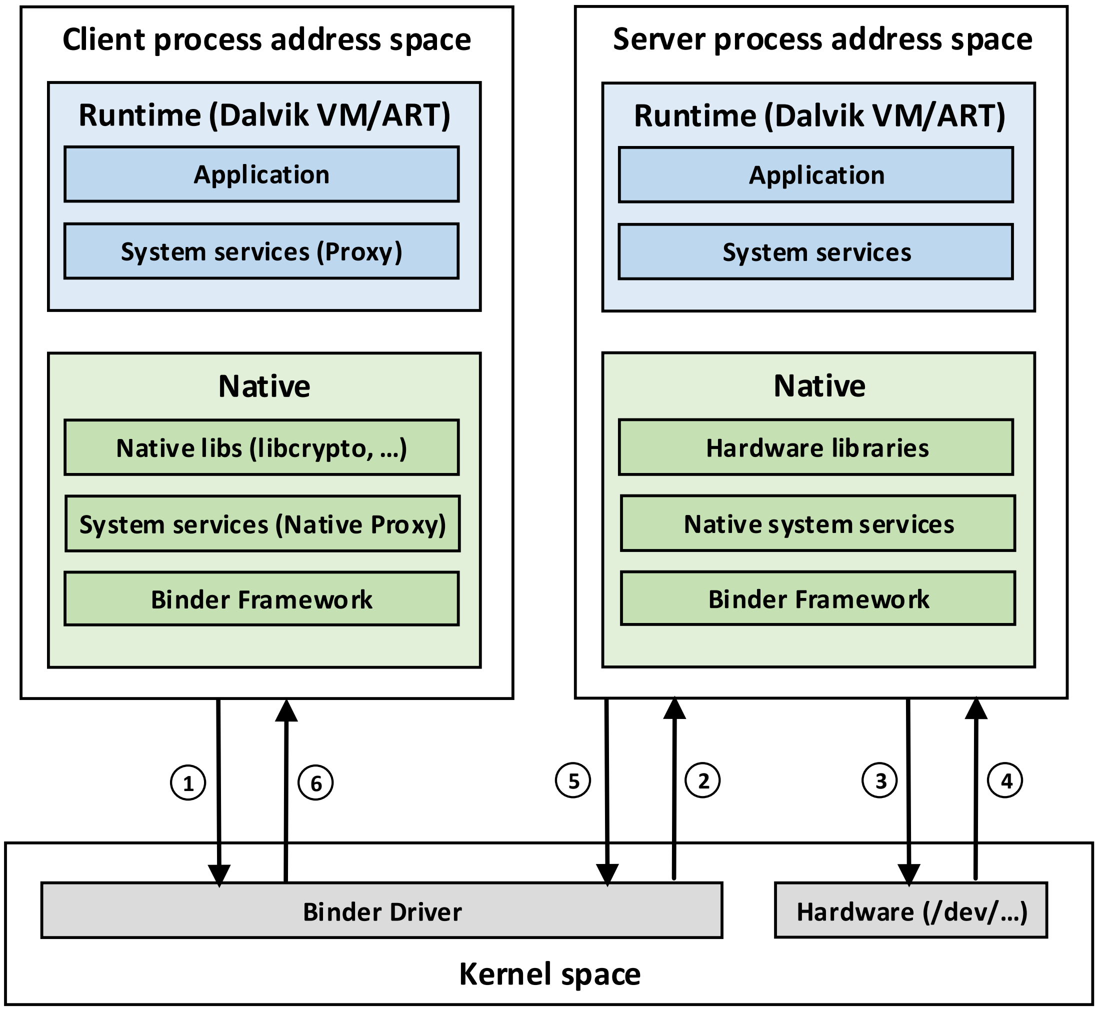


Figure 2.1: Communication between apps and services on Android

access the Android Framework for UI, accessing Content Providers, Location and other services. Each app runs with a unique user and group ID [2]. Although the services appear (to the apps) to be in the same process space, since they call methods on object in their address space, in reality these services —daemon-like UNIX processes that provide features such as sending a text message, accessing sensors, drawing UIs, and sending intra- and inter-thread messages— also run in their own sandbox (in a separate process). This can be see in Figure 2.1.

Communication with these services heavily relies on a well-defined inter-component communication mechanism[2] known as Binder [62], which is accessed through an `ioctl` system call. The Android (Java API) Framework can be seen in Figure 2.2 (in olive and

[2]The terms inter-component communications, inter-process communications, and Binder transactions are used interchangeably in this thesis.

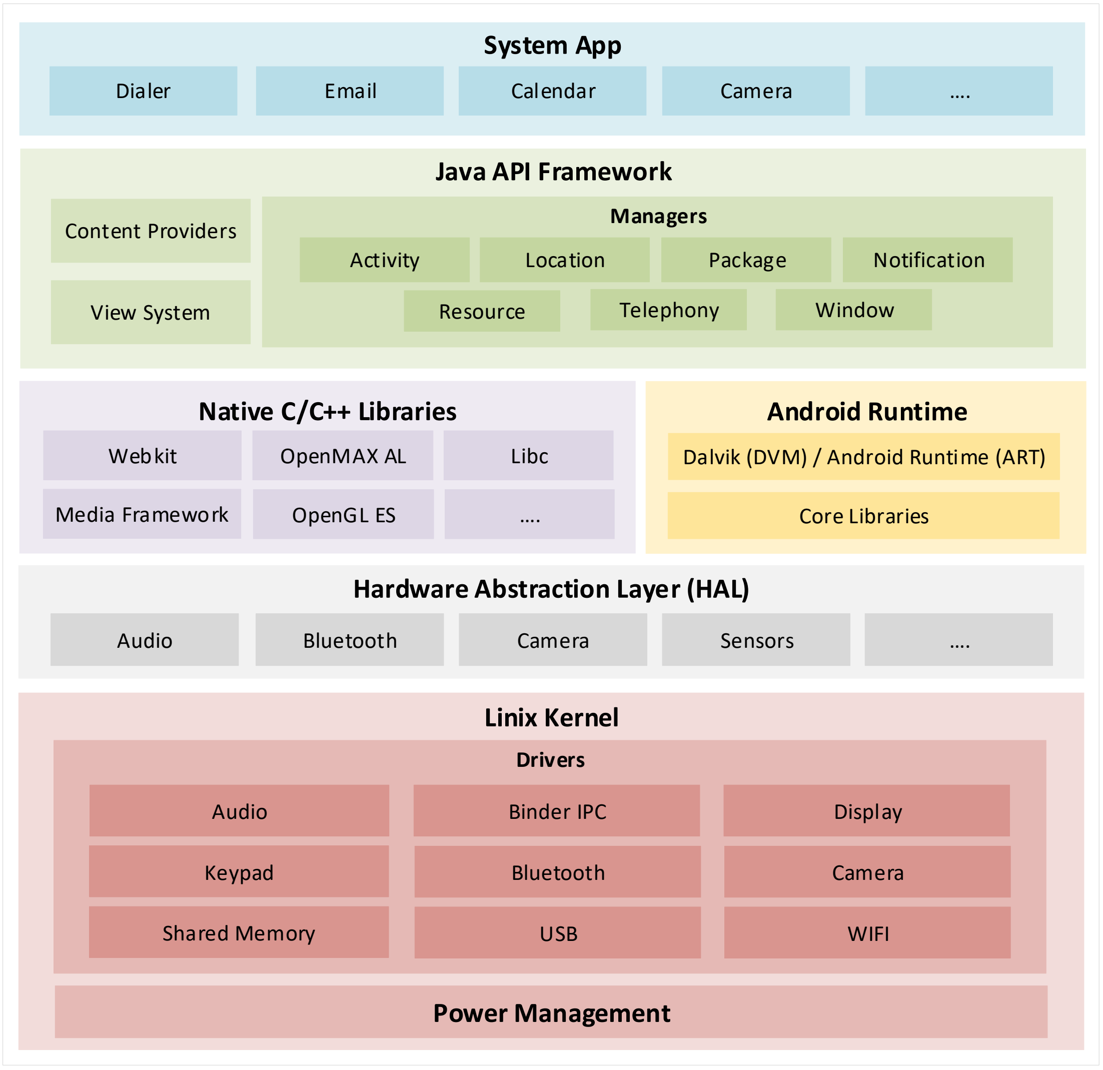


Figure 2.2: Android Architecture

green) along with the rest of the components of the Android system. Apps utilise either the Dalvik VM (prior to Lollipop) or the Android Runtime (Lollipop and later). As apps execute they interact with the framework utilising API calls written to the SDK, if a given API call accesses a service then this communication will happen via Binder.

Binder itself is accessed via the `ioctl` system call to the `/dev/binder` device [62]. In [71], Fattori et al. demonstrate that it is possible to reconstruct high-level operations from introspection of low-level system calls, including the Binder `ioctl`. The author of this thesis further expanded on the work described in [71] building a version agnostic dynamic analysis solution known as CopperDroid v2.

Prior to the Android version codenamed Lollipop (2014), apps were executed in an instance of the Dalvik VM, a Java-like virtual machine. Many analysis tools modified the Dalvik VM and used their own modified version to analyse apps for information leakage and other (potentially) malicious behaviours [23] [69] [88]. However, as of Android Lollipop, apps are compiled into native code at installation time and use the Android runtime (ART) instead, which reduces the overhead introduced by the execution of bytecode [77]. A side-effect of ART is that Dalvik VM based analysis is no longer possible. This appears to have had an impact on Android security research resulting in fewer papers at recent security conferences. The work contained in this thesis attempts to overcome the difficulty caused by the shift to ART and enable further research on current Android versions.

2.2 Malware Analysis Techniques

Malware analysis today utilises static analysis, dynamic analysis or a combination of both, known as hybrid analysis. Static and dynamic analysis both have their strengths and weaknesses and hybrid analysis attempts to combine the strengths of each form in order to overcome the weaknesses.

2.2.1 Static Analysis

Static analysis works by analysing a given app without executing it. This theoretically enables the analysis of all code paths within a given app, however, it can also be defeated by inserting complex tasks such as encryption, cryptographically secure hashing. It is also not able to analyse code that uses obfuscation techniques such as `Java` reflection or relies on code that is downloaded and used only when the program actually executes. Sharif et al. [74] describe a mechanism for hindering static analysis of malware through conditional code obfuscation. Techniques previously used for obstructing static analysis of malware on traditional desktop operating systems still apply for mobile malware, including encryption, polymorphism and others. Furthermore, there are legitimate reasons for obfuscating and encrypting Android apps in order to protect intellectual property rights and other potentially valuable data.

There are tools that exist today for this very reason to stop Android apps from being decompiled and repackaged, one such tool is DexGuard [41]. As of Android Jellybean (version 4.1), the Google Play Store encrypts all paid apps and even installs them using the encrypted APK [40] [22]. Furthermore, [60] describes the limits of static analysis and describes how binary code can be obfuscated at a binary level to defeat static analysis tools. While the paper is focused on Windows and Linux programs, one could also envision a version that includes native code in an Android app that is obfuscated in a similar manner.

2.2.2 Dynamic Analysis

In contrast, dynamic analysis works by executing a given app and analysing the operations that it performs while running [7] [19]. Doing so makes it possible to analyse code that is downloaded as well as code that is obfuscated since the interaction with the system will be similar regardless of whether or not code is obfuscated. However, dynamic analysis may miss certain behaviours simply because it is unlikely to execute all the code in the app. Using stimulation techniques it is possible to overcome this to elicit otherwise hidden behaviours as described in [78].

2.2.3 Hybrid Analysis

Hybrid malware analysis seeks to build on the strengths of these two approaches while minimizing the weaknesses of each. SmartDroid [16] utilises static analysis of a mobile app in order to determine how activities switch and what functions may be invoked (using function call graphs). SmartDroid then utilises dynamic analysis to traverse those paths in an attempt to trigger malware behaviours.

2.3 The Android Emulator

There is an inherent difference between full-system virtualisation and full-system emulation. Although both virtualisation and emulation result in a full-system including peripherals being made available to the guest operating system, how each achieves this is fundamentally different. Products such as VMWare, VirtualBox and Windows Hyper-V are all full system virtualisation solutions and make use of hardware specific features to support this virtualisation (often running with a higher-privilege level than even the host operating system). These provide a virtualised environment wherein a guest operating system and all its programs can execute without adversely affecting the host. The guest operating system and programs all execute code that is native to the CPU, generally x86/x86-64 code. Executing code from a different CPU architecture is not possible.

Full system emulators, however, such as the Quick Emulator (QEMU) [8], emulate the target environment and can execute code with a different CPU architecture from that of the host. For example, QEMU supports x86, x86-64, MIPS, ARM, PowerPC and other CPU architectures running on an x86 host. Emulation is achieved through real-time translation from one instruction set to another (for example ARM to x86). Emulators don't require any hardware support in order to function, however, they run slower than virtualisation as every instruction has to be translated from the guest CPU to the equivalent host CPU instruction(s).

The Android emulator is based on the QEMU emulator with support added for Android devices, including sensors, modems and battery support. Its primary function is to enable developers to test and debug their applications directly from their desktop systems. To this end, the emulator can also simulate phone calls, incoming or outgoing SMS messages, location, rotation and other features necessary for the development and testing of applications [33]. This emulated environment has also proven useful for malware analysis since, being open source, it can be modified to support virtual machine introspection. For instance, Google Bouncer, DroidScope, Andrubis and others make use of the Android emulator in order to analyse apps for malware [58], [88], [55].

2.4 Related Work

The research described in this thesis covers several areas, including systems that perform dynamic analysis of Android apps/malware, static analysis of the Android framework, information leakage detection and anti-evasion mechanisms to elicit behaviours in evasive malware. While there has been prior work in these areas, the novel approach taken in this thesis provides a model that enables high-performance, portable and extensible version agnostic analysis/manipulation of Android apps and malware.

Malware research and analysis on traditional operating systems such as Windows or Linux utilises a system call based approach as all interesting actions on these operating systems occur through system calls. System calls on ARM use the `swi` instruction, with a system call number in `r7` and parameters being passed in through `r0-r5` (and the return value in `r0`). For other processor types, such as `x86` or `amd64`, a similar model is used with the `sysenter` and `sysret`. In the absence of an analysis system, any currently executing app is effectively opaque with regards actions the app performs. The research undertaken in this thesis seeks to make the actions apps perform transparent so that their behaviours can be reconstructed and analysed.

Reina et al. [71] demonstrate that a purely system call based approach (similar to that used in traditional desktop operating systems) can provide details about high-level operations on Android. A system call based approach has the potential to work regardless of the application runtime environment, however, the work described in the paper excludes certain interfaces critical to the functioning of Android apps, such as `IActivityManager` and `IServiceManager`. While this research demonstrates that both low-level and high-level operations can be reconstructed using system calls alone, it fails to handle several core interfaces. Furthermore, while this can determine the simple parameters, such as strings and integers, in Binder calls, complex parameters such as `Location`, `Uri` or `Intent` objects cannot be reconstructed. With the reconstruction of Binder incomplete, much pertinent information remains opaque.

Tam et al. [78] demonstrate that building on this research and augmenting the Binder IPC reconstruction with an unmarshaling Oracle, enables the full reconstruction of both low-level and high-level operations. The unmarshaling Oracle reconstructs complex Android objects, such as `Intent` and `Uri`. However, while state of the art, it utilises `Java` reflection for reconstructing complex objects and is prohibitively expensive with regards performance. Thus the Oracle cannot be invoked in real-time and is therefore unable to support manipulation of unmarshaled objects as the overhead results in Android watchdog timers firing.

The following sections describe the different approaches that prior work has utilised for dynamic analysis in addition to static analysis related to that utilised in this thesis. The related static analysis research described entails program analysis in general and has not been applied to Android prior to that carried out for the purposes of this thesis.

2.4.1 Dynamic Analysis

Although there are existing analysis platforms for Android, none provides a similar level of portability, extensibility and performance as CopperDroid v2.

DroidScope [88] is framework that enables the development of dynamic analysis tools for Android apps and malware. It uses an out-of-the-box approach (much like CopperDroid v2) by instrumenting the Android emulator but certain modes of operation (for instance taint tracking) can incur significant overhead. Furthermore, it uses 2-level virtual machine introspection (VMI) [30] to determine information about the currently executing system and exposing this to plug-ins via a series of hooks and APIs. DroidScope plug-ins can leverage these hooks and APIs to perform different levels of analysis such as system calls, single instruction tracing or taint tracking. However, as DroidScope is reliant on the Dalvik VM tracing, the introduction of ART renders it moot. In addition, while it is invoked for system calls, it does not provide any introspection into the parameters of those system calls.

Therefore, DroidScope without the Dalvik support does not achieve much more than the CopperDroid v2 plug-in manager that is integrated into QEMU. CopperDroid v2, by comparison, performs all dynamic analysis seamlessly through automatically generated system call and Binder reconstruction code and therefore does not require a 2-level VMI approach. Plug-ins are invoked as operations occur without requiring them to manage disparate contexts.

AppsPlayground [69] is an automatic dynamic analysis system for Android, providing taint tracing and kernel level system call inspection. It relies on TaintDroid [23] for taint tracking and is built upon the Dalvik VM. Furthermore, it is not extensible so even if it could be made to work on current Android versions, it does not provide a means for building on the information it provides.

CopperDroid v1 [71] uses a system call based approach to reconstruction and has support for AIDL-defined Binder interfaces. While taking an innovative approach using a modified AIDL-parser to generate Binder reconstruction code in `Python`, it is not able to reconstruct `Java` based interfaces. This results in it missing critical interactions between apps and interfaces that form the core of the Android framework (such as `IActivityManager` and `IServiceManager`).

Furthermore, CopperDroid v1 is unable to unmarshal complex Android objects without assistance and this is provided by an unmarshaling Oracle. The Oracle is implemented in `Java` and utilises `Java` reflection to reconstruct complex objects, resulting in extremely poor performance (on the order of seconds for such reconstruction). In addition, the Oracle runs as a service in an standard Android emulator instance with CopperDroid v1 communicating with it via TCP, which adds additional overhead. The Oracle is therefore only utilised for offline analysis in post-processing of the results of a given app execution.

In contrast, the static analysis performed by the CopperDroid v2 compiler results in `C++` unmarshaling code that is integrated directly into CopperDroid v2 resulting in μs level performance for unmarshaling entire interface method calls and replies. The automatically generated code also enables real-time manipulation of system calls, interface method parameters and complex objects through plug-ins providing the ability to build further research on top of CopperDroid v2.

2.4.2 Static Analysis

As far as the author is aware, there is currently no body of work that statically analyses the Android Framework in order to enhance dynamic analysis of Android applications by producing unmarshaling code automatically. Edge Miner [11] does statically analyse the Android Framework, however it uses the analysis to generate API summaries towards understanding implicit control flow transitions for information leakage detection.

Program slicing was first proposed in 1979 to aid in debugging, program maintenance and other tasks [87]. Slicing in the traditional context generally results slices being computed for a given program variable as it relates to the variables flow through the program. The original idea stems from the fact that seasoned developers intuitively follow a single variable through a debugger to understand how the program is affecting it, in order to isolate a problem [86]. Many other slicing techniques have since been introduced including those more suited to object oriented software [87]. Liang and Harrold propose performing object slicing based on a system dependency graph [53] as this is able to capture more of the context than earlier works. While most focus on single fields or variables in a given object, the author requires slicing as it relates to marshaling and unmarshaling. Thus, while the author uses a system dependency graph, program slicing is performed not on the basis of a given field in a class, but rather in terms of how every field and method in a class relate to the marshaling and unmarshaling of the class. This is determined which members of a given class are accessed by interfaces in the course of interface method invocations.

2.4.3 Information Leakage

There are different approaches to detecting the leakage of sensitive information. The first of these is static taint analysis which analyses a given app without executing its code. AndroidLeaks uses static taint analysis to locate sources of sensitive information and uses tainting to determine if the data can reach a sink resulting in transmission to the network [31]. However, it uses a coarse grained approach to tracking tainting at the object level as opposed to at the field level. FlowDroid implements a more precise method that works at the field and variable level, avoiding false positives by determining *when* a variable is tainted so that accessing a *sink* prior to being tainted doesn't result in a false positive [4]. Edge Miner analyses the entire Android framework to determine implicit control flow transitions [11].

Static taint analysis can be thwarted by using `Java` reflection to access sensitive information without the analysis system being aware and hence avoiding being tainted.

Dynamic taint analysis tracks information flow from *sources* to *sinks*, however, it does so by actually executing the app and analysing it at runtime. TaintDroid [23] modifies the Dalvik interpreter and implements variable level taint tracking. AppFence [42] modifies TaintDroid by adding support for obfuscation and encryption. AppsPlayground [69] is a dynamic analysis system that performs security analysis of Android apps but uses TaintDroid for tracking information leakage. All of these approaches rely on the Dalvik VM and therefore cannot be used on Android Lollipop and later versions. Additionally they cannot detect leaks in native code.

The Android Runtime (ART) compiles `Java` apps into native code upon installation. TaintART [77] replaces the ART compiler with one that implements taint analysis in native code using a similar model to TaintDroid. Although this addresses the change in the runtime, leaks originating from app native code written in `C++` will not be tracked.

Another approach works by analysing data at the network level to determine if sensitive information is being leaked. Some, such as AntMonitor [51], PrivacyGuard [75] and HayStack [70], perform on-device analysis by using the built-in VPNService present in Android. These then scan the network traffic for hard coded identifiers or the user's private information and therefore cannot detect leaks if the data is obfuscated or encrypted. ReCon [72] also uses a VPN approach and can handle simple forms of obfuscation, but not encryption that is performed within the app.

AGRIGENTO [17] uses a black-box differential analysis based approach while controlling sources of non-determinism (random values, timing values, system values, encrypted values and patching `JavaScript` code). It establishes a baseline by executing an app while controlling non-determinism and capturing network (`HTTP` and `HTTPS`) traffic. Subsequent runs change a single piece of sensitive information each time, capturing the network traffic and comparing it to the traffic captured before. AGRIGENTO uses a risk analysis based approach to attempt to mitigate cases where non-determinism is affected by a source out of its control by executing it as many times as needed to determine which traffic changes even if all other data remains the same.

2.4.4 Anti-Evasion

Google Bouncer [58] is a dynamic analysis platform for detecting malicious behaviour in apps submitted to the Google Play Store. Not much is known about it except what has been determined by apps submitted for the express purpose of fingerprinting the environment. While it does modify some sensitive information (presumably for anti-evasion), such as the network operator, phone number and device ID other properties are unchanged.

AppsPlayground [69] is a dynamic analysis based system for performing security analysis of apps using the Android emulator. It modifies the phone number, `IMEI`, `IMSI`, Android ID and others in order to evade detection. It further modifies the `build.prop` file within the emulator to mimic a real device.

Details of the implementation are not published making it difficult to perform any comparison, however, implementing the same feature set using a `Python` plug-in built on top of CopperDroid v2 requires very little effort.

Chapter 3

Efficient Version-Agnostic Automatic System Call Introspection for Android

3.1 Introduction

This chapter describes CopperDroid v2, as designed and implemented by the author, and provides the context required for understanding how the hand-written and automatically generated code are integrated to produce deep inspection of apps running on Android. It further describes the static analysis based on the bionic utilities which generates basic system call reconstruction code. In addition, the fixed portions of Binder IPC via the `ioctl` to the `/dev/binder` device, are also covered including how they integrate with code generated through static analysis of the Android Framework, described in Chapter 4.

System calls are the core mechanism through which programs communicate with the operating system on which they are running, whether that is Windows, macOS or Linux. System calls let apps access local files or devices (`open`, `read`, `write`, `ioctl` and `close`), get the current time (`gettimeofday`) as well as send and receive data across the network (`sendto`, `recvfrom` etc). Every operating system has its own set of system calls and new versions often add system calls. Android is a mobile operating system that runs on top of a Linux kernel. All interactions that occur between apps running on Android and the operating system or other apps and services is accomplished through system calls [71].

The Linux kernel used by Android continues to provide additional system calls with each new release. Therefore, analysing apps on Android requires inspection of the system calls available for a given version. As new Android versions are released, Google drops support for older versions with the net effect that newer apps no longer need to target those earlier versions. Thus, if the analysis tools are not able to seamlessly execute on later versions, then Android malware simply needs to raise the lowest version they support in order to avoid being analysed on older platforms. Since Android is an open source platform, the system calls it supports for each version can be determined through simple static analysis of the bionic_utils [32], described in 3.6.

CopperDroid version 1 as described in [71] demonstrates that a system call based approach along with deep inspection of ioctl calls related to Binder enables the reconstruction of high-level behaviours (both malicious and benign) on unmodified Android images. CopperDroid version 1 is limited to Froyo 2.2.3 and relies on the Remote Serial Protocol (RSP) debug protocol to communicate with the Android emulator [33] (built on top of QEMU [8] as described in Section 2.3) via its GNU Debugger (GDB) stub. Reliance on the RSP debug protocol combined with a `Python` implementation of CopperDroid v1 results in significant performance issues and cause watchdog timers to fire on later Android versions, thereby making it infeasible as a long term solution.

CopperDroid v2 as described in this thesis shares the name and also utilises system call based analysis, however, the similarity ends there. Rather than significantly modify the Android emulator (as was done with CopperDroid v1), the author of this thesis uses a modular approach, adding a plug-in manager to QEMU and implementing CopperDroid v2 as a plug-in. The plug-in interface that is exposed from QEMU invokes plug-ins for every system call entry and exit (in addition to initialisation/shutdown of the emulator). Each system call entry and exit includes a context that enables to plug-in to query guest registers in addition to reading from or writing to guest memory. Both the plug-in manager (residing in QEMU) and the CopperDroid v2 plug-in are implemented in `C++`. The plug-in approach requires significantly less modification to QEMU enabling faster adoption of newer emulators[1]. Futhermore, a plug-in approach is used to simplify execution in other environments, including on a device. The plug-in uses an abstract interface to access information regarding system call information (including parameters) and to read/write memory in the currently executing process. Thus, a driver or process can implement the plug-in manager interface and invoke the CopperDroid v2 plug-in for system calls thus having CopperDroid v2 analyse a given app in real-time. Since CopperDroid v2 itself supports plug-ins, a `C++` plug-in can perform additional analysis include policy enforcement or on-device malware detection. The latter could similarly be attained through automatically generating Berkeley Packet Filter support. This is discussed further in Chapter 8.

[1]Since the initial implementation the author has had to move to newer emulators on two occasions, both times requiring less than 30 minutes to complete the work.

Process Name	Process Id	Thread Id	Thread Info	System Call	Parameter 1	Parameter 2	Parameter 3	Return Value
??	??	??	0xe03ac000	5	0xa68f3cf0	0x00020002		38
...	...	...	...	...	...	...	...	...
??	??	??	0xe03ac000	54	38	0xc0186201	0xa68f3cf0	0
...	...	...	...	...	...	...	...	...‘

Table 3.1: System calls without context.

While CopperDroid v2 performs dynamic analysis of apps and services executing within an instance of the Android emulator, any version specific code that CopperDroid v2 needs to operate is entirely automatically generated. The publication [78] combined the results of CopperDroid v1 as well as CopperDroid v2. The latter, CopperDroid v2 (developed entirely by the author of this thesis), was used for testing execution on Android Lollipop as mentioned in the publication. Furthermore, the reconciling of the PendingIntent return value was also achieved using CopperDroid v2.

As part of its operation, CopperDroid v2 collects the details of all system calls occurring in every process running in the Android instance. To reduce the size of the data collected prior to running a specific app sample, only system calls that affect file descriptors are tracked (`open`, `openat`, `close`, `socket`, `pipe`, `fork`, `clone`, `dup`, `dup2`, `fcntl` etc.). System calls that affected file descriptors are tracked at all times in order to ensure that any descriptors to the `/dev/binder` device are tracked across processes clones, forks and descriptor duplication (dup/dup2/dup3). Additionally, another research project built on CopperDroid v2 and needed to track file descriptors for all files accessed on the system (from boot), which can only be achieved if descriptors are tracked from startup. Once the given sample is installed, capturing of all system calls is enabled and the app in question is executed.

To illustrate the challenge in reconstructing system calls and Binder ICC across versions, Table 3.1 provides an example of system calls captured with little to no context. The reason for the lack of context is due to the fact up to this point, the author hasn't described the necessary mechanisms for interpreting the data present within a given system call. The current information defined in this table (system call numbers, parameters and return value) is standard for all system calls on ARM processors as is the method for determining the Thread Info address on Linux kernels. This chapter and Chapter 4 revisit this table, illustrating how each advancement enables greater context, with Chapter 6 illustrating modification of a portion of a specific system call's data.

Serialisation of the data is managed using a two-stage serialisation process to minimise the overhead of inspecting any given system call. The first stage is a lightweight serialiser that simply adds the data for each system call to a `std::queue` protected by a lock. Second stage serialisation is either managed at the end of a given analysis, with all data being kept in memory, or periodically on a separate thread of execution (using POSIX threads). The latter occurs based on a timeout (using seconds) or if a memory threshold is reached.

3.2 Design

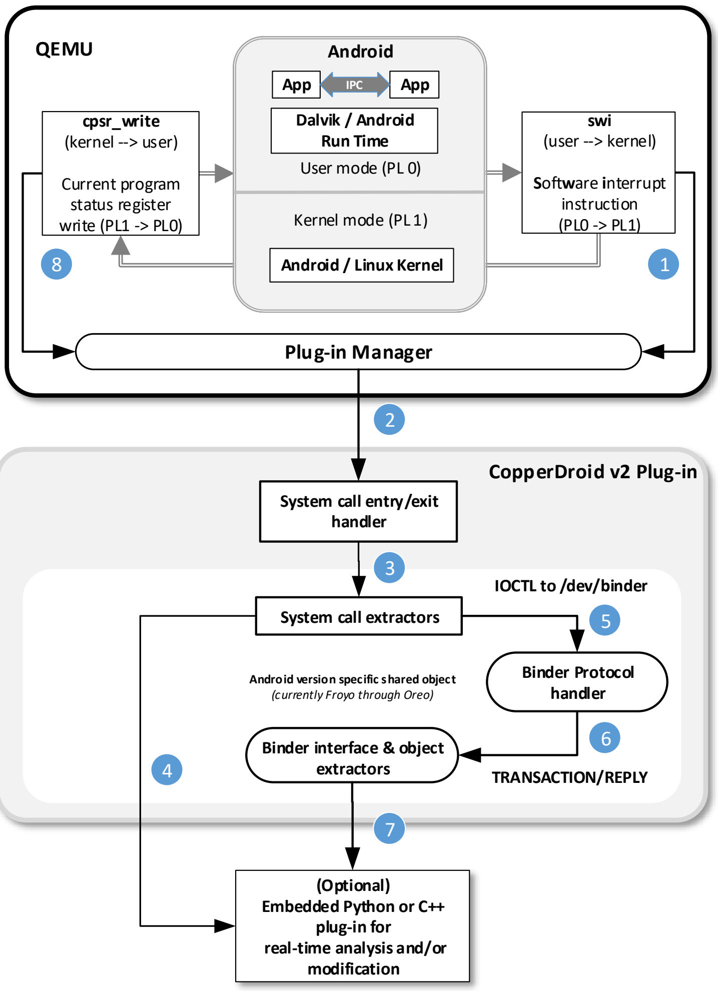

QEMU
Android
App
IPC
App
Dalvik / Android Run Time
User mode (PL 0)
Kernel mode (PL 1)
Android / Linux Kernel
cpsr_write (kernel --> user)
Current program status register write (PL1 -> PL0)
swi (user --> kernel)
Software interrupt instruction (PL0 -> PL1)
Plug-in Manager
CopperDroid v2 Plug-in
System call entry/exit handler
System call extractors
IOCTL to /dev/binder
Binder Protocol handler
Android version specific shared object (currently Froyo through Oreo)
Binder interface & object extractors
TRANSACTION/REPLY
(Optional) Embedded Python or C++ plug-in for real-time analysis and/or modification
1
2
3
4
5
6
7
8

Figure 3.1: CopperDroid v2 Architecture

CopperDroid v2 has a modular design with interlocking pieces. This is important as it greatly simplifies supporting multiple versions of Android (currently Froyo through Oreo) seamlessly.

The Android emulator (QEMU) is modified to support a plug-in manager that provides an abstract interface invoked for system call entry/exit. An additional abstract interface enables the CopperDroid v2 plug-in to query virtual CPU register information as well as read from/write to guest memory (see Figure 3.2).

On the ARM platform system calls are handled via the software interrupt (`swi`) instruction, which transitions from privilege level 0 (User mode) to privilege level 1 (Kernel mode) invoking the Android/Linux Kernel to handle the system call and then returning via the `cpsr_write` instruction from privilege level 1 back to privilege level 0. The plug-in manager in QEMU is invoked for each of these transitions and in turn invokes the CopperDroid v2 plug-in for system call entry and exit. The design of the abstract interface as implemented in the plug-in manager supports running on the x86 or MIPS platforms too, however, the current implementation of CopperDroid v2 runs on the ARM 32-bit platform.

The core functionality in CopperDroid v2 is implemented in a plug-in that connects to and utilises the interfaces provided by the plug-in manager. The CopperDroid v2 plug-in tracks processes, threads, kernel objects (such as devices and files) and analyses the fixed `ioctl` structures of the Binder protocol. The reconstruction of system call information is achieved with the help of the Android bionic utilities [32], while the interface and object reconstruction is accomplished through static analysis of the Android framework to automatically generated reconstruction code (described in Chapter 4). The code generated for system call and textsfBinder IPC reconstruction is hosted in an Android version specific shared object that is loaded by the CopperDroid v2 plug-in during initialisation. This shared object provides an interface for system call reconstruction and a second for Binder IPC reconstruction.

This design allows the core management functionality (process/thread/kernel objects etc.) to be shared across all Android versions while simultaneously enabling per-version reconstruction of system calls and Binder IPC.

The dynamically loadable module (shared object) built for each supported Android version consists of approximately 30 lines of hand-written code (see Figure 3.2) for initialisation (described later in this chapter). The remaining code, $>$100,000 lines of code per Android version, is automatically generated through static analysis of different areas of the Android source and is used to produce system call reconstruction and Binder interface/object marshalers/unmarshalers in `C++` automatically during the build process.

Furthermore, the CopperDroid v2 plug-in itself supports plug-ins, written in either `Python` or `C++`, that are invoked for system calls and Binder operations and are provided with the fully reconstructed data extracted by CopperDroid v2. Plug-ins to CopperDroid v2 can operate on this data and make changes to system calls, Binder interface calls and complex Binder objects. The plug-ins are either hosted through the use of embedded Python or, in the case of C++ plug-ins, natively as shared objects.

Supporting Python and C++ plug-ins enables complex analysis and modification of system calls and Binder IPC in real-time, fueling the potential for further research built on top of CopperDroid v2.

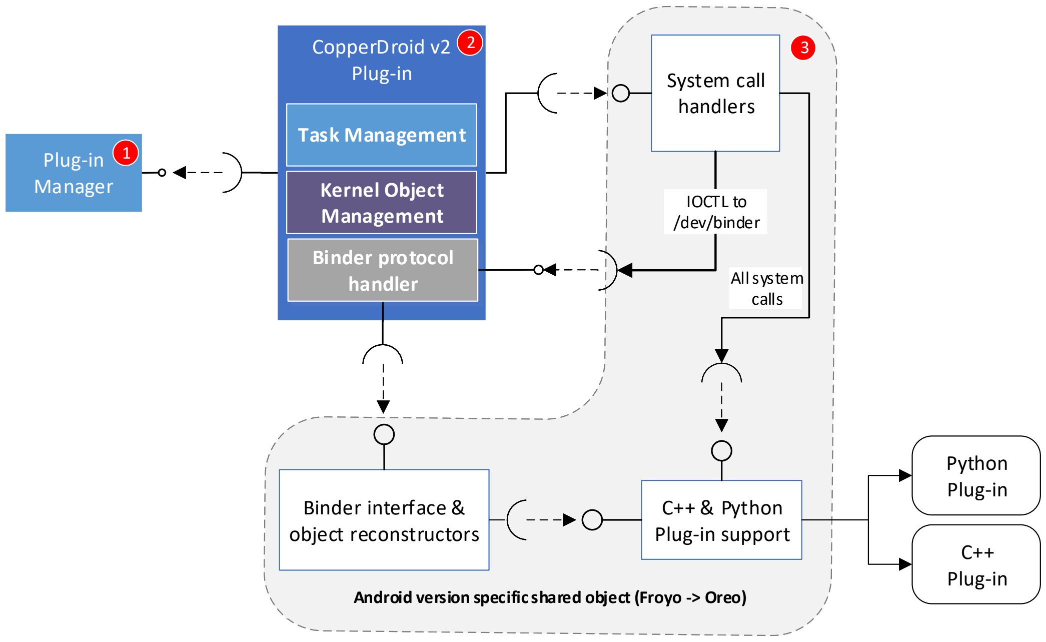


Figure 3.2: CopperDroid v2 Design

Figure 3.2 illustrates the design of CopperDroid v2. The plug-in manager (1 in the figure), CopperDroid v2 plug-in (2 in the figure) including Task management, Kernel Object Management (file descriptors for devices) etc. and the Binder protocol handler are all hand-written. The Binder protocol handler is responsible for parsing the fixed ioctl structures of the Binder protocol and is described in Section 3.9.

The Binder protocol has not changed in format in the last ten years. The only change has been to normalize types to make supporting 64-bit platforms. Specifically types such as void* to binder_uintptr_t, signed long to _u32 and size_t to binder_size_t. These values are all 32-bit on 32-bit platforms and 64-bit on 64-bit platforms. The version number has been 7 for the last ten years (for 32-bit), with a version number of 8 being added for 64-bit platforms. Thus, a hand-coded implementation for parsing the Binder protocol is sufficient for all the current versions (the protocol is similarly unchanged in Android Pie). However, automatically implementing a parser for this protocol is possible by utilising static analysis of the binder.h header file, however, that is not in the scope of this thesis, and given stable nature of the protocol thus far, is not a likely requirement.

As mentioned earlier, the system call handlers and Binder IPC reconstruction code is generated using static analysis and hosted in a version-specific shared object (③ in Figure 3.2).

The static analysis is further used to automatically generator boilerplate plug-in code (in both `C++` and `Python`) for system calls as well as Binder IPC. Although the static analysis used for system calls relies on the `SYSCALLS.TXT` parsing mentioned above (described later in this chapter), Binder is handled using static analysis Android Framework `Java` source code and is significantly more complex. This is described in detail in Chapter 4.

3.2.1 Plug-in Manager

Introducing a plug-in manager to QEMU limits the scope of the changes required in QEMU without any loss of functionality. Implementing the CopperDroid v2 plug-in using a set of abstract interfaces simplifies porting CopperDroid v2 to other environments, such as a different emulator or even a physical device. Furthermore, modifying the CopperDroid v2 plug-in does not require rebuilding the emulator each time. CopperDroid v1 made significant modifications to QEMU and required a non-trivial amount of time to support new emulator versions. Conversely, CopperDroid v2 can be ported to a new emulator version in less than 30 minutes.

A plug-in context (`IPlugInContext`) is passed to the plug-in for every system call entry and exit, permitting the plug-in to access registers and guest memory and is defined in Listing A.2 (in Appendix A). The `IRegisters` interface abstracts the underlying registers enabling a single plug-in context definition to support multiple CPU architectures. When a plug-in is dynamically loaded, the plug-in manager locates its initialisation routine (described in Listing A.4) and calls it, passing in the `IPluginManager` interface shown in Listing A.3 thus enabling the plug-in to register itself.

The CopperDroid v2 plug-in implements the `IPlugInEvents` interface (see Listing A.5) and, once registered, is invoked for every system call entry and exit. Registration occurs before any code is executed in the guest, so every system call is observed by CopperDroid v2 from startup to shutdown.

3.2.2 The CopperDroid v2 Plug-in

As mentioned earlier, the CopperDroid v2 functionality is implemented as plug-in. This is important for the following reasons:

- Reduces the cost of moving to a new emulator
- Supports running in new environments, possibly a different emulator or even on a device
- Reduces the time required to rebuild and test CopperDroid v2 (only the plug-in needs to be recompiled not the emulator)

System calls occur very frequently on any operating system and Android is no exception. Debugging any issues within the context of an emulator is an issue because watchdog timers in Android fire almost immediately and have to be disabled in order to single-step; disabling them can cause side-effects within the emulator. Additionally, debugging a specific instance of a system call is almost impossible because of the frequency at which these occur. The author of this thesis, therefore, built a test framework to (minimally) simulate the emulated environment and perform specific system calls of interest in order to debug the paths without issue.

3.3 CopperDroid v2 Initialization

Since CopperDroid v2 performs dynamic analysis for Android using virtual machine introspection without modification to the guest, it needs to understand certain details about the Android version that is executing within the emulator. This information is provided in the configuration file passed to CopperDroid v2 during initialisation by the plug-in manager. The configuration file uses a text-based Google Protocol Buffer (protobuf[2]) file format [39]. The format is shown in Listing A.1 in Appendix A. The configuration file for Android KitKat is shown in Listing A.6 as an example.

The offsets shown in the listing are used for obtaining the thread ID (TID), process ID (PID) as well as the package or executable name of the currently executing program using the thread info address obtained through the current stack pointer (shown in Listing 3.1).

[2]Protocol Buffer and protobuf are used interchangeably in this thesis.

Process Name	Process Id	Thread Id	Thread Info	System Call	Parameter 1	Parameter 2	Parameter 3	Return Value
com.example.binderdemo	1504	1504	0xe03ac000	5	0xa68f3cf0	0x00020002		38
...	...	...	...	...	...	...	...	...
com.example.binderdemo	1504	1504	0xe03ac000	54	38	0xc0186201	0xa68f3cf0	0
...	...	...	...	...	...	...	...	...

Table 3.2: System calls with process name, process ID and thread ID correctly determined.

```
...
uint64_t ThreadInfoAddress = Context->GetRegisters()->r(sp,0) & ~(GetConfiguration()->
    GetKernelStackSize()-1);
...
```

Listing 3.1: Determining thread info address using the current stack pointer

The kernel stack size is also present in the configuration file definition and is implicitly defaulted to 8 kilo-bytes. It can be explicitly defined if it changes for a given version or platform (for example 64-bit rather than 32-bit).

The values obtained from the configuration enable the CopperDroid v2 plug-in to load the platform specific shared object (for example, `kitkat4_4_2.so` in Listing A.6 in Appendix A) and initialize it. The `PlugInInformation` section results in the shared object initializing a `Python` or `C++` plug-in (if specified). If `InvokePlugIn` is false, the plug-in will not be loaded.

The offsets provided in the file are used internally within the core CopperDroid v2 code to obtain the process ID (PID), thread ID (TID) and name of the program initiating the system call (described in Section 3.5). The method for accessing this information is the same across Android versions, it is only the offsets that can differ.

With the process name, process ID and thread ID identified, some information missing in Table 3.1 can be included, as can be seen in Table 3.2.

After emulator startup the CopperDroid v2 plug-in will only analyse system calls that affect certain system state unless other requested. For example, file descriptors can be forked into processes even though the name is only provided during an open/openat call. Therefore, from startup, CopperDroid v2 will monitor `open`, `openat`, `close`, `dup`, `dup2`, `dup3`, `fork`, `clone` (where a new process is created) and `fcntl` (where a handle is duplicated) system calls. This is done within CopperDroid v2 to ensure that `/dev/binder` descriptors are tracked correctly, however, it's required to permit plug-ins to CopperDroid v2 to track state as well.

CopperDroid v2 relies on knowing which descriptors refer to `/dev/binder` in order to ensure that Binder IPC is interpreted correctly. `ioctl` codes are only unique within a given device and during early testing of CopperDroid v2 the author discovered an `ioctl` to a different device with a code identical to the `BINDER_WRITE_READ` `ioctl` code (described in Section 3.9). CopperDroid v1 only checks the `ioctl` code and misinterprets the buffer provided, whereas CopperDroid v2 only treats `ioctl` to the `/dev/binder` device as Binder related.

The CopperDroid v2 plug-in provides a `TCP/IP` interface for receiving instructions. Therefore, during a given analysis run, a message will be sent to the CopperDroid v2 plug-in through `TCP/IP` interface to start capturing all system calls, at which point every system call will be inspected. The configuration file can also request this to begin from startup and in that case every system call from startup to shutdown is fully inspected. CopperDroid v2 supports a variety of commands over `TCP/IP` including some that are passed through to `Python` or `C++` plug-ins. These include commands to switch scenarios or serialise the captured information to disk.

3.4 Platform Initialization

As mentioned earlier in this chapter, CopperDroid v2 supports running multiple Android versions with the same core code base, with all version specific code is implemented in per-version shared objects. Therefore, during initialisation the shared object for the specific version is loaded and its `InitializePlatform` function is called, passing along the configuration mentioned in Section 3.3. A plug-in configuration file for the `Python` or `C++` plug-in is also provided. An output parameter is set to the version specific system call handler routines (exposed via the `ISysCalls` interface and automatically generated as part of the build process) to permit the core CopperDroid v2 code to invoke the interface for every system call entry and exit. This is permits the core CopperDroid v2 code to remain version-agnostic.

3.5 System Call Introspection

As mentioned earlier in this chapter, the modified QEMU used for CopperDroid v2 intercepts the `swi` instruction and invokes the plug-in manager, which in turn calls the `EnterSyscallEvent()` method on the `IPlugInEvents` interface (see Listing A.5 in Appendix A) in the CopperDroid v2 plug-in before returning to QEMU to actually switch modes and execute the system call. Returning from the kernel mode to user-mode (via the `cpsr_write` instruction) invokes the plug-in manager which calls the `ExitSyscallEvent()` method in the CopperDroid v2 plug-in prior to returning from the system call.

The `ISysCalls` interface (see Listing A.8 in Appendix A) is of key importance in keeping CopperDroid v2 version agnostic. Newer versions may have different system calls that alter state and hence need to be tracked, for instance, Lollipop introduced the `dup3` system call which duplicates a given file descriptor. As Section 3.3 mentions, CopperDroid v2 needs to be aware of any file descriptors that reference the `/dev/binder` device to correctly handle Binder IPC. Therefore the core CopperDroid v2 plug-in calls the `IsTrackedSystemCall` method on the `ISysCalls` interface to determine if a given system call should be inspected when full inspection is not active.

As outlined in Section 3.2, shared objects are automatically generated for every supported Android version. The shared object includes an implementation of the `ISysCalls` interface with the `CallEntryStub()` method being invoked for every system call entry and the `CallExitStub()` method for system call exit.

The `CallEntryStub` routine obtains the necessary information about the currently executing task in the guest (PID, TID and name) and invokes an entry stub of the format shown on line 1 at the top of Listing A.8 (in Appendix A). The relevant stub is a static method in the class that has an entry in an `unordered_map` with the system call number being the key. When a given system call occurs, the corresponding stub is invoked and allocates a `SystemCallEvent` that stores the parameters, task information and other relevant data.

Most system calls return to the caller, however, some such as `exit` or `exit_group` do not return to the caller and hence don't have an exit stub. For those that do return (which is the majority), the system call event information (`SystemCallEvent` class instance) is inserted in an `unordered_map` with the task information as the key (process ID + thread ID). When the system call exits, the same information (process ID + thread ID) is used to locate the `SystemCallEvent` instance that is passed to the exit stub which captures the return value and output parameters. If a `Python` or `C++` plug-in is present then it will be invoked for both the system call entry and exit, where it can inspect or manipulate any of the parameters or, in the case of system call exit, the return value. Once the system call processing (entry and exit) has completed, the `SystemCallEvent` instance is added to the first stage (lightweight) serialiser, described later in this chapter.

For non-exiting system calls, the entry stub is invoked (including any registered plug-ins) and upon return the `SystemCallEvent` instance is immediately added to the first stage serialiser).

Some system calls have a conditional exit, for example `execve` will exit if it fails but not if it succeeds. CopperDroid v2 manages this by inserting `SystemCallEvent` instance for the conditional-exit system call into a table on per-processor basis and then checking for any pending conditional system calls then next time a system call entry occurs on the same processor. A system call entry can only occur on the same processor without an exit occurring the system call did not exit (which indicated success in the case of `execve`) and in that case, the system call is removed from the table and added to the first stage serialiser.

Plug-ins CopperDroid v2 can modify system call parameters and return values in order to alter the behaviour as needed. Chapter 6 and Chapter 7 demonstrate how this can be used to good effect with minimal effort in a `Python` plug-in to perform research in information leakage detection and anti-evasion.

3.6 Automatic System Call Reconstruction

This section describes the process involved in generating system call reconstruction code for all supported Android versions. The final result of this process is a system call handler implementing the `ISysCalls` interface described in the previous section that correctly reconstructs all system calls for a given Android version.

Generating the system call reconstruction code for each Android version requires several distinct stages. The first of which produces a system call table that captures the names, return values, parameters and system call numbers of all system calls that version. This involves utilising code obtained from the bionic utilities and is described in detail in Section 3.6.1.

The second stage determines whether a given system call's pointer parameters are filled in by the caller (inbound) or by the system call (outbound). The process involved in determining parameter direction is described in detail in Section 3.6.2 and modifies the system call table generated in stage one to include the parameter direction information.

CopperDroid v2 serialises all data captured during a run to protobufs which support scalar value types of `double`, `float`, `int32`, `int64`, `uint32`, `uint64`[3] along with `bool`, `string` and `bytes` (the latter being an arbitrary sequence of bytes). Thus, the third stage of the reconstruction code generation determines which Protocol Buffer type best matches the `C` type of a parameter and is described in Section 3.6.3.

The fourth stage utilises the system call table from stage one (augmented with the parameter direction information from stage two), a list of non-returning system calls and the protobuf types to generate a class derived from the `ISysCalls` interface, this is described in Section 3.6.5. The generated class provides the version specific system call reconstruction code for each supported Android version.

[3]There are other integer types but aren't relevant here

Return Type	Function Name[:Syscall Name[:callid]](Parameter List)	System Call Number or Presence or Architecture
Froyo through JellyBean		
ssize_t	read(int, void*, size_t)	3 (syscall number)
KitKat		
ssize_t	read(int, void*, size_t)	1 (supported on ARM platform)
Lollipop through Oreo		
ssize_t	read(int, void*, size_t)	all (supported platforms)

Table 3.3: Different formats of SYSCALLS.TXT entries across versions illustrated by read system call

3.6.1 Generating System Call Information

Android's bionic [32] utilities provide information on all the supported system calls for a given Android version including the processor architecture on which the system call is valid (x86 or ARM). The supported system calls reside in a file named SYSCALLS.TXT. Support for parsing this file exists in the form of a Python class also present in the bionic utilities. Android makes use of these components to produce the system call stubs during the build process. CopperDroid v2, however, uses them to assist in the generation of system call reconstruction code.

Since each version of Android has a different format for this file, it is necessary to use the corresponding parser to determine the system calls valid for a given Android version including the return values and parameters. Prior to Android KitKat (4.4.2) the SYSCALLS.TXT file included each system call number with the relevant system call (as can be seen in Table 3.3).

However, for Android KitKat and later versions, the system call number is not present in SYSCALLS.TXT and has to be obtained by parsing the unistd.h header for the Android version in question.

Android's bionic utilities also includes gensyscalls.py [35] and this file contains code to scan the unistd.h file and obtain the system call numbers for a given platform. The author of this thesis therefore built an equivalent function using the gensyscalls.py code as a starting point to build the system call information, this can be see in Listing A.9 (in Appendix A).

The method ParseSyscalls() uses the parser from the bionic utilities and constructs a table of all the system calls including the name, parameters and return value types. This table is constructed with the same structure for all Android versions permitting a single code generator to produce the reconstruction code for all system calls across any Android version.

3.6.2 Determining System Call Parameter Direction

System calls can have parameters that are either inbound or outbound, or both. For example, the `write` system call has inbound parameters for file descriptor, buffer and buffer length with a return value of the number of bytes written. The `read` system call on the other hand has a buffer that is outbound, as it is filled in by the system call, and will contain the data only on return from the system call. The `ioctl` system call has a parameter that is both inbound and outbound. As system call entry and exit are two separate events, CopperDroid v2 needs to have sufficient context as to when a specific parameter should be captured (on entry or exit or both).

Furthermore, since CopperDroid v2 executes outside of the system, when it attempts to access memory that is paged out, the guest OS does not page in the memory, rather, the read fails. Therefore CopperDroid v2 tracks whether or not reading a specific memory region succeeded and if not, re-reads the memory on system call exit. The author also tested hooking into the paging code for the ARM platform within QEMU and re-reading the memory when a memory read that was missed on system call entry was paged-in by the guest OS. This works, however, re-reading inbound parameters on system call exit proved sufficient and the author does not wish to risk potentially destabilizing the emulator on such a critical code path. However, if required in the future, re-reading memory when it is paged in by the guest OS is a viable option.

Currently CopperDroid v2 uses heuristics to determine input versus output parameters, specifically, examining system call names to see if they include ’read’, ’recv’, ’pipe’, ’socketpair’ as well as ’get’ among others and if so, non-`const` pointer parameters are marked as outbound. Thus far every system call outbound parameter is correctly handled. The author has considered writing a program to scour the online Linux documentation to determine parameter direction and parameter names, however, parameter direction is already adequately handled in a 30 line Python module using the heuristics mentioned above.

3.6.3 Determining Parameter Types

Each entry in the generated system call table includes parameter information that CopperDroid v2 uses to determine the protobuf types to which a given parameter should be mapped. Scalar types such as integers and boolean parameters use the equivalent protobuf type, whereas `char*` are mapped to the `string` protobuf type. For `C` structures or `void*` parameters, the `bytes` type is used with CopperDroid v2 automatically capturing the buffer into a `bytes` allocation based on the structure’s size. CopperDroid v2 also includes a hand-written `bytes` class (separate from the protobuf `bytes`) that is capable of reading memory from and writing memory to the guest using a functor shown in Listing A.10.

When system calls are invoked, scalar parameters are assigned using the appropriate type (since they represent a numeric value only), however, types such as `char*`, `void*` or `sockaddr_storage` are copied from the guest into a `bytes` instance on the host.

3.6.4 Accessing Guest Data

While the emulator runs on the host system using host memory, accessing memory managed by the guest operating system requires copying data into or out of the guest's address space. The `IPlugInContext` class (shown earlier in Listing A.2) provides methods for copying data between host and guest.

There are two functors that CopperDroid v2 uses, `CopyFromGuest` and `CopyToGuest` (shown in Listing A.11). This allows the `bytes` class to be agnostic of routines used for actually copying the memory. The `CopyFromGuest` functor uses the `ReadMemory()` method of the plug-in context, while the `CopyToGuest` function uses the `WriteMemory()` method. Using a functor interface makes it possible to pass the copy routine to the appropriate method without needing to define separate methods for each.

Listing A.12 shows two methods which are integral to copying memory between the host and the guest. The `SetData` method is used extensively throughout the `ISysCalls` implementation for copying non-scalar data from the guest to the host. Instances of `bytes` can be re-used and will discard their current allocations and update them to new ones. The `m_DataAddress` field stores the address from which the memory was originally read.

There are multiple overloaded methods for `SetData` and `WriteToAddress` which will use the stored address when re-trying a copy.
`WriteToAddress` is only used to modify the guest when a change is made in a `Python` or `C++` plug-in. The `void* CopyContext` is currently always an `IPlugInContext` interface, however, to keep the `bytes` class agnostic of the rest of the system this is passed in as a `void` pointer.

Although protobufs have a `bytes` type, serialisation of this type needs to be of type `std::string`, since the length will be used to determine the data to copy as opposed

Linux Type	Protobuf Type
int	int32
int64_t	int64
ssize_t	int64
uint32_t	uint32
uint64_t	uint64
mode_t	uint32
size_t	uint64
char*, const char*	string
void*	bytes
struct sockaddr, msghdr, stat etc.	bytes

Table 3.4: Mapping of `C` types to Protobuf types

to a null terminated string. The `bytes` class therefore has an `AsString` method which returns a `std::string` instance initialised with the data.

3.6.5 System Call Reconstruction Code Generation

With the types correctly determined, the fourth stage proceeds to generate the `ISysCalls` implementation. For every entry in the system call table, an entry and exit stub is built (for non-returning calls such as `exit` or `exit_group` only an entry stub is created). These stubs contain all the code necessary to capture the parameters (using their type information) and corresponding return values. Parameters are captured into a structure similar to the protobuf type that most suits them (as per Table 3.4). For example, structures such as `sockaddr` are copied into a `bytes` class instance from the guest using the size of `sockaddr_storage` structure, to ensure the full address is captured. Specialisers in the code generation choose the appropriate structures to use for a given type and generate the appropriate code. Similarly there are specialisers that handle specific system calls differently, for instance, the `ioctl` stub is specialised to call into the Binder reconstruction whenever an `ioctl` occurs to a file descriptor matching the `/dev/binder` device.

The code generated by the system call reconstruction for the `open` system call entry stub is shown in Listing A.13. Line 40 shows the allocation of an instance of a `CSystemCallOpen` class, which is automatically generated using the type information for the system call captured above. The automatically generated source code for this class is shown in Listing A.14 (in Appendix A.2).

Process Name	Process Id	Thread Id	Thread Info	System Call	Parameter 1	Parameter 2	Parameter 3	Return Value
com.example.binderdemo	1504	1504	0xe03ac000	open (5)	/dev/binder	0x00020002		38
...	...	...	...	...	...	...	...	...
com.example.binderdemo	1504	1504	0xe03ac000	ioctl (54)	38	0xc0186201	0xa68f3cf0	0
...	...	...	...	...	...	...	...	...

Table 3.5: System calls that includes the system call names and parameters.

CopperDroid v2 enables plug-ins to modify any system call's parameters when they are invoked for that system call. This provides the ability to significantly alter both the input and output of a system call, as plug-ins are invoked prior to system call entry/exit occurring in the guest.

Choosing `C++` types that match equivalent protobuf types simplifies the generation of the Google Protocol Buffers file that enables serialisation of all the collected system call information and Binder IPC communication across all versions with a single protobuf definition. Deserialisation code is also automatically produced (in `C++` and `Python`) permitting the resulting protobuf file to be analysed with tools written in either language.

The final stage of the code generation automatically produces boilerplate code for Python and `C++` plug-ins for system call handling. `C++` classes are also automatically generated for each system call and associated parameters in order to allow passing a single entity between CopperDroid v2 and `Python` or `C++` plug-ins and permitting the tracking of modifications made by plug-ins. Listing A.14 demonstrates the automatically generated class that encapsulates the data of the `open` system call (and is instantiated on line 40 of Listing A.13 in Appendix A.2). Chapter 6 demonstrates a plug-in redirecting an open of a non-existent file in the guest to one located on the host.

With the system call code generation complete, additional missing portions of information our available and can be seen in Table 3.5. While at this point all the information for a given system call will be captured, however, understanding the contents of the Binder `ioctl` data requires additional work and is described in Section 3.9.

Parameter 3 of the `ioctl` system call in Table 3.5 refers to a instance of the `BINDER_WRITE_READ` structure and will be described later in this chapter. The data that is captured at this point will simply include the raw data of the structure and can be seen in Listing 3.2.

```
00000000  08 00 00 00 08 00 00 00  90 25 43 b8 00 01 00 00   .........%C.....
00000010  30 00 00 00 88 24 43 b8                            0....$C.
```

Listing 3.2: Binder `ioctl` Buffer data

3.7 Serialisation of System Call Data

As mentioned in Section 3.3, serialisation of the captured system call and Binder data is managed using a two-stage serialisation process (to ensure minimal overhead for system call inspection). The first stage is a lightweight serialiser that simply adds the data for each system call to a `std::queue` protected by a lock. Second stage serialisation either occurs at the end of a given analysis, with all data being kept in memory and then serialised to storage, or periodically on a separate thread of execution (using `POSIX` threads). The latter occurs based on a timeout (using seconds) or if a memory threshold is reached. The timeout period as well as the memory threshold have default values, however, these can be adjusted too.

As mentioned in Section 3.6, the results obtained from analysing an app with CopperDroid v2 are serialised using Google Protocol Buffers, a message based language-independent, platform-neutral mechanism for the serialisation of structured data. Protobuf messages don't store their length internally, therefore using a single file for serialisation requires including the length along with a given message. CopperDroid v2 therefore uses a file with a header that contains details of the file (including a magic number and file version). While for each system call event, the length of the serialised message is stored followed by the message (as shown in Figure 3.3).

System call data is serialised in the SysCallEvent message (as shown in Listing 3.3). The basic structure of a serialised system call is a header, which contains information about the system call, such as the system call number in addition to information identifying the process making the call. When serialising the system call, the header is filled in as well as an optional event field matching the system call, for example readevent, writeevent or ioctlevent. When protobufs are serialised, only required fields and optional fields that have been set are included. Therefore, although the message definition is large, each message stored includes only the data relevant to the system call.

The code generators which generate the serialisation code also produce deserialisation code in `Python` and `C++`. This deserialisation code assists in post-processing of CopperDroid v2 output for reconstructing (potentially malicious) behaviours.

3.8 CopperDroid v2 Viewer

As mentioned in Section 3.7, system call data from a given run is serialised into Google Protocol Buffers. The serialisation is performed using the information gathered from

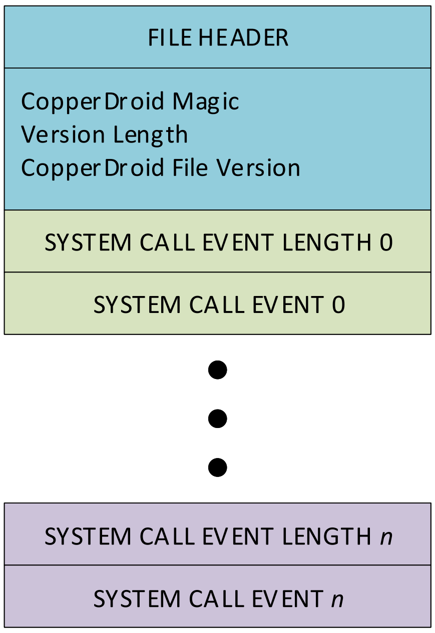


Figure 3.3: CopperDroid v2 Serialisation File Structure

```
message SysCallEvent {

    enum SYSCALLNUMBER {
        ...
        SysCallEnumRead = 3;
        SysCallEnumWrite = 4;
        ...
    }

    message Header {
        required SYSCALLNUMBER syscallnumber = 1;
        optional string timestamp = 2;
        optional int64 processid = 3;
        optional int64 threadid = 4;
        optional int64 threadinfo = 5;
        optional string name = 6;
        optional int64 eventid = 7;
    }
    ...
    message SysCallRead {
        optional int64 returnvalue = 1;
        optional int32 parameter0 = 2;
        optional bytes parameter1 = 3;
        optional uint64 parameter2 = 4;
    }

    message SysCallWrite {
        optional int64 returnvalue = 1;
        optional int32 parameter0 = 2;
        optional bytes parameter1 = 3;
        optional uint64 parameter2 = 4;
    }
    ...
    required Header header = 1;
    optional SysCallRead readevent = 1000003;
    optional SysCallWrite writeevent = 1000004;
    ...
}
```

Listing 3.3: Portion of the Google protobuf definition file for system call events

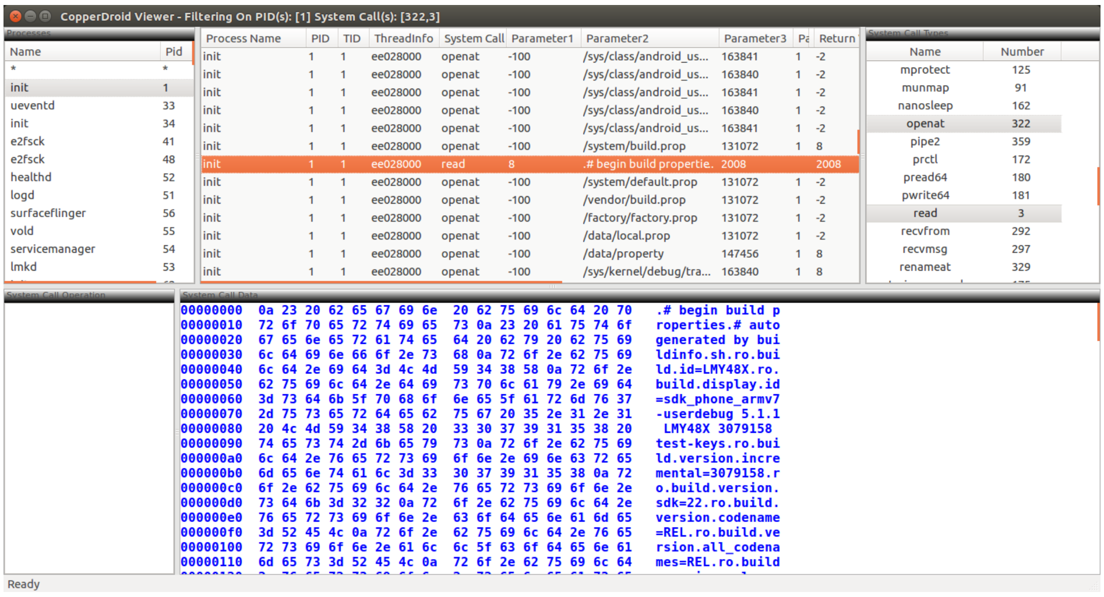
parsing SYSCALLS.TXT which also enables deserialisation code to be automatically generated in both Python and C++. The Python deserialisation code is utilised for post-processing the protobufs for a given app's run and plays a key role in producing low-level, high-level and composite behaviours utilised in malware analysis and classification. In the DroidScribe publication [21], 5246 app samples were classified using the features extracted by analysing the protobufs. The code which analysed the protobufs was writing entirely by the author of this thesis and uses the deserialisation code mentioned earlier. Support for composite behaviours was also added to improve the feature set available for machine learning, for instance, a combination of open, read, close is a composite behaviour of FILE_READ, whereas open, write, close is a behaviour of FILE_READ_WRITE. Similarly, a socket, bind and listen is a NETWORK_SERVER, where as socket and connect is a NETWORK_CLIENT.

The author (of this thesis) implemented a text based Python script to allow filtering of specific apps (based on Process ID), Binder interface method calls and system calls. However, the sheer number of system calls collected while running a given app makes it incredibly difficult to look at a specific system call or Binder interface call and follow what is occurring. Therefore, the author added automatically generated C++ deserialisation support for system calls and the Binder protocol (the latter is described in Section 3.9) and implemented a more useful tool, the CopperDroid v2 Viewer.

The CopperDroid v2 Viewer is implemented in C++ using the deserialisation code and presents a Graphical User Interface (GUI) that permits filtering on process, system call (including system call parameters and return value) as well as specific interfaces and methods on those interfaces. Furthermore, it also fully displays all the different Binder operations for both transactions (calls) and replies enabling one to analyse each component accordingly. The concept is akin to a combination of the system call tracer strace and a network packet analyser such as Wireshark or the Windows Network Monitor (NetMon). However, rather than network packets, the CopperDroid v2 Viewer is able to inspect and display the contents of Binder IPC communications, in addition to regular system calls.

Figure 3.4 is an example of an openat call of /system/build.prop followed by a read of the file. The title bar shows the current filters being applied to the list of system calls. Multi-select can be used for viewing multiple processes and system calls simultaneously. Alternatively, selecting the * at the top of a pane (either *Processes* or *System Call Types*) clears the filters for that type.

Using multi-select of processes in conjunction with specific Binder interface calls allows one to view both the client and server side of a given invocation. Figures B.1,



CopperDroid Viewer - Filtering On PID(s): [1] System Call(s): [322,3]

Processes

Name	Pid
*	*
init	1
ueventd	33
init	34
e2fsck	41
e2fsck	48
healthd	52
logd	51
surfaceflinger	56
vold	55
servicemanager	54
lmkd	53

Process Name	PID	TID	ThreadInfo	System Call	Parameter1	Parameter2	Parameter3	Pa	Return
init	1	1	ee028000	openat	-100	/sys/class/android_us...	163841	1	-2
init	1	1	ee028000	openat	-100	/sys/class/android_us...	163840	1	-2
init	1	1	ee028000	openat	-100	/sys/class/android_us...	163841	1	-2
init	1	1	ee028000	openat	-100	/sys/class/android_us...	163840	1	-2
init	1	1	ee028000	openat	-100	/sys/class/android_us...	163841	1	-2
init	1	1	ee028000	openat	-100	/system/build.prop	131072	1	8
init	1	1	ee028000	read	8	.# begin build propertie.	2008		2008
init	1	1	ee028000	openat	-100	/system/default.prop	131072	1	-2
init	1	1	ee028000	openat	-100	/vendor/build.prop	131072	1	-2
init	1	1	ee028000	openat	-100	/factory/factory.prop	131072	1	-2
init	1	1	ee028000	openat	-100	/data/local.prop	131072	1	-2
init	1	1	ee028000	openat	-100	/data/property	147456	1	8
init	1	1	ee028000	openat	-100	/sys/kernel/debug/tra...	163840	1	8

System Call Types

Name	Number
mprotect	125
munmap	91
nanosleep	162
openat	322
pipe2	359
prctl	172
pread64	180
pwrite64	181
read	3
recvfrom	292
recvmsg	297
renameat	329

System Call Operation

System Call Data

```
00000000  0a 23 20 62 65 67 69 6e  20 62 75 69 6c 64 20 70   .# begin build p
00000010  72 6f 70 65 72 74 69 65  73 0a 23 20 61 75 74 6f   roperties.# auto
00000020  67 65 6e 65 72 61 74 65  64 20 62 79 20 62 75 69   generated by bui
00000030  6c 64 69 6e 66 6f 2e 73  68 0a 72 6f 2e 62 75 69   ldinfo.sh.ro.bui
00000040  6c 64 2e 69 64 3d 4c 4d  59 34 38 58 0a 72 6f 2e   ld.id=LMY48X.ro.
00000050  62 75 69 6c 64 2e 64 69  73 70 6c 61 79 2e 69 64   build.display.id
00000060  3d 73 64 6b 5f 70 68 6f  6e 65 5f 61 72 6d 76 37   =sdk_phone_armv7
00000070  2d 75 73 65 72 64 65 62  75 67 20 35 2e 31 2e 31   -userdebug 5.1.1
00000080  20 4c 4d 59 34 38 58 20  33 30 37 39 31 35 38 20    LMY48X 3079158
00000090  74 65 73 74 2d 6b 65 79  73 0a 72 6f 2e 62 75 69   test-keys.ro.bui
000000a0  6c 64 2e 76 65 72 73 69  6f 6e 2e 69 6e 63 72 65   ld.version.incre
000000b0  6d 65 6e 74 61 6c 3d 33  30 37 39 31 35 38 0a 72   mental=3079158.r
000000c0  6f 2e 62 75 69 6c 64 2e  76 65 72 73 69 6f 6e 2e   o.build.version.
000000d0  73 64 6b 3d 32 32 0a 72  6f 2e 62 75 69 6c 64 2e   sdk=22.ro.build.
000000e0  76 65 72 73 69 6f 6e 2e  63 6f 64 65 6e 61 6d 65   version.codename
000000f0  3d 52 45 4c 0a 72 6f 2e  62 75 69 6c 64 2e 76 65   =REL.ro.build.ve
00000100  72 73 69 6f 6e 2e 61 6c  6c 5f 63 6f 64 65 6e 61   rsion.all_codena
00000110  6d 65 73 3d 52 45 4c 0a  72 6f 2e 62 75 69 6c 64   mes=REL.ro.build
```

Ready

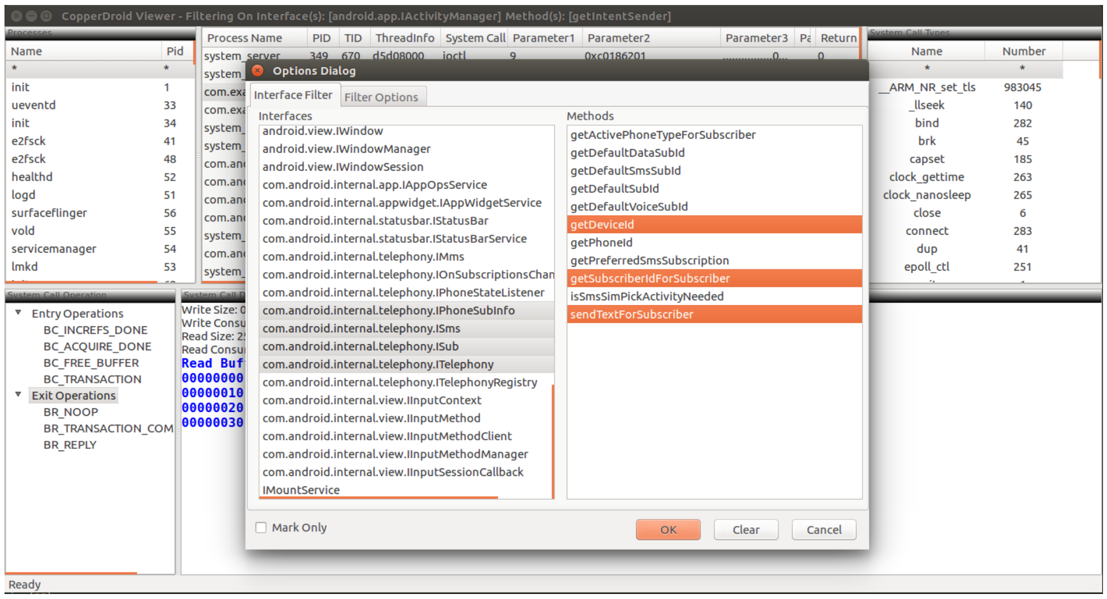
Figure 3.4: Lollipop 5.1.1 `openat` and `read` of `/system/build.prop`

B.2, B.3 and B.4 in Appendix B all show both the client and the server in the system call window (although only the client side Binder details are visible as the client side calls are selected).

The various portions of the CopperDroid v2 Viewer are populated by the contents of the protobuf file, this includes system call types, processes and the system calls invoked by each process. Additionally, every interface and method invoked is included in the options dialog (shown in Figure 3.5). Therefore, if a given system call or interface isn't invoked it won't appear in the viewer.

While automated analysis can achieve a lot, there are times when it is necessary to have a person inspect an operation or set of operations. For instance, when Google Bouncer flags something, a human operator will look at it to confirm whether or not the action is malicious [10]. The CopperDroid v2 Viewer has proven very useful to the author in debugging issues with Binder reconstruction and in verifying that changes made by plug-ins are marshalled correctly. Plug-ins are invoked prior to each system call (or Binder call) actually being invoked and before the data has been serialised resulting in plug-in modified data being collected in the protobuf.

Information leakage detection and anti-evasion described in Chapter 6 and Chapter 7 respectively replace entire files by intercepting `open`, `lseek`, `fstat`, `read` and `close`. The CopperDroid v2 Viewer makes it possible isolate the calls involved in opening a file, determining the length and actually reading it by allowing one to filter and mark specific system call types in a given process and then jump from instance to instance while still seeing all the other system calls occurring within that process.



CopperDroid Viewer - Filtering On Interface(s): [android.app.IActivityManager] Method(s): [getIntentSender]

Processes

Name	Pid
*	*
init	1
ueventd	33
init	34
e2fsck	41
e2fsck	48
healthd	52
logd	51
surfaceflinger	56
vold	55
servicemanager	54
lmkd	53

Process Name	PID	TID	ThreadInfo	System Call	Parameter1	Parameter2	Parameter3	Pa	Return
system_server	349	670	d5d08000	ioctl	9	0xc0186201	...............0...		0

System Call Types

Name	Number
*	*
__ARM_NR_set_tls	983045
_llseek	140
bind	282
brk	45
capset	185
clock_gettime	263
clock_nanosleep	265
close	6
connect	283
dup	41
epoll_ctl	251

System Call Operation

- Entry Operations
  - BC_INCREFS_DONE
  - BC_ACQUIRE_DONE
  - BC_FREE_BUFFER
  - BC_TRANSACTION
- Exit Operations
  - BR_NOOP
  - BR_TRANSACTION_COM
  - BR_REPLY

Options Dialog

Interface Filter | Filter Options

Interfaces
- android.view.IWindow
- android.view.IWindowManager
- android.view.IWindowSession
- com.android.internal.app.IAppOpsService
- com.android.internal.appwidget.IAppWidgetService
- com.android.internal.statusbar.IStatusBar
- com.android.internal.statusbar.IStatusBarService
- com.android.internal.telephony.IMms
- com.android.internal.telephony.IOnSubscriptionsChan
- com.android.internal.telephony.IPhoneStateListener
- com.android.internal.telephony.IPhoneSubInfo
- com.android.internal.telephony.ISms
- com.android.internal.telephony.ISub
- com.android.internal.telephony.ITelephony
- com.android.internal.telephony.ITelephonyRegistry
- com.android.internal.view.IInputContext
- com.android.internal.view.IInputMethod
- com.android.internal.view.IInputMethodClient
- com.android.internal.view.IInputMethodManager
- com.android.internal.view.IInputSessionCallback
- IMountService

Methods
- getActivePhoneTypeForSubscriber
- getDefaultDataSubId
- getDefaultSmsSubId
- getDefaultSubId
- getDefaultVoiceSubId
- getDeviceId
- getPhoneId
- getPreferredSmsSubscription
- getSubscriberIdForSubscriber
- isSmsSimPickActivityNeeded
- sendTextForSubscriber

☐ Mark Only

Figure 3.5: Lollipop 5.1.1 Options Dialog showing several interfaces and methods

The Options Dialog in Figure 3.5 includes a *Mark Only* check-box which, when selected, will highlight system calls matching the filters in blue while leaving all others black. The Alt-N and Alt-P key combinations then loop through the next and previous instances of matching system or Binder calls.

3.9 Binder ICC Reconstruction

This section describes how the fixed structures passed to `ioctl` for the `/dev/binder` device are parsed. It also describes how the interface and method call in a given invocation are determined, however, actual call parameters and return value marshaling is described in the Chapter 4. This chapter does, however, describe the interaction between the work performed here with that of the marshaling code generated as a result of the static analysis described in Chapter 4.

3.9.1 Introduction

Binder interprocess communication is accomplished using a kernel driver that is responsible for moving the information for calls and their responses between client and server (as outlined in Section 2.1). It should be noted that the client and server refer to the initiator of an IPC request and the target to which that request is sent. For example, while the LocationManager service is the server-side of a location information query, when it sends a one-way location update notifications to an app that requested them, it is at that point the client and the app is a server.

Binder exposes a device `/dev/binder` and implements a protocol[4] on top of the `ioctl` system call provided by the Linux kernel. This protocol provides the necessary infrastructure to support fast interprocess communication between apps and services. Apps may also communicate with one another, provided the interface definition has been shared. Furthermore, services communicate with one another too, which is all performed through Binder IPC.

Binder IPC has the same basic architecture as Remote Procedure Calls (RPC) on other platforms. When a specific interface method is invoked, a proxy/stub component for that interface marshals the parameters into a single allocation (a `Parcel` in Binder's case), it then invokes the remote method and copies or maps the allocation into the target server. The server then unmarshals the parameters from this allocation and invokes the specific method passing in the unmarshaled parameters. Replies are handled in the same way with the difference that they are marshaled on the server, copied or mapped into the client and then unmarshaled and returned to the original caller.

The `Parcel` structure plays a key role in marshaling information between client and server. There is also a `Parcel` class provided to marshal and unmarshal data to and from raw data contained within the `Parcel`. `Parcels` can also be accessed using native `C++` code via the `Java` Native Interface (JNI). The author of this thesis implemented a hand-written `Parcel` class in CopperDroid v2. There are a few reasons for this, firstly, the hand-written class is aware of host and guest memory and can therefore move data back and forth as needed. Secondly, an attacker could, in theory, attack Binder by specially crafting `Parcels` [62] [26] and a separate and distinct implementation could make it harder to attack CopperDroid v2 through this (since both implementations would have to be similarly compromised). Thirdly, the `Parcel` class implementation in Android uses a combination of `Java` and native code (including the JNI), complicating automatic code generation. Finally, there are some constructs in `Java` that are tricky to translate to `C++` (but not impossible) and implementing the class by hand is faster.

[4]The author refers to this as the Binder Protocol in this thesis

3.9.2 The Binder Protocol

The well-defined `ioctl` mechanism mentioned above passes different structures between the app and the `/dev/binder` device, the most interesting of these being the `binder_write_read` structure (see Listing 3.4), and the marshaled data contained in the location pointed to by either the `write_buffer` or `read_buffer` members of this structure. The Binder driver has a *command* and *return* protocol. These are generally both present in a single client-server round trip. The `write_buffer` contains the commands sent from the client to the server. The `read_buffer` contains the reply from the server.

```
struct binder_write_read {
    binder_size_t       write_size; /* bytes to write */
    binder_size_t       write_consumed; /* bytes consumed by driver */
    binder_uintptr_t    write_buffer;
    binder_size_t       read_size;  /* bytes to read */
    binder_size_t       read_consumed;  /* bytes consumed by driver */
    binder_uintptr_t    read_buffer;
};
```

Listing 3.4: binder_write_read structure

Binder IPC calls work in the following manner, a `BINDER_WRITE_READ ioctl` is sent to the Binder driver, the `write_buffer` containing the commands, with the `write_size` specifying the complete size of the buffer, while `write_consumed` contains the length of the buffer actually utilised[5].

The `write_buffer` contains one or more `BC_*` commands (such as those in Listing 3.5). Commands are packed into the `write_buffer` one after another. While one may be tempted to simply loop through every looped through every byte in the `write_buffer` up to `write_size-4` and checking every 4 bytes to see if they match either `BC_TRANSACTION/BC_REPLY` or `BR_TRANSACTION/BR_REPLY`, this could lead to issues. For example some commands contain addresses which could potentially have the same value as `BC_TRANSACTION (0x40286300)`, `BC_REPLY (0x40286301)`, `BR_TRANSACTION (0x80287202)` or `BR_REPLY (0x80287203)`, leading to undefined behaviour including instability.

```
enum binder_driver_command_protocol {
    BC_TRANSACTION = _IOW('c', 0, struct binder_transaction_data),
    BC_REPLY = _IOW('c', 1, struct binder_transaction_data),
    /*
    * binder_transaction_data: the sent command.
    */
    ...
    BC_INCREFS = _IOW('c', 4, __u32),
```

[5]The author of this thesis obtained the information regarding the Binder protocol through experimentation and inspection of the Binder driver source code

```
    BC_ACQUIRE = _IOW('c', 5, __u32),
    BC_RELEASE = _IOW('c', 6, __u32),
    BC_DECREFS = _IOW('c', 7, __u32),
    /*
    * int:  descriptor
    */
    ...
};
```

Listing 3.5: Binder command protocol (code omitted for brevity)

Only `write_size` bytes need to be examined, as the caller knows the size of the buffer it is sending and only includes that many bytes. The `write_consumed` value is updated by the Binder driver (invoked after CopperDroid v2 inspects the `ioctl` data). CopperDroid v2, however, needs to intercept and potentially modify a given transaction prior to the `ioctl` actually being executed (and reaching the Binder driver) therefore, the `write_size` is used in determining what is contained with the `ioctl` request. An example of a `write_buffer` including `write_size` and `write_consumed` can be seen in Figure B.5, in Appendix B.

```
enum binder_driver_return_protocol {
    BR_ERROR = _IOR('r', 0, __s32),
    /*
    * int: error code
    */

    BR_OK = _IO('r', 1),
    /* No parameters! */

    BR_TRANSACTION = _IOR('r', 2, struct binder_transaction_data),
    BR_REPLY = _IOR('r', 3, struct binder_transaction_data),
    /*
    * binder_transaction_data: the received command.
    */
    ...
    BR_INCREFS = _IOR('r', 7, struct binder_ptr_cookie),
    BR_ACQUIRE = _IOR('r', 8, struct binder_ptr_cookie),
    BR_RELEASE = _IOR('r', 9, struct binder_ptr_cookie),
    BR_DECREFS = _IOR('r', 10, struct binder_ptr_cookie),
    /*
    * void *:   ptr to binder
    * void *: cookie for binder
    */
    ...
};
```

Listing 3.6: Binder return protocol (code omitted for brevity)

The `read_buffer` similarly contains one or more `BR_*` return commands (such as those in Listing 3.6). Return commands are also packed into the `read_buffer` one after another. Replies to Binder calls are returned using a `BR_REPLY` return command.

CopperDroid v2 processes the return path on `ioctl` exit and uses the `read_consumed` value when analysing the `read_buffer`. As the caller does not know ahead of time how many bytes will be in the response, a larger buffer, specified in `read_size`, is always sent (often 256 bytes). However, only the bytes actually used for the reply are needed and these are reflected in the `read_consumed` value. An example of a `read_buffer` including `read_size` and `read_consumed` can be seen in Figure B.6.

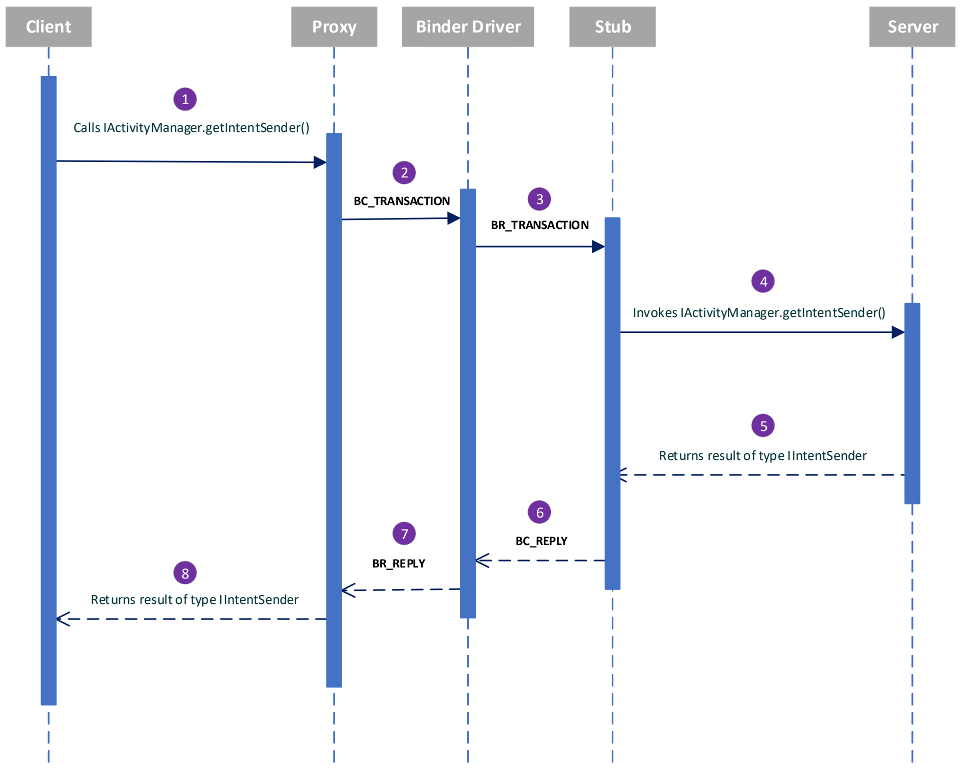

Client
Proxy
Binder Driver
Stub
Server
1
Calls IActivityManager.getIntentSender()
2
BC_TRANSACTION
3
BR_TRANSACTION
4
Invokes IActivityManager.getIntentSender()
5
Returns result of type IIntentSender
6
BC_REPLY
7
BR_REPLY
8
Returns result of type IIntentSender

Figure 3.6: Sequence of an app call to IActivityManager getIntentSender and corresponding reply

CopperDroid v2 fully interprets both the *command* and *return* protocols and is able to reconcile calls and their responses correctly. Full protocol support also opens the path for further research into the resiliency of Binder to attacks targeting its referencing of objects (fpr which malware could certainly craft attacks using native code and accessing the `/dev/binder` device directly).

Figure 3.6 is an example sequence diagram for the `IActivityManager getIntentSender()` interface method call[6]. On the client side, the proxy marshals the parameters into a `Parcel` and issues an ioctl to the `/dev/binder` device which includes a BC_TRANSACTION (2) structure with the data pointing to the `Parcel`. The `/dev/binder` driver translates any flat_binder_object structures between processes and completes a waiting `ioctl` on the server passing it a BR_TRANSACTION (3) which includes the (potentially) modified `Parcel`. The server stub unmarshals the `Parcel` into the parameters and invokes the `getIntentSender()` method (4).

[6]The IActivityManager getIntentSender() method is used as a running example throughout this thesis because it is a non-trivial interface method to reconstruct and demonstrates several different facets of Binder communication.

The reply performs the same steps in reverse using BC_REPLY and BR_REPLY on the server and client sides respectively.

In general multiple operations are sent at a time, for example, BC_INCREFS, BC_FREE_BUFFER and BC_TRANSACTION can often be seen in a single write buffer. In the case of an immediately completed request, it is not unusual to see a BR_NOOP, BR_TRANSACTION_COMPLETE and BC_REPLY in the read buffer associated with a given request.

Most Binder calls are synchronous and return on the same thread as they were initiated[7]. Others are one-way calls that reply via a broadcast receiver. Binder transaction requests include a code, which translates to the name of the method that the Binder transaction invokes, and an interface token, which represents the interface to which the method belongs. Conversely, Binder transaction replies include only an exception code of 0 (if successful) and the return result. In case of a failure the reply consists of a non-zero exception code and a string detailing the exception. The return value of the target function is contained within the reply and is dependent on the return type of the function, either a primitive or complex type. Therefore, when examining Binder replies, it is necessary to know the return type associated with a given call. This information is present in the Android Interface Definition Language (AIDL) files for most interfaces [34] and in `Java` files for some interfaces such as `IActivityManager`, `IContentProvider` and `IServiceManager`.

`Intents` permit a particular app or service to carry out an operation on behalf of a caller. These can either be acted upon immediately or held for later use via a `PendingIntent`. For example when sending an SMS text message, including a sent and a delivery `PendingIntent` enables the Telephony Service to notify the caller when the message is sent and once it has been delivered. The `PendingIntent` is obtained via a call to the `IActivityManager.getIntentSender()` method.

Figure 3.7 demonstrates the call (Binder transaction) that registers the PendingIntent while Figure 3.8 shows the response with the handle to the PendingIntent.

The second field in the binder_transaction_data structure as shown in Figure 3.7 is the `code` field and refers to the interface method to which the transaction is referring.

All Android interfaces include an `onTransact()` method in their stub and the `code` specifies which transaction to execute. An example `onTransact()` method is shown in Listing 4.4 in Section 4.6. The prototype for the method can be seen in List-

[7]The author determined this by debugging Binder `ioctl` system call entry/exit using the CopperDroid v2 emulator

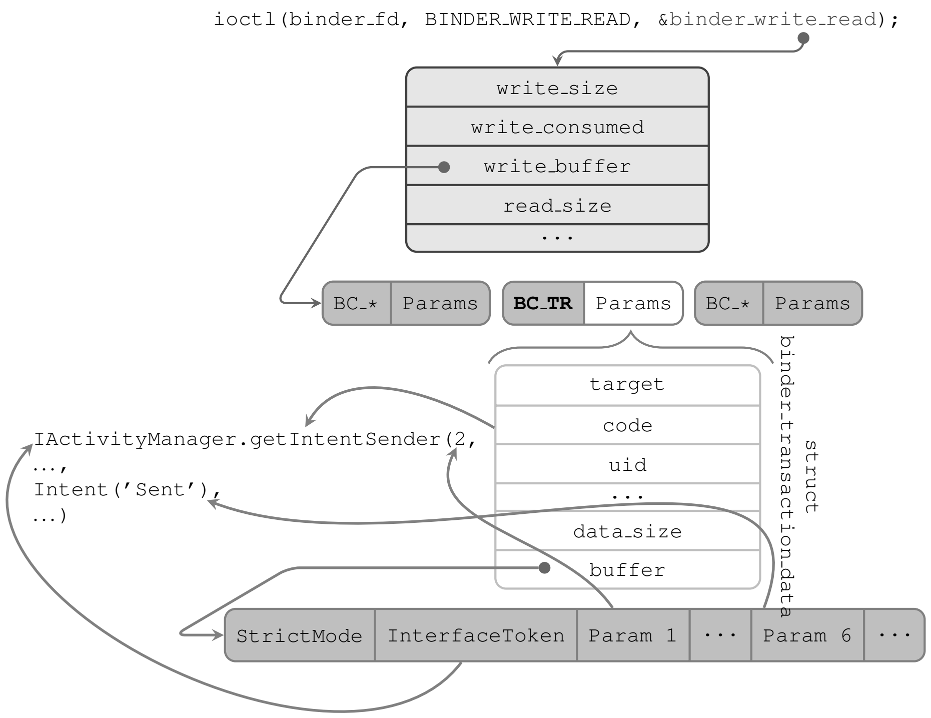
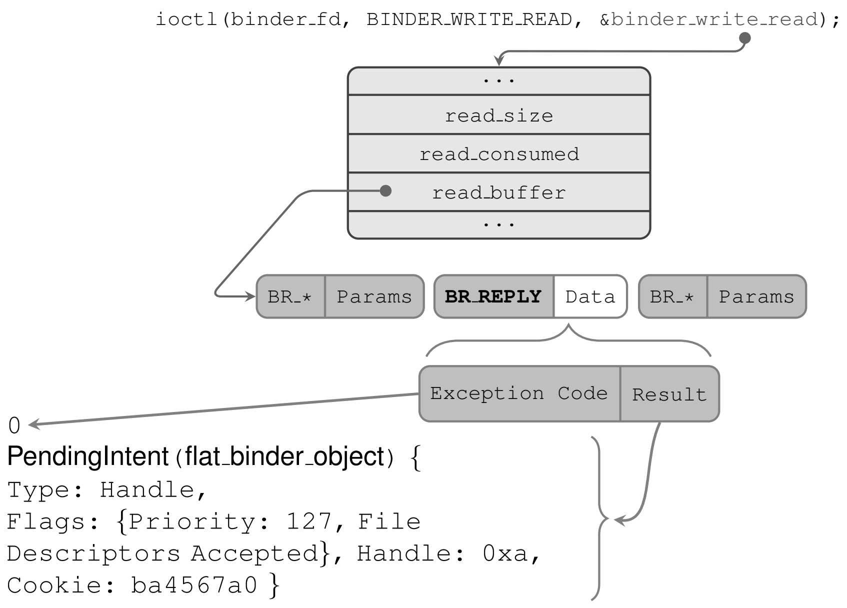


Figure 3.7: An example of a Binder transaction that registers a pending Intent with the IActivityManager that is later used for notification



ioctl(binder_fd, BINDER_WRITE_READ, &binder_write_read);
...
read_size
read_consumed
read_buffer
...
BR_* Params
BR_REPLY Data
BR_* Params
Exception Code
Result
0
PendingIntent (flat_binder_object) {
Type: Handle,
Flags: {Priority: 127, File
Descriptors Accepted}, Handle: 0xa,
Cookie: ba4567a0 }

Figure 3.8: An example of a Binder reply that receives the information corresponding to the PendingIntent registered with the IActivityManager

ing 3.7.

```
@Override
public
boolean onTransact(int code, Parcel data, Parcel reply, int flags) throws
    RemoteException;
```

Listing 3.7: Interface onTransact method prototype

Listing 3.12 is taken from a test program written by the author that encrypts sensitive data and sends it via SMS. The program is used to test which portions of entropy in the system need to be controlled for secure key generation to generate the same keys from the initial seed and will be further described in Chapter 6. How the code in this example is marshaled into a `Parcel` can be seen in Figure 3.9. Chapter 4 describes in detail the analyses that ultimately enable the automatic unmarshaling of such Binder transactions.

Binder interfaces optionally have a `StrictMode` policy at the start of the `Parcel` (stored as a 32-bit integer). However, not all interfaces and interface methods have `StrictMode` settings. CopperDroid v2 needs to determine the target interface for a given `Parcel` which requires parsing the raw `Parcel`.

For an interface without `StrictMode` policy, this would just involve reading the interface descriptor string (32-bit integer *length* plus the number of Unicode characters denoted by the *length*, plus any padding which may be present) as per Table 3.6, later in this chapter.

All components of a `Parcel` are 4-byte aligned, which is why a `byte` is stored as a 32-bit integer (see Table 3.6 later in this chapter). However, if `StrictMode` policy is present, then that needs to be read first and **then** the interface descriptor needs to be read. Therefore CopperDroid v2 uses heuristics to determine if `StrictMode` is present and does so by determining if the 32-bit integer at the beginning of the `Parcel` has a value higher than that of the length of the `Parcel`, in which case it is `StrictMode` policy. If the length is smaller than the total `Parcel` then the two `uint16_t` values immediately following the first 32-bit integer are analysed to determine if they are valid `UNICODE` characters. If not, the first 32-bit integer is `StrictMode` policy, otherwise the first 32-bit integer is the length of the interface descriptor string.

If it is determined that `StrictMode` is present then the policy value is read and stored and the presence of `StrictMode` is noted, if not, then that is noted too. The interface descriptor is read from the `Parcel` next and this is used to invoke code to unmarshal the rest of the `Parcel`. There is a non-heuristic based approach that can be taken at some point (if it is necessary). The `binder_transaction_data` structure

```
struct binder_transaction_data {
    /* The first two are only used for bcTRANSACTION and brTRANSACTION,
    * identifying the target and contents of the transaction.
    */
    union {
        __u32   handle; /* target descriptor of command transaction */
        binder_uintptr_t ptr;   /* target descriptor of return transaction */
    } target;
    binder_uintptr_t    cookie; /* target object cookie */
    __u32       code;       /* transaction command */

    /* General information about the transaction. */
    __u32       flags;
    pid_t       sender_pid;
    uid_t       sender_euid;
    binder_size_t   data_size;  /* number of bytes of data */
    binder_size_t   offsets_size;   /* number of bytes of offsets */

    /* If this transaction is inline, the data immediately
    * follows here; otherwise, it ends with a pointer to
    * the data buffer.
    */
    union {
        struct {
            /* transaction data */
            binder_uintptr_t    buffer;
            /* offsets from buffer to flat_binder_object structs */
            binder_uintptr_t    offsets;
        } ptr;
        __u8    buf[8];
    } data;
};
```

Listing 3.8: Binder Transaction Data used in Binder IPC

```
typedef struct _flat_binder_object {
/* 8 bytes for large_flat_header. */
__u32   type;
__u32   flags;

/* 8 bytes of data. */
union {
binder_uintptr_t    binder; /* local object */
__u32           handle; /* remote object */
};

/* extra data associated with local object */
binder_uintptr_t    cookie;
} flat_binder_object;
```

Listing 3.9: Flat Binder Object used for referencing data specific to either the client or server

in Binder has a `target` field which is either a handle or a pointer value (see Listing 3.11). From the client side, this is a Binder handle value representing the service in question and is obtained by a call the `IServiceManager.getService()` method (with the handle being returned in a `flat_binder_object`). Since CopperDroid v2 observes everything in the system, the `IServiceManager.getService()` method call and reply can be used to track services. Thus far this has not been necessary, but remains a possibility should it be required.

A `flat_binder_object` is returned from the `getIntentSender()` call, shown in Listing 3.9. This structure is used for passing references to objects between processes with the Binder driver updating the `binder/handle` field to reference the object within the process in question.

CopperDroid v2 wraps the `flat_binder_object` structure into a class called `BinderReference`, this class includes helper methods for determining the object in question is a handle, reference or file descriptor. It also includes an `AsString()` method that returns a string representation of a `flat_binder_object`, as can be seen from the following text copied from output of the CopperDroid v2 viewer. This is discussed in more detail in Chapter 4.

```
android.app.IActivityManager
getIntentSender(type = 1, packageName = "com.example.binderdemo", token = BinderReference {
     Type:"Binder Strong Reference", Flags: {"Priority: 127, File Descriptors Accepted" },
     Binder: 0x0, Cookie: 0 }, resultWho = "(null)", requestCode = 0, intents = [ "{ "mAction
     " : "Sent", "mData" : null, "mType" : "(null)", "mPackage" : "(null)", "mComponent" :
     null, "mFlags" : 0, "mCategories" : [ ], "mExtras" : null, "mSourceBounds" : null, "
     mSelector" : null, "mClipData" : null, "mContentUserHint" : -2 }" ], resolvedTypes = [
     ], flags = 0, options = null, userId = 0)
```

Listing 3.10: IActivityManager getIntentSender call parameters

The result is also a `flat_binder_object` and therefore encapsulated by a `BinderReference` class.

```
android.app.IActivityManager
getIntentSender()
exceptionCode = 0
res = BinderReference { Type:"Binder Handle", Flags: {"Priority: 127, File Descriptors
     Accepted" }, Binder: 0x8, Cookie: a5243f80 }
```

Listing 3.11: IActivityManager getIntentSender reply data

The `offsets_size` field refers to the size of the `offsets` field in `binder_transaction_data` (see Listing 3.11) where each offset is of type `binder_uintptr_t` which is 4 bytes in length on a 32-bit Android system and 8 bytes in length on a 64-bit Android system.

Listing 3.13 shows the full `Parcel` which is produced by calling the `sendText` method on the `ISms` interface. The target refers to the service that the app is communicating with, in this case the SMS service.

The code that produces this parcel is shown in Listing 3.12 and a visual representation is shown in Figure 3.9. The `Parcel` in Listing 3.13 is color coded to demonstrate the different components that make up the `Parcel` and these match those shown in Figure 3.9.

The first 4 bytes are the `StrictMode` policy (in gray). The next portion is the interface token, followed by the package name and all the different parameters.

The `flat_binder_objects` present in the `Parcel` are prefixed by a violet 32-bit value (0x00000001) representing the boolean value of true, indicating the presence of the `flat_binder_object`. If null had been specified in the call, then this would have been 0x00000000 (false). A null string is encoded as -1 (0xffffffff) and can be seen

```java
public class MainActivity extends AppCompatActivity implements View.OnClickListener {
    ...
    private Button m_SendSMS;
    PendingIntent piSent;
    PendingIntent piDelivered;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        piSent = PendingIntent.getActivity(this, 0, new Intent("Sent"), 0);
        piDelivered = PendingIntent.getActivity(this, 0, new Intent("Delivered"), 0);
        ...
    }
    ...
    @Override
    public void onClick(View view) {
        ...
        if (view == m_SendSMS) {
            ...
            byte[] encryptedData = SecurityUtils.encryptAES(seedValue, LeakString);
            String encryptedText;
            encryptedText = AESHelper.toHex(encryptedData);

            SmsManager smsManager = SmsManager.getDefault();
            smsManager.sendTextMessage("0123456789", null, encryptedText, piSent,
                piDelivered);
        }
        ...
    }
}
```

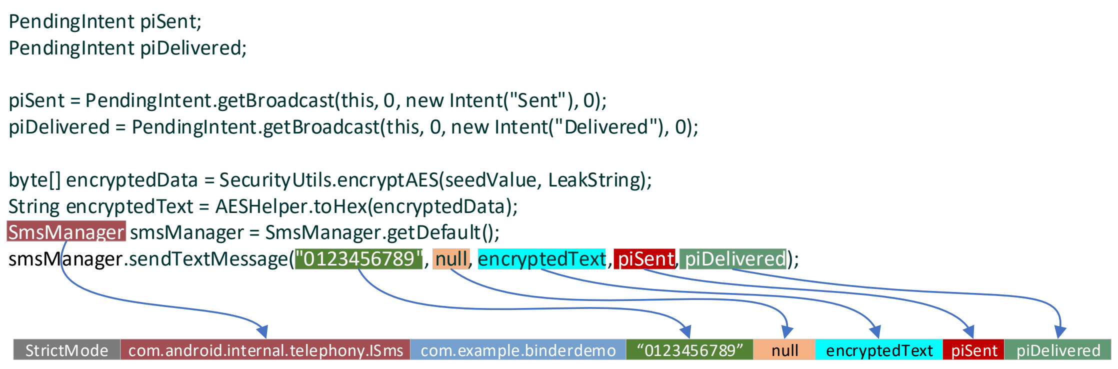
Listing 3.12: Sample showing creation of PendingIntents which are used in a SMS sendText call

```
Target: 32, Cookie: 0, Code: 10, Flags: 16, SenderPid: 0,
SenderEuid: 0, Datasize: 536, OffsetsSize: 8

Buffer:
    00000000  04 03 00 00 23 00 00 00 63 00 6f 00 6d 00 2e 00   ....#...c.o.m...
    00000010  61 00 6e 00 64 00 72 00 6f 00 69 00 64 00 2e 00   a.n.d.r.o.i.d...
    00000020  69 00 6e 00 74 00 65 00 72 00 6e 00 61 00 6c 00   i.n.t.e.r.n.a.l.
    00000030  2e 00 74 00 65 00 6c 00 65 00 70 00 68 00 6f 00   ..t.e.l.e.p.h.o.
    00000040  6e 00 79 00 2e 00 49 00 53 00 6d 00 73 00 00 00   n.y...I.S.m.s...
    00000050  01 00 00 00 16 00 00 00 63 00 6f 00 6d 00 2e 00   ........c.o.m...
    00000060  65 00 78 00 61 00 6d 00 70 00 6c 00 65 00 2e 00   e.x.a.m.p.l.e...
    00000070  62 00 69 00 6e 00 64 00 65 00 72 00 64 00 65 00   b.i.n.d.e.r.d.e.
    00000080  6d 00 6f 00 00 00 00 00 0a 00 00 00 30 00 31 00   m.o.........0.1.
    00000090  32 00 33 00 34 00 35 00 36 00 37 00 38 00 39 00   2.3.4.5.6.7.8.9.
    000000a0  00 00 00 00 ff ff ff ff a0 00 00 00 37 00 33 00   ............7.3.
    000000b0  34 00 34 00 39 00 31 00 33 00 31 00 44 00 44 00   4.4.9.1.3.1.D.D.
    000000c0  30 00 30 00 36 00 30 00 46 00 45 00 41 00 43 00   0.0.6.0.F.E.A.C.
    ...
    000001d0  32 00 42 00 35 00 45 00 39 00 33 00 37 00 31 00   2.B.5.E.9.3.7.1.
    000001e0  34 00 32 00 44 00 39 00 32 00 36 00 00 00 00 00   4.2.D.9.2.6.....
    000001f0  01 00 00 00 85 2a 68 73 7f 01 00 00 08 00 00 00   .....*hs........
    00000200  00 00 00 00 01 00 00 00 85 2a 68 73 7f 01 00 00   .........*hs....
    00000210  0a 00 00 00 00 00 00 00                           ........

Offsets:
    00000000  f4 01 00 00 08 02 00 00                           ........
```

Listing 3.13: Parcel from a calling sendText on the ISms interface

in orange in Listing 3.13 where the source address parameter is omitted (and null passed instead).

The offsets shown below the buffer are offsets into the parcel where the `flat_binder_object` structures are located, when the Binder driver is invoked in



```
PendingIntent piSent;
PendingIntent piDelivered;

piSent = PendingIntent.getBroadcast(this, 0, new Intent("Sent"), 0);
piDelivered = PendingIntent.getBroadcast(this, 0, new Intent("Delivered"), 0);

byte[] encryptedData = SecurityUtils.encryptAES(seedValue, LeakString);
String encryptedText = AESHelper.toHex(encryptedData);
SmsManager smsManager = SmsManager.getDefault();
smsManager.sendTextMessage("0123456789", null, encryptedText, piSent, piDelivered);
```

Figure 3.9: Parceling of a call

a given process, prior to transferring the `Parcel` to the target process, any `flat_binder_objects` in the parcel are translated to their counterparts in the target parcel. For example, while the handle value for the *sent* `PendingIntent` object is 0x00000008 in the listing shown below, it is 0x0000000f in the target process (com.android.phone) – obtained by examining the communication between client and server using the CopperDroid v2 Viewer, developed by the author.

Table 3.6 demonstrates examples of how different types are packed within a `Parcel`. This is not an exhaustive list, however, it should be noted that all values are always 4-byte aligned, so even boolean values (requiring one byte of storage) are represented using 4 bytes.

For transactions, the parcel contains the interface name and all of the parameters being passed to the method, including both simple and complex types. Perhaps the most well-known of the internal Android interfaces is the IActivityManager, as it is central to supporting Android apps. As shown in figure 3.7, when an Android app registers a `PendingIntent`, this is managed through the `IActivityManager` method, `getIntentSender()`. In this case, the Binder reply contains a Binder handle reference. Figure 3.8 (earlier in the chapter) depicts the reply which in this case contains a Binder handle reference.

Content providers also register with the Activity Manager and apps that utilise a given content provider communicate with the Activity Manager (via the IActivityManager interface) to obtain a reference to the content provider in question. This is covered in more detail in Chapter 6 as it is relevant in determining when a content provider is queried for (potentially sensitive) information.

Public interfaces are shared via AIDL files and CopperDroid v2 initially used an AIDL compiler[8] modified by the author to generate `C++` code to reconstruct interface

[8]Compiler in this thesis refers to a source-to-source compiler

Parcel Type	Packing
int	int32
byte	int32 0-255
bool	int32 0 : false 1 : true
String	int32 length + Unicode characters + padding -1 : null 0 : empty string
Array	int32 item count + array -1 : null
Bundle	int32 length + BNDL magic + data -1 : null
Objects	object data -1 : null

Table 3.6: Packed representation of types within a Parcel

communications. However, as in CopperDroid v1, this only assists in obtaining the overall structure of the interface method call, but can not reconstruct complex objects. It also suffers from the same shortcoming of CopperDroid v1 where the length of complex objects within a `Parcel` can only be determined by actually unmarshaling them (which results in any simple types following complex objects being incorrectly reconstructed) and that is where the unmarshaling Oracle is required. However, as mentioned in Chapter 2, the unmarshaling Oracle suffers from performance and stability issues. For example, it throws an exception which crashes the Oracle when an object containing embedded `IBinder` types is included in the `Parcel` and it attempts to reconstruct the obejct without the correct context. It is also unable to reconstruct interfaces such as `IActivityManager` (implemented in `Java`) as it needs to be provided with all the types contained a `Parcel`.

3.9.3 Parsing The Binder Protocol

Section 3.9.1 describes a `Parcel` class written by the author which enables CopperDroid v2 to determine the interface name specified within a given `Parcel` while

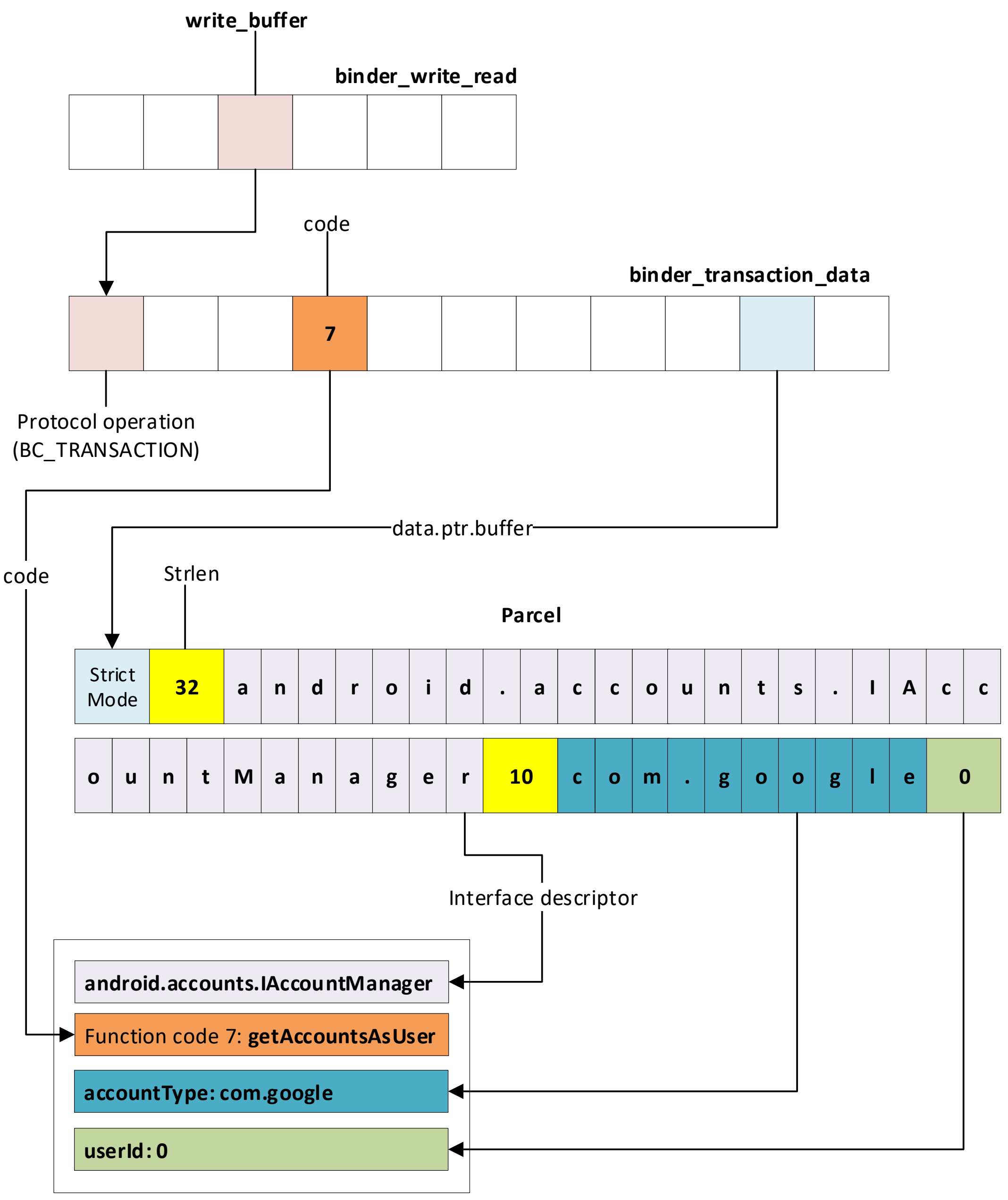


Figure 3.10: Breakdown of a Binder Transaction

also handling the optional `StrictMode` policy. Figure 3.10 demonstrates how the `binder_write_read` structure is used to access a given operation's `binder_transaction_data` and finally the `Parcel`. For every structure that the Binder protocol includes, the author implemented classes that understand how to parse and even modify each Binder structure. These classes assist in unmarshaling and, in the case of modifications, marshaling a `Parcel` correctly.

The `open` system call entry handler (`open_stub_enter_syscall()`) mentioned in Section 3.6.5 instantiates a BinderObject class instance if the /dev/binder device is opened. This class inherits from the KernelObject class defined in Listing A.15 in Appendix A.

Only /dev/ashmem and /dev/binder are currently tracked internally within CopperDroid v2. While CopperDroid v2 tracks anonymous shared memory objects (ashmem), it currently doesn't examine them in much detail. Support was added for this because it was thought that PendingIntent may be returned using an ashmem region, however, the author disproved this idea by determining that they are in fact allocated by calling the IActivityManager interface's getIntentSender() method.

The BinderObject class derives from the KernelObject class and implements the GetObjectType(), IoCtl() and IoCtlExit() calls. When the CopperDroid v2 ioctl handler is invoked on entry, it attempts to locate the corresponding KernelObject using the file descriptor. If it locates an object, it then invokes the IoCtl method. The BinderObject class's IoCtl method then proceeds to parse the Binder protocol. For a BINDER_WRITE_READ ioctl, this involves copying the binder_write_read structure from the target address in the guest to the host. The write_buffer is copied next and each *command* is stored in an instance of a BinderOperation class (written by the author) which has the requisite knowledge on the contents of each *command* and *return* type (BC_*/BR_*).

The BinderOperation class holds the logic for parsing the top level of every *command* and *return*. It also encapsulates a BinderTransaction class which knows how to parse the fixed portions of Binder transactions and replies. The BinderTransaction class allocates a Parcel object and initializes it with the data obtained from the guest and then parses it. The BinderObject class tracks transactions and replies using the thread ID of the calling thread. When a BC_TRANSACTION or BR_TRANSACTION occurs, and the transaction is not one-way, the details of the transaction are extracted (interface, method, parameters etc.) and the current operation is inserted into an unordered_map using the current task's thread ID as the key (line 27 in Listing A.16). When a subsequent reply (BC_REPLY or BR_REPLY) occurs, the previous transaction's operation is located using the thread ID as the key (line 14 in Listing A.16).

3.9.4 Extracting Interface Call Information

Extraction of Binder data for transactions and replies is accomplished using the IInterfaceDataExtractor interface, shown in Listing 3.14. This interface is implemented using code automatically generated through static analysis of the Android Framework and is described in Chapter 4.

```
class IInterfaceDataExtractor
{
public:
    virtual STATUS ExtractData(string InterfaceName, int code, Parcel* parcel,
        BinderTransaction* binderTransaction, bool isTransaction) = 0;
};
```

Listing 3.14: IInterfaceDataExtractor interface

When the `ExtractOperationData()` is called (line 17 of Listing A.16), it will first utilise the `ExtractData()` method shown Listing 3.14 to unmarshal the `Parcel`. If a plug-in is present, it will call the plug-in passing in the unmarshaled data wrapped in a class (described in Chapter 4). If the plug-in modifies the `Parcel` data then the data will be marshaled into a new `Parcel` and overwritten in the same memory location as the original `Parcel` on the guest. Currently modifications to `Parcels` cannot exceed the original data size due to the fact that CopperDroid v2 is executing outside of the guest and can't allocate memory in the guest to handle larger `Parcels`.

3.9.5 Serialising Binder Information

Section 3.7 covers how system calls are serialised in CopperDroid v2, but does not cover Binder. When an `ioctl` is serialised, the `IOCTL_TYPE` is set to `IoCtlGeneral` by default, however, if the `ioctl` is to the `/dev/binder` device, then the appropriate Binder type is set (see Listing A.17). All Binder types include the `IoCtlBinderFlag` value and this flag is later used for determining which of the Binder structures to use.

Serialisation occurs only after plug-ins have been invoked, therefore any changes made by a plug-in will be reflected in the resultant protobuf file. The author has considered keeping a copy of the original data too and this may be added in the future.

3.10 Summary

In this chapter, the author describes the design of CopperDroid v2, including how static analysis of the system call information aids in the automatic generation of system call reconstruction code thereby enabling a version-agnostic approach. Additionally, the author described how the parameters of each system call are encapsulated into a classes that can be shared via a plug-in interface, permitting `Python` and `C++` plug-ins to analyse and modify system call parameters and behaviour in real-time.

The author further describes how the fixed portions of the Binder protocol are parsed and how this interacts with the `IInterfaceDataExtractor` interface in order to invoke the unmarshaling code for a given interface.

With regards to Table 3.5, parameter 3 can now be correctly interpreted to reconstruct the `BINDER_WRITE_READ` structure and included operations. The example included in the table is captured from a notification to the `ILocationListener onLocationChanged` method, but at this point (based on what is covered thus far) one can determine the interface and method code, but cannot extract the contents of the marshaled data.

In the next chapter, the author describes how the Android framework source is statically analysed to produce a model that is applied in extracting code code pertinent to Binder ICC. Application of the model is achieved through the CopperDroid v2 source-to-source compiler and demonstrates how static analysis of the Android framework can be used to automatically produce `C++` reconstruction code for unmarshaling interfaces and objects in real-time. While this design and implementation are followed in this thesis, there are other ways which offer a different trade-off, this is discussed in Chapter 8.

Chapter 4

Efficient Version-Agnostic Automatic Interface Introspection for Android

4.1 Introduction

Binder IPC plays a significant role in Android as described in Chapter 2. Services are hosted in external processes and Binder enables apps to communicate with those services via interface method calls. Interface method call transactions from apps are routed through Binder to the appropriate service and replies are routed from services to the calling app in a similar manner. Therefore, it is necessary to interpret both the app to service calls as well as the service to app replies in order to fully analyse high-level operations.

As mentioned in Section 3.9.2 most interfaces are shared via AIDL files which the AIDL parser uses to generate `Java` code that marshals the parameters and return values for interface method calls. However, `IActivityManager`, `IContentProvider`, `IServiceManager` and others are implemented directly in `Java`. Using a modified AIDL parser can be used to generate reconstruction code for AIDL based interfaces, however, it does not address those interfaces implemented in `Java`. The latter are only present in the form of Java code, split into two files, with the interface declaration in one file and the proxy/stub code in a second file. For example, IActivityManager.java and ActivityManagerNative.java in the case of the IActivityManager interface.

While investigating the source of the `PendingIntents` used in the call to `ISms` interface's `sendText()` method, the author originally considered that the Binder handles may be file descriptors (possibly for `/dev/ashmem` objects), however, the values did not match any in the process. The author therefore continued examining each Binder call that occurred prior to the `ISms` interface's `sendText()` call. Doing so led to the discovery that the `IActivityManager.getIntentSender()` method returns the handle value which uniquely identifies the `PendingIntent` object within a given process.

This example is one of many that emphasise how critical the `ActivityManager` is in the functioning of Android. For instance, without the ability to reconstruct the `IActivityManager` interface, there is no way of determining if an app accesses contacts or the SMS inbox, as these can only be determined by reconstructing other methods on the `IActivityManager` interface.

Therefore, given the central role that the `ActivityManager` plays in Android, it is clear that reconstructing behaviours for apps is incomplete without the ability to unmarshal these interfaces. Such interfaces remain opaque to the unmarshaling Oracle too, as the parameter types need to be determined by CopperDroid v2 and passed to the Oracle in conjunction with the `Parcel`.

The author considered several options for achieving this goal. For example, using the ANTLR [65] parser generator with a `Java` grammar, the author implemented a program that translated the `Java` code back into an AIDL interface. However, using the AIDL parser on this AIDL file did not produce the correct results, for several reasons including the fact that some method codes in the `IActivityManager` have to be fixed as they are called from `C++`, as shown in Listing 4.1.

```
// Please keep these transaction codes the same -- they are also
// sent by C++ code.
int START_RUNNING_TRANSACTION = IBinder.FIRST_CALL_TRANSACTION;
int HANDLE_APPLICATION_CRASH_TRANSACTION = IBinder.FIRST_CALL_TRANSACTION+1;
int START_ACTIVITY_TRANSACTION = IBinder.FIRST_CALL_TRANSACTION+2;
int UNHANDLED_BACK_TRANSACTION = IBinder.FIRST_CALL_TRANSACTION+3;
int OPEN_CONTENT_URI_TRANSACTION = IBinder.FIRST_CALL_TRANSACTION+4;
```

Listing 4.1: Fixed transaction codes used by native code

Another option is writing a simple parser that just determines parameter types (by looking for `readStrongBinder()`, `readInt()` and other operations) for each line in a given transaction case statement, however, this quickly becomes far too complex to manage. As Listing 4.2 demonstrates, some statements cross multiple lines, or read in a value from the `Parcel` that is then assigned elsewhere or conditionally read another value depending on the first value read. Parsing this naively could result in the wrong logic being applied and incorrectly unmarshaling the values.

It is clear that these approaches cannot succeed and, moreover, don't solve the problem of unmarshaling complex objects. Yet, in order to automatically match PendingIntent registration with usage (and other methods), it is necessary to utilise the IActivityManager interface source code to assist in interpreting the Binder transaction and responses for that interface.

```
...
case START_ACTIVITY_TRANSACTION:
{
    data.enforceInterface(IActivityManager.descriptor);
    IBinder b = data.readStrongBinder();
    IApplicationThread app = ApplicationThreadNative.asInterface(b);
    String callingPackage = data.readString();
    Intent intent = Intent.CREATOR.createFromParcel(data);
    String resolvedType = data.readString();
    IBinder resultTo = data.readStrongBinder();
    String resultWho = data.readString();
    int requestCode = data.readInt();
    int startFlags = data.readInt();
    String profileFile = data.readString();
    ParcelFileDescriptor profileFd = data.readInt() != 0
    ? data.readFileDescriptor() : null;
    Bundle options = data.readInt() != 0
    ? Bundle.CREATOR.createFromParcel(data) : null;
    int result = startActivity(app, callingPackage, intent, resolvedType,
    resultTo, resultWho, requestCode, startFlags,
    profileFile, profileFd, options);
    reply.writeNoException();
    reply.writeInt(result);
    return true;
}
...
```

Listing 4.2: StartActivity transaction case statement in IActivityManager

The author also considered using a parser generator to produce code for parsing `Java`, however, one issue with a grammar generated parser is that adapting to language changes can be very difficult.

Thus, the author developed a back-tracking recursive descent parser for `Java` 1.7 (the version Android was using at the time). Android Nougat 7.0.0 and later use `Java` 1.8 and the author extended the parser to handle the new language features in less than a week. CopperDroid v2 is implemented in `C++`, therefore, an ideal solution that unmarshals interfaces and objects would also be written in `C++`, to reduce overhead and simplify integration.

The Android interface code is designed solely to marshal parameters and return values between client and server. However, dynamic analysis of these interfaces relies on the ability to reconstruct the communication, not implement it. The author therefore uses static program analysis of the Android framework to enhance the dynamic analysis of Android apps; specifically in reconstructing their interprocess communication.

The `Java` parser is a front-end supported by several back-ends with different purposes. At a high-level, the interface back-end analyses the interface abstract syntax tree and produces code for marshaling and unmarshaling the different interface methods. An object back-end analyses every `Java` source file in the Android framework and produces a type system that includes every object class for a given version of Android.

Every method call (including return type and parameter types) and field declaration (including type) is included, as are class relationships such as nesting and inheritance. This back-end also produces a call graph and a data dependency graph using the methods and fields as nodes/vertices and the interaction between them as edges (for example a method from one class accessing a field from its own class or another class).

A second object back-end is invoked for a given `Java` object and uses the call graph and data dependency graph (producing a system dependency graph (SDG)) to extract (through slicing) only those portions relevant to marshaling and unmarshaling the object. A final stage in the both interface and the second object back-ends is to produce `C++` reconstruction code.

This chapter describes in detail how the parser/translator (source-to-source compiler) works and how it utilises code slicing [86] [53], splicing [5], transformation [83] and type translation to generate working `C++` code that enables real-time analysis and manipulation of Binder operations.

4.2 Static Analysis of the Android Framework

There are many abstractions that are used for static analysis and each has its benefits and drawbacks. For instance, Soot [82] operates on `Java` bytecode and provides the means to manipulate bytecode (including translating into other languages). However, the amount of context at a given level declines the further one gets from the source. For example, while a variable may be named *index* at the field level, it's name doesn't exist once it is compiled into bytecode or native code. Source can also be parsed and then operated on using control flow graphs which may also lose context.

Therefore, the static analysis used in CopperDroid v2 operates on the Abstract Syntax Trees (ASTs) built from the source code of the Android framework. This allows CopperDroid v2 to keep class, method, field, variable and parameter names as well as the original structure. There are instances where the code structure is altered, however, this is intentional and usually due to either syntax differences between `Java` and `C++` or to enable reconstruction.

4.3 Abstract Syntax Tree Structure

The CopperDroid v2 parser uses an irregular heterogeneous AST, where a given node in the tree is specific to the type it represents and the components contained therein. For instance, an assignment expression contains a left side expression, an assignment operator (`=, -=, +=, /=, *=, |=, &=, <<=, >>= or >>>=`) and a right expression. The left side of an assignment is usually an identifier (variable, or field name) but can also be a field access expression, for example, `location.mBearing = 0;`.

The goal of the AST is not to generate executable code directly (where an intermediate representation such as LLVM IR might apply), but rather to generate `C++` code for inclusion into CopperDroid v2 thereby enabling real-time reconstruction and manipulation. Using this AST structure enables using both the visitor design pattern [24] as well as external tree walkers [64]. The latter make it simpler to modify whole portions of the tree whereas the former allow scope tracking for variables, parameters and fields. The CopperDroid v2 compiler has nine visitors each performing a different task. For example, one visitor is responsible for managing name mappings, as some identifier names are valid in `Java` but not in `C++`, such as `NULL` and `string` (when using the `std` namespace). The name mapper traverses the AST replacing any instance of a given name with one that doesn't clash with a `C++` type or `#define`.

All non-abstract nodes in the AST implement a `Write()` method that generates code specific to the node. While `Java` and `C++` are closely related, there are differences. For example, `Java` generally has only the dot operator (`.`) for class member access (with the exception of method references in `Java` 1.8 which use the double colon notation (`::`)). However, `C++` uses different notations depending on the context, for example, non-pointer fields and methods are accessed using the dot notation, pointer fields and methods use the arrow notation (`->`) and static members use the double colon notation. The context in which a particular member is being used affects the notation, thus the nodes responsible for member access (field access and method calls) also encode this information.

As mentioned earlier, CopperDroid v2 also generates boilerplate plug-in code in `Python` and while the root AST node is the same, there are specific `Python` AST nodes as well. These implement a `Write` method that generates `Python` code.

4.4 CopperDroid v2 Compiler Design

The source-to-source compiler described in this chapter consists of a hand-written lexer, `Java` parser and several back-ends. Its ultimate goal is to produce working `C++` code that can marshal and unmarshal Binder IPC in real-time.

Each interface method call has its own parameters and return value (if present), for example, that shown earlier in Listing 4.2. Unmarshaling these consists of either reading a specific type from the `Parcel`, for instance `readString()` and `readInt()`, or creating an object from the `Parcel` using the `createFromParcel()` method for a given object. In some cases classes are instantiated by calling a `readFromParcel()`

```java
1      ...
2      case STOP_SERVICE_TOKEN_TRANSACTION: {
3          data.enforceInterface(IActivityManager.descriptor);
4          ComponentName className = ComponentName.readFromParcel(data);
5          IBinder token = data.readStrongBinder();
6          int startId = data.readInt();
7          boolean res = stopServiceToken(className, token, startId);
8          reply.writeNoException();
9          reply.writeInt(res ? 1 : 0);
10         return true;
11     }
12     ...
```

Listing 4.3: IActivityManager stopServiceToken interface method

method as opposed to calling `createFromParcel()`, demonstrated in Listing 4.3 line 4.

Many Android objects contain other objects, for instance, the `Intent` object contains `Rect`, `ComponentName`, `Uri` and `Bundle` objects. Each object maintains its own marshaling code and invokes the marshaling code of objects that it uses. This led to the key insight that all the code needed for marshaling and unmarshaling interfaces and Android objects is contained within the Android source code. A second key insight is that the only portion of the Android object source code needed is that which intersects with an interface operation. The latter point is important, because the goal is not to implement full support for every aspect of a given object, but rather to determine the portions involved in marshaling and use those to generate the required code.

The compiler supports three separate modes of operation. The first mode builds the type system and system dependency graph across all Android objects (in a given version). The second mode analyses interfaces and updates the system dependency graph to mark code for inclusion while also producing the interface method marshaling code. The third mode uses the system dependency graph to determine which classes, methods and fields to include and generates object marshaling code for all objects accessed by the interfaces.

4.4.1 Type System and System Dependency Graph

The CopperDroid v2 compiler uses a sqlite3 database as the storage for both the type system and system dependency graph. When CopperDroid v2 is built, the compiler is invoked (per-version) and checks to see if the type system has already been constructed for that version. If it has not been constructed then every `Java` file in a given version is parsed and the resulting ASTs analysed over four passes. The first pass captures all the fields and class names for a given file, while the second pass adds every class located in a given `Java` source file to the database, including fields, methods, base classes, outer classes (if nested) and class type (`enum`, `interface` or `class`).

The third pass determines which class every field access or method call belongs to, using the database. Each entity is automatically assigned a node identifier. The final pass analyses every field and method to locate the edges between methods, fields and classes, using method calls, field initialisers, new operators and array instantiations. This also takes inheritance into account, for example, the `Uri` class has an abstract `writeToParcel(Parcel, int)` method that it inherits from the `Parcelable` interface. When it is determined that an inherited (and inheritable) method is called, then any calls to the method on the base class also result in edges to the derived classes. The code for determining the called methods correctly distinguishes between overloaded methods (having the same name with different parameters). Every edge determined through this process results in a record being added to the *edge* table which contains the current node and the node it is accessing. This information is sufficient for building the system dependency graph and contains the raw data involved.

Typically top-down parsers, such as recursive descent parsers start at the top of the grammar and work their way down. However, the CopperDroid v2 parser also permits a partial parse starting a specific point in the grammar. As mentioned above, the type system includes all methods (including parameters and return types) and all fields (including their type). When building the call graph by following method calls and field accesses, as a variable is accessed, the class involved is determined from the variable declaration (using the currently active scope and working outwards) is read into memory from the database. The database representation is cached in-memory (so future reads don't access the disk) and each method signature is parsed, methods are stored in the form of `MethodName(Parameter1,Parameter2,...)` (a method with no parameters is simply `MethodName()`). Thus the methods are parsed into an AST (starting at the `ParseMethod()` entry point in the parser) and then added to the class and used for matching with those found in the current `Java` code. Similarly, field types are parsed into an AST as well (starting with the `ParseType` entry point) to obtain the type. As code is traversed, field types are used to determine which nodes need to be included.

The method matching code is utilised by many different components (including visitors) and first verifies that the methodName matches and then checks each argument to make sure the types match the declared parameters. This can either be a literal matching a given type or an exact type match (`int`, `boolean`, `Intent` or `String`) or it can be a more flexible match, for example, a `null` will check for any type that could be `null` (typically objects). Additionally, since the call matching occurs multiple times including before and after type translation (discussed later in this chapter), some arguments are checked to see if they match any of the possible types. For example, on `Java` a boolean argument can be either `boolean` or `Boolean`, however, in `C++` only `bool` is valid, therefore when a `true` or `false` literal are encountered, all possibilities are checked. If a method parameter is a base class (determined using the type system) of a given argument (and no better match exists) then the method with the base class will be chosen as matching the call.

Each method and field entry in the type system includes a boolean value that indicates if it should be included once the class is built. Initially this boolean is set to `false` for all methods and fields, as there is no reliable mechanism to determine from the objects alone which methods and field should be included. While one could argue that anything accessing a `Parcel` should be included, there are supporting methods (from other classes, sometimes invoked directly by interfaces) that don't operate on `Parcels` and yet need to be included.

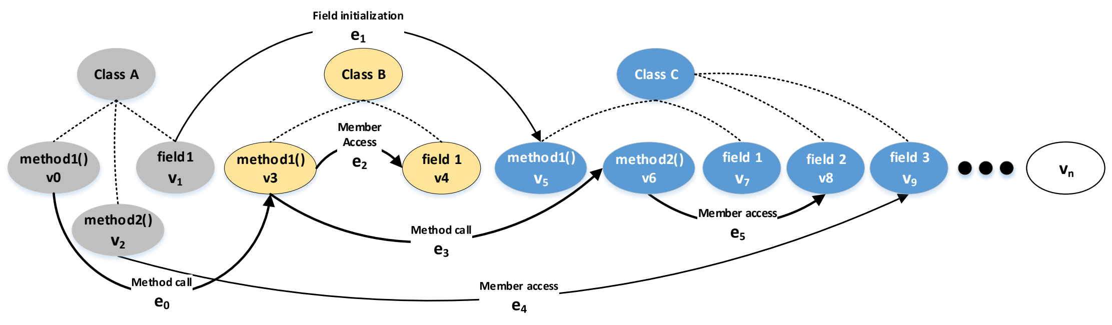


Figure 4.1: High-level representation of class relationships with vertices (methods/fields) and edges (field access/method call)

Figure 4.1 is an example of the class relationships as determined through analysis of all the Android objects for a given version. The type system contains information about each class including its methods and fields. The system dependency graph is constructed using the method calls and fields accessed by a given method or field initialiser.

However, the graph constructed at this point is incomplete in that is cannot identify which fields and methods need to be included for a given class and even if the class itself is required for marshaling. The `Uri` class has multiple nested classes, many of which do not play any role in marshaling. Therefore, more context is required in order to build a complete system dependency graph that captures all the necessary information required to produce marshaling code for Android objects. This process is described in the following section.

4.5 Unmarshaling Android Interfaces

Binder interfaces in Android are defined in either `AIDL`, `Java` or `C++`. The `AIDL` compiler that is included in the Android SDK compiles `AIDL` into `Java` source code. CopperDroid v2 uses that standard `AIDL` compiler to produce interface marshaling code for `AIDL` defined interfaces in `Java`. The CopperDroid v2 compiler analyses the `Java` interface code in order to build the unmarshaling code. While the author could have continued using a modified `AIDL` compiler for the `AIDL` interfaces, doing so would require maintaining two separate compilers, both of which need to work together to produce the final marshaling code. Therefore, the simpler route is to use the standard `AIDL` compiler to produce `Java` and have the CopperDroid v2 compiler analyse the `Java` code in the same way it analyses those interface handwritten in `Java`.

As mentioned in Section 4.4, a front-end for `Java` is used in conjunction with several back-ends to statically analyse Android interfaces and objects and automatically produce code that is able to dynamically reconstruct Binder ICC calls. This analysis includes slicing/splicing of the marshaling code in interfaces and objects, code and type transformation and the automatic generation of code that enables reconstruction of Binder operations in real-time. While slicing typically involves individual statement paths related to a given variable [86] [54] [9], the approach CopperDroid v2 utilises is to include all statements, fields and methods related to marshaling and unmarshaling Binder ICC calls.

The front-end for `Java` produces an Abstract Syntax Tree (AST) after parsing a `Java` file (for an interface or Android object). The AST is passed through to the back-end where forward and backward slicing is used to extract the code and data relevant to a Binder ICC call. Slicing in CopperDroid v2 is achieved using a system dependence graph (where vertices represent methods and fields and edges represent data and control dependencies to those methods and fields). This is similar to the system dependence graph described in [54]. Splicing in the context of this paper refers to either removing a portion of an AST and attaching it elsewhere, or inserting a new AST into an existing AST.

The *system dependency graph* referred to as S, is constructed by parsing all the Android object `Java` source files into ASTs and producing vertices for each class method or field encountered (over three passes to capture all the necessary information). A final pass examines every statement and produces edges for every method call or field access (whether direct, indirect, as a super call to a base class or field initialisation or argument to a method call). Calls to overloaded methods (which share a name but differ in parameters only) are fully supported with method resolution being done at the call site based on the arguments to the call.

The *system dependency graph* is typically constructed once, as it involves parsing all the Android framework `Java` source code for each supported version (requiring around 7-10 hours to analyse the 8 million lines of code from Froyo to Oreo). As the source code for each Android version is static, constructing this once per-version and then re-using it thereafter is sufficient (but can be overridden). Each Android interface `Java` source file is parsed into ASTs and these are sliced/spliced as described in Section 4.6. The final AST for each interface produced by the slicing/splicing is then analysed to determine the vertices that are accessed within the *system dependency graph S* and for every vertex v_i accessed, all edges e_j are followed recursively until all vertices $v_i...v_m$ are marked for inclusion inside the type system (described in Section 4.2). This information is utilised in the object slicing phase, described in Section 4.8. Figure 4.2 illustrates how this works (at a high-level). The lines and vertices marked in red represent those that a given interface case statement accesses. Since this analysis is performed for every `onTransact()` case statement (and all statements contained there-in), once complete, all accessed vertices are marked for inclusion.

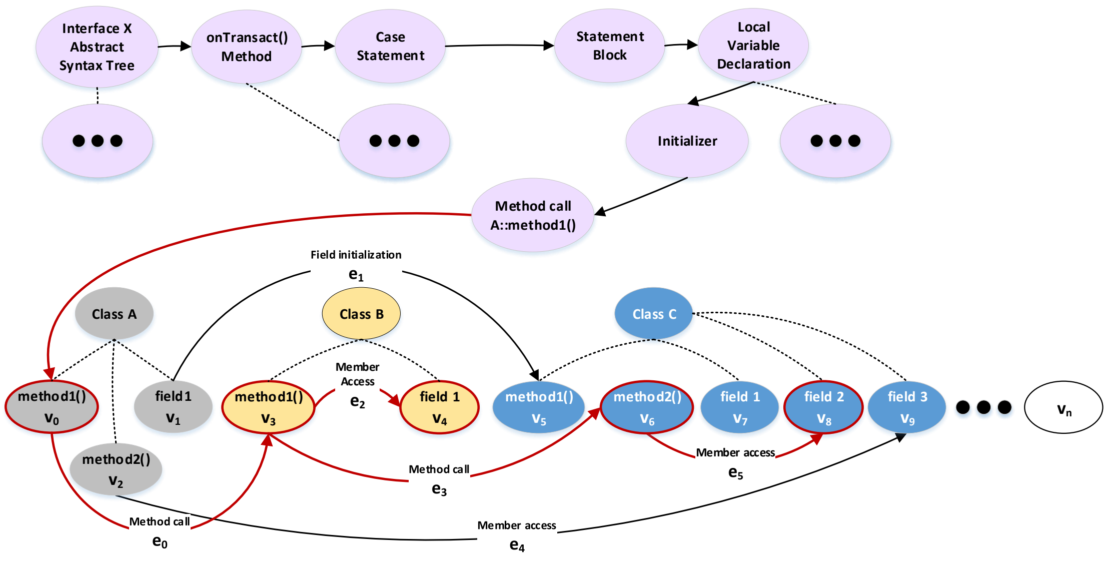


Figure 4.2: High-level representation of class relationships with vertices (methods/fields) and edges (field access/method call) included due to an interface accessing one of the classes.

While a Uri is instantiated using a createFromParcel method, subclasses of the Uri class are instantiated using a readFrom() method. Therefore when forward slicing the Uri class, all methods in all classes invoked along the createFromParcel call graph need to be sliced too. In the case of the IActivityManager getIntentSender() example, the ActivityManager instantiates an `Intent` object using its createFromParcel method which in turn instantiates (among others), a Uri object which attempts to instantiate the correct subclass using its readFrom method. An example of how this is sliced is shown in Figure 4.3 which shows a combination a control flow and data dependency graph which focuses on the Intent, ClipData and Uri classes. Many methods and data members were omitted for brevity, however some are present in the diagram but excluded from the slice (outside the dashed line) were shown for demonstration purposes. These include the `Intent(Intent)` and `Intent(String)` constructors of the `Intent` class and the `parsePath()` method of the `Uri` class. The following sections describe how this slicing is performed.

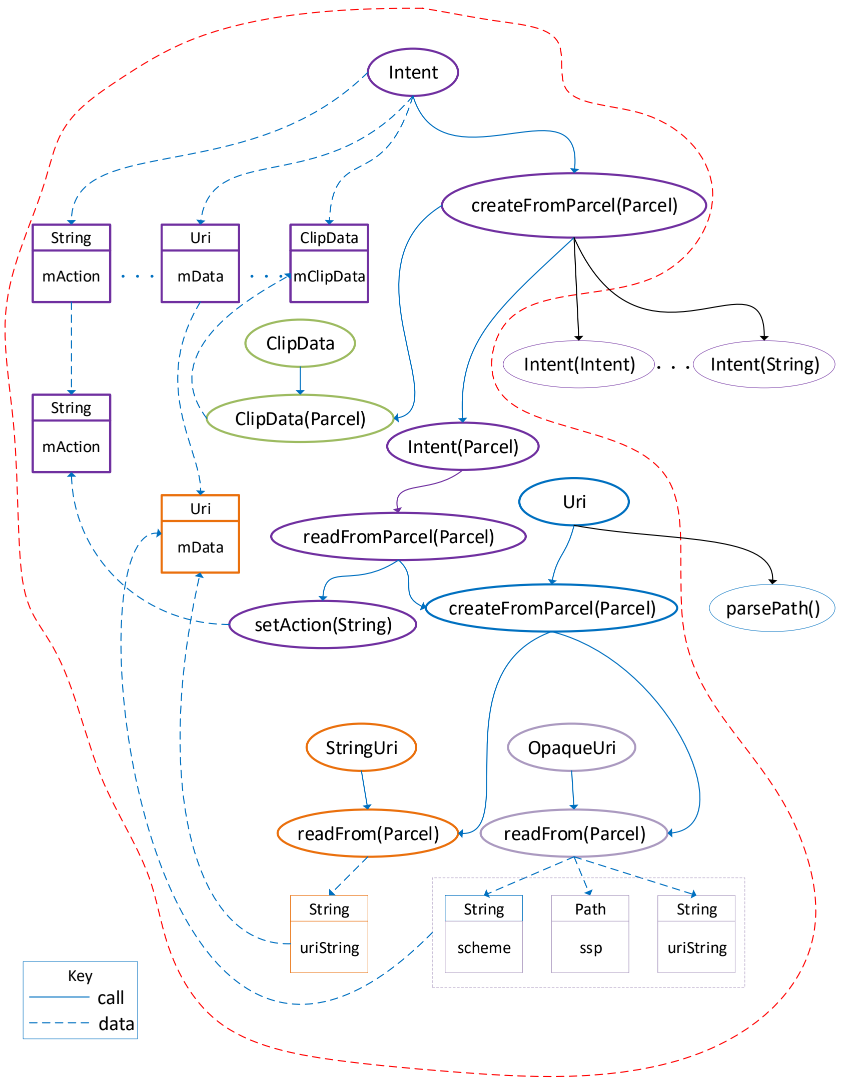


Figure 4.3: Slicing of code and data for an `Intent` object.

4.6 Interface Code Slicing

There are typically four components to a given Binder interface method call. The client-side call marshaling, server-side call unmarshaling, server-side reply marshaling and client-side reply unmarshaling. These components are contained within the `onTransact` method of the interface stub and the interface method in the proxy. Intercepting a given Binder call and potentially modifying it requires separating these four components as follows.

As mentioned in Section 3.9.2, all interfaces have an `onTransact` method in their stub. This method contains a portion of the marshaling code for a given interface call, with the rest of the marshaling code residing in the interface method in the proxy. When CopperDroid v2 analyses a given interface, it walks the AST for the proxy class and adds every method it encounters to an `unordered_map` (these methods typically have the same name as the public name, for example `getIntentSender()`, although they can differ). Next the `switch(code)` statement is located and every case is analysed individually using multiple passes. The first pass analyses every method call in the statement block and determines if the method matches one of the methods from the proxy (using the method outlined in Section 4.2. Typically calls to the interface method will either be of the form `methodName(Parameters)` or `this.methodName(Parameters)`. By determining the AST node responsible for calling the service in question, the call and reply can be ascertained. Determining which code is involved in the call versus the reply is required for constructing the code that performs unmarshaling and marshaling of a given interface method.

This initial pass also creates a def-use chain using the parameters to the `onTransact` method and any variable declared locally within the case statement. The def-use chain is updated for every use, including assignment between variables and when passed as parameters to method calls. The highlighted portions in Listing 4.4 illustrate how the def-use chains apply to interface method parameters. The `if-else` statement is included in any slices as the request intents and resolved types are included. The interface method call arguments are of particular interest as any variables that affect this method call are typically required for unmarshaling the call on the server side. There are exceptions which will be discussed shortly.

A second pass uses the def-use chain to remove any statements not involved in marshaling (for example statements that log information). This is followed by a pass that performs type conversions, for example, `IBinder` to `BinderReference`, `String` to `javaString` (described in Chapter 5) and others. A subsequent pass determines which statements are involved in marshaling the reply to the client and stores these separately. The transaction unmarshaling code is used to build a case statement similar to that of the `Java` code for unmarshaling the current transaction.

```
public abstract class ActivityManagerNative extends Binder implements IActivityManager
{
    ...

    @Override
    public boolean onTransact(int code, Parcel data, Parcel reply, int flags) throws
        RemoteException
    {
        switch (code) {
            ...
            case GET_INTENT_SENDER_TRANSACTION: {
                data.enforceInterface(IActivityManager.descriptor);
                int type = data.readInt();                                    ①
                String packageName = data.readString();
                IBinder token = data.readStrongBinder();
                String resultWho = data.readString();
                int requestCode = data.readInt();
                Intent[] requestIntents;
                String[] requestResolvedTypes;

                if (data.readInt() != 0) {
                    requestIntents = data.createTypedArray(Intent.CREATOR);
                    requestResolvedTypes = data.createStringArray();
                } else {
                    requestIntents = null;
                    requestResolvedTypes = null;
                }

                int fl = data.readInt();
                Bundle options = data.readInt() != 0 ? Bundle.CREATOR.createFromParcel
                    (data) : null;
                int userId = data.readInt();

                IIntentSender res = getIntentSender(type, packageName, token,
                resultWho, requestCode, requestIntents, requestResolvedTypes, fl,
                    options, userId);

                reply.writeNoException();                                     ②
                reply.writeStrongBinder(res != null ? res.asBinder() : null);

                return true;
            }
            ...
        }
    }
}
```

Listing 4.4: IActivityManager onTransact stub method case statement for the getIntentSender method (code omitted for brevity).

Next the interface method as analysed and the code separated into the transaction marshaling code and the reply unmarshaling code. The transaction marshaling code is used to build a class that is initialised within the transaction case statement and passed for plug-ins to permit them to use and potentially manipulate the transaction.

```

class ActivityManagerProxy implements IActivityManager
{
    ...
    public IIntentSender getIntentSender(int type,
    String packageName, IBinder token, String resultWho,
    int requestCode, Intent[] intents, String[] resolvedTypes, int flags,
    Bundle options, int userId) throws RemoteException
    {
        Parcel data = Parcel.obtain();
        Parcel reply = Parcel.obtain();

        data.writeInterfaceToken(IActivityManager.descriptor);        ③
        data.writeInt(type);
        data.writeString(packageName);
        data.writeStrongBinder(token);
        data.writeString(resultWho);
        data.writeInt(requestCode);
        if (intents != null) {
            data.writeInt(1);
            data.writeTypedArray(intents, 0);
            data.writeStringArray(resolvedTypes);
        } else {
            data.writeInt(0);
        }
        data.writeInt(flags);
        if (options != null) {
            data.writeInt(1);
            options.writeToParcel(data, 0);
        } else {
            data.writeInt(0);
        }
        data.writeInt(userId);

        mRemote.transact(GET_INTENT_SENDER_TRANSACTION, data, reply, 0);

        reply.readException();
        IIntentSender res =                                           ④
            IIntentSender.Stub.asInterface(reply.readStrongBinder());
        ...
    }
    ...
}
```

Listing 4.5: IActivityManager proxy with getIntentSender method (code omitted or edited for brevity).

Listing 4.4 shows a case statement in the onTransact method and the getIntentSender method in the proxy and shows a portion of the code needed for unmarshaling and marshaling a given interface method. If this code were to be executed as-is (in the correct environment), it would simply marshal the data between the client and the server. While this is necessary for Android to function, only the marshaling and unmarshaling code associated with Binder IPC calls in both the proxy and stub are required to unmarshal the parameters associated with a given interface call. Additionally both simple and complex types need to be unmarshaled, however, that process is described in Section 4.8.

In the case statement for the `onTransact` method (shown in Listing 4.4), the server-side unmarshaling code (shown in ①, lines 12-30) is sliced and transformed before being added to a corresponding switch statement in a `C++` transaction reconstruction class. In addition, a class is built for every interface method that includes the transaction marshaling code (shown in ③, line 56-77) in Listing 4.5 and is instantiated from within the case statement.

Contained within the getIntentSender method is the reply unmarshaling code, shown in ④ (lines 80-82) in Listing 4.5; this code is added to a corresponding switch statement in `C++` reply reconstruction class. These classes are invoked by CopperDroid v2 when a call to the IActivityManager interface occurs. A class is also produced which includes the reply marshaling code (shown in ②, lines 35-36 in Listing 4.4). Code not included in boxes ①, ②, ③ or ④ is removed from the final output as it is not required for marshaling or unmarshaling. All four slices are needed to fully unmarshal and marshal both transactions and replies, anything less loses vital information.

Once the abstract syntax tree for the reconstruction has been produced, it is examined and the Android object slices are computed by following all method calls and field accesses to Android objects through the vertices and edges being followed and marking every vertex for inclusion in the slice. Method signatures are utilised at call sites to manage overloaded methods, with the signature consisting of the method name and parameter types. Wildcard matching is used for indeterminate types such as null.

The classes built for the marshaling code are not required for reconstruction, however, they are automatically produced in order to make it possible to share the unmarshaled transaction and replies with plug-ins. Plug-ins can examine and modify transactions and replies through these classes and call methods to modify them. If the data is modified then the changes are written back into the guest, permitting the plug-ins to manipulate the calls and replies.

The above algorithms are formalized in Algorithm 4.1 and Algorithm 4.2.

When the algorithms mentioned above are applied to every interface method for a given interface, they produce code that unmarshals a transaction into another generated class that is passed to a plug-in (if present). If the plug-in modifies the transaction, the changes are marshaled in a parcel and written to the guest. Replies are handled in a similar manner and any modifications by plug-ins are written to the guest.

ALGORITHM 4.1: Parsing transaction case statement to build transaction unmarshaling code and reply marshaling code

```
Data: Java onTransact method case statement
Result: C++ onTransact transaction case statement and reply class
1  for each $AST$ node do
2      build def-use chains;
3      if interface method located then
4          exit loop;
5      end
6  end
7  for each interface method parameter do
8      mark def-use chain for inclusion;
9  end
10 Build unmarshaling slice with included def-use chains;
11 for each $AST$ node after interface method do
12     if return statement then
13         exit loop;
14     end
15     add statement to reply marshaling code;
16 end
17 Build reply marshaling slice using reply code;
18 Build reply class and splice marshaling slice into WriteToParcel method;
```

ALGORITHM 4.2: Parsing interface method code to build transaction marshaling code and reply unmarshaling code.

```
Data: Interface method code (for example IActivityManager.getIntentSender())
Result: C++ onTransact reply case statement and transaction class
1  for each $AST$ node do
2      if transact method located then
3          exit loop;
4      end
5      add node to transaction marshaling code
6  end
7  build transaction class and splice marshaling slice into WriteToParcel method;
8  for each $AST$ node after transact method do
9      if return statement then
10         exit loop;
11     end
12     add statement to reply code;
13 end
14 Build reply unmarshaling slice using reply code;
15 Build reply case statement and splice reply unmarshaling slice;
```

```
class IDataExtractor
{
public:
    virtual STATUS BuildTransactionData(int code, Parcel* data, BinderTransaction*
        binderTransaction) = 0;
    virtual STATUS BuildReplyData(int code, Parcel* reply, BinderTransaction* binderReply) =
        0;
};
```

Listing 4.6: IDataExtractor interface which is invoked to extractr the transactions and replies for a given message

Listing A.18 shows the class automatically generated for extracting interface transactions and replies. The class inherits from the `IDataExtractor` interface (seen in Listing 4.6) and has abstract methods that are invoked for either a transaction or a reply. These functions are invoked separately as the transaction occurs on the `ioctl` system call entry while replies occur on system call exit and may even occur on the exit of a subsequent`ioctl` system call exit (though on the same thread). Looking at the code in Listing A.18, one can see that it is very similar to the server-side unmarshaling code in Listing 4.4 denoted by (1).

This is the `C++` equivalent of code required to unmarshal the server side of the `IActivityManager`'s `getIntentSender()` method. The types have been translated and the correct member access notation chosen as appropriate, arrow-notation for pointer members and double-colon notation for static members. The `TypedObjectArray`, `TypedArray` and `javaString` classes are hand-coded to bridge the gap between `Java` and `C++` (described in Chapter 5).

As mentioned earlier, every interface method call transaction results in an equivalent case statement being created for transaction unmarshaling and marshaling (similarly for replies). These classes inherit from the `IInterfacePlugInTransactionData` interface and implement the methods shown in Listing 4.7. A destructor is only generated if a given class has fields that are pointers (and hence require being deleted). The `CActivityManagerGetIntentSenderTransaction` class shown in Listing A.19 has dynamically allocated fields (pointers) and therefore includes a destructor.

Each generated class has a `m_IsModified` field which is initially set to `false`, however, if a plug-in calls a `Set` method in the class, this is then set to `true`. When the plug-in completes its processing and returns to CopperDroid v2, the `IsModified()` method is invoked on the current class instance and, if this returns `true`, the `WriteToParcel()` method is called, updating the `Parcel`. CopperDroid v2 then copies the `Parcel` into the guest at the same address as the original `Parcel`.

```
class IInterfacePlugInTransactionData
{
    public:
    virtual ~IInterfacePlugInTransactionData() {};
    virtual std::string GetTransactionCodeString() = 0;
    virtual bool IsModified() = 0;
    virtual void WriteToParcel(Parcel* InterfaceParcel) = 0;
};
```

Listing 4.7: IInterfacePlugInTransactionData interface for supporting plug-ins modifying transactions and replies

The `WriteToParcel()` method shown in Listing A.19 is the `C++` analogue of the transaction marshaling `Java` code shown earlier in Listing 4.5 in the box labeled (3). This code is required to re-marshal any changes made by plug-ins into a `Parcel` that replaces the original `Parcel` in the guest.

Reply handling is similar to that of transactions. The reply unmarshaling code is extracted from the proxy method shown in Listing 4.5 (4). However, the CopperDroid v2 compiler adds some additional code to handle the case when an exception is raised. In the case of a successful reply, the exception code will be 0 and the result will be available for unmarshaling, otherwise the exception code and string are captured.

It is possible to change a successful response into a failure response by calling the `SetExceptionCode()` method on the reply class instance (shown in Listing A.21 - code omitted for brevity) and thus fail otherwise successful calls from a plug-in.

As mentioned earlier in this section, there are exceptions to the method described thus far. While this works for almost all cases, there are some transactions that don't have an associated method. There are also methods that don't return a result and instead update an output parameter, for example the `IActivityManager getMemoryInfo` transaction. Therefore, prior to generating the necessary unmarshaling and marshaling code the CopperDroid v2 compiler analyses every variable in the case statement's AST to determine which `Parcel` it is accessing. Since the `onTransact()` method parameters are known, the CopperDroid v2 compiler knows which is the transaction and which is the reply `Parcel`. Similarly, the order of the declaration of the `Parcels` in the interface method determine their function. Interface marshaling code generated by the Android `AIDL` compiler use `_data` and `_reply` as opposed to `data` and `reply`, however, the name mapper visitor mentioned earlier normalizes these to match the hand-written names.

Once all the code slices have been determined and the resultant Abstract Syntax Tree constructed for each slice, the resultant ASTs are analysed to determine which nodes/vertices have to be marked for inclusion. Listing A.22 shows the method that is invoked for every case statement in the `onTransact()` method. This is performed over multiple passes and when actually building the final extractors, analyses all the edges to determined which nodes need to be included. The `CObjectBuilder` class (present in Listing A.22) simplifies construction of the classes used for sharing the marshaled data with plug-ins, including the production of `pybind11` bindings for `Python` support.

Listing A.23 demonstrates one of the visitor classes (CEdgeAnalyzer) used to follow every edge and determine every node affected by the interface being analysed. This results in all reachable classes, methods and fields being marked for inclusion to enable the object code slicing (described in the next section) to include only the code relevant to Binder IPC.

4.7 Interface Code Generation

As mentioned earlier, every AST node has a `Write()` method that outputs code specific to that node. As an interface is being constructed, the ASTs for the transaction and reply case statements are spliced (inserted into) the method relevant to them, either `BuildTransactionData` or `BuildReplyData`. Once this has been completed, the class is already present in the form of an AST, with type translation done and member access and pointer information correctly updated.

Generating the code is as simple as calling the `Write()` method in the class. Similarly, at this point all the transaction and reply objects have been captured and these are all written first, followed by the class. The final portion added to the class statement are the transaction codes. The code in Listing A.24 generates every line of code necessary for interface marshaling.

Code generation for plug-in support is described in Section 5.3, which details the various portions involved in supporting `Python` and `C++` plug-ins.

4.8 Android Object Code Slicing

4.8.1 Overview

Since Android is open source, all of the complex objects are also available in source form, these are mostly in the form of `Java` code although some do have native (`C++`) code for performance sensitive tasks. Complex objects such as `Intent`, Uri, LocationRequest, SmsRawData are instantiated through a static CREATOR createFromParcel() routine [78] which then instantiates objects use within those classes, such as Rect, Bundle and ComponentName. These, in turn, instantiate other objects, for example the `TextUtils` class instantiates a `SpannableStringInternal` object that relies on the `ArrayUtils.idealIntArraySize()` method.

The system dependency graph mentioned in Section 4.2 is constructed by parsing all the Android object source files into ASTs. The ASTs are analysed and all vertices (for methods/fields) and edges (for method calls and field accesses) computed. When the Android interfaces are parsed (as described in Section 4.6), all vertices accessed by the interface are marked for inclusion and edges from those vertices are recursively followed until all accessed vertices are marked. This greatly simplifies the Android object slicing as all the classes, fields and methods have already been computed. The last stage, described in this section, involves walking each Android object's AST and building the slice. This is not unlike native code linkers that exclude unreferenced code from executables while linking [52]. The difference is that the compiler described here is excluding code at the source level as opposed to the object or native code level. Source-to-source translation is difficult even for closely related languages like `Java` and `C++`, therefore reducing the code to only that involved in marshaling simplifies this task significantly.

4.8.2 Constructing Android Objects

The CopperDroid v2 compiler has a visitor class named `CObjectTranslator` which is involved in building the type system and system dependency graph described in Section 4.2. This class also performs the final stage which constructs the slices responsible for object marshaling and unmarshaling using the system dependency graph. The `CObjectTranslator` performs both functions because there is a great deal of overlap in the logic involved in creating the type system and system dependency graph as well as the final stage of slicing and code generation. A second class named `CObjectBuilder` simplifies construction of the resultant object classes and is also used in building the transaction and reply classes for plug-in support (seen earlier in Listing A.22).

Constructing the type system and the system dependency graph produces a database containing all the object classes, method names, parameters and return types as well as fields and their types in a given Android version, but doesn't produce any code. The interface analysis marks any classes, fields and methods required for marshaling in the database in addition to producing the marshaling code for a given interface, slicing as needed. This section describes how the slicing is accomplished with the next few sections describing how ASTs are transformed (to support `C++`) and how the code generation works.

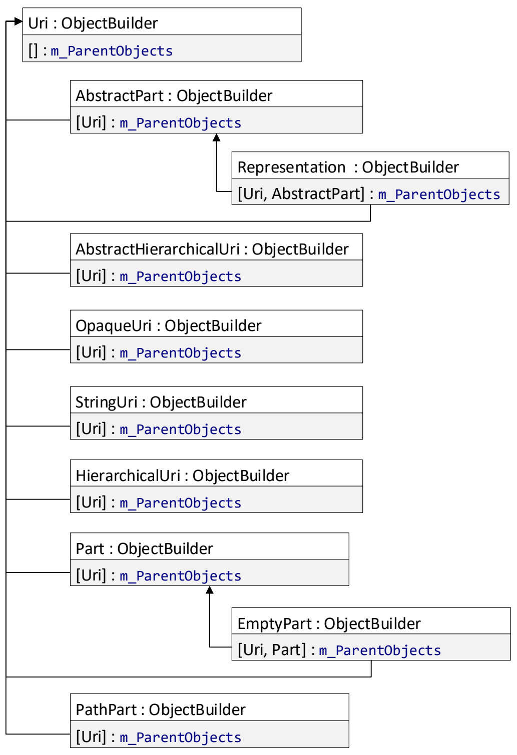
The CopperDroid v2 compiler is invoked once for each `Java` object source file with ASTs being constructed for any classes present in the file. As each class is analysed, a `CObjectBuilder` instance is created with the same name as the current class. If any nested classes are encountered, then instances of `CObjectBuilder` are created for these too with the outer (parent) `CObjectBuilder` instance being recorded. Figure 4.4 shows an example of this using the `Uri` class hierarchy. This only shows the classes that are finally included in the slice, however, there are others, such as `Builder`, `PathSegments`, that are excluded for brevity.

The parents of a given nested class need to be tracked for several reasons. There are many nested classes with duplicate names, for example there are 32 nested classes named `Builder` in KitKat 4.4.2 and 116 in Oreo 8.0.0 (see Listing 4.8). When resolving a given class using its name, the potential parents include all the outer classes of the current class. Additionally, `C++` requires fully qualified names to be used when accessing nested classes (even if the code accessing nested classes is within the same class hierarchy). Therefore, the code generation needs to be aware of any nesting in order to generate `C++` that will actually compile (and work).

```
salah@Pholus:~/Enlistments/cd2/plugin/copperdroid/platforms/KitKat-4.4.2/java$
   sqlite3 types.db "select count(name) from class where name='Builder';"
32

salah@Pholus:~/Enlistments/cd2/plugin/copperdroid/platforms/Oreo-8.0.0/java$
   sqlite3 types.db "select count(name) from class where name = 'Builder';"
116
```

Listing 4.8: Number of instances of Builder nested classes in KitKat and Oreo

The `CObjectBuilder` for a given class is populated with methods and fields of the class, including modifiers such as `public`, `static` or `const`. These properties are translated into `C++` equivalents described in Section 4.11.



Uri : ObjectBuilder
[] : m_ParentObjects
AbstractPart : ObjectBuilder
[Uri] : m_ParentObjects
Representation : ObjectBuilder
[Uri, AbstractPart] : m_ParentObjects
AbstractHierarchicalUri : ObjectBuilder
[Uri] : m_ParentObjects
OpaqueUri : ObjectBuilder
[Uri] : m_ParentObjects
StringUri : ObjectBuilder
[Uri] : m_ParentObjects
HierarchicalUri : ObjectBuilder
[Uri] : m_ParentObjects
Part : ObjectBuilder
[Uri] : m_ParentObjects
EmptyPart : ObjectBuilder
[Uri, Part] : m_ParentObjects
PathPart : ObjectBuilder
[Uri] : m_ParentObjects

Figure 4.4: Object Builder Hierarchy for `Uri.java`

Similarly, class names are tracked and each time a new class is encountered its name is recorded while it is being analysed and then discarded when analysis is complete. These names are used when locating a given class in the type system, as the backing store is a relational database, several records could be found and the correct one extracted. Usually, the potential parents are passed into the `SQL` query, however, there are instances where this isn't possible and resolution has to be handled by using the full hierarchy. Figure 4.5 portrays how the class name list (stored in a `std::vector`) changes as ASTs are walked.

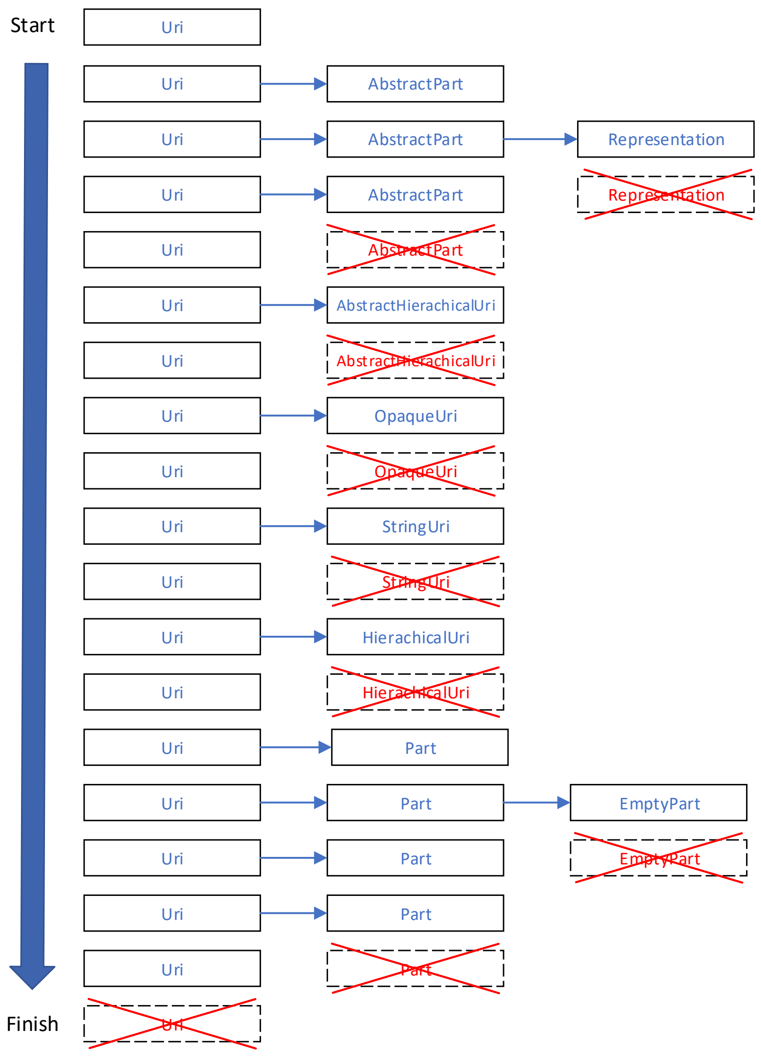


Figure 4.5: Class Traversal for `Uri.java`

The CopperDroid v2 compiler makes several passes over the full AST for a given `Java` source file with each pass building on the previous pass' work. The initial pass captures all of the information about every class in the AST.

The second, and final, pass performs type translation (including fully qualifying nested classes) and AST transformation to ensure that valid `C++` code is produced. These are both discussed in subsequent sections.

Only the `CObjectBuilder` class instances are involved in the final stage, which is responsible for finally slicing the code. This occurs in several stages, the first re-orders the `CObjectBuilder` instances to ensure that inherited and parent classes occur prior to other classes in the final source. The second stage removes any `CObjectBuilder` instances for classes that aren't used (determined through the SDG). The third stage trims methods and fields that are **not** referenced (using the SDG). The next stage ensures that all complex object types are declared as pointers, for example `Parcel*` as opposed to `Parcel`. It also determines if a constructor calls another constructor in the same class or a base class (a `super()` call) and if so, moves the call from the statement block into an initialiser for the current constructor. Additionally, statically declared non-scalar fields and constants are moved from inline initialisation to out-of-line initialisation.

Listing 4.9 shows a small portion of the `Uri` class with the `Part` nested class inheriting from the `AbstractPart` class. Two static non-scalar fields with initialisers are shown along with a constructor that invokes a base class constructor using a `super()` call. While valid in `Java`, this requires significant transformation to work in `C++`.

```
class Uri {
    ...
    static abstract class AbstractPart {
        ...
        volatile String encoded;
        volatile String decoded;

        AbstractPart(String encoded, String decoded) {
            this.encoded = encoded;
            this.decoded = decoded;
        }
        ...
    };
    ...
    static class Part extends AbstractPart {

        /** A part with null values. */
        static final Part NULL = new EmptyPart(null);

        /** A part with empty strings for values. */
        static final Part EMPTY = new EmptyPart("");

        private Part(String encoded, String decoded) {
            super(encoded, decoded);
        }
        ...
    }
}
```

Listing 4.9: Nested Part class with inline initialisation and a super call

The first set of transformations result in the `AbstractPart` and `Part` classes being forward declared within the `Uri` class. The `AbstractPart` class also inherits the `Object` class, this has to do with managing parity with some of the `Java` assumptions and is described in Chapter 5. Finally the `Part` class inherits from the `AbstractPart` class using its fully qualified name (required in `C++`), as can be seen in Listing 4.10. The CopperDroid v2 compiler automatically separates declarations and definitions of objects into a header and source file if an output source file name is provided; otherwise everything remains in the header (which is the case for interfaces). The `CLASS_INSTANCE_ID` will be covered later, but is used to support transformation of `instanceof` into a `C++` compatible implementation.

```
class Uri : public Parcelable
{
public:
    ...
    class AbstractPart;
    class Part;
    ...
};
...
class Uri::AbstractPart : public virtual Object
{
public:
    class Representation;

    AbstractPart(javaString encoded, javaString decoded);
    ...
    javaString encoded;
    javaString decoded;
    static constexpr uint64_t CLASS_INSTANCE_ID = 2574566498548164050U;
};
...
class Uri::Part : public Uri::AbstractPart
{
public:
    class EmptyPart;

    Part(javaString encoded, javaString decoded);
    ...
    static Uri::Part* _NULL_;
    static Uri::Part* EMPTY;
    ...
    static constexpr uint64_t CLASS_INSTANCE_ID = 9516640767813523839U;
};
...
```

Listing 4.10: Nested Part class declaration in header

While constant scalar fields such as integers or floats can be initialised inline (for example `CLASS_INSTANCE_ID` in Listing 4.10), complex types such as strings or objects have to be initialised out-of-line. Therefore, the CopperDroid v2 compiler declares these in the class header but initializes them in the source file (seen in Listing 4.11). Furthermore, `super` calls and alternate constructor invocations from constructors are converted to constructor initialisers. For non-constructor super calls, the base class is explicitly used. For example `super.MyMethod(parameters)` would be translated to `BaseClass::MyMethod(parameters)`. The type system is used to locate the appropriate base-class for a given method signature.

```
Uri::Part* Uri::Part::_NULL_ = new Uri::Part::EmptyPart(nullptr);
Uri::Part* Uri::Part::EMPTY = new Uri::Part::EmptyPart("");

Uri::Part::Part(javaString encoded, javaString decoded) : AbstractPart(encoded, decoded)
{
}
```

Listing 4.11: Nested Part class with out-of-line static field inialization and base class constructor invocation

Algorithm 4.3 formally describes the algorithm used in computing the slices for a given object.

ALGORITHM 4.3: Android object slicing for unmarshaling and marshaling.

```
Data: Android object code (for example Intent)
Result: C++ class for unmarshaling/marshaling Android object using the system
        dependency graph (SDG)
1  for each $AST$ node in class do
2      if node is a method then
3          if method is marked for inclusion in SDG then
4              add method for inclusion on current class;
5          end
6      end
7      if node is a field then
8          if field is marked for inclusion in SDG then
9              add field for inclusion on current class;
10         end
11     end
12 end
13 Trim excluded methods and fields from CObjectBuilder instance.
```

4.8.3 Splicing Code into Android Objects

The object slicing described in the previous section creates a `CObjectBuilder` instance containing ASTs for all the marshaling and unmarshaling code of a given Android object. However, in order to extract the necessary data and make it possible for plug-ins to analyse and manipulate objects, additional code is required.

Each `CObjectBuilder` instance stores methods and fields. The CopperDroid v2 compiler inspects every field in a given object and creates `Get` and `Set` methods for every non-constant field. Listing A.25 shows a portion of the `Location` generated object class with the `AsString()` method as well as get and set routines for latitude and longitude.

Using the fields in a given object, an `AsString()` support method is produced. This method returns a string representation of a given object in the form of a `JSON` object. The variable fields in the object are included based on their type, for example, scalar types are added using the appropriate format string (%d, %f etc.), strings are included using `.c_str()` and nested objects by calling their `AsString()` method. This can be seen in Listing A.26 which demonstrates how all the fields of the `Location` object are used in construction of the `JSON` represention.

All these components are spliced into the Abstract Syntax Tree (AST) of the object class similar to [3] [18], [5] and [45], except using ASTs.

Java notation	C++ equivalent	Example
string.equals(value)	string == value	name.equals(”selector”) → name == ”selector”
asInterface(x) / asBinder(x)	x	asInterface(data.readStrongBinder()) → data−>readStrongBinder();
type.StaticMember	type::StaticMember	Intent.ACTION_MAIN → Intent::ACTION_MAIN
type. CREATOR. createFromParcel()	type::createFromParcel()	Intent.CREATOR.createFromParcel() → Intent::createFromParcel()
string.intern	String	mActions.add(action.intern()); → mActions−>add(action);
fully.qualified.type	type	java.lang.String → string

Table 4.1: The types of transformations used to convert the Java code for interfaces and objects into equivalent C++ code

4.9 Abstract Syntax Tree Transformation

While Java and C++ are relatively closely related (in terms of syntax, structure and keywords), it is necessary to transform the code and translate types to produce reconstruction code. For example, while Java uses dot-notation for field access, method calls and access to static fields in classes, C++ uses dot-notation for statically allocated objects, arrow-notation for dynamically allocated structs/objects (pointers) and double colon notation for static member access.

The ASTs for field access or method calls are examined and necessary adjustments made in the generated C++ code. Table 4.1 shows all of the different transformations used in order to generate reconstruction code in C++.

Simple types in Java are translated to a C++ equivalent, for example String → string or boolean → bool. Other translations also take place, for example the Java null is translated into the C++11 nullptr. The type translator is seeded with some static conversions (String → javaString, Boolean/boolean → bool, Float → float, IBinder → BinderReference) to simplify the process. When methods are called, the return type is examined and if needed, the type of the assignee is changed to match the translated type.

A pass is done over the class' Abstract Syntax Tree (AST) [3] [18] which switches each AST node to C++. During this pass all types are also translated from Java to C++.

The transformations that take place can either remove a portion of an AST, move a portion elsewhere or insert a new portion (splicing). Figure 4.6 shows what is actually occurring in when a super call in `Java` is transformed into a field initialisation with a method call to the base class while re-using the arguments to the original call (in earlier listings 4.9, 4.10 and 4.11). The super call AST node is removed from the AST and deleted. This is one of many examples of the types of transformations that occur in the CopperDroid v2 compiler in order to produce working code for marshaling Android objects.

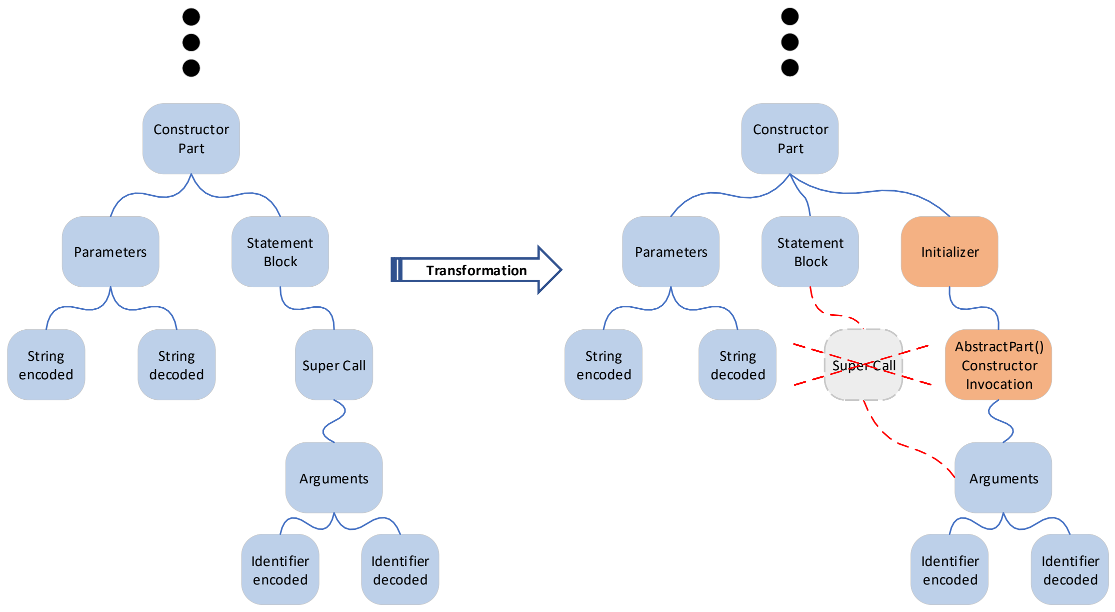


Figure 4.6: Abstract Syntax Tree Transformation for the Uri::Part example shown in the previous section

4.9.1 Transforming `Java` `enum`'s to `C++`

`Java` and `C++` both have an enumeration type, however, there are a few significant difference between them. `Java` `enum`'s have implicit methods such as `name()` and `valueOf()`. They also support inheritance and can have explicitly defined methods and also hold a value (their current state). Listing A.27 shows the `SupplicantState` enumeration which inherits from the `Parcelable` class and therefore implements the `writeToParcel()`, `createFromParcel()` and `newArray` methods. The CopperDroid v2 compiler therefore transforms enumerations into a `C++` equivalent with the same semantics.

As mentioned in Section 4.4.1, the CopperDroid v2 stores a schematic of every class the it encounters in a given version of Android in the type system. This includes the type of class: `interface`, `class` or `enum`. When interface are analysed, the SDG is updated for enumerations too. Therefore, when the source file for a given enumeration is analysed and the enumeration is marked for inclusion, the CopperDroid v2 compiler builds a `CObjectBuilder` instance for a class with the same name. It then splices the enumeration values into a nameless nested `enum` within the class. Any methods in the `Java` enumeration referenced by interfaces are added to the class, along with explicit implementations of `name()` and `valueOf()`. Both of these use the enumeration value identifier names to build these routines. An example is present in Listing A.28.

Since the `name()` and `valueOf()` methods are implicit, there is no `Java` code that can be used to produce these methods. Therefore, the AST for the `name()`, `valueOf()` implicit methods as well as a default constructor, a copy constructor and `operator==` and `operator=` are built by hand. These are spliced into the class that is partially constructed using the code present in the `Java` enumeration. Listing A.29 provides an example of the code used to produce the `valueOf()` method.

4.10 Type Translation

While `Java` and `C++` have several types in common and are structurally similar, not all types are equivalent. While primitive types such as `int`, `float` and `double` are the same, there is no `String` class in `C++`. While one code argue that there is a `std::string` class, the semantics of these differ. The author therefore implemented the `javaString` class (described in Chapter 5) which has similar semantics as the `Java String` class in that it can be `null`, empty("") or hold a value ("Some Text"). This is one of several differences between `Java` and `C++`.

4.10.1 Binder Interface Types

Android interfaces are passed `flat_binder_objects` when used as arguments to interface method calls or return values from calls. As mentioned previously, CopperDroid v2 wraps `flat_binder_objects` within the `BinderReference` class. Since there are many different interfaces in Android, CopperDroid v2 infers the type based on how it is parceled. If something is unmarshaled using the `Parcel` method `readStrongBinder()` or marshaled using the `writeStrongBinder()` method, then its type is converted to `BinderReference`. Any calls to `asInterface()` and `asBinder()` result in the same type conversion. The `flat_binder_objects` are a reference (or handle) to a given object within the current context rather than the object itself, CopperDroid v2 translates their type to `BinderReference` so that they can be tracked if needed.

4.10.2 Array Types

C++ arrays are fixed length (statically allocated) while Java arrays are dynamically allocated and variable length. Fixed size arrays of primitive types are unchanged, however, variable length arrays of non-pointer types are translated to TypedArray (derived from std::vector) and pointer types are translated to TypedObjectArray. The reason for this distinction is that dynamically allocated objects need to be freed once the array is freed. These two classes are described in Chapter 5.

Similarly, other array types, such as ArrayList, List are also translated to either TypedArray or TypedObjectArray, depending on the type argument type. Unifying the types simplifies the need to support multiple classes with the same implementation. This is discussed further in Chapter 5.

4.10.3 Hash Table Types

There are many different hash table classes used in Android, such as HashMap, ArrayMap and Hashtable. These are automatically translated to either a C++ HashMap implementation or an ObjectValueHashMap. The latter manages the lifetimes of dynamically allocated objects using reference counting. Similarly, unifying the hash table types reduces the number of implementations necessary to support the functionality.

4.11 Code Generation

Once the static program analysis just described for a given object is complete, two code generation back-ends are invoked. The first of these automatically produces C++ header and source files for the current Java file. Any classes contained with the current Java file that is referenced by interfaces either directly or indirectly are written to the appropriate file (class declarations in the header file and class implementations in the source file). The second automatically produces pybind11 bindings that allow Python plug-ins to access data members of Android interface method calls and objects by exposing Get and Set routines [43]. Examples of these bindings are shown in Section 5.3 where plug-in code generation is described in more detail.

As mentioned earlier in this chapter, the CObjectBuilder class is constructed using a given Java class's AST. The CObjectBuilder class hierarchy for the Uri class is illustrated in Figure 4.4, shown earlier. When analysis of the classes contained within a given Java source file is complete, the CopperDroid v2 compiler invokes the

Process Name	Process Id	Thread Id	Thread Info	System Call	Parameter 1	Parameter 2	Parameter 3	Return Value
com.example.binderdemo	1504	1504	0xe03ac000	open (5)	/dev/binder	0x00020002		38
...	...	...	...	...	...	...	...	...
com.example.binderdemo	1504	1504	0xe03ac000	ioctl (54)	38	0xc0186201	See Listing 4.12	0
...	...	...	...	...	...	...	...	...

Table 4.2: System calls with full context further described in Listing 4.12.

CObjectTranslator::BuildObjects() method. This performs the final stage of producing C++ code for marshaling and unmarshaling Android objects. The classes are ordered to ensure that they are declared prior to use, following which the BuildObject() method is invoked for every CObjectBuilder instance for the current source file. The BuildObject() method performs many tasks, including trimming any methods or fields not involved in marshaling/unmarshaling, moving field initialisations of complex types from inline to out-of-line, adding implicit methods (for enumerations) and several others, preparing the object(s) for code generation.

The CObjectBuilder class also generates a unique 64-bit class ID using the fully qualified name of the current class, as can be seen in Listing A.30. This hash is assigned to the CLASS_INSTANCE_ID static field of all generated classes and used for implementing instanceof semantics in C++. The author wrote a tool which used the type system class information to generate hashes for every single one of the fully qualified class names for each Android version from Froyo (which has 3591 classes) to Oreo (which has 12187) classes. There were no hash collisions within any of the versions. The author considered using C++ runtime type information (rtti) but it wasn't clear if the semantics were identical, therefore the solution described here is used instead.

The CopperDroid v2 compiler generates an InstanceOf() method for every class in a given source file. The InstanceOf() method checks the provided class instance ID against the current classes instance ID, if that doesn't match then any base classes inherited from directly are also queried which follows the same pattern.

The next stage produces C++ ASTs (via the BuildClass() method) for each class referenced by interfaces and inserts the field initialisations and method definitions into the source document AST, it also generates the Python bindings for all referenced classes. The BuildClass() method produces either a class wholly inlined (if the source document is null) or it produces a class declaration with method prototypes in the header and method definitions in the source file AST. The header document is constructed using the AST for the class produced by the BuildClass() method and each is inserted into the header document AST in the order previously determined. Finally, the Write() method is called on each document's AST (header, source and python bindings) which produces C++ source code for all of these.

```
    BR_TRANSACTION:
        android.location.ILocationListener.onLocationChanged(Location)
        Location:
            mProvider :  "gps"
            mTime :  1556678061000
            mElapsedRealtimeNanos :  172928871078
            mLatitude :  -74.000000
            mLongitude :  40.710000
            mHasAltitude :  true
            mAltitude :  100.000000
            mHasSpeed :  false
            mSpeed :  0.000000
            mHasBearing :  false
            mBearing :  0.000000
            mHasAccuracy :  true
            mAccuracy :  20.000000
            mExtras :
                mMap :  (null)
                mParcelledData :
                    String Value :  "satellites"
                mHasFds :  false
                mFdsKnown :  true
                mAllowFds :  true
            mIsFromMockProvider :  false
```

Listing 4.12: Reconstructed Binder interface call for the ILocationListener.onLocationChanged() method

Through the application of static analysis on the Android framework source code and the code that is automatically generated by the CopperDroid v2 source-to-source compiler, Binder ICC can be fully reconstructed. The example illustrated in earlier chapters and again in Table 4.2 contains a call to the `ILocationListener onLocationChanged` method with a `Location` object. This can be seen fully reconstructed in Listing 4.12. Furthermore, the resultant code enables the modification of reconstructed interface calls parameters and including any Android objects.

4.12 Evaluation

In order to ensure that the model correctly reconstructs Binder ICC and Android objects, the author performed several tests. Leveraging the CopperDroid v2 Viewer, in conjuction with several test apps, the author verified that the information being reconstructed and displayed in the Viewer was, in fact, the same information as had been passed through in the test application.

However, writing many tests in this manner is inefficient and won't cover several of the more complex interface methods. Thus a plug-in to CopperDroid v2 was utilised to access objects from different interfaces and methods and read values from objects present in the method and update a specific value with itself. While this has no change on the actual data included in the Binder ICC request, it does force CopperDroid v2 to reparcel the original data. During initial testing, there were some values that were being remarshaled incorrectly (due to the author assuming a byte type would require one byte, however, as mentioned earlier all integer types are 4 bytes in length for Binder).

Incorrectly parceled requests result in an exception in Android which is visible through `adb logcat`. Severely malformed parcels can also hang Android, which the author encountered at several points while working on this.

In verifying the accuracy of Binder reconstruction, the author realised that while there are thousands of Binder interface methods and Android objects, it is only necessary to verify several key constructs. Namely, the primitive types that Binder supports and arrays of either Android objects or simple types. No matter how complex any given object or how many objects it encapsulates, all marshaling eventually leads to primitive types or arrays being written into the parcel. While the author tested various interfaces including complex `IActivityManager` methods, `Location` objects (since they contain several types including `Bundle`) and the `ISms` interface, the key factor in all of these is whether or not the primitive types and array were being marshaled/unmarshaled correctly. Approximately 40 interface methods (across different interfaces) were tested to ensure that as many combinations were handled as possible. Initially the number was higher, however, the author reduced the number of tests once the key success factor was determined.

Through these tests the author is able to establish that the proposed model with the automatically generated code is able to correctly unmarshal and marshal Binder ICC with complete accuracy. This is the expected outcome as the source code is a source of truth for the Android objects and Binder ICC.

4.13 Summary

In this chapter the author described how static analysis of the Android framework `Java` source is leveraged to produce `C++` marshaling code for both objects and interfaces. This enables a version-agnostic method of reconstructing complex ICC operations in real-time. This include the details on how a complete type system is constructed as well as how the intersection between interfaces and objects provides the necessary information for correctly slicing the objects for marshaling and unmarshaling. The author further described how the results of the static analysis is used to provide plug-ins with the ability to perform analysis and manipulation of the interface calls and complex objects in real-time.

The next chapter describes how the work performed in this chapter and the previous chapter are integrated to provide a whole system that enables seamless analysis and manipulation of system calls and Binder interfaces and objects. It also evaluates CopperDroid v2 in terms of performance and extensibility.

Chapter 5

Integrating CopperDroid v2's Components

5.1 Introduction

In Chapter 3, the author describes the architecture and design of CopperDroid v2 including the core portions that manage state, the automatic reconstruction of system calls and parsing teh fixed portions of the Binder protocol. Chapter 4 describes the analysis performed by the CopperDroid v2 compiler to automatically generate C++ code that marshals and unmarshals Binder interfaces and objects. While this accomplishes the vast majority of the work required to effectively enable reconstruction and manipulation of system calls and Binder operations, combining all this into a working solution requires some additional pieces.

This chapter describes the portions not covered in detail previously, including how all the pieces are integrated to provide an end-to-end solution.

5.2 Classes that Provide Parity with Java

The differences between the Java and C++ languages are more extensive than would first appear. Furthermore, Android introduces additional classes that are implemented partially in Java and partially in C++. While it is technically possible to use static analysis for both the Java and C++ portions and generate a unified C++ implementation for these classes, as this is a time-consuming task the author considers it to be future work. The author therefore chose to achieve parity with Java by hand implementing these classes. The most critical of these is the Parcel class, mentioned earlier, as well as collection and hash table classes. The Parcel class is hand implemented to provide parity with the Android Java class, however, there are some differences. The

Java class utilises native code (written in C++) to achieve some of the core functionality which includes retrieving objects (using the flat_binder_object) from the Binder Object Manager. This, however, requires the correct context in order to work and while the author has investigated this and has some ideas on how to achieve this goal, it requires additional research. Therefore, the class written by the author simply maintains the flat_binder_objects in their original form without attempting to locate them in the current task's address space.

5.2.1 The javaString Class

Java strings can either be null, empty ("") or contain some text. The C++ string class, however, only supports a string being empty or containing some text. The distinction is important though, while certain strings in Android parcels can be empty, if they are null, an exception is raised. One such example is the ComponentName(Parcel) class constructor (shown in Listing 5.1), where an empty string is valid, however, a null string is not. The author initially translated null string comparisons to empty string comparisons (since the C++ string class doesn't support null semantics), however, it quickly became clear that this isn't a viable solution as both parameters can be empty in a valid parcel.

```
public final class ComponentName implements Parcelable, Cloneable, Comparable<ComponentName>
     {
    private final String mPackage;
    private final String mClass;

    ...
    public ComponentName(Parcel in) {
        mPackage = in.readString();
        if (mPackage == null) throw new NullPointerException("package name is null");
        mClass = in.readString();
        if (mClass == null) throw new NullPointerException("class name is null");
    }
    ...
}
```

Listing 5.1: ComponentName Java constructor with Parcel input

The author therefore implemented a new class called javaString that handles null semantics in addition to strings being either empty or containing data. This class permits null assignment and comparisons while still supporting all the usual string methods, thus enabling a simpler translation to C++ as shown in Listing 5.2.

```
class ComponentName : public Parcelable
{
public:
    ...
    ComponentName(Parcel* in);
    ...

private:
    javaString mPackage;
    javaString mClass;
    ...
};

ComponentName::ComponentName(Parcel* in)
{
    mPackage = in->readString();
    if (mPackage == nullptr) {
        throw logic_error("package name is null");
    }
    mClass = in->readString();
    if (mClass == nullptr) {
        throw logic_error("class name is null");
    }
}
```

Listing 5.2: ComponentName C++ constructor with Parcel input

The `javaString` class makes use of the `C++11` `nullptr_t` type in order to differentiate `null` assignments from other types. The class tracks `null` using a boolean value, clearing the underlying string in the case of `nullptr` being assigned.

5.2.2 The Object Class

All `Java` classes implicitly inherit from the `Object` class and several Android classes use the `Object` class as arguments when the actual type can vary. Therefore, the CopperDroid v2 compiler adds a virtual `Object` base class to classes that don't inherit from other classes (since they would have inherited from the `Object` class already). This enables the CopperDroid v2 compiler to generate code that works correctly with methods that have parameters of type `Object`. Furthermore, this class provides referencing counting which is used to assist with object lifetime. While most objects are used and deleted directly (for example class fields), objects can also be added to an array or hash table and can sometimes be copied into multiple collections. For example, objects in a hash table also be copied into a vector-like array or objects into a different hash table. The `TypedObjectArray` and `ObjectValueHashMap` classes described in the following sections utilise the reference counting to ensure that objects are only freed once all instances are no longer in use. The `Object` class also exposes a pure-virtual method `InstanceOf()` which provides the same semantics as the `Java` `instanceof` operator.

The `Object` class also includes `Reference()` and `Dereference` methods that atomically increment and decrement a reference count (initialised to 1 for a newly instantiated object). When this reference count reaches 0, the object is deleted. This assists the `TypedObjectArray` and `ObjectValueHashMap` template classes in managing object lifetimes.

5.2.3 The TypedArray/TypedObjectArray Classes

The `TypedArray` and `TypedObjectArray` classes (seen in various code listings in Chapter 4) provide a variable length array-type template class for parity with `Java`. `TypedArray` inherits from the `std::vector` class and the `TypedObjectArray` class inherits from the `TypedArray` class. As mentioned in Section 4.8 the CopperDroid v2 compiler splices `AsString()` methods into all of the classes it generates and these methods are generated to either use type specific formatting when outputting a string (%d, %ld, %f or %s) or in the case of objects, invoking their `AsString()` methods. Both `TypedArray` and `TypedObjectArray` include an `AsString()` method that produces an array in `JSON` form ([entry_0, entry_1, . . . , entry_n]).

The distinction between simple types and objects is necessary to ensure this is handled correctly. When the CopperDroid v2 compiler translates arrays into either `TypedArray` or `TypedObjectArray`, it examines the type associated with the array using the type system described in Chapter 4. If the type is a complex object, then it translates the array to a `TypedObjectArray` otherwise it translates it to `TypedArray`.

Listing A.34 includes a portion of the `TypedArray` class including the default constructor, destructor and `AsString()` method. The latter outputs the individual entries to a `stringstream` which is converted back into a string that is returned to the caller. This enables generated objects to seamlessly output their arrays without requiring additional generated code.

The `TypedObjectArray`, as shown in Listing A.35, inherits from the `TypedArray` class and also implements the `AsString()` method. However, since all of the entries in the array are objects themselves, it invokes their `AsString()` methods in order to construct the stream and resultant string. Since all of these objects are dynamically allocated, the destructor invokes the `clear()` method which dereferences every object before removing it from the array. This invokes the `Derefernce()` method on the underlying object (described earlier), thus enabling the correct management of object lifetimes.

The `TypedObjectArray` class includes an assignment operator that references and object assigned to it. Methods that remove entries from the array dereference the objects prior to removal. Implementing the class this way allows for seamless translation from `Java` to `C++` reducing the amount of transformation or code splicing that would otherwise be required.

5.2.4 The HashMap/ObjectValueHashMap Classes

Just as the `TypedArray/TypedObjectArray` classes are used for simplifying the translation from `Java` to `C++`, so too are the `HashMap` and `ObjectValueHashMap` classes. As with the previous example the `HashMap` class is used when both the key and the value are simple types (for example `int`, `float` or `string`), while `ObjectValueHashMap` is used with the key is a simple type but the value is a complex object. Analogous to the `TypedObjectArray` class described in the previous section, the `ObjectValueHashMap` uses reference counting for managing object lifetimes and invokes the `AsString()` method on its entries when its `AsString()` method is invoked. The `HashMap` class simply outputs its entries to a stream.

5.3 Plug-in Code Generation

The `CActivityManagerGetIntentSenderTransaction/Reply` classes, along with all other interface method transactions/replies, are shared with `Python` plug-ins using pybind11 bindings as described in Section 4.6. Some of the code for the automatically generated classes can be seen in Chapter 4, Listing A.19 and Listing A.21. The `Python` plug-in code generation, mentioned in comment in Listing A.24, uses the `Set` and `Get` methods automatically produced for every non-constant field of a class to permit access to the interface call data. These methods are shared with `Python` plug-ins using automatically generated bindings, shown in Listing A.36. `C++` plug-ins do not require these bindings as they can include the automatically generated headers directly.

The Python code generator consists of 14 hand-coded AST classes which are congruent to their `Java` counterparts; for example, if statements, methods, return statements as well as an auto-indenter (for Python). During the code generation phase, the `Java` AST is examined and used to automatically produce the Python plug-in boilerplate code described below. The CopperDroid v2 compiler automatically generates a plug-in entry point module (named `copperdroidplugin.py`) using the `Python` ASTs.

For every interface analysed, CopperDroid v2 generates a `Python` plug-in module (Listing A.39) that invokes a given routine use the interface code string as a key to a dictionary. While the integer code value could have been used, the string is clearer.

In addition to producing the plug-in module, the CopperDroid v2 compiler also produces boilerplate code that includes comments listing the methods available each method interface call's class. An example is provided in Listing A.40.

The plug-in boilerplate code generated is placed into multiple files, two per interface (for `Python`) or a single header (for `C++`) and one as the plug-in module (`Python`) or a source and header for `C++`. Figure 5.1 illustrates the directory structure of the some of the files produced for both the `Python` and `C++` plug-in boilerplate code.

```
...
└─arm
  ├─pythonplugin
  │ ├─activitymanagerplugin.py
  │ ├─activitymanagerroutines.py
  │ ├─...
  │ ├─copperdroidplugin.py
  │ ├─...
  │ ├─systemcallplugin.py
  │ ├─systemcallroutines.py
  │ └─...
  └─plugin
    ├─activitymanagerplugin.h
    ├─...
    ├─plugin.h
    ├─plugin.cpp
    ├─...
    ├─systemcallsplugin.h
    └─...
```

Figure 5.1: Automatically generated plug-in boilerplate directory/file layout

5.4 Evaluation

The author evaluates CopperDroid v2 in terms of accuracy and diversity in the reconstruction of Android apps' behaviours, its incurred overhead as well as extensibility to fuel further research.

CopperDroid v2 contains approximately 1.5 million lines of C++ code. Of this 86,000 lines are hand-written and implement the static program analysis and type hierarchy outlined earlier and the core state management functionality in the CopperDroid v2 plug-in. The remaining code is automatically generated as a result of the static analysis described in Chapter 3 and Chapter 4.

The author carried out experiments on a 1st gen Quad-core Core i7 2.93Ghz with 16GB RAM, running Ubuntu 16.04 using the Android emulator (QEMU) running KitKat 4.4.2 to perform controlled tests. While CopperDroid v2 seamlessly supports Android versions from Froyo to Oreo, a large portion of the initial testing took place on KitKat and, to ensure consistency, the remaining testing was continued using that version. Chapter 6 includes experiments executed on KitKat 4.4.2 and Lollipop 5.1.1 for its analysis and run recently on different hardware.

5.4.1 Behaviours

The author analysed 4,500 apps, consisting of the top 50 paid and free apps in each category (2,500 total) on the Google Play Store [38], as well as 2,000 known malware samples all obtained through AndroZoo [1] (December 2017). CopperDroid v2 breaks down behaviours into three categories: system calls, Binder ICC (e.g., calls to Binder interfaces) and composite (e.g., system calls where semantic-dependency form cluster of related low-level actions that represent high-level behaviours [78], such as data sent over network).

Each app was analysed by CopperDroid v2 for 3 minutes; the author observed overall 387,153,582 system calls (128 unique), 1,687,267 Binder ICCs (294 unique) and 5,505,832 composite (40 unique) behaviours across the ∼2500 benign apps. The malware apps yielded 92,185,408 system calls (122 unique), 349,949 Binder ICCs (234 unique), and 1,520,954 composite (41 unique) behaviours, overall[1]. Table 5.1 reports a small excerpt of the total behaviours reconstructed.

In contrast, [78] relied on `Java` reflection to reconstruct a limited view of Binder semantics (e.g., only a subset of requests). This incurred a prohibitive overhead, further exacerbated by the reliance on the GDB stub and the RSP protocol for intercepting system calls.

5.4.2 Performance

The author evaluated the performance of CopperDroid v2 with the PassMark v2.0.1 [66] benchmark app. The author also includes 2D benchmarks as graphical events are implemented as Binder calls. the CPU tests, Memory tests, 2D Graphics tests and Disk tests. The author also ran CFBench to gauge its applicability, however, since it is CPU and memory based and (as mentioned in Section 3.1) CopperDroid v2 only intercepts system calls, there was negligible difference (with CopperDroid v2 outperforming the

[1]Please note that unique number refers only to raw events with no argument reconstruction.

Behaviour Class	**Subclass**	**Benign**	**Malware**
Network Access	Generic	89,171	46,377
	HTTP	7,620	8,070
	DNS	0	1
FS Access	Read	886,472	458,943
	Write	33,052	93,123
Access Personal Info.	Phone	4,915	19,144
	Accounts	1,080	61
	Location	682	861
Exec. External App.	Generic	2,672	3,280
	Priv. Esc.	12	47
	Shell	30	128
	Inst. APK	0	0
	Create Process	1,930	1,907
Make/Alter Call	—	0	5
Binder Telephony	getDeviceId	1,782	13,299
	getSubscriberId	295	2,473
	getDeviceSvn	2	97
	getLine1Number	157	808
Anti-Debug	—	20	549,938

Table 5.1: Excerpt of the behaviour breakdown on the (potentially) benign apps and known malware (source: AndroZoo [1]).

plain emulator at times) between running with or without CopperDroid v2 and therefore the author have not included the results of this benchmark here. However, the display system in Android makes use of Binder calls for managing surfaces. Since Binder is implemented via an ioctl system call, CopperDroid v2 is invoked and therefore 2D graphics benchmarks were included.

Benchmarks were performed using a vanilla Android emulator, a CopperDroid v2-enhanced Android emulator, and a CopperDroid v2-enhanced Android emulator with Python plug-in enabled[2].

It is important to note that the `Python` and `C++` plug-in support is not required for CopperDroid v2 to function and indeed all application behaviour capture takes place irrespective of any plug-in. The plug-in support is simply an add-on that simplifies the task of enabling other research opportunities.

As CopperDroid v2 intercepts system calls, it captures the data associated with each system call (parameters, buffers pointed to by parameters and return values) and serialises this data to disk on a secondary thread within the same Android emulator. This

[2]It is worth remarking that Python plug-in support eases the development of complex behavioural analysis, but its absence does not affect in any way the functionalities of CopperDroid v2.

Type	**Emulator**	**CopperDroid v2 no plug-in**	**CopperDroid v2 with Python plug-in**
Disk	44,324	34,548 (22.06% overhead)	22,335 (49.61% overhead)
2D Graphics	407	267 (34% overhead)	205 (49.6% overhead)

Table 5.2: Overall Benchmark score for Disk and 2D Graphics performance (10 runs)

introduces overhead since memory read from the guest OS is dependent on the system calls, the size of the buffers etc. The data collected is currently analysed offline, after a given execution, which allows us to reconstruct behaviours upon which to build further analyses.

The disk benchmarks shown in Table 5.2, show an overhead 22.05% and 49.61% for CopperDroid v2 with no plug-in and with a plug-in respectively. The overhead without the plug-in is due to CopperDroid v2 copying and logging all of the memory for every read/write operation that is occurring in these benchmarks. For live scenarios where behaviour is reconstructed for enforcing security policies, logging will not be enabled and therefore the overhead of copying/logging would not feature.

Table 5.3 shows the actual throughput in all three scenarios. The internal read benchmark uses 32K buffers for all read operations, micro-benchmarks built into CopperDroid v2 show that the majority of its processing of reads complete in under 12μs, however, there are also some cases where processing can take as long as 1ms. This is due to copies of guest memory taking longer in certain instances (for example due to it being paged out on the host system). The internal write benchmark uses a combination of 4K and 16K and 32K buffers and this results in more overhead. While the overall throughput is higher, there is some cost involved in invoking CopperDroid v2 for every system call, and smaller buffers (while faster to copy) require more system calls. However, as mentioned earlier, the copy can be deferred to an on-demand basis should a plug-in wish to examine a specific process's read/write requests for making security policy decisions in real-time.

The additional overhead in the case of a `Python` plug-in is due to the fact that an embedded `Python` plug-in is being invoked for every read/write operation (system call entry/exit) which incurs overhead associated with cross language interactions. These tests were executed prior to the author adding `C++` plug-in support and thus the measurements only include `Python` plug-ins. `Python` plug-ins are easy to implement and, being interpreted, can be quickly modified and rerun, in addition `Python` can be embedded in a `C/C++` program; therefore the author chose this as a language to support. Additionally, a significant amount of the malware analysis code for CopperDroid v2 is

Type	Emulator only	CopperDroid v2 no plug-in	CopperDroid v2 with plug-in
Internal Read (MB/s)	26.89	26.01 (3.37% overhead)	23.68 (13.58% overhead)
Internal Write (MB/s)	45.80	31.15 (47.04% overhead)	21.58 (112.22% overhead)
External Read (MB/s)	158.34	104.64 (51.31% overhead)	45.05 (251.52% overhead)
External Write (MB/s)	35.38	27.83 (27.10% overhead)	16.82 (110.34% overhead)

Table 5.3: Performance benchmarks for disk read/write operations

Operation (per second)	Emulator only	CopperDroid v2 no plug in	CopperDroid v2 with plug in
Solid Vectors	642	451 (42.35% overhead)	298 (115.44% overhead)
Transparent Vectors	513	316 (62.34% overhead)	247 (107.69% overhead)
Complex Vectors	10.4	9.67 (7.55% overhead)	7.18 (44.85% overhead)
Image Rendering	805	504 (59.72% overhead)	374 (115.24% overhead)
Image Filters	34.4	29.4 (17.01% overhead)	22.8 (50.88% overhead)

Table 5.4: Performance benchmarks for 2D graphics operations

written in `Python` and supporting `Python` plug-ins makes it easy to invoke this code in real-time and perform the analysis online as opposed to offline.

The performance for 2D graphics operations are also impacted by CopperDroid v2 as these make extensive use of the Binder in order to update the display, although the actual data transfer is done to memory mapped regions. Table 5.4 demonstrates the impact in performance depending on the task at hand. Processor intensive operations (like Complex Vectors) actually have less overhead since most of the work is done independently of system calls/Binder interaction.

`IActivityManager` is central to Android apps (including start tasks, checking permissions and many others) and prior to this work, automatic reconstruction of apps - `IActivityManager` interactions were not been addressed. The `IActivityManager` supports 260 unique methods (on Oreo) callable by Android apps, with the `IPackageManager` having the next highest at 182. Table 5.5 therefore shows micro-benchmarks for the `IActivityManager.getIntentSender()` method and the `IPhoneSubInfo.getDeviceId()` method. While the `getDeviceId()` benchmark with the `Python` plug-in seems to have a large overhead, this is likely due to a context switch occurring during this call. Each micro-benchmark was run 1,500 times as a warmup and then a further 15,000 times with the latter runs being used for determining the overhead. Individual measurements of the overhead breakdown contained in Table 5.6 show that the time spent in both the Binder reconstruction and the plug-in invocation is very little.

Binder call	Emulator only	CopperDroid v2 no plug-in	CopperDroid v2 with plug-in
getIntentSender	1,590.47μs	1,818.87μs (14.36% overhead)	2,296.53μs (44.39% overhead)
getDeviceId	2,490.8μs	2,922.10μs (17.32% overhead)	4,065.25μs (63.21% overhead)

Table 5.5: Micro-benchmarks Binder operations (15k runs)

Operation	System Call Total	Binder Entry	Plug-in	Binder Exit	Plug-in
getIntentSender	123.82μs	25.99μs	N/A	9.27μs	N/A
getDeviceId	130.73μs	1.50μs	N/A	6.84μs	N/A
getIntentSender (with Python plug-in)	200.40μs	48.48μs	23.49μs	32.75μs	23.76μs
getDeviceId (with Python plug-in)	189.06μs	23.15μs	21.70μs	31.19μs	24.03μs

Table 5.6: Micro-benchmarks μs breakdown for Binder operations (15k runs)

The `getIntentSender()` method was chosen as an example as reconstructing the `Intent` array instantiates multiple other complex objects (e.g., `ComponentName`, `Uri`, `Bundle`, `Rect`, `ClipData`, `ClipDescription`) required on the order of seconds to reconstruct [78]. Conversely, CopperDroid v2 needs only 300μs to reconstruct the entire Binder ICC call from the app through the `/dev/binder` ioctl (150μs on the client and 150μs on the server). Similarly, `getDeviceId()` was chosen to show a low-overhead yet successful reconstruction of a Binder reply. While both cases are micro-benchmarks, they measure the full path from the `Java` app through the Binder device to the `Java` server and back.

While some benchmarks show a fairly large overhead, these are worst case scenarios and the cost can be reduced by only copying the relevant buffers when needed. As CopperDroid v2 is invoked for every system call for every process from the start of boot until capture is stopped or the emulator exits, the author took the geometric mean of every system call from boot to emulator exit and this showed an overhead of 16.8% with system call and Binder plug-ins and a 9% impact without any plug-ins but still doing full capture of system call data and reconstruction of Binder calls.

CopperDroid v2 also supports micro-benchmarks with a much finer granularity that are included in every protobuf (if enabled). For instance, full reconstruction of a `getIntentSender()` call including `Intent` parameters requires only 25.99μs (using a geometric mean), when measuring the reconstruction time alone. In contrast, just reconstructing a single `Intent` object with the unmarshaling oracle (using a handcrafted test that only included the `Intent` object) in [78] takes 20s. This represents a 76.95 million % improvement in performance. These results were obtained by enabling the micro-benchmarks for Binder which measures the time required to unmarshaling each interface method call. Plug-in invocations are measured as well in order to correctly determine the time spent in the unmarshaling versus the plug-in.

5.4.3 Extensibility

CopperDroid v2's design allows for low-overhead online introspection of Android apps' events. This enables to introspect and modify any system call argument, including ICC payload, which facilitates ICC-related fuzzing analyses, anti-VM evasions, and eases the effort required to port CopperDroid v2 on a real-device for realtime security policy enforcement.

To ease this task further, the author enhanced CopperDroid v2 to support `Python` and `C++` plug-ins.[3] As the static analysis described in Chapter 4 produces both `C++` classes and Python bindings as well as boilerplate `Python` and `C++` plug-in code for a given Android interface, manipulating Binder requests and replies is a straightforward task. In particular, every method of a given interface produces a function with the same parameters of `Task` and `Object`. The `Task` parameter in each function provides information about the currently executing thread, whereas the `Object` parameter provides access to the methods valid for that given `Object`. In the case of a Binder reply (versus a Binder transaction) these typically include a result as well as exception-related methods. The exception information can be used to mask an exception and make it appear as though an event occurred, or not.

Listing 5.3 shows the code that produces a pseudo-random IMEI number whenever CopperDroid v2 captures `getDeviceId` replies. The Python plug-in code for the `getDeviceId` method, checks to see if it is being called in the context of a given thread and only manipulate such resources in that case.

The plug-in support in CopperDroid v2 works for both interface transactions and replies, making it possible to manipulate a transaction to an interface and modify the data prior to it reaching the service in question. As shown in Section 5.4.2, while there is some overhead in invoking the Python plug-in, this is negligible to have any measurable effect on operations in general.

5.4.4 Use Cases

Through the combination of real-time reconstruction/modification and plug-in support, CopperDroid v2 enables the following three (and potentially more) use cases.

[3]The same actions can be achieved from within CopperDroid v2 by writing C++ code; here, the author show how CopperDroid v2 analysis framework and its support of plug-ins ease the task further.

```
def GetDeviceIdReply(Task, Object):
    # Valid methods for Task
    ...
    # GetProcessName

    # Valid methods for Object of type: CPhoneSubInfoGetDeviceIdReply
    ...
    # SetResult(ResultValue)

    if Task.GetProcessName() == "com.example.binderdemo":

        newIMEI = ""
        for x in range(0,15):
            newIMEI += str(random.randint(0,9))

        Object.SetResult(newIMEI)
```

Listing 5.3: Python plug-in code for introspecting and manipulating ICC-related objects.

Malware Analysis. CopperDroid v2 has already been utilised in the analysis of 3000+ malware samples, however, all of these were done based on post-processing data captured during execution. With the Python plug-in support in CopperDroid v2 it is possible to perform this analysis in real-time. While there are advantages to storing the data (as is currently done), for example for use in machine learning, being able to run these in real-time saves storage space and can provide more immediate results.

Anti-evasion. There are many techniques that malware can use to determine if it is running inside an emulator varying in their degree of difficulty, as mentioned in [84]. In the case of Android, some of the simplest ones are to query for things such as the device ID, subscriber ID, the line1 number or voice mail number (among others). As shown in Listing 5.3, it is easy to manipulate this type of information from within a Python plug-in. Other properties such as battery status is also used for determining whether or not an emulator is in use. Apps that query for battery status are notified by the batteryPropertiesChanged method of the IBatteryPropertiesListener interface, using the BatteryProperties class, which can be modified using a Python plug-in to mimic the characteristics of actual charging.

App/Service Fuzzer and error injection. BinderCracker [26] is an automatic testing framework for Android Services. Due to the fact that Binder calls contain both primitive and complex types, BinderCracker needed to record many Binder calls and build a dependency graph between them in order to understand how complex objects fit together. BinderCracker also needed to instrument the `Parcel` unmarshaling methods to understand the simple data types at each point, and then modify them accordingly.

This, however, did not give BinderCracker much context in terms of what specific primitive types represented. As mentioned in Section 3.9.2, parcels contain both simple and complex types. BinderCracker needs to record seed transactions and uses those to determine the primitive types of complex Android objects.

CopperDroid v2, however, automatically generates classes for complex Android objects, such as Intent, Rect, Bundle etc. As these are made available to Python plug-ins, it is simple to manipulate these objects directly as well as handles to such objects as they are passed around, as can be seen in Listing 5.4 and Listing 5.5.

```python
def SendTextTransaction(Task, Object):
    ...
    # Valid methods for Object of type: CSmsSendTextTransaction
    ...
    # GetSentIntent()
    # SetSentIntent(SentIntentValue)
    # GetDeliveryIntent()
    # SetDeliveryIntent(DeliveryIntentValue)
    ...
    sentIntent = Object.GetSentIntent()
    deliveryIntent = Object.GetDeliveryIntent()

    Object.SetSentIntent(deliveryIntent)
    Object.SetDeliveryIntent(sentIntent)

    ...
```

Listing 5.4: Python example that swaps the sentIntent and deliveryIntent (ellipsis denote code omitted for brevity)

This functionality can be used to build something similar to BinderCracker with far more context available with regards interfaces, parcels and complex objects, potentially enabling more thorough fuzzing. For example, rather than simply manipulating a floating point number, knowing it referred to a Global Positioning System (GPS) coordinate would permit a targeted change to perhaps cause a specific result.

```python
def GetIntentSenderTransaction(Task, Object):
    ...
    # Valid methods for Object of type: CActivityManagerGetIntentSenderTransaction
    ...
    # GetIntents()
    ...
    for intent in Object.GetIntents().GetElements():
        if (intent.GetAction() == "Sent"):
            intent.SetAction("SENT")
    ...
```

Listing 5.5: Python example which changes an Intent's action (ellipsis denote code omitted for brevity)

5.4.5 Limitations

Currently CopperDroid v2 does not support purely native Android interfaces which are implemented entirely in C++ with no AIDL file. There are 24 such interfaces (approximately 6.22%) of the total number of interfaces in Android, as opposed to the 363 Java Android interfaces (as of Oreo). However, this constitutes very little of the entire Android interface surface area, as the number of interface methods is 3.75% of the total number in Android. The percentage of lines of code involved in these C++ interfaces is 2.87% of the total in Android across all Java and C++ interfaces.

It is not difficult to support those interfaces written in C++ but would require parsing the C++ code for the interfaces and transforming them in a similar manner to that which CopperDroid v2 employs for the Java interfaces. This is something that the author is considering. The purely native interfaces are mostly used for the graphics surface/window management, event connectors, and hardware sensors (such as accelerometers and gyroscopes). While the reconstruction of graphics and window management is of little value, the ability to manipulate sensors via a plug-in does have potential applications.

The design of the CopperDroid v2 compiler is such that only a new C++ front-end needs to be written using the same AST classes (which already support both Java and C++) as those employed for managing Java interfaces. The code structure in the native interfaces is sufficiently similar to that of Java that little to no modification is required to perform the necessary object slicing required to support interface marshaling and unmarshaling. Therefore, not currently supporting native interfaces is not a design issue in CopperDroid v2, but simply due to the constraints of time (and is something that the author plans to address in the future).

The unmarshalling Oracle [78], by comparison, uses Java reflection to reconstruct complex objects and therefore cannot reconstruct the native C++ interfaces. In addition, the Oracle has to be passed the parameter types of a given interface parcel in order to reconstruct the contents. Conversely, the approach described in this work, utilising static analysis to automatically generate reconstruction code for Android interfaces, allows automatic unmarshalling and remarshalling of parcels. This is something that the unmarshalling Oracle (used in CopperDroid v1) is unable to achieve due limitations in its design.

5.5 Summary

This chapter described how the various portions of CopperDroid v2 are integrated to provide a system that enables seamless reconstruction of system calls and Binder interfaces and complex objects in real-time. It further described those hand-written classes that assist in bridging the gap between `Java` and `C++`, as well as the boilerplate plug-in code generation. CopperDroid v2 was further evaluated in terms of performance, flexibility and extensibility for use a platform for further research.

Chapter 6 describes a plug-in built on CopperDroid v2 that uses black-box differential analysis to perform information leakage detection of sensitive information through the real-time manipulation of sensitive items.

Chapter 6

Information Leakage Detection

6.1 Introduction

The preceding chapters (Chapter 3, Chapter 4 and Chapter 5 describe how static analysis is leveraged to automatically produce a framework that enables real-time reconstruction and manipulation of Binder ICC and Android objects. This chapter builds on the model described thus far and, through the use of a plug-in to CopperDroid v2, describes a differential analysis approach to detecting the leakage of sensitive information on Android.

The leakage of sensitive and personally identifiable information (PII) is a well-known of issue, however, it is also a challenging problem to address. While malware may seek to intentionally leak this information, even otherwise benign apps can do so whether intentionally or unintentionally [72] [69] [23] [77] [4]. The reasons for collecting sensitive information vary, for example some apps may choose to identify users so that they can provide a unified experience across all devices on which a user has the app installed. Others may wish to monetize the app through targeted ads. If a particular app encodes the sensitive information in a way that can't be reversed, for example using a SHA-256 hash the leakage will likely be less risky, however, should still require user consent.

The Federal Trade Commission does fine Android app developers when they breach privacy laws [14] [13] [15]. Following suit, the European Union has introduced stricter privacy laws to safeguard against sensitive information being leaked and will fine those found to be doing so [27]. Therefore, it is likely that more app developers will attempt to obfuscate or encrypt sensitive information prior to transmitting it in order to avoid being fined.

Prior research on information leakage detection uses static or dynamic taint analysis, for example TaintDroid [23] and TaintART [77]. However, there are ways to work around taint analysis through implicit flows (using information contained in a given portion of data to make control flow decisions that transmit data based on the control path taken), or rarely tracked explicit flows, such as writing to a file and later reading from and then transmitting the data [23]. Furthermore, these require some modification to the runtime (either Dalvik or ART) in order to work. While dynamic taint analysis can detect information leakage from downloaded code, native code that leaks information directly via system calls may be undetected. TaintDroid [23] attempts to mitigate this by ensuring the only native libraries included in the firmware can be loaded, citing the fact that only 5% of the apps at that time included shared objects.

Static taint analysis, on the other hand decompiles apps and then uses tools to construct control-flow graphs that are used determine how information flows from sources (where sensitive information is accessed) to sinks (where it is transmitted off the device) through the app [4]. This approach, while likely to have high accuracy, is both time consuming and resource intensive [72]. Furthermore, it cannot be used to detect leaks in native code or code downloaded by the app. It may also result in false positives if, for instance, a path between source and sink can never be executed.

Another approach is to use network flow analysis to detect sensitive information being transmitted over the network [51] [70] [75] [57]. However, this approach only works with data sent in the clear or using commonly known obfuscations such as Base64 encoding. When an app uses its own custom encoding or encrypts data prior to sending it to the network, then network flow analysis fails to detect the leak.

A further approach is to use black-box differential analysis through multiple executions of a program, with differing inputs and analysing the resultant `HTTP` and `HTTPS` traffic. An example of this is the Privacy Oracle [44]. Another project (AGRIGENTO) extends this model by controlling various sources of non-determinism as apps are executed and analyses `HTTP` and `HTTPS` traffic from these apps [17]. While the majority of apps do use `HTTP` or `HTTPS` for transmitting data across the network, this is by no means a guarantee.

A differential analysis approach is based on the intuition that if the system is sufficient controlled then any data transmitted from the system will only change if a specific piece of (tracked) information is explicitly changed. Thus if one can control the non-determinacy in Android and only modify sensitive information, detecting differences in transmitted information makes it possible to detect information leakage. The work described in this chapter is akin to record/replay, however, it is a very lightweight approach and is evaluated later in this chapter.

The author of this thesis also uses differential analysis conceived independently of the work described in [17]. This approach is necessitated due to CopperDroid v2 executing outside the Android guest and operating entirely on system call data.

This chapter describes a `Python` plug-in that is used to support the differential analysis by permitting modification of sensitive information building on the boilerplate code generated by the CopperDroid v2 compiler. This provides an example of using the plug-in infrastructure that CopperDroid v2 exposes while also demonstrating its use in application of information leakage detection.

6.2 Overview

The plug-in support and boilerplate code generated by the static analysis (described in Chapter 4) simplifies the development of various security-related research tools. This chapter demonstrates the versatility of CopperDroid v2 with a plug-in that utilises the control it has over the guest operating system, both through system calls and Binder operations, to aid in the detection of sensitive information leakage. This is achieved through running app/malware samples multiple times and changing only one specific sensitive datum per execution. To ensure that each run is identical, non-determinism is reduced by always returning the same values from operations that could be used for entropy.

As Chapter 5 shows, the `Python` plug-in boilerplate code produced by the CopperDroid v2 compiler includes a `copperdroidplugin.py` file which has the following routines that can be invoked from CopperDroid v2:

1. InitializePlugIn(ConfigFileName)
2. SetPackageName(PackageName)
3. UninitializePlugIn()
4. InvokeInterfacePlugIn(Task, InterfaceName, Type, Method, Object)
5. InvokeSystemCallPlugIn(Entry, Task, SystemCall, Object)
6. UpdateScenario(ScenarioName)
7. Serialize()

Functions 1-5 are invoked for all plug-ins, while functions 6 and 7 were added to support launching the emulator and running multiple scenarios sequentially. In the latter case, the CopperDroid v2 automation framework stops the app between scenarios, instructs the plug-in to serialise its data, performs some cleanup and then relaunches the app with a new scenario. This feature exists in order to avoid starting the emulator instance for every iteration and saves 5+ minutes per-scenario.

6.2.1 Sources of Non-determinism

Performing differential analysis requires that non-deterministic inputs be controlled so that only the data actually modified intentionally affects the final output. In Android there are multiple sources of non-determinism that have to be controlled, for instance, random number sources such as the `Random` or `SecureRandom` classes, system values such as the remaining disk or memory space of the system, timing data such as the current time or date, as well as information obtained from network sources. For the purposes of this thesis, the latter is excluded due to time constraints, however, it can be trivially added in the future. The experiments described later in this chapter use a modified version of the app stimulation described in [78], where every activity is launched and MonkeyRunner is used to stimulate the all apps in the same way. The automation also starts every activity in a given app, stimulating the app, waiting for a fixed period of time and then moving to the next activity.

Sensitive information accessed by malware can be transmitted in the clear, encrypted or obfuscated (such as using Base64 encoding). When encrypting data, malware can use either secure key generation or a simple random number generator (such as the `Random` class in `Java`). Through experimentation, the author determined that only three sources of entropy need to be controlled to ensure the same secure cryptographic keys are generated for every run when the `SecureRandom` class is used. The keys will change during a run, however, the initial starting key and subsequent keys will always stay in sync. These three pieces of information are the Process Id (PID), data read from the `/dev/random` device and the current millisecond or nanosecond time (as determined using the `gettimeofday()` system call).

However, in KitKat and earlier versions the `Random` class determines the starting seed (if none is provided) by adding the current millisecond time to the
`identityHash()` for the calling instance of the `Random` class. Thus any apps or malware using the `malware` utilising the `Random` class on versions prior to Lollipop can result in false positives (but not false negatives). While AGRIGENTO [17] replaced the random functions that apps utilise, CopperDroid v2 does not modify the image or

the app. However, beginning with Lollipop, the initial seed for the `Random` class is obtained by combining the current nanosecond time (obtained from `gettimeofday()`) with a constant (either through addition or exclusive-OR). Therefore, returning a consistent value from `gettimeofday()` will result in the initial seed for the `Random` class remaining the same through every execution on Android Lollipop and subsequent versions. Apps also utilise other sources for entropy, such as the `IActivityManager` method `getMemoryInfo()` or the `StatFs.getAvailableBlocksLong()` method.

6.2.2 Sensitive Information Sources

The most commonly leaked information on Android includes the device identifier (IMEI), subscriber identifier (IMSI), location, contacts [23]. Additionally, some malware forwards or leaks SMS messages [48] [49]. The plug-in described in this chapter also tracks potential leakage of the phone number, voicemail number and SIM serial number. Any sensitive values that need to be changed are specified in `JSON` and are accessed via information in the plug-in configuration file (example shown in Listing 6.1).

Some information is also included which is static across all configurations to ensure that entropy is controlled. For example, the `MemoryInfo` property contains all of data related to the `IActivityManager.getMemoryInfo()` method, while the `Location` field contains timestamp information that would otherwise vary with every query even if the location is unchanged. Furthermore, the `StatFs` data is used to return consistent data for different paths from the the `statfs64` system call.

The flags field specifies which sensitive information to track for a given execution and the results of a run are written to the output file specified. Once all runs are complete another tool analyses the output against the baseline (which controls the entropy but doesn't modify any sensitive data).

6.2.3 Sinks

While apps or malware may access sensitive data, information leakage only occurs if they transmit the data to an external entity. Therefore, whenever a sensitive datum is accessed, a flag is set (specific to what was accessed) and all subsequent data transmitted from the process or any processes it launches is logged in the output file. The sensitive information is called a *source* and the transmission point a *sink*. In the case of this plug-in the sendto(), `sendmsg()`, `write()` and `writev()` (for sockets) and SMS `sendText()`, `sendMultipartText()` and `sendData()` are all tracked.

Although only 2% of apps leak sensitive information over SMS that would still mean approximately 60,000 apps on the Google Play Store provided the same percentage applies [55].

6.3 Approach

The analysis of a given app is accomplished in the following manner, an app is executed four times in succession (stopping, resetting state and restarting it between executions) with certain properties that affect non-determinism being controlled. If at any point during a given execution sensitive information is accessed, then that access is logged and any action that transmits data (such as `sendto()`, `sendmsg()`, `write(sockedfd)` or `sendText()`) is subsequently logged. Each entry includes flags that determine *what* information was accessed, be it the `IMEI`, `IMSI` or some other sensitive datum. The flags remain set once something is accessed, since there is no guaranteed method (using a purely system call approach) to determine if the next sink actually transmitted the datum or if perhaps multiple items were accessed and then sent together. The initial four runs establish a baseline for the app.

Once the baseline has been determined, the app is run an additional eight times during which only one sensitive information source is altered per run, for example Location, Contacts or `IMEI`. The sources of private information altered are as follows: `IMEI` (device ID), `IMSI` (subscriber ID), Location, phone number, `ICCID` (Sim card number), voice mail number, contacts and SMS messages.

The plug-in to CopperDroid v2 uses `JSON` files for managing each execution. A `JSON` file used for establishing the baseline can be seen in Listing 6.1.

```
{
    "isBaseline" : true,
    "OutputFileName" : "$RESULTS_DIRECTORY/baseline.json",
    "Flags" : [
        "TELEPHONY_ALL",
        "LOCATION",
        "CONTACTS",
        "SMS_READ"
    ],
    "Properties" : [
        {
            "name" : "Pid",
            "value" : 1260
        },
        {
            "name" : "Tid",
            "value" : 1354
        },
        {
            "name" : "Location",
            "value" : {
                "Provider" : "gps",
                "Time" : 1542349162000,
                "ElapsedRealtimeNanos" : 239759509138,
                "IsFromMockProvider" : false
            }
        },
        {
            "name" : "MemoryInfo",
            "value" : {
                "availMem" : 613654528,
                "totalMem" : 766939136,
                "threshold" : 150994944,
                "lowMemory" : false,
                ...
            }
        }
    ],
    "AddFile" : [
        {
            "target" : "/data/data/com.cd2.preptool/files/rules.json",
            "redirected" : "/test/plugin/infoleak/data/rules.default.json"
        }
    ],
    "StatFs" :  [
        {
            "target" : "/",
            "value" : "009988b400788...0673a10534b2"
        },
        ...
        {
            "target" : "/storage/sdcard",
            "value" : "c08a8007882b0...0673a10534b2"
        },
        {
            "target" : "/data",
            "value" : "c08a87b400788...0673a10534b2"
        },
        ...
        {
            "target" : "*",
            "value" : "009988b400788...0673a10534b2"
        }
    ]
}
```

Listing 6.1: Sample JSON configuration file for the baseline properties

The flags array indicates which sensitive data is pertinent to the current run, however, all sensitive information access is tracked in all instances. The flags only serve to isolate which were modified in the individual run (or in the case of the baseline to indicate that all need to be considered). As the plug-in is invoked by CopperDroid v2 for a specific piece of sensitive information, it updates the flags to indicate which items have been been accessed.

The stimulation that the CopperDroid v2 automation framework uses, analyses `APK` permissions and checks for Location and SMS read permissions. If location permissions are included in the apps manifest, then location updates are sent 10 times per run, with the same co-ordinates each time. Similarly if SMS read permissions are requested then several SMS messages are sent to the device. Both of these communicate via the Android Debug Bridge (adb).

For the most part, the plug-in modifies sensitive information in different areas by simply adding code to the system call or interface handler that is automatically generated as a result of the static analysis performed by the CopperDroid v2. The two exceptions are contacts and SMS messages, which are updated using a tool described later in this chapter. However, access to contacts or SMS is detected using `IActivityManager` interface and is described in Section 6.3.3.

While the baseline appears to be updating the Location information, it is only doing so in order to ensure that timing information included in a Location update does not differ across runs, the co-ordinates are unchanged except for the specific Location run.

The code generated by the CopperDroid v2 compiler greatly simplifies the manipulation of data in real-time. The following sub-sections illustrate how access to a given sensitive resource is determined and, where applicable, how the values are changed.

6.3.1 Telephony related information

The various telephony related resources are accessed using the `Telephony` interface, `IPhoneSubInfo`. This interface is responsible for providing information such as the `IMEI` (Device Id), `IMSI` (Subscriber Id) and phone number. While CopperDroid v2 unmarshals both Binder calls (transactions) and replies, no parameters are passed into the transaction and therefore only the automatically generated reply routines need to be modified. Listing 6.2 illustrates how the device ID returned by `IPhoneSubInfo.getDeviceId()` call is (potentially) modified depending on the data specified in the `JSON` configuration for a given run. If the configuration does not specify a device ID, then the result is unchanged, although the value is still logged. If a different value is specified for the device ID, then the modified value is returned.

Subscriber ID, phone number and other sensitive telephony data is all modified in this way. The automatically generated boilerplate code for each interface method's reply is modified to either return a specific value or the default (if none is specified) in addition to logging the access.

```
def GetDeviceIdReply(Task, Object):

    # Valid methods for Task
    # GetProcessIdentifier
    # GetThreadIdentifier
    # GetProcessName

    # Valid methods for Object of type: CPhoneSubInfoGetDeviceIdReply
    # GetResult()
    # SetResult(ResultValue)
    ...

    if packageinfo.IsTrackedPackage(Task.GetProcessName()) or packageinfo.IsTrackedProcess(
        Task.GetProcessIdentifier()):

        present, deviceId = config.GetConfig().GetProperty("DeviceId")

        if present != False:
            Object.SetResult(deviceId)
        else:
            deviceId = Object.GetResult()

        event = events.Event(events.ACCESS_PERSONAL_INFO, "DeviceId", Task.GetProcessName(),
             Task.GetProcessIdentifier(), Task.GetThreadIdentifier(), events.
             FLAG_TELEPHONY_DEVICE_ID)
        event.SetData(deviceId)
        events.GetEvents().AddEvent(event)
```

Listing 6.2: Plug-in code to modify DeviceId

6.3.2 Location Information

An app can obtain the current location information through one of two methods, it can either call the `ILocationManager.getLastKnownLocation()` method, or it can request location updates. In the latter case, a given app requests location updates from the `ILocationManager` and passed in an instance of a class that implements the `ILocationListener` interface. Once registered, any update to the location (even with the same information) results in the app's class being invoked on its `onLocationChanged()` method.

The info-leak plug-in implemented by the author handles both cases, although only the code for the callback is shown in Listing 6.3. As can be observed by referring to the listing, the `JSON` configuration can update any field in the `Location` object.

While an app is more likely to send only the latitude, longitude, altitude and other information pertinent to the devices current location, it's entirely possible that the entire location object could be serialised and sent. Furthermore, as the location includes timestamps an app could attempt to use this for entropy. Therefore, the *Time* and *ElapsedRealtimeNanos* fields are modified in the baseline case (and all non-location cases) to ensure they are unchanged across invocations. Only the location run does not update the timestamp information and leaves it as the value set by the location service. However, latitude, longitude and altitude are all modified by the configuration.

```
def OnLocationChangedTransaction(Task, Object):

    # Valid methods for Task
    # GetProcessIdentifier
    # GetThreadIdentifier
    # GetProcessName

    # Valid methods for Object of type: CLocationListenerOnLocationChangedTransaction
    # GetLocation()
    # SetLocation(LocationValue)

    if packageinfo.IsTrackedPackage(Task.GetProcessName()) or packageinfo.IsTrackedProcess(
         Task.GetProcessIdentifier()):
        present, locationModifiers = config.GetConfig().GetProperty("Location")

        if present != False:
            Location = Object.GetLocation()

            ...

            if "Time" in locationModifiers.keys():
                Location.SetTime(locationModifiers["Time"])

            if "ElapsedRealtimeNanos" in locationModifiers.keys():
                Location.SetElapsedRealtimeNanos(locationModifiers["ElapsedRealtimeNanos"])

            if "Latitude" in locationModifiers.keys():
                Location.SetLatitude(locationModifiers["Latitude"])

            if "Longitude" in locationModifiers.keys():
                Location.SetLongitude(locationModifiers["Longitude"])

            if "Altitude" in locationModifiers.keys():
                Location.SetAltitude(locationModifiers["Altitude"])
                Location.SetHasAltitude(True)

            ...

            if "IsFromMockProvider" in locationModifiers.keys():
                Location.SetIsFromMockProvider(locationModifiers["IsFromMockProvider"])
            Object.SetLocation(Location)
        else:
            Location = Object.GetLocation()
        event = events.Event(events.ACCESS_PERSONAL_INFO, "Location", Task.GetProcessName(),
            Task.GetProcessIdentifier(), Task.GetThreadIdentifier(), events.FLAG_LOCATION)
        event.SetData(repr(Location))
        events.GetEvents().AddEvent(event)
```

Listing 6.3: ILocationListener onLocationChanged() callback

6.3.3 Contacts and SMS

Content providers such as Contacts and SMS return results to queries using shared memory regions. These regions contain the returned data in the form of `SQLite` cursors. The author has utilised CopperDroid v2 to determine how this information is shared and how to manipulate it, however, this is still being researched. Therefore, the author developed a tool (described in the next section) that modifies the Contacts and SMS messages for each iteration.

Detecting that an app is accessing a content provider (such as contacts or SMS) requires tracking state through `IActivityManager` interface method invocations. The first of these is the `getContentProvider` method call which queries for a given content provider. The reply contains a Binder handle corresponding the content provider in question. As mentioned in Section 3.9.2, Binder calls are synchronous and are either one way (such as the `ILocationListener.onLocationChanged()` call mentioned earlier) or return their result on the same thread, but not necessarily on exit from the current system call (it can occur on a subsequent `ioctl` exit).

Additionally, while tracking the `getContentProvider` method call enables the plug-in to determine than an app has queried for the content provider, unless a call to the method `refContentProvider()` occurs, the sensitive data in question has not been accessed. Therefore a second call `refContentProvider` call must be inspected as it is passed a Binder handle that uniquely identifies the content provider being queried. Only by tracking the initial `getContentProvider` method call and response, can one determine which provider a given `Binder` handle to which it refers.

Therefore, the info-leak plug-in monitors all calls to the `IActivityManager.getContentProvider()` method, adding a pending handle entry to a dictionary using the process ID and thread ID as the key along with the name of the provider being requested. This is necessary as this is the only instance in which the content provider name will be passed as a parameter to a Binder interface.

```
def GetContentProviderTransaction(Task, Object):

    # Valid methods for Task
    # GetProcessIdentifier
    # GetThreadIdentifier
    # GetProcessName

    # Valid methods for Object of type: CActivityManagerGetContentProviderTransaction
    ...
    # GetName()
    ...

    if packageinfo.IsTrackedPackage(Task.GetProcessName()) or packageinfo.IsTrackedProcess(
        Task.GetProcessIdentifier()):
        InsertPendingHandleEntry(Task.GetProcessIdentifier(), Task.GetThreadIdentifier(),
            Object.GetName())
```

Listing 6.4: GetContentProvider call

The plug-in is subsequently invoked when the reply is dispatched back to the app and examines the data included in the call. As Section 4.10.1 mentions, `flat_binder_object` instances are encapsulated in a `BinderReference` class that has methods to simplify handling these structures.

One such method is the `IsHandle()` method that determines the type of the `flat_binder_object` and returns true only if the data contained therein refers to a Binder handle. The `GetCph()` method on the automatically generated reply class returns an instance of the `IActivityManager.ContentProviderHolder` class. As with all such classes in the `Android` framework, the CopperDroid v2 compiler automatically generates `Get` and `Set` methods for all variable fields and exposes these through `Python` bindings. Thus the connection field is made available for query by the plug-in reply handler method shown in Listing 6.5.

If a pending handle entry can be located for the current process/thread, which will be the case immediately following a call to `getContentProvider`, then the reply handler method determines if the connection `flat_binder_object` contains a handle and if so, inserts the handle into a Binder handle table along with the name and deletes the pending handle entry.

When a content provider is released in an app (for instance if a method local variable goes out of scope), then the `removeContentProvider()` method on the `IActivityManager` interface is invoked with the Binder handle in question. When this occurs, the info-leak plug-in deletes the handle entry from the table specific to the current process.

```
def GetContentProviderReply(Task, Object):

    # Valid methods for Task
    # GetProcessIdentifier
    # GetThreadIdentifier
    # GetProcessName

    # Valid methods for Object of type: CActivityManagerGetContentProviderReply
    # GetCph()
    ...
    if packageinfo.IsTrackedPackage(Task.GetProcessName()) or packageinfo.IsTrackedProcess(
         Task.GetProcessIdentifier()):
        pendingHandleEntry = FindPendingHandleEntry(Task.GetProcessIdentifier(), Task.
             GetThreadIdentifier())

        if pendingHandleEntry != None:
            cph = Object.GetCph();

            if cph.GetConnection().IsHandle():
                InsertBinderHandleEntry(Task.GetProcessIdentifier(), cph.GetConnection().
                     GetHandle(), pendingHandleEntry.GetResourceName())

            DeletePendingHandleEntry(Task.GetProcessIdentifier(), Task.GetThreadIdentifier())
```

Listing 6.5: Get Content Provider reply

While the previous two plug-in methods are used to determine the handle referring to a given content provider, the actual access to sensitive data is only determine when it is used. For that to happen, a call to the `IActivityManager` method `refContentProvider()` occurs, in which the info-leak plug-in locates the handle entry referring to the handle provided to the method call and determines if it matches either of the providers of interest. If so, the access is logged as can be observed in Listing 6.6.

```
def RefContentProviderTransaction(Task, Object):
    ...

    # Valid methods for Object of type: CActivityManagerRefContentProviderTransaction
    # GetConnection()
    ...

    if packageinfo.IsTrackedPackage(Task.GetProcessName()) or packageinfo.IsTrackedProcess(
         Task.GetProcessIdentifier()):
        connection = Object.GetConnection()

        if connection.IsHandle():
            binderHandleEntry = FindBinderHandleEntry(Task.GetProcessIdentifier(), connection
                 .GetHandle())

            if binderHandleEntry != None:
                if binderHandleEntry.GetResourceName() == "com.android.contacts":
                    event = events.Event(events.ACCESS_PERSONAL_INFO, "Contacts", Task.
                         GetProcessName(), Task.GetProcessIdentifier(), Task.
                         GetThreadIdentifier(), events.FLAG_CONTACTS)
                    event.SetData("Contacts Accessed")
                    events.GetEvents().AddEvent(event)

                if binderHandleEntry.GetResourceName() == "SMS":
                    event = events.Event(events.ACCESS_PERSONAL_INFO, "SMS", Task.
                         GetProcessName(), Task.GetProcessIdentifier(), Task.
                         GetThreadIdentifier(), events.FLAG_READ_SMS)
                    event.SetData("SMS Accessed")
                    events.GetEvents().AddEvent(event)
```

Listing 6.6: Reference Content Provider

The actual contents related to either Contacts or SMS is specified in `JSON` data file on the host system. A tool is executed once per-run that attempts to open the file from its local `/data/...` folder and the info-leak plug-in intercepts the `open` and `openat` system calls and opens a file from the host if requested. A *fake* file descriptor (of value 10000 or higher) is returned and with all subsequent operations being handled by the plug-in. For example, following an `open` system call, a call is made to the `fstat64` system call to determine the file size, which the plug-in responds to with the appropriate value (depending on the file size on the host system). All `read` requests then return data as requested and `close` system call results in the plug-in removing the descriptor from the descriptor table for the current process.

The plug-in handles either the addition of a new file (should the file not exist in the guest) or replacement of a file in the guest with one from the host, although the latter is currently only used in the next chapter. Thus as each iteration is run, the plug-in can specify a new data file resulting in the different behaviour without having to modify the guest image (other than installing the preparation tool). The preparation tool is only required in order to simplify the running of multiple information leakage passes booting the emulator only once per app.

6.3.4 Tracking System State

CopperDroid v2 and its plug-ins observe the whole Android system (with the exception of kernel components) and can therefore track not just a given package but also any processes it launches. The automation framework first installs the preparation tool and the app being tested and then starts the full system capture. Thus calls to Binder interfaces are only made visible to the plug-in once the app has been installed. There are apps and malware than install other packages and knowing what processes to include in the collected data is critical to ensuring that all relevant data is captured.

When a package is installed a call is made to the `IPackageInstallObserver` interface to notify the system of the package installation. As the CopperDroid v2 automation is controlled by the author, any packages installed after capture has started are initiated by the app in question. Therefore the plug-in adds any packages installed to the list of tracked packages as demonstrated in Listing 6.7.

```python
def PackageInstalledTransaction(Task, Object):
    ...
    # Valid methods for Object of type: CPackageInstallObserverPackageInstalledTransaction
    # GetPackageName()
    ...
    packageinfo.InsertPackage(Object.GetPackageName())
```

Listing 6.7: Tracking package installation

Similarly when a process is forked or cloned resulting in a new distinct process (i.e. not a thread), then it is necessary to track any actions taken by the the new process. Failing to do so would allow an app to query sensitive data and then launch a program that transmits the data. If the new process is not tracked then the system would fail to detect the transmitted data.

Listing 6.8 demonstrates how the plug-in handles the `clone` system call, similar code is used for the `fork` and `vfork` system calls. However, the author has only ever observed the `clone` system call being used in Android that is by no means a guarantee. An app could utilise native code to perform a `fork` and `execve` in an attempt to evade detection.

```
def SystemCallClone(Entry, Task, Object):
    ...
    # Valid methods for Object of type: CSystemCallClone
    ...
    # GetFlags()
    # GetReturnValue()
    ...
    if Entry == false:
        newProcessPid = Object.GetReturnValue()
        parentProcessId = Task.GetProcessIdentifier()
        flags = Object.GetFlags()

        if (newProcessPid != parentProcessId) and
            ((flags & (CLONE_THREAD | CLONE_FILES)) == 0):

            InheritProcessHandleTable(parentProcessId, newProcessPid)

            if packageinfo.IsTrackedPackage(Task.GetProcessName()) or
               packageinfo.IsTrackedProcess(Task.GetProcessIdentifier()):

                packageinfo.InsertProcess(Task.GetProcessIdentifier())
```

Listing 6.8: Tracking processes through clone

6.4 Controlling Sources of Non-determinism

A key requirement in differential analysis such as that described in this chapter is minimizing or removing non-determinism. As described earlier there are multiple sources of non-determinism that can affect the output produced by a given app. For instance, timing values obtained from the system, the process ID (PID) of the current process and others. This section describes how each is controlled.

6.4.1 Randomly Generated Values

There are a variety of methods that an app can utilise to obtain randomly generated values. For instance the `/dev/random`, `/dev/urandom` and `/dev/arandom` devices can all provide random values. The `SecureRandom` class combines output from the `/dev/urandom` device, the current process `PID` and the current time in order to seed its generator. The `Python` plug-in described in this chapter addresses this by always returning 0's from any reads to these devices. While the author tested different values including a sequence of chosen values that always initiated at the same point the first time a given process or thread read from these devices, returning 0's was sufficient for this example (see Listing 6.9).

The `randomUUID() Java` method [63] returns a randomly generated `UUID`. However, it relies on the same data sources as the `SecureRandom` class and therefore can be controlled in the same way.

```
def SystemCallRead(Entry, Task, Object):

    # Valid values for Entry
    # True  -- this is being invoked on system call entry
    # False -- this is being invoked on system call exit

    # Valid methods for Task
    # GetProcessIdentifier
    ...

    # Valid methods for Object of type: CSystemCallRead
    # GetFileDescriptor()
    # SetFileDescriptor(FileDescriptor)
    # GetBuffer()
    # SetBuffer(Buffer)
    # GetCount()
    # SetCount(Count)

    if Entry == False:
        ...
        fileDescriptor = Object.GetFileDescriptor()

        handleTableEntry = FindHandleEntry(Task.GetProcessIdentifier(), fileDescriptor)

        if handleTableEntry != None:
            fileObject = handleTableEntry.GetObject()
            identifier = fileObject.GetIdentifier()

            if identifier in ["/dev/random","/dev/urandom","/dev/arandom"]:

                bufferData = bytearray(Object.GetBuffer())

                for i in range (0, Object.GetCount()):
                    bufferData[i] = 0

                Object.SetBuffer(bytes(bufferData))
        ...
```

Listing 6.9: Read system call controlling the values returned from the `/dev/urandom device`

6.4.2 Timing Values

Apps can request timing information from the system using calls such as `System.currentTimeMillis()` or `System.nanoTime()`. Both of these APIs make use of the `gettimeofday()` system call in order to obtain the timing values. If a process of interest calls the `gettimeofday()` system call, the plug-in modifies the output to the same value every time (as shown in Listing 6.10). While the value chosen in this case is 0 for both, the author has verified that any value that stays constant (or predictable, for instance returning a specific set of values in order for a given process/thread) results consistent output across iterations. However, in order to mitigate the risk of external events resulting in additional `gettimeofday()` calls and affecting the results, the author kept these constant. Doing so had no observable side-effects in apps, which the author verified by manipulating apps through their user-interface within the CopperDroid v2 emulator while a `Python` plug-in set the values to 0.

```
def SystemCallGettimeofday(Entry, Task, Object):
    ...
    # Valid methods for Task
    # GetProcessIdentifier
    # GetThreadIdentifier
    # GetProcessName

    # Valid methods for Object of type: CSystemCallGettimeofday
    # GetTimeValue()
    # SetTimeValue(TimeValue)
    ...

    if Entry == False and (packageinfo.IsTrackedPackage(Task.GetProcessName()) or
                           packageinfo.IsTrackedProcess(Task.GetProcessIdentifier())):

        seconds = 0
        nanoseconds = 0

        timeBuffer = bytearray(8)

        struct.pack_into("<i", timeBuffer, 0, seconds)
        struct.pack_into("<i", timeBuffer, 4, nanoseconds)

        Object.SetTimeValue(bytes(timeBuffer))
```

Listing 6.10: Modification of gettimeofday system call data

6.4.3 System Related Data

As mentioned earlier, some of the secure cryptographic random number generators make use of the process ID as a source of entropy (as determined by the author through experimentation). Apps can also make use of other system values such as those provided by `StatFs`, such as `getAvailableBytesLong()` or the `IActivityManager` interface method `getMemoryInfo()`. As CopperDroid v2 is invoked for all system calls and Binder operations, any plug-in registered with CopperDroid v2 is invoked for these too. Thus a plug-in simply has to implement code in the appropriate system call or interface method's generated boilerplate code to modify the output as required.

The process ID can be controlled by modifying the output of the `getpid()` system call which is very simple to accomplish, as shown in Listing 6.11.

```
def SystemCallGetpid(Entry, Task, Object):
    ...

    # Valid methods for Object of type: CSystemCallGetpid
    # GetReturnValue()
    # SetReturnValue(ReturnValue)

    if Entry == False and (packageinfo.IsTrackedPackage(Task.GetProcessName()) or
                           packageinfo.IsTrackedProcess(Task.GetProcessIdentifier())):

        present, pid = config.GetConfig().GetProperty("Pid")

        if present:
            Object.SetReturnValue(pid)
```

Listing 6.11: Handler for getpid() system call

The plug-in written for this example also modifies the output of the `StatFs` and `getMemoryInfo()` APIs mentioned above, using the values provided in the configuration file for each iteration. Listing 6.12 illustrates how simple this is to do by modifying the automatically generated reply handler for the interface method.

```
def GetMemoryInfoReply(Task, Object):

    # Valid methods for Task
    # GetProcessIdentifier
    # GetThreadIdentifier
    # GetProcessName

    # Valid methods for Object of type: CActivityManagerGetMemoryInfoReply
    # GetOutInfo()
    # SetOutInfo(OutInfoValue)
    ...

    if packageinfo.IsTrackedPackage(Task.GetProcessName()) or
       packageinfo.IsTrackedProcess(Task.GetProcessIdentifier()):

        present, memoryInfoModifiers = config.GetConfig().GetProperty("MemoryInfo")

        if present != False:
            memoryInfo = Object.GetOutInfo()
            ...
            if "availMem" in memoryInfoModifiers.keys():
                memoryInfo.SetAvailMem(memoryInfoModifiers["availMem"])

            if "totalMem" in memoryInfoModifiers.keys():
                memoryInfo.SetTotalMem(memoryInfoModifiers["totalMem"])

            if "threshold" in memoryInfoModifiers.keys():
                memoryInfo.SetThreshold(memoryInfoModifiers["threshold"])
            ...

            Object.SetOutInfo(memoryInfo)
```

Listing 6.12: IActivityManager getMemoryInfo() reply handler

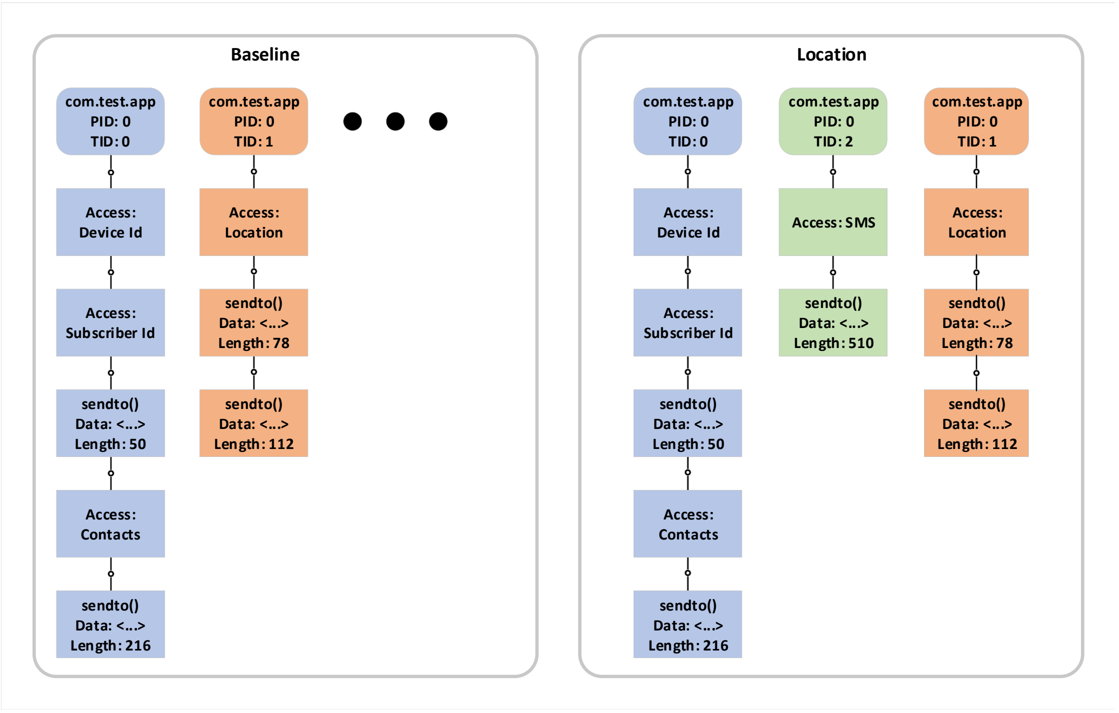
In the case of the `StatFs` class, when the `statfs64` system call is invoked, the data to use is determined by the path passed into the system call. If the path doesn't match any provided in the configuration then the data matching the `"*"` entry used. The data in the configuration is passed in as hexadecimal and is converted to binary data upon loading a given configuration. This binary data is simply copied over the whole `statfs64` buffer.

6.5 Detecting Information Leakage

Android apps can have multiple threads within their process and may also launch other processes, either from the firmware or downloaded. Therefore, when performing differential analysis it is necessary to determine which process/thread in each iteration match those in the base line and other iterations. As mentioned earlier, the logging of information only begins the first time the app being analysed accesses sensitive information. Once that occurs every instance of data being transmitted off the device is recorded. Thus if an app sent thousands of packets of data prior to accessing anything sensitive, those would not be recorded as it could not result in information leakage.

Every *event* recorded in the logs includes the process or package name as well as the process and thread ids for the app in question. The data contained in a given log for a specific iteration is separated based on the process name, process ID and thread ID in which the event occurred. The process and thread identifiers are not important in terms of their values (since these change as processes are launched and can't be controlled), however, the analysis utilises them to assign psuedo-identifiers to each process/thread. For instance the main process of an app is assigned a process ID of 0 and thread ID of 0, while the next thread is a assigned a process ID of 0, but a thread ID of 1. This is true for the baseline. However, for the other iterations, process name and events are used to identify which processes/threads match those in the baseline and all are assigned values accordingly.

The process and thread pseudo-identifier assignments use the contents of events to determine which thread or process matches, for example, if the device ID and subscriber ID is accessed or the size of the data being transmitted and how it is transmitted. A app could use `sendto()` or `sendmsg()` to transmit data off the device, these are noted separately and therefore assist in the construction of the apps profile. Figure 6.1 illustrates how this selection is performed and process/thread pseudo-identifier assignments determined.



Baseline
com.test.app PID: 0 TID: 0
Access: Device Id
Access: Subscriber Id
sendto() Data: <...> Length: 50
Access: Contacts
sendto() Data: <...> Length: 216
com.test.app PID: 0 TID: 1
Access: Location
sendto() Data: <...> Length: 78
sendto() Data: <...> Length: 112
Location
com.test.app PID: 0 TID: 0
Access: Device Id
Access: Subscriber Id
sendto() Data: <...> Length: 50
Access: Contacts
sendto() Data: <...> Length: 216
com.test.app PID: 0 TID: 2
Access: SMS
sendto() Data: <...> Length: 510
com.test.app PID: 0 TID: 1
Access: Location
sendto() Data: <...> Length: 78
sendto() Data: <...> Length: 112

Figure 6.1: App Process and Thread identification through event analysis

Following this assignment, the actual data transmitted is examined. The four baseline iterations are compared to determine what data, if any, differs between them since non-deterministic information is controlled as much as is possible. This permits the analysis to determine which data transmitted, if any, differs without changes to the environment. This assists in determining the potential for false positives.

Only one of sensitive information is modified per-iteration, therefore, the final stage of the analysis focuses only on the transmitted data and removes any matching the baseline or differing across all iterations (although these are tracked to avoid false negatives). Finally the analysis determines which flags are set for each transmission event remaining and determines if this matches what the current configuration is modifying. If so, the leak is logged, otherwise the examination continues until all events have been analysed and any leaks identified.

While the listing shown earlier in this chapter for the baseline (Listing 6.1) enables flags for all types of sensitive information, the configuration for the others only include a flag specific to the sensitive datum being modified. Listing 6.13 only modifies the location information and this is specified in the `JSON` configuration for the run.

The `LOCATION` textual flag name is used as a lookup in a dictionary that maps to the value 0x00000020. Thus when considering events logged for a given app, the flag is used to ensure that only data transmissions occurring *after* the location is accessed will be considered.

```
{
    "isBaseline" : false,
    "OutputFileName" : "$RESULTS_DIRECTORY/modifiedlocation.json",
    "Flags" : [
        "LOCATION"
    ],
    "Properties" : [
        ...
        {
            "name" : "Location",
            "value" : {
                "Latitude" : 47.6704449,
                "Longitude" : -122.1187072,
                "Altitude" : 17
            }
        }
        ...
    ],
    ...
}
```

Listing 6.13: Portion of location leak detection configuration showing the flags to check for this iteration

Unlike other work in this area [72] [17], this plug-in does not specifically target `HTTP` or `HTTPS` traffic, instead it analyses any data transmissions that leave the device. While `HTTP` traffic is prevalent among Android apps, it is not the only option.

6.6 Evaluation

As mentioned earlier, the approach taking in this chapter uses differential analysis over multiple executions of a given app.

6.6.1 Experiment Setup

The author used an extended version of the framework mentioned in [78]. This creates a new Android Virtual Device (AVD) with the requested Android version and then starts the emulator. Once boot is complete, two apps are installed, one is a tool which is used to prepare the system between runs and the second is the app that is being analysed.

The preparation tool (preptool) is used to simplify the management of contacts and SMS messages on the system. While contacts can be added using `adb` the author has found the performance less than optimal. While launching the preparation tool (preptool) is simple, providing it with the data for a given run requires some innovative thinking.

Since CopperDroid v2 is invoked for all system calls in all processes on Android, the author added code in the preptool to open and read a non-existent file named `rules.json` from its data directory. Using the CopperDroid v2 Viewer, the author determined the path to the file in question and added support to the plug-in to redirect the file read to the host. The plug-in handles the `open` , `lseek` (to obtain the file length) as well as the `read` and `close` calls. This provides a simple means of passing different configuration information to the preparation tool for each iteration without requiring any changes to the guest image.

```
...
"AddFile" : [
    {
        "target" : "/data/data/com.cd2.preptool/files/rules.json",
        "redirected" : "/test/plugin/infoleak/data/rules.default.json"
    }
]
...
```

Listing 6.14: Configuration for redirecting reads to the host for the preptool configuration

Prior to every iteration (including the initial baseline), the preptool cleans up state by deleting SMS messages and contacts and then adding any contacts or SMS messages specified in the configuration file. Each iteration has its own configuration and the file is redirected into the guest only when the preptool attempts to open it when starting each time. This enables the execution of multiple distinct configurations without requiring a new emulator instance.

```
{
    "DeleteExistingContacts" : true,
    "DeleteExistingSms" : true,
    "Contacts" : [
        {
            "Name" : "Donald Duck",
            "PhoneNumber" : "425-882-8000"
        },
        {
            "Name" : "Daffy Duck",
            "PhoneNumber" : "206-218-1000"
        },
        {
            "Name" : "Goofy",
            "PhoneNumber" : "206-517-1000"
        }
    ]
}
```

Listing 6.15: Default rules

The framework utilised for the experiments included in the [78] paper runs only a single instance of the emulator at the time. Therefore the experiments were executed in many virtual machines (across multiple servers) each running a copy of Ubuntu and running one CopperDroid v2 emulator per virtual machine.

The experiments carried out here, however, were run in a single Ubuntu virtual machine hosted on an 8-core Intel Core i7-7820x with 128GB RAM. The system has 16 logical processors (with hyper-threading) 14 of which were assigned to the VM and up to 120GB RAM available for the VM to use.

The automation framework, as modified by the author, makes it possible to run as many instances of the emulator as the system can handle. The author also implemented an automation tool in `C++` that takes as input a list of all the sample `APKs` to run and launches workers to manage as many instances as requested.

Each worker launches its own instance of the CopperDroid v2 automation framework that creates an AVD, installs the preptool, the sample requested and runs all the required iterations. The automation worker monitors the execution of the framework and once a sample is completed, records the success and requests a new sample, starting a new framework and hence emulator instance. There are situations when the emulator can hang (even an unmodified emulator) and if this occurs (determined through a time-out), the automation worker kills the whole process group and restarts it automatically. The experiments carried out for the purposes of this chapter ran 12 samples concurrently, requiring as much as 120GB RAM at times.

For each sample, the app was run for 10 minutes in each of the iterations (4 baseline + 8 targeted) requiring at least 120 minutes to complete (although a time-out of 180 minutes was specified). Between iterations, the preptool was run which deleted the contacts and SMS messages and then added any specified in the configuration file.

The reason for the 10 minute runtime is to match that used by the AGRIGENTO [17]. However, that work was performed on a device, while this work is on an emulator, with the latter running slower. In general apps can be run automatically through all of their activities in less than 10 minutes, even in an emulator, however, some malware will set an explicit timer for 3 minutes to avoid triggering any malicious actions while being analysed. Thus a 10 minute timer is sufficient for the purposes of this thesis.

While the author would have preferred to run each iteration on a clean VM, not enough time remained prior to the thesis deadline to do so. The author does plan to redirect the data folders of apps to the host in the future for multiple reasons, including the fact that doing so means the plug-in can retain files that the app downloads (and tries to delete after execution), as well as making it easier to clean up state between iterations.

Experiments were run on Android KitKat 4.4.2 and Lollipop 5.1.1. The latter is useful in that the non-secure `Random` class's initial seed can be controlled by returning the same values from the `gettimeofday()` system call.

6.6.2 Datasets

This author used the same datasets as the `AGRIGENTO` [17] paper as that includes the `ReCon` [72] and `BayesDroid` [80] datasets. The datasets include a total of 1030 apps, including 750 apps from the `AppsApk.com` market and other sources, as well as the top 100 free Google Play apps at the time of those papers. Using the same datasets makes it possible to determine the efficacy of the approach followed as compared to those determined in the aforementioned papers.

6.6.3 Identifying Potentially Leaking Apps

As previously mentioned, there is no simple way to control the initial seed of the `Random` class on versions prior to Android Lollipop. However, not controlling this information can lead to false positives but not false negatives. Therefore, the initial run of the experiments was performed against KitKat 4.4.2 to identify apps that had the potential to leak. This is determined by analysing the logs produced for a given app and checking if *any* transmission of data occurred after accessing sensitive information.

Performing this analysis reduced the number to apps needed for the experiments to 338. These apps were then executed on Lollipop 5.1.1 in addition to KitKat and the results of the experiments follow in the Section 6.6.4.

6.6.4 Results

The experiments performed in the manner mentioned in Section 6.6 yielded a total of 195 apps that leaked sensitive information. Based on the results shown in Table 6.1, this falls between the ReCon network flow analysis approach and the differential analysis of network traffic approach of AGRIGENTO. Among the leaks detected, the most prevalent were device ID and subscriber ID, however, SMS messages, location and contacts were also leaked.

There are several potential reasons why the number detected is lower than that discovered in AGRIGENTO. Due to the time constraints, the experiments had to be run for all iterations of a given app without fully resetting the app state, either by creating a new emulator instance or uninstalling/reinstalling the app. Thus if an app stored sensitive information in its storage without querying for it again, it would not have received the information as modified by the plug-in. Furthermore, leakage detection of the Android ID was not included in the experiments as that would have required creating a new emulator instance. Lastly, the stimulation that performed by the CopperDroid v2 automation framework launches the main activity, waits for approximately 30 seconds and then launches the next activity finally returning to the original activity. Although the author used MonkeyRunner to send keystrokes and taps to the app, some paths that result in information leakage may have been missed.

Tool	**Approach**	**#Apps Detected**
FlowDroid	Static taint analysis	44
AppAudit	Static and dynamic taint flow	46
Andrubis/TaintDroid	Dynamic taint analysis	72
ReCon	Network flow analysis	155
Info-leak plug-in to CopperDroid v2	*Differential analysis of network traffic*	*195*
AGRIGENTO	Differential analysis of network traffic	278

Table 6.1: Comparison of different approaches including the plug-in described in this chapter [17]

The info-leak plug-in, developed by the author in just 6 days, still yielded a significant number of leaks, many of which the author confirmed by examining the captured network traffic. Listing 6.16 is an example of network traffic sent in the clear that leaks the device ID as modified by the plug-in. Some apps leaked information using base64 encoding, although the plug-in did not have to decode the data (since it relies on the differences between transmissions), the author did so manually to verify the leaks.

```
POST /bbmf_boss/application_createSession.action HTTP/1.1
Accept: */*
Content-Type: text/html
```

```
User-Agent: Dalvik/1.6.0 (Linux; U; Android 4.4.2; sdk Build/KK)
Host: ads.adlayout.net
Connection: Keep-Alive
Accept-Encoding: gzip
Content-Length: 321

{"screenSize":"480*320","osVersion":"4.4.2","packageName":"com.menueph.util.funtorch","
    deviceModel":"sdk","lng":"0","osType":"Android","existMarket":"0","version":"5","
    timezone":"-5","sdkKey":"D4A3655874","appVersion":"2.8","responseType":"json","
    carrier":"Android","language":"en","lat":"0","deviceId":"091287362661728"}


POST /mobile!install.action HTTP/1.1
Accept: */*
Content-Type: application/x-www-form-urlencoded
Connection: Keep-Alive
User-Agent: Dalvik/1.6.0 (Linux; U; Android 4.4.2; sdk Build/KK)
Host: www.searchmobilestuff.com
Accept-Encoding: gzip
Content-Length: 165

appKey=9adb1850-a666-488a-8ee7-e4bc6de78293&language=en&packageName=com.menueph.util.
    funtorch&deviceModel=sdk&deviceId=091287362661728&appVersion=2.8&carrier=Android
```

Listing 6.16: com.menueph.util.funtorch leaking device ID

In addition to providing an example of information leakage detection using differential analysis, the results of these experiments also demonstrate the ease with which research can be conducted using the plug-in infrastructure that CopperDroid v2 provides.

6.6.5 Record/Replay challenges

As mentioned earlier in Section 6.4, sources of non-determinism are controlled in order to ensure that the only changes in transmitted data occur through the plug-in explicitly modifying a sensitive datum **and** the app being executed leaking that (modified) sensitive datum. Thus, for each run, the values of (otherwise) non-deterministic data need to be controlled.

In general this has no side-effect on the execution of Android or the apps. However, setting the result of the `clock_gettime` system call to the same (unchanging) value every time will hang Android. None of the entropy used by apps relies on this system call, however, the author tested controlling this value for each process/thread and determined that provided the value constantly increases *at any rate* on every query in a given process/thread, Android is unaffected. This was verified using the plug-in described here but starting every thread at the same initial value and then increasing at the same rate, tracking the new values per thread.

6.7 Summary

This chapter described how the various portions of CopperDroid v2 are integrated to provide a system that enables seamless reconstruction of system calls and Binder interfaces and complex objects in real-time. It further described those hand-written classes that assist in bridging the gap between `Java` and `C++`, as well as the boilerplate plug-in code generation. CopperDroid v2 was further evaluated in terms of performance, flexibility and extensibility for use a platform for further research.

Chapter 6 describes a plug-in built on CopperDroid v2 that uses black-box differential analysis to perform information leakage detection of sensitive information through the real-time manipulation of sensitive items.

Chapter 7

Anti Evasion for Malware Analysis

7.1 Introduction

This chapter describes the techniques utilised by Android malware to avoid analysis by anti-malware systems. It further describes a plug-in that modifies otherwise static information that malware accesses in order to determine its execution environment.

Malware and malware detection can be likened to a digital arms race, as one side advances, the other has to advance accordingly to combat the increased threat. Malware has existed in form or another since the early 1980s and has evolved as new platforms have been introduced. Over time malware has also evolved in its attempts to evade detection [79]. The Brain virus is one of the first known instances of evasive malware, hooking the `int 13h` interrupt to redirect disk reads to an uninfected boot sector in order to hide itself [25].

The introduction of software marketplaces, such as the Apple AppStore and the Google Play Store, has made it significantly easier for developers to make their software available for installation. While many legitimate apps are available from these stores, the threat of malware making it onto these stores is ever-present. Examples such as the HummingBad and HummingWhale software are two examples of malware that was accepted onto the Google Play Store but also significatly benefited its authors financially [61] [12].

Given the large number of apps available the various stores and the rate at which they are being developed, it is necessary to scale up the rate at which these can be analysed. While mobile device hardware is becoming increasingly powerful, it still doesn't compare to what can be achieved using traditional desktop systems. The Google Play Store, for example, uses the emulator-based Google Bouncer to analyse apps for potential threats. This allows analysis tools to execute many apps simultaneously on multiple instances of the Android emulator in parallel. As mentioned in Section 6.6, 12 instances of the CopperDroid emulator were run in parallel on an 8-core, 128GB RAM system to ensure the experiments were completed in time. With better financial resources, far more is possible.

There are, however, distinct differences between emulators and actual devices, making it possible for malware to detect an emulated environment and not exhibit any malicious behaviours. These characteristics may be something simple like retrieving and checking the `IMEI` (which is all zeros by default on an emulator) to the timing of memory access with the cache enabled and disabled. Techniques such as the cache-timing differences or processor performance counter analysis, both described in [68], are hard to thwart but they also require expertise to implement correctly. In many instances malware authors may prefer to perform simpler tests to detect an emulated environment.

This chapter does not attempt to solve the issues related to detecting virtualised environments such as those described in [68]. Rather it seeks to demonstrate how the static analysis performed by the CopperDroid v2 compiler coupled with automatically generated plug-in code can be used to further research on Android and in so doing assist in the detection of evasive malware through anti-evasion techniques.

7.2 Evasive Techniques Malware Utilise

Successfully emulating a full system completely is a very difficult task that may never be fully realised. There are a myriad of potential pitfalls that can interfere with doing so, including performance characteristics, device presence or lack thereof, sensor timing and values and a host of others. However, detecting an emulated environment can also, potentially, require a significant amount of work. The more accurately a given system can be accurately emulated, the harder the malware author must work to detect the environment.

As with anything, there are trade-offs that need to be made. Does the malware author spend a significant amount of energy trying to determine processor cache timings of analyse the performance of 2D and 3D graphics? Or can simpler techniques be used, such as readily available device properties? From the perspective of one constructing an emulated environment, can device properties be easily changed in order to force malware authors to work harder to detect the environment?

Prior work in this area has identified several heuristics that malware can use to identify an emulated environment [84] [67] [29]. These can be separated into several categories, namely, static heuristics, dynamic heuristics or virtualisation heuristics.

7.2.1 Static Heuristics

Static properties are those which generally do not change on either a device or in an emulated environment. For instance, the device ID (`IMEI`) remains the same for the lifetime of a device. The SIM card serial number (`ICCID`), however, while static for the most part could change in a real device if the card is replaced. These values, however, generally do not change in an emulator and additionally are initialised to well-known values. For example, the device ID (`IMEI`) is ”000000000000000” in an emulator, but has an actual value on a real device. Similarly the subscriber ID (`IMSI`) has a constant well-known value in the Android emulator but a real value on a device. Additional properties such as the device model, build information, fingerprint or product are also static well-known values, for example *sdk* or *generic*.

Table 7.1 lists the various static values on a given Android system that can be used to determine if the app/malware is executing inside an emulator. These values are obtained from different sources, for example, the `BUILD` values are all present in the `/system/build.prop` file, while the `IMEI` (device ID) and others are obtained from the Telephony class of interfaces. Table 7.2 includes potential values for `Build.HOST` as these can vary widely, however, some are known to be associated with emulators. These constant static values make it easy for malware to detect an emulated environment without much effort.

Network routing information is another area that evasive malware can target as this is fixed in the emulator where addresses are can be in the 10.0.2/24 subnet or 0.0.0.0. Routing information is obtained by reading from `/proc/net/tcp`.

7.2.2 Dynamic Heuristics

Android devices have various sensors such as accelerometers, gyroscopes, GPS, gravity and others. These sensors provide dynamic data based on movement or orientation of a device. The `SensorManager` is responsible for exposing these sensors to apps. Apps register with the `SensorManager` for callbacks related to specific sensor and are then receive calls to their `onSensorChanged()` method. In an actual device the frequency and timing of sensor updates can vary, however, simulated sensors tend to have a regular period (with very minor discrepancies) [67]. Thus, malware can utilise this knowledge to detect simulated sensors and hence an emulated environment.

API Method	**Value**	**Meaning**
Build.ABI	armeabi	Likely an emulator
Build.ABI2	unknown	Likely an emulator
Build.BOARD	unknown	Emulator
Build.BRAND	generic	Emulator
Build.DEVICE	generic	Emulator
Build.FINGERPRINT	contains: generic, sdk etc.	Emulator
Build.HARDWARE	goldfish	Emulator
Build.HOST	Various	See Table 7.2
Build.ID	FRF91	Emulator
Build.MANUFACTURER	unknown	Emulator
Build.MODEL	sdk	Emulator
Build.PRODUCT	sdk	Emulator
Build.RADIO	unknown	Emulator
Build.SERIAL	null	Emulator
Build.TAGS	test-keys	Emulator
Build.USER	android-build	Emulator
TelephonyManager.getDeviceId()	000000000000000	Emulator
TelephonyManager.getLine1Number()	155552155xx	Emulator
TelephonyManager.getNetworkCountryIso()	us	Possible Emulator
TelephonyManager.getNetworkType()	3	Possible Emulator (EDGE network)
TelephonyManager.getNetworkOperator().substring(0,3)	310	Possible Emulator or USA device
TelephonyManager.getNetworkOperator().substring(3)	260	Possible Emulator or T-Mobile USA device
TelephonyManager.getPhoneType()	1	Possible Emulator or GSM phone
TelephonyManager.getSimCountryIso()	us	Possible Emulator
TelephonyManager.getSimSerialNumber()	89014103211118510720	Emulator
TelephonyManager.getSubscriberId()	310260000000000	Emulator
TelephonyManager.getVoiceMailNumber()	15552175049	Emulator

Table 7.1: Static values that malware can query to detect an emulated environment [84]

Device	**Build.HOST**
Emulator	apa27.mtv.corp.google.com
Emulator	android-test-15.mtv.corp.google.com
Emulator	android-test-13.mtv.corp.google.com
Emulator	android-test-25.mtv.corp.google.com
Emulator	android-test-26.mtv.corp.google.com
Emulator	vpbs30.mtv.corp.google.com
Emulator	vpak21.mtv.corp.google.com
Motorola Droid	android-test-10.mtv.corp.google.com
HTC EVO 4G	AA137
Samsung Charge	SEI-26
Samsung Galaxy Tab7	SEP-40
Samsung Galaxy Nexus	vpak26.mtv.corp.google.com

Table 7.2: Build.Host values for different environments [84]

7.2.3 Virtualisation Heuristics

Physical devices execute code using the native instruction set of their CPU. Although a bytecode instruction set such as `Java` is translated into native instructions, this is still faster than an additional level of translation. Emulators, however, may need to translate code between instruction sets, in addition to any translation that occurs due to a bytecode based instruction set. Therefore, the performance characteristics between a device and an emulator are likely to differ significantly. Furthermore, emulators often take short cuts to improve performance, such as not updating the the virtual program counter on every instruction [67]. While such information is harder to access, since it likely involves writing native code, it is still an avenue that malware can utilise to detect an emulated environment.

7.3 Anti-Evasion built upon CopperDroid v2's plug-in infrastructure

For the purposes of this thesis, this chapter will focus on the static values as mentioned in the previous section. These represent the low-hanging fruit that malware authors can use with relative ease to determine if their app is being run within an emulator. Changing all these values by hand would require significant effort and malware could also fingerprint the environment during execution and pass information back to the author enabling them to detect the modified environment in the future. Therefore, a good solution would not only simplify changing the values for the otherwise static information, doing so should require a minimum of effort. Reducing the effort in modifying these values ensures that they can change frequently to avoid fingerprinting by malware.

Detecting the emulator via sensors and virtualisation heuristics requires significantly more work and while certainly feasible, it is likely that malware authors will utilise a simpler approach until that is no longer possible.

The information leakage plug-in described in Chapter 6 provides mechanisms to modify many of the common characteristics that evasive malware use in their attempt to evade analysis in an emulated environment. In addition to the `IMEI` and `IMSI`, evasive malware may also rely on device specific details, such Model, Manufacturer, Build Identifier and the IP tables (to check for a 10.0.2.x IP addresses) [84], [67], [29] to determine the presence of an emulated environment. The device specific details are stored in a `/system/build.prop` and `/system/vendor/build.prop` files (the former is always present, the latter varies based on the device in question). During boot Android reads these files and make the properties contained therein available via the android.os.Build.<value> property.

The author therefore added the ability to redirect existing files to the host (in addition to adding files as was done for the preparation tool). By doing so, the plug-in is able to provide a modified `build.prop` with values that do not match the emulated environment without having to modify the image. This functionality is also used (at least for the short term) to return network routing information that does not match an emulator. The author obtained this by capturing the output of a Linux system into a file and then redirecting the read there. These are shown in Listing 7.1. The example shown is for KitKat, however, the author also has `build.prop` files for Lollipop and others.

The author extended the info-leak plug-in rather than writing a second one for two reasons, firstly, almost all the necessary functionality for modifying static values is already present in the info-leak plug-in. Secondly, doing so simplifies combining information leakage detection with anti-evasion, making it possible to detect information leakage in apps that attempt to evade analysis through emulator detection. A set of experiments was carried out on the 1030 samples used for the information leakage detection, however, after analysing the data and performing a second set of experiments targeting anti-evasion only (without information leakage), the author determined that none of the apps in question were employing any evasion techniques.

```
...
"ReplaceFile" : [
    {
        "target" : "/system/build.prop",
        "redirected" : "/test/plugin/infoleak/data/kitkat.build.prop"
    },
    {
        "target" : "/system/vendor/build.prop",
        "redirected" : "/test/plugin/infoleak/data/kitkat.build.prop"
    },
    {
        "target" : "/proc/net/tcp",
        "redirected" : "/test/plugin/infoleak/data/tcp"
    }
],
...
```

Listing 7.1: Support for redirecting existing files to the host for modifying static information

As mentioned in Section 6.3, configuration information is passed to the plug-in in the form of a `JSON` file. Chapter 6 described the different properties that are manipulated, including device ID, subscriber ID, phone number, voicemail number etc. Since many of these are used by malware to detect emulators, changing these parameters can be achieved simply by specifying the appropriate entry in the configuration. The network type is included as an additional value and sets the network type to `LTE` (Listing 7.2). The operator and other values are not as critical as they are also valid on a physical device (the author has a T-Mobile USA phone).

The location information is included so that the properties (if queried) provide realistic data, for example, the co-ordinates (including the altitude are realistic) and the speed is sufficiently slow (but non-zero) that querying multiple times is unlikely to change the value.

```
    ...
    "Properties" : [
        {
            "name" : "DeviceId",
            "value" : "736356289373622"
        },
        {
            "name" : "SubscriberId",
            "value" : "820164381256781"
        },
        {
            "name" : "PhoneNumber",
            "value" : "+14258001111"
        },
        {
            "name" : "VoiceMailNumber",
            "value" : "+14258788811"
        },
        {
            "name" : "SimCardSerialNumber",
            "value" : "98104012312215801702"
        },
        {
            "name" : "NetworkType",
            "value" : 13
        },
        {
            "name" : "Location",
            "value" : {
                "Provider" : "gps",
                "Latitude" : 47.6704449,
                "Longitude" : -122.1187072,
                "Altitude" : 207.0,
                "Speed" : 0.001,
                "Bearing" : 72.134,
                "Accuracy" : 11.125,
                "IsFromMockProvider" : false
            }
        },
    ...
```

Listing 7.2: Plug-in configuration modifying all the necessary properties accessed through the Telephony Manager

The author modified the plug-in to add support for modifying the network type as returned from the `ITelephony.getNetworkType()` method including in Listing 7.3. The value specified in the `JSON` configuration indicates an `LTE` network.

```
def GetNetworkTypeReply(Task, Object):

    # Valid methods for Task
    # GetProcessIdentifier
    # GetThreadIdentifier
    # GetProcessName

    # Valid methods for Object of type: CTelephonyGetNetworkTypeReply
    # GetResult()
    # SetResult(ResultValue)
    ...

    if packageinfo.IsTrackedPackage(Task.GetProcessName()) or packageinfo.IsTrackedProcess(
         Task.GetProcessIdentifier()):
        present, networkType = config.GetConfig().GetProperty("NetworkType")

        if present != False:
            Object.SetResult(networkType)
```

Listing 7.3: ITelephony getNetworkType method handler in the plug-in for modifying the network type

The author modified KitKat, Lollipop and other version's `build.prop` files using information obtained from different sources to match those of actual devices. For example, information from a Marshmallow device's file was use to modify the KitKat file shown in Listing 7.4.

```
# begin build properties
# autogenerated by buildinfo.sh
ro.build.id=KK442
ro.build.display.id=pixel 4.4.2 KK442 3462041 release
ro.build.version.incremental=3462041
ro.build.version.sdk=19
ro.build.version.codename=REL
ro.build.version.release=4.4.2
ro.build.date=Thu Nov 10 05:52:02 UTC 2016
ro.build.date.utc=1478757122
ro.build.type=user
ro.build.user=dailybuild
ro.build.host=server006
ro.build.tags=release-keys
ro.product.model=Pixel
ro.product.brand=v2
ro.product.name=Pixel
ro.product.device=Pixel
ro.product.board=pxlv2
ro.product.cpu.abi=armeabi-v7a
ro.product.cpu.abi2=armeabi
ro.product.manufacturer=Google
ro.product.locale.language=en
ro.product.locale.region=US
ro.wifi.channels=
ro.board.platform=pxlv2
# ro.build.product is obsolete; use ro.product.device
ro.build.product=Pixel
# Do not try to parse ro.build.description or .fingerprint
ro.build.description=pixel-user 4.4.2 KK442 3462041 release-keys
ro.build.fingerprint=Google/pixel/pixel:4.4.2/KK442/3462041:user/release-keys
ro.build.characteristics=default
# end build properties
```

Listing 7.4: Modified KitKat `build.prop` made to resemble that of an actual device

While the combination of the `JSON` configuration to the plug-in and the redirection of the `build.prop` file in the emulator handles almost all of the properties that evasive malware can potentially use to detect the emulator environment, one still remains. The code, which the author found in [29] and included here for convenience in Listing 7.5, is from an actual malware sample Android/Pincer.A. The first portion of the if statement is referring to the network provider name obtained by calling the `TelephonyManager.getNetworkOperatorName()` method. The author has worked to determine how this value is propagated up to the app (since it is set in the modem), but it requires additional research.

```
    if (str3.toLowerCase().equals("android") || str1.equals("000000000000000")) || (str1.
         equals("012345678912345")) || (str2.equals("15555215554")) || Build.MODEL.
         toLowerCase().equals("sdk")) || (Build.MODEL.toLowerCase().equals("generic")))
        com.security.cert.b.a.a.a(paramContext, true);
    else
    {
        str8 = "false";
        continue;
    }
```

Listing 7.5: Android/Pincer.A malware emulator detection source

The author therefore implements a single line change to the emulator's modem (shown in Listing 7.6) in order to change the provider to something that malware would not recognise as an emulator. The author does plan to make it possible to change this via configuration as opposed to manually in the future. With this change in place (the author verified the API now returns the modified value), the malware anti-analysis detection in Listing 7.5, will not detect the emulator and will execute its payload. The author uses the T-Mobile network name since other properties listed in Table 7.1 refer to the device being on the T-Mobile network and the incongruence resulting in choosing another network or inventing a new one could be used for emulator detection.

```
#define  OPERATOR_HOME_NAME  "T-Mobile" // Previously: "Android"
```

Listing 7.6: Change to android_modem.c to return a different network name

Listing 7.7 is obtained directly from a copy of the Dendroid malware source [81]. The malware checks multiple `BUILD` properties to determine if it is executing in an emulator and if not, it executes its payload. Since all of the values shown below clearly do not match those provided in the `build.prop` substituted through the plug-in, this malware will also execute its payload.

```
if("google_sdk".equals(Build.PRODUCT ) || "google_sdk".equals(Build.MODEL) ||
    Build.BRAND.startsWith("generic") || Build.DEVICE.startsWith("generic")
    || "goldfish".equals(Build.HARDWARE))
{
}
else
{
    PreferenceManager.getDefaultSharedPreferences(getApplicationContext()).edit().putBoolean(
        "Start", true);
    initiate();
}
```

Listing 7.7: Dendroid Emulator Detection Source

In all the samples run thus far by the author, none exhibited evasive techniques, even though clearly malware does exist that attempts to evade detection in emulated environments. Mobile malware is still relatively new as compared to desktop malware and thus it is likely that as the platform matures, so too will malware advance to avoid detection, likely leading to an increase in evasive malware instances. The author developed a test program to implement the two methods for determining the presence of an emulator as per the two malware samples (Android/Pincer.A and Dendroid). In both cases, the emulator was not detected when anti-evasion was enabled, but was detected if it wasn't enabled.

7.4 Summary

In this chapter the author describes enhancements to the information leakage plug-in to easily mask the static properties of the Android emulator to support anti-evasion. Although requiring very little additional code and some configuration information, implementing the evasion routines from two evasive malware samples resulted in the heuristic determining that it was not running in an emulator.

Chapter 8

Discussion

8.1 Introduction

This chapter compares Copperdroid v1 with CopperDroid v2 in addition to discussing different approaches that could be taken with respect to the utilising the model proposed in this thesis. As mentioned before, the model establishes a clear relationship between Binder interfaces and Android objects and makes it possible to determine only those portions specifically related to Binder ICC. While the author has chosen to utilise slicing and transformation of the `Java` ASTs to automatically generate `C++` code, other implementations are possible with their own benefits and trade-offs.

8.2 CopperDroid v1 versus CopperDroid v2

8.2.1 CopperDroid v1

As mentioned earlier in this thesis, CopperDroid v1 demonstrates that a system call only based approach to app/malware analysis is viable, however, it requires the assistance of an unmarshaling Oracle for reconstructing Binder ICC and Android objects. Furthermore, the performance of CopperDroid v1 and the unmarshaling Oracle limit the usability across Android versions. In [78], Tam et al. describe the performance of CopperDroid v1, including the fact that system wide analysis incurs more than a 200% performance overhead. While the targeted analysis has a lower overhead (20-35%), it has the potential to miss actions that may be important. In the context of CopperDroid v1, targeted means only certain *interesting* system calls. The problem with focusing on specific system calls and excluding others is that side-effects of calls not included could potentially enable exploitation of the system. For instance, the exploitation of speculative execution on `x86` processors in the SPECTRE and MELTDOWN malware to access kernel memory is an example of an unexpected side-effect albeit on CPUs [46] [56]. Similarly an otherwise innocuous system call could potentially be exploited to perform an execution of privilege or other attack. Thus it is imperative that any approach aimed at app/malware analysis not be limited by its design such that certain operations cannot be analysed.

CopperDroid v1 only supports execution on Android Froyo version 2.2.3 with the performance characteristics limiting adoption of later versions. The design and implementation of CopperDroid v1 requires significant modification of the Android emulator as well as a second unmodified emulator instance for hosting the unmarshaling Oracle, with communication with the occurring via TCP/IP. While the basic system call support in CopperDroid v1 is mostly automatically generated (with several hand-written portions including `ioctl` code for Binder) and the AIDL-based Binder interfaces, there is a significant amount of hand-written code to enable reconstruction of parts of the core Binder service interfaces `IActivityManager` and `IServiceManager`. Furthermore, CopperDroid v1 is designed specifically for use with the Android emulator, without considering other potential execution options. Additionally, the design is not extensible and therefore doesn't provide a mechanism through which further research can be conducted.

8.2.2 CopperDroid v2

CopperDroid v2, by contrast, analyses *every system call across all processes* with only a 7-12.22% overhead even when fully reconstructing Binder ICC. All version specific code in CopperDroid v2 is automatically generated, making adoption of a new Android version relatively simple. As described in Chapter 4, the automatically generated code reconstructs Binder ICC and Android objects with minimal overhead (on the order of nanoseconds) as opposed to the order of seconds as required by the unmarshaling Oracle. Furthermore, CopperDroid v2 is itself extensible through plug-ins which are given full access to reconstructed system call and Binder interfaces and Android objects. The aforementioned plug-ins can analyse and manipulate system calls, Binder interfaces and Android objects. Table 8.1 compares the characteristics of CopperDroid v1 and CopperDroid v2. While not explicitly included, one additional feature is the possibility of execution in other environments. CopperDroid v2 is designed with flexibility of the execution environment in mind and is thus implemented using the abstract plug-in interface. However, as mentioned in Chapter **??**, other options are possible including the use of Berkeley Packet Filters, described in the following section.

CopperDroid Version	Android Versions	New Version Cost	AIDL interfaces	Java interfaces	Real-time	Modify in-vivo	Extensible
CopperDroid v1	Froyo 2.2.3	2+ weeks	✓	✗	✗	✗	✗
CopperDroid v2	Froyo 2.2.3 → Oreo 8.0.0	1 hour	✓	✓	✓	✓	✓

Table 8.1: Comparison of qualitative properties of CopperDroid v1 versus v2

CopperDroid Version	System call overhead	Binder Reconstruction Overhead
CopperDroid v1	0.0910ms	>1 second (including Oracle)
CopperDroid v1	0.0001ms	10μs

Table 8.2: Performance comparison between CopperDroid v1 versus v2

Table 8.2 shows the micro-benchmark performance of different system calls and the reconstruction of Binder ICC. The system call processing overhead for CopperDroid v1 is obtained from [78] while the Binder overhead is obtained from timing tests carried out by the author of this thesis using CopperDroid v1. The system call overhead of CopperDroid v2 is calculated across *all* system calls in a given execution of CopperDroid v2 using a geometric mean. This includes any reads or writes occurring to/from the file system and the associated copying of such data for logging purposes. CopperDroid v1, however, only inspects certain *interesting* system calls and thus the performance information provided in [78] may not provide an accurate assessment.

As a qualitative example, using the Android Emulator's Graphic User Interface (GUI) is simply not possible with CopperDroid v1, even with the limited system call capture it performs. However, the Android Emulator instance is fully usable in the case of CopperDroid v2, even though every single system call is analysed in real-time.

In the case of the Binder ICC performance tests, only calls with complex objects are included as the processing of simple types for CopperDroid v2 only registers in nanoseconds (and is indistinguishable from the margin of error). While the table includes the processing time for the Oracle (in the case of CopperDroid v1), it should be noted that the Oracle is not invoked during execution of the emulator, but rather during post-processing (due to the overhead incurred by accessing the Oracle). Therefore, the performance cost of the Oracle is included to illustrate the difference between the model followed in the Oracle versus that provided through the static analysis approach taken in this thesis.

8.3 Supporting On-Device Execution

As described in Chapter 3.2, CopperDroid v2 has modular design, with the core functionality being implemented as a plug-in to an abstract interface. While one of the goals of this design is to limit the modification required to support a new emulator version, a second goal is to enable the execution of the CopperDroid v2 in other environments, including on a device. The Linux kernel does not support C++ code, therefore, the current implementation of the plug-in manager and the CopperDroid v2 plug-in would likely need to execute in a user-mode process and trace all system calls (for example via ptrace). However, this mode of execution would add additional overhead in analysing apps, however, this approach does not expose the kernel to attacks through carefully crafted Parcels.

Another potential approach is to leverage the Berkeley Packet Filter [59] support provided in Android [37] to enable on device analysis of Android apps/malware. As described in Chapter 4, static analysis is utilised to determine the precise relationship between Binder interfaces, calls and Android objects. This model makes it possible to extract only the portions of the Android framework source code directly related to Binder ICC. The design described in Chapter 4 constructs an abstract syntax tree which is then analysed by visitor classes to extract the necessary code for reconstructing Binder interfaces and Android objects. The current visitor classes maintain a similar code structure to the original transforming those portions necessary to a support C++ implementation, however, this is only one potential solution. The visitor paradigm described here applies to building on the support already provided by the CopperDroid v2 compiler, however, any approach that follows the model described in this thesis can achieve the same ends. The key insight, that only Binder interfaces and their intersection with Android objects is required to reconstruct Binder ICC, is all that is needed to achieve this goal. The system dependency graph generated through the static analysis makes it possible to isolate those portions directly involved in Binder and determine how Parcels are accessed for a given interface or object.

By constructing a higher level model that determines the relationship between interfaces and objects and Parcels, one can automatically generate Berkeley Packet Filters for the various interfaces and Android objects. For example, in some cases one value in a parcel determines how another is interpreted, thus the data read from the Parcel must take this into account. One example is to check the presence of an object, for example, if a complex object can be null, then a boolean (uint32) is included that is set to 1 if the object is present or 0, if it is not. Arrays serialized into a parcel are similarly handled, generally there is a boolean to determine presence, followed by a count indicating the number of items followed by a packed array of the items. Thus, in order to support Berkeley Packet Filters a visitor class can construct a control flow graph and data dependency graph and utilising the two in tandem, construct the precise semantics of a given interface and it's interaction with Parcels and Android objects. While the system dependency graph constructed by CopperDroid v2 determines which portions of a given object are to be included in the object, based on its intersection with interfaces, the control flow graph as these relate to `Parcels` is required to automatically construct the analogous Berkeley Packet Filter code.

As mentioned in Chapter 3.2, the core Binder protocol is largely unchanged over the last 10 years from a code perspective and is identical in terms implementation for 32-bit systems. The only modification has been to add support for 64-bit pointers (for protocol version 8, 32-bit support still uses protocol version 7). Therefore, implementing Berkeley Packet Filter support would require parsing this protocol much like CopperDroid v2 so today, albeit in `C` as opposed to `C++`.

There are certain advantages in leveraging the Berkeley Packet Filters in Android as this permits real-time analysis directly from the kernel, avoiding a process context switch or kernel to user transition. However, this has its own risks, for instance, malware could potentially craft a `Parcel` to exploit the kernel analysis code and potentially elevate its own privilege. Furthermore, any deep anti-malware analysis would likely be executing in user-mode to reduce the kernel footprint, since any related databases and machine learning libraries would probably not support kernel mode execution.

As mentioned in Chapter 1, the author also considered developing a language which captured the relationships between interfaces, objects and `Parcels`. Such a language would contain the data and control flow as it pertained to a specific interface method and `Parcel`. Listing 8.1 shows a mock-up of a potential language for capturing the semantics of the `getIntentSender` of the `IActivityManager` interfaces, including the flow control where one `Parcel` member is dependent on another.

```
{
    type <- int,
    packageName <- String,
    token <- Binder,
    resultWho <- String,
    requestCode <- int,
    {
        _ <- int != 0 ? requestIntents <- array(Intent),
                        requestResolvedTypes <- array(String)
                      : requestIntents <- null,
                        requestResolvedTypes <- null
    },
    fl <- int,
    {
        _ <- int != 0 ? options <- Bundle : options <- null
    },
    userId <- int
}
```

Listing 8.1: Mock-up of language describing relationship between interface method and `Parcel` for the getIntentSender method call

It should be noted that this is simply a mock-up, although the author has considered multiple approaches to this problem and this captures the semantics. However, `Intent` and `Bundle` need to be defined, although this can be achieved through the system dependency graph described in Chapter 4 combined with analysis of the methods invoked in the respective objects.

```
{
    noException -> Exception,
    res -> Binder
}
```

Listing 8.2: Mock-up of language describing relationship between interface method and `Parcel` for the getIntentSender method reply

Similarly, Listing 8.2 describes the `Parcel` structure included in the response to the getIntentSender method call. By constructing all of the client/server unmarshaling/-marshaling semantics, real-time analysis and manipulation could be achieved through the generation of the appropriate code for a given target. Furthermore, this language could also be translated into intermediate representation such as LLVM IR [50] in order to simplify construction of other solution.

Chapter 9

Conclusions and Future Work

Mobile malware is an ever present threat that has managed, on multiple occasions, to bypass the protections put in place to hold them at bay and make it onto mobile marketplaces [61] [12] [28] [47]. Combating this threat requires a solution that is able to scale to meet the increasing number of apps, while enabling new research with ease. The solution should also be agnostic of the Android version and runtime.

The model described in this thesis leverages static analysis to determine only the components necessary for Binder ICC. CopperDroid v2 as described in this thesis implements the model described to automatically generate per-version system call and Binder reconstruction code. In so doing, it meets the criteria described above in that it enables version and runtime agnostic reconstruction and manipulation of system calls, Binder ICC and Android objects in real-time.

By automatically generating boilerplate plug-in code that is able to observe and modify reconstructed system calls and Binder interfaces and object, further analyses and research can be built with ease in either `Python` or `C++`.

Through micro-benchmarks the author established that reconstruction and modification of complex Binder objects requires only micro-seconds (μs). `Python` plug-ins add little overhead while `C++` plug-ins are almost undetectable in their performance overhead (within the margin of error).

Experiments show that CopperDroid v2 outperformed the state-of-the-art both in terms of accuracy and diversity of reconstructed behaviors, and incurred overhead, paving the way to realistic dynamic analysis of Android apps.

The author evaluated the performance and effectiveness of CopperDroid v2 for real-time analysis and modification as well as enabling further research scenarios. CopperDroid v2 has already been used in the analysis of 10000+ app samples and the author believes that further research can be built using CopperDroid v2 plug-in infrastructure, thereby taking advantage of its real-time reconstruction and modification capabilities.

The author demonstrates the reconstruction and modification capabilities of CopperDroid v2 through the information leakage plug-in. The plug-in was developed end-to-end in just 6 days, highlighting the ease with which research can be developed on top of the platform provided by CopperDroid v2. The information leakage detection plug-in seamlessly modifies system call data and return values as well as Binder objects to implement a differential analysis based detection algorithm. The author further demonstrates combating evasive malware by modifying static properties of the system in real-time with minor additions to the information leakage plug-in.

9.1 Future Work

As described in Chapter 8, CopperDroid v2 is designed as a plug-in to an abstract interface. This design is intended to permit execution in other environments, including on a device. However, another approach is to extend the CopperDroid v2 compiler using an AST visitor to produce Berkeley Packet Filter code for use with the Android support for Berkeley Packet Filters.

The author intends to further explore the creation of a language that expresses the semantics of the relationship between Binder interfaces, methods and Android objects, with the goal that reconstruction and manipulation code can be automatically generated for a given scenario, including on device execution. Furthermore, such a language may also enable further research that is yet to be considered.

Although Android malware currently appear to be using simple techniques to detect emulated environments, as anti-evasion improves, so too will malware's anti-analysis. Therefore, with the relatively recent introduction of C++ plug-in support the author plans to implement an on device version of CopperDroid v2 enabling additional research on actual devices while simultaneously alleviating the need for anti-evasion.

With the ability to modify system calls and Binder operations with ease, a fuzzing system can be constructed on top of CopperDroid v2 to allow security testing of apps and services.

Appendix A

Detailed Source Listings

A.1 CopperDroid v2 Configuration File Format

```
//
// This file contains the definition for the Configuration files used for
// CopperDroid.
//
// The Configuration "Message" is really just a container in the format
// that Google Protobufs use. Protobufs support both a binary and a text
// format and the data contained within this protobuf definition is used
// in a textual format as configuration.
//

message Configuration {

    //
    // The version information is to make it clear what a given config file
    // is targetting.
    //

    message AndroidVersion {
        required string Name = 1;
        required string Version = 2;
        required string Path = 3;
    }

    //
    // The TargetSystem assists in describing which architecture a given
    // configuration file is targetting as well as other values relevant
    // to the system (for example kernel stack size).
    //

    message TargetSystem {
        enum ARCHITECTURE {
            ARM = 0;
            x86 = 1;
            x86_64 = 2;
            ARM64 = 3;
        }

        optional ARCHITECTURE Architecture = 1 [default = ARM];
        optional int32 KernelStackSize = 2 [default = 8192];
        optional int32 MessageHeaderLength = 3 [default = 28];
        optional int32 IoVectorStructureSize = 4 [default = 8];
        optional int32 PointerSize = 5 [default = 4];
    }

    //
    // The kernel offsets are to allow control of the offsets within the kernel
    // data structures to match a given system, without requiring multiple
    // binaries depending on the given emulation target OS. While these values
    // have been effectively unchanged for some time, they can differ between
    // processor architectures. By making these configurable via a file means
    // that the same code base can support different target versions without
    // requiring a rebuild; a different configuration file is sufficient.
    //

    message KernelOffsets {
        optional int32 OFFSET_TASK = 1 [default = 12];
        optional int32 OFFSET_COMM = 2 [default = 724];
        optional int32 OFFSET_THREADID = 3 [default = 492]; // This is really the PID in the
             kernel but is actually the thread ID in Linux.
        optional int32 OFFSET_PROCESSID = 4 [default = 496]; // This is really the TGID in
            the kernel but is actually the process ID in Linux.
        optional int32 OFFSET_MM = 5 [default = 456];
        optional int32 OFFSET_MM_START_STACK = 6 [default = 144];
        optional int32 OFFSET_MM_ARG_START = 7 [default = 148];
        optional int32 OFFSET_MM_ARG_END = 8 [default = 152];
        optional int32 OFFSET_MM_ENV_START = 9 [default = 156];
        optional int32 OFFSET_MM_ENV_END = 10 [default = 160];
        optional int32 OFFSET_MSGHDR_NAME = 11 [default = 0];
        optional int32 OFFSET_MSGHDR_NAME_LEN = 12 [default = 4];
        optional int32 OFFSET_MSGHDR_IOVEC = 13 [default = 8];
        optional int32 OFFSET_MSGHDR_IOVEC_LEN = 14 [default = 12];
        optional int32 OFFSET_IOVEC_BASE = 15 [default = 0];
        optional int32 OFFSET_IOVEC_LEN = 16 [default = 4];
    }

    message AutoSaveThresholds {
        optional uint32 TimeThreshold = 1 [default = 30];
        optional uint32 SizeThreshold = 2 [default = 104857600];
    }

    message PlugIn {
        enum PLUG_IN_TYPE {
            PYTHON = 0;
            CPP = 1;
        }

        required bool InvokePlugIn = 1 [default = false];
        optional PLUG_IN_TYPE PlugInType = 2 [default = PYTHON];
        optional string PlugInPath = 3 [default = "."];
        optional string ModuleName = 4 [default = ""];
    }

    message Benchmark {
        required bool EnableBenchmarks = 1 [default = false];
        optional bool BenchmarkSystemCalls = 2 [default = false];
        optional bool BenchmarkBinderNoReconstruction = 3 [default = false];
        optional bool BenchmarkBinder = 4 [default = false];
        optional bool BenchmarkPlugIn = 5 [default = false];
    }

    required AndroidVersion Version = 1;
    optional TargetSystem System = 2;
    optional KernelOffsets Offsets = 3;
    optional AutoSaveThresholds Thresholds = 4;
    optional PlugIn PlugInInformation = 5;
    optional Benchmark BenchmarkSettings = 6;
    optional bool CaptureFromStartup = 7 [default = false];
}
```

Listing A.1: Google Protobuf definition file for CopperDroid v2 to support multiple versions with a single core plug-in.

A.2 CopperDroid v2 Source Listings

A.2.1 Chapter 3 Source Code

```
class IRegisters {
public:
    virtual const long int r(REGISTER Register, int banked_mode) = 0;
    virtual const long int PageDirectoryBase() = 0;
    virtual void set_r(REGISTER Register, long int Value) = 0;
};

class IPlugInContext {
public:
    virtual IRegisters* GetRegisters() = 0;
    virtual bool ReadMemory(long int TargetAddress, void* Buffer, unsigned long Length) = 0;
    virtual bool WriteMemory(long int TargetAddress, void* Buffer, unsigned long Length) = 0;
    virtual bool ReadString(long int TargetAddress, char** String) = 0;
    virtual void FreeString(char* String) = 0;
    virtual bool ReadInteger32(long int TargetAddress, int32_t* Value) = 0;
    virtual bool ReadInteger64(long int TargetAddress, int64_t* Value) = 0;
    virtual bool ReadUnsignedInteger32(long int TargetAddress, uint32_t* Value) = 0;
    virtual bool ReadUnsignedInteger64(long int TargetAddress, uint64_t* Value) = 0;
    virtual uint64_t GetUniqueCpuIdentifier() = 0;
};
```

Listing A.2: Interface for accessing guest memory and state.

```
class IPlugInManager {
public:
    virtual bool RegisterPlugin(const char* Name, IPlugInEvents* Events) = 0;
    virtual bool IsTargetBigEndian() = 0;
};
```

Listing A.3: The IPluginManager interface which a plug-in invokes to register itself

```
//
// Function(s) which plug-in must support:
//   InitializePlugin (signature is show below as InitializePluginFn).
//

typedef bool (*InitializePluginFn)(IPlugInManager* Manager, const char* Parameters, const
     char* Port, const char* PlugInConfigFileName);
```

Listing A.4: Method which a plug-in must have in order to be initialised after being loaded

```
class IPlugInEvents {
public:
    virtual void EnterSyscallEvent(IPlugInContext* Context, ARCHITECTURE Architecture, int
         SyscallNumber) = 0;
    virtual void ExitSyscallEvent(IPlugInContext* Context, ARCHITECTURE Architecture) = 0;
};
```

Listing A.5: An Interface that plug-ins implement for system call entry/exit.

```
Version
{
    Name: "KitKat"
    Version: "4.4.2"
    Path : "plugin/copperdroid/objs/kitkat4_4_2.so"
}

System
{
    Architecture:ARM
}

Offsets
{
    OFFSET_THREADID: 504
    OFFSET_PROCESSID: 508
    OFFSET_MM : 468
    OFFSET_MM_ARG_START: 144
    OFFSET_MM_ARG_END: 148
}

PlugInInformation
{
    InvokePlugIn: true
    PlugInPath : "plugin/infoleak/python/"
    ModuleName: "copperdroidplugin"
}
```

Listing A.6: Configuration file for Android KitKat 4.4.2

```
...
extern "C" bool InitializePlatform(std::string VersionString, GuestConfiguration*
     Configuration, std::string PlugInConfigFileName, ISysCalls** SystemCalls)
{
    IPlugIn* PlugIn = nullptr;

    SetConfiguration(Configuration);

    if (Configuration->IsPlugInPresent()) {
        if (Configuration->GetPlugInType() == PYTHON) {
            PlugIn = InitializePythonPlugIn(Configuration->GetPlugInPath(), Configuration->
                 GetPlugInModuleName(), PlugInConfigFileName);

        } else {
            InitializePlugInFn InitializePlugIn;
            void* handle;
            string libraryName = Configuration->GetPlugInPath() + "/" + Configuration->
                 GetPlugInModuleName();

            handle = dlopen(libraryName.c_str(), RTLD_LAZY);
            if (handle == nullptr) {
                printf("Error: Unable to load plug-in. %s\n", dlerror());
                return false;
            }

            InitializePlugIn = (InitializePlugInFn) dlsym(handle, "InitializeCppPlugIn");
            if (InitializePlugIn == nullptr) {
                printf("Error locating InitializePlugIn routine: %s\n", dlerror());
                return false;
            }

            PlugIn = InitializePlugIn(PlugInConfigFileName);
        }
        systemCalls.SetPlugIn(PlugIn);
    }

    *SystemCalls = &systemCalls;

    return true;
}
```

Listing A.7: Initialization routine for initializing a given platform such as KitKat or Nougat.

```
typedef SystemCallEvent* (*SyscallEntryStubFunction)(ISysCalls* This, SystemInformation*
     System, TaskInformation* Task, IPlugInContext* Context, IPlugIn* PlugIn);
typedef void (*SyscallExitStubFunction)(ISysCalls* This, SystemInformation* System,
     TaskInformation* Task, IPlugInContext* Context, SystemCallEvent* event, IPlugIn* PlugIn)
     ;

class ISysCalls
{
public:
    virtual bool IsTrackedSystemCall(const int SysCallNumber) = 0;
    virtual SystemCallEvent* CallEntryStub(ISysCalls* ThisPointer, SystemInformation* System,
          IPlugInContext* Context, const int SyscallNumber) = 0;
    virtual void CallExitStub(ISysCalls* thisPointer, SystemInformation* System,
         IPlugInContext* Context, const int SyscallNumber, SystemCallEvent* Event) = 0;
    virtual IInterfaceDataExtractor* GetInterfaceExtractor() = 0;
    virtual void SetPackageName(string PackageName) = 0;
    virtual void UninitializePlugIn() = 0;
    virtual void UpdateScenario(string ScenarioName) = 0;
    virtual void SerializePlugIn() = 0;
};
```

Listing A.8: ISysCalls interface for version agnostic system call inspection.

```
def ParseSyscalls(FileName, UnistdPath):

    parser = SysCallsTxtParser()
    parser.parse_file(FileName)

    syscallNumberMapping = scanunistd.scan_linux_unistd_h(UnistdPath)

    syscalls ={}

    for syscall in parser.syscalls:
        if syscall['armid'] == 1 or syscall['common'] == 1:
            allvalues = syscall['decl'].split()
            returntype = allvalues[0]

            if '*' in allvalues[1]:
                returntype += allvalues[1]

                if '*' in allvalues[2]:
                    returntype += allvalues[2]

            syscallEntry = [returntype, syscall["name"], syscall["params"]]
            syscalls[syscallNumberMapping[syscall["name"]]] = syscallEntry

    parser = None

    return syscalls
```

Listing A.9: ParseSysCalls which uses the bionic utilities SysCallsTxtParser and unistd.h scanner to produce the system call table

```
class ICopyFunctor {
public:
    virtual bool operator()(void* Context, byte* Destination, const byte* Source,
        size_t Length) = 0;
};
```

Listing A.10: Abstract interface for a functor that can be passed to a function allowing copy to or from a guest

```
class CopyFromGuest : public ICopyFunctor {
public:
    bool operator()(void* Context, byte* Destination, const byte* Source, size_t Length)
    {
        IPlugInContext* PlugInContext = (IPlugInContext*)Context;

        return PlugInContext->ReadMemory((long int) Source, (void*) Destination, Length);
    }
};

class CopyToGuest : public ICopyFunctor {
public:
    bool operator()(void* Context, byte* Destination, const byte* Source, size_t Length)
    {
        IPlugInContext* PlugInContext = (IPlugInContext*)Context;

        return PlugInContext->WriteMemory((long int) Destination, (void*) Source, Length);
    }
};
```

Listing A.11: Functor implementations that perform a copy from or to a guest

```
class bytes
{
public:
    ...
    bool SetData(const void* Data, size_t Length, ICopyFunctor& CopyData, void* CopyContext);
    {
        byte* Bytes = nullptr;

        try {
            Bytes = new byte[Length];

        } catch (std::bad_alloc) {
            return false;
        }

        if (m_Buffer != nullptr) {
            delete[] m_Buffer;
            m_Buffer = nullptr;
        }

        if (CopyData(CopyContext, Bytes, (const byte*) Data, Length) == false) {
            delete[] Bytes;

            return false;
        }

        m_DataAddress = (void*)Data;
        m_Buffer = Bytes;
        m_BufferLength = Length;
        m_Offset = 0;

        return true;
    }

    ...
    bool WriteToAddress(byte* DataAddress, ICopyFunctor& CopyData, void* CopyContext);
    {
        m_DataAddress = DataAddress;

        return CopyData(CopyContext, DataAddress, (const byte*) m_Buffer, m_BufferLength);
    }
    ...
};
```

Listing A.12: Excerpt of `bytes` class illustrating copying data to/from guest

```
...
static SystemCallEvent* open_stub_enter_syscall(ISysCalls* This, SystemInformation* System,
     TaskInformation* Task, IPlugInContext* Context, IPlugIn* PlugIn)
{
    ...
    SystemCallEvent* event = new SystemCallEvent;
    Parameters* parameters = new Parameters;
    Parameter* parameter0 = new Parameter;
    Parameter* parameter1 = new Parameter;
    Parameter* parameter2 = new Parameter;
    KernelObject* object = nullptr;
    struct timeval timeStamp;
    char* TempBuffer;

    gettimeofday(&timeStamp, nullptr);
    event->SetTimeStamp(timeStamp);

    event->CaptureParameterRegisters(Context->GetRegisters()->r(r0,0), Context->GetRegisters
         ()->r(r1,0), Context->GetRegisters()->r(r2,0));
    if (Context->ReadString(Context->GetRegisters()->r(r0,0), &TempBuffer) != false) {
        object = ObjectManager::CreateObject(string(TempBuffer));
        parameter0->SetValue((const char*)TempBuffer);
        Context->FreeString(TempBuffer);
    }

    parameter1->SetValue((int)Context->GetRegisters()->r(r1,0));
    parameter2->SetValue((mode_t)Context->GetRegisters()->r(r2,0));
    parameters->Add(parameter0);
    parameters->Add(parameter1);
    parameters->Add(parameter2);

    event->SetParameters(parameters);
    if (object != nullptr) {
        event->SetKernelObject(object);
        if (object->GetObjectType().GetType() == TypeBinder) {
            BinderObject* binderObject = (BinderObject*)object;
            binderObject->SetInterfaceExtractor(This->GetInterfaceExtractor());
        }
    }

    if (PlugIn != nullptr) {
        CSystemCallOpen* systemCallOpen = new CSystemCallOpen(true, event, (parameters->
             GetParameter(0)->GetValue()->Value.string_value != nullptr) ? *parameters->
             GetParameter(0)->GetValue()->Value.string_value : "", parameter1->GetValue()->
             Value.int32_value, parameter2->GetValue()->Value.int32_value);

        event->SetPlugInData(systemCallOpen);
        PlugIn->InvokePlugInForSystemCall(true, Task, "open", systemCallOpen);

        if (systemCallOpen->IsModified() != false) {
            systemCallOpen->Update(Context);
        }
    }

    return event;
}
...
```

Listing A.13: Entry stub for the `open` system call

```
class CSystemCallOpen : public ISystemCallPlugInData
{
public:
    CSystemCallOpen(bool IsEntry, SystemCallEvent* SystemCallEventObject, string PathName,
         int32_t Flags, int32_t Mode) : mModified(false), mSystemCallEvent(
         SystemCallEventObject), mIsEntry(IsEntry), mPathName(PathName), mFlags(Flags), mMode
         (Mode) {}

    void SetPathName(string PathName)
    {
        mPathName = PathName;
        mModified = true;
    }

    string GetPathName()
    {
        return mPathName;
    }
    ...
    void SetReturnValue(int32_t ReturnValue)
    {
        mReturnValue = ReturnValue;
        mModified = true;
    }

    int32_t GetReturnValue()
    {
        return mReturnValue;
    }

    bool IsModified()
    {
        return mModified;
    }
    ...
    void Update(IPlugInContext* PlugInContext)
    {
        CopyToGuest copyToGuest;
        bytes data;

        data = mPathName;
        data.WriteToAddress((byte*)mSystemCallEvent->GetParameterRegister(0), copyToGuest, (
             void*)PlugInContext);

        PlugInContext->GetRegisters()->set_r(r1, mFlags);
        PlugInContext->GetRegisters()->set_r(r2, mMode);

        if (mIsEntry == false) {
            PlugInContext->GetRegisters()->set_r(r0, mReturnValue);
        }
    }

private:
    bool mIsEntry;
    string mPathName;
    int32_t mFlags;
    int32_t mMode;
    int32_t mReturnValue;
    bool mModified;
    SystemCallEvent* mSystemCallEvent;
};
```

Listing A.14: Class that encapulates the `open` system call enabling `Python` and `C++` plug-ins to inspect and alter parameters

```
class KernelObject
{
public:
    virtual ~KernelObject(){};
    virtual ObjectType GetObjectType() = 0;
    virtual bool IoCtl(SystemInformation* System, TaskInformation* Task, IPlugInContext*
         Context, uint64_t IoCtlCode, uint64_t BufferAddress, IPlugIn* PlugIn) = 0;
    virtual bool IoCtlExit(SystemInformation* System, TaskInformation* Task, IPlugInContext*
         Context, IEventData** EventData, IPlugIn* PlugIn) = 0;
};
```

Listing A.15: KernelObject base class

```
bool BinderObject::ExtractData(SystemInformation* System, TaskInformation* Task,
    IPlugInContext* Context, BinderEventData* EventData, bytes& DataBuffer, bool Entry,
    IPlugIn* PlugIn)
{
    bool result;
    BinderOperation* binderOperation;

    while (DataBuffer.LengthRemaining() != 0) {
        size_t bufferOffset;
        BinderOperation* pendingOperation = nullptr;

        bufferOffset = DataBuffer.GetOffset();
        binderOperation = new BinderOperation((BINDER_OPERATION_TYPE)DataBuffer.
            ReadUnsignedInteger32(), bufferOffset, Task, PlugIn);

        if (binderOperation->IsReply()) {
            pendingOperation = GetPendingOperation(Task->GetThreadIdentifier());
        }

        if (binderOperation->ExtractOperationData(DataBuffer, Context,
            m_InterfaceDataExtractor, pendingOperation) != false) {
            EventData->AddOperation(binderOperation, Entry);

            if (binderOperation->PendingReply()) {

                //
                // Reference the operation.
                //

                binderOperation->Reference();
                AddPendingOperation(Task->GetThreadIdentifier(), binderOperation);
            }
        }

        if (pendingOperation != nullptr) {
            pendingOperation->Dereference();
        }
    }

    return true;
}
```

Listing A.16: BinderObject tracking transactions and replies

```
message SysCallEvent {
    enum SYSCALLNUMBER {
        ...
        SysCallEnumIoctl = 54;
        ...
    }
    ...
    message SysCallIoctl {
        enum IOCTL_TYPE {
            IoCtlGeneral = 1;
            IoCtlBinderFlag = 0x00100000;
            IoCtlBinderWriteRead = 0x00100001;
            ...
        }
        enum BINDER_OPERATION_TYPE {
            BC_TRANSACTION = 1;
            BC_REPLY = 2;
            ...
            BR_TRANSACTION = 20;
            BR_REPLY = 21;
            ...
        }
        enum BINDER_TRANSACTION_FLAGS {
            TF_ONE_WAY = 0x01;
            ...
        }
        enum FLAT_BINDER_OBJECT_TYPE {
            BINDER_TYPE_BINDER = 0x73622a85;
            BINDER_TYPE_WEAK_BINDER = 0x77622a85;
            BINDER_TYPE_HANDLE = 0x73682a85;
            ...
        }
        ...
        message BINDER_WRITE_READ_DATA {
            optional uint64 write_size = 1;
            optional uint64 write_consumed = 2;
            optional bytes write_buffer = 3;
            optional uint64 read_size = 4;
            optional uint64 read_consumed = 5;
            optional bytes read_buffer = 6;
        }
        message BINDER_FLAT_OBJECT {
            required FLAT_BINDER_OBJECT_TYPE type = 1;
            optional int32 flags = 2;
            optional bytes write_buffer = 3;
            optional uint64 read_size = 4;
            optional uint64 read_consumed = 5;
            optional bytes read_buffer = 6;
        }
        message BINDER_TRANSACTION_DATA {
            optional uint64 target = 1;
            optional uint64 cookie = 2;
            optional uint32 code = 3;
            optional uint32 flags = 4;
            ...
            optional uint64 dataSize  = 7;
            optional uint64 offsetsSize = 8;
            optional bytes buffer = 9;
            optional bytes offsets = 10;
            ...
        }
        message BINDER_REFERENCE_DATA {
            required uint64 address = 1;
            optional bytes data = 2;
        }
        message BINDER_TRANSACTION {
            required BINDER_TRANSACTION_DATA transactiondata = 1;
            optional bytes parcel = 2;
            ...
            optional bytes interfaceToken = 4;
            ...
            repeated BINDER_REFERENCE_DATA binderstrongreferencedata = 9;
        }
        message BINDER_OPERATION {
            required BINDER_OPERATION_TYPE type = 1;
            optional BINDER_TRANSACTION transaction = 2;
            ...
        }
        ...
        optional IOCTL_TYPE ioctltype = 5;
        optional BINDER_WRITE_READ_DATA binderEntryWriteRead = 6;
        optional BINDER_WRITE_READ_DATA binderExitWriteRead = 7;
        repeated BINDER_OPERATION entryBinderOperation = 8;
        repeated BINDER_OPERATION exitBinderOperation = 9;
        ...
    }
}
```

Listing A.17: `ioctl` serialisation structure with Binder support

A.2.2 Chapter 4 Source Code Listings

```
class CActivityManagerExtractor : public IDataExtractor
{
public:
    STATUS BuildTransactionData(int code, Parcel* data, BinderTransaction* binderTransaction)
    {
        switch (code) {
            ...
            case GET_INTENT_SENDER_TRANSACTION:
            {
                int type = data->readInt();
                javaString packageName = data->readString();
                BinderReference* token = data->readStrongBinder();
                javaString resultWho = data->readString();
                int requestCode = data->readInt();
                TypedObjectArray<Intent*> requestIntents;
                TypedArray<javaString> requestResolvedTypes;

                if (data->readInt() != 0) {
                    requestIntents = data->createTypedArray<Intent>();
                    requestResolvedTypes = data->createStringArray();
                } else {
                    requestIntents = nullptr;
                    requestResolvedTypes = nullptr;
                }

                int fl = data->readInt();
                Bundle* options = data->readInt() != 0 ? Bundle::createFromParcel(data) :
                    nullptr;
                int userId = data->readInt();

                CActivityManagerGetIntentSenderTransaction*
                    activityManagerGetIntentSenderTransaction = new
                    CActivityManagerGetIntentSenderTransaction(type, packageName, token,
                    resultWho, requestCode, requestIntents, requestResolvedTypes, fl,
                    options, userId);

                binderTransaction->SetPlugInTransactionData(
                    activityManagerGetIntentSenderTransaction);
                ...
            }
            ...
        }
        ...
    }
    ...
};
```

Listing A.18: IActivityManager extractor showing the getIntentSender transaction unmarshaling code

```
class CActivityManagerGetIntentSenderTransaction : public IInterfacePlugInTransactionData
{
public:
    CActivityManagerGetIntentSenderTransaction(int type, javaString packageName, BinderReference*
         token, javaString resultWho, int requestCode, TypedObjectArray<Intent*> intents, TypedArray
         <javaString> resolvedTypes, int flags, Bundle* options, int userId) : m_Transaction("
         GET_INTENT_SENDER_TRANSACTION"), mType(type), mPackageName(packageName), mToken(token),
         mResultWho(resultWho), mRequestCode(requestCode), mIntents(intents), mResolvedTypes(
         resolvedTypes), mFlags(flags), mOptions(options), mUserId(userId)
    {
    }
    ...
    void WriteToParcel(Parcel* data)
    {
        data->writeInterfaceToken("android.app.IActivityManager");
        data->writeInt(mType);
        data->writeString(mPackageName);
        data->writeStrongBinder(mToken);
        data->writeString(mResultWho);
        data->writeInt(mRequestCode);
        if (mIntents != nullptr) {
            data->writeInt(1);
            data->writeTypedArray(mIntents, 0);
            data->writeStringArray(mResolvedTypes);
        } else {
            data->writeInt(0);
        }
        data->writeInt(mFlags);
        if (mOptions != nullptr) {
            data->writeInt(1);
            mOptions->writeToParcel(data, 0);
        } else {
            data->writeInt(0);
        }
        data->writeInt(mUserId);
    }
    ...
    bool IsModified()
    {
        return m_Modified;
    }
    ...
    virtual ~CActivityManagerGetIntentSenderTransaction()
    {
        if (mToken != nullptr) {
            delete mToken;
            mToken = nullptr;
        }
        if (mOptions != nullptr) {
            delete mOptions;
            mOptions = nullptr;
        }
    }
};
```

Listing A.19: Generated class for getIntentSender transaction handling permitting modification through plug-ins (code omitted for brevity).

```
class CActivityManagerExtractor : public IDataExtractor
{
public:
    ...
    STATUS BuildReplyData(int code, Parcel* reply, BinderTransaction* binderReply)
    {
        switch (code) {
            ...
            case GET_INTENT_SENDER_TRANSACTION:
            {
                int exceptionCode = reply->readException();

                if (exceptionCode == 0U) {
                    BinderReference* res = reply->readStrongBinder();

                    CActivityManagerGetIntentSenderReply* activityManagerGetIntentSenderReply
                         = new CActivityManagerGetIntentSenderReply(res);

                    binderReply->SetPlugInTransactionData(activityManagerGetIntentSenderReply
                        );
                    ...
                } else {
                    javaString exceptionMessage = reply->readString();

                    CActivityManagerGetIntentSenderReply* activityManagerGetIntentSenderReply
                         = new CActivityManagerGetIntentSenderReply(nullptr, exceptionCode,
                        exceptionMessage);

                    binderReply->SetPlugInTransactionData(activityManagerGetIntentSenderReply
                        );
                }
            }
            ...
        }
        ...
    }
    ...
};
```

Listing A.20: IActivityManager extractor showing the getIntentSender reply (code omitted for brevity).

```
class CActivityManagerGetIntentSenderReply : public IInterfacePlugInTransactionData
{
public:
    CActivityManagerGetIntentSenderReply(BinderReference* res, int ExceptionCode = 0U,
        javaString ExceptionMessage = "") : m_Transaction("GET_INTENT_SENDER_TRANSACTION"),
        mRes(res), mExceptionCode(ExceptionCode), mExceptionMessage(ExceptionMessage)
    {
    }

    ...

    void WriteToParcel(Parcel* reply)
    {
        if (mExceptionCode == 0U) {
            reply->writeNoException();
            reply->writeStrongBinder(mRes != nullptr ? mRes : nullptr);
        } else {
            reply->writeInt(mExceptionCode);
            reply->writeString(mExceptionMessage);
        }
    }
    ...
    bool IsModified()
    {
        return m_Modified;
    }

    BinderReference* GetRes()
    {
        return mRes;
    }

    void SetRes(BinderReference* ResValue)
    {
        mRes = ResValue;
        m_Modified = true;
    }

    int GetExceptionCode()
    {
        return mExceptionCode;
    }

    void SetExceptionCode(int ExceptionCodeValue)
    {
        mExceptionCode = ExceptionCodeValue;
        m_Modified = true;
    }
    ...
};
```

Listing A.21: Generated class for getIntentSender reply handling permitting modification through plug-ins (code omitted for brevity).

```
void CInterfaceTranslator::ExtractCase(CCaseStatement* CaseStatement, CCaseStatement**
    TransactionCaseStatement, CCaseStatement** ReplyCaseStatement)
{
    CFormatType formatType;
    CClass* transactionObjectClass;
    CObjectBuilder* transactionObjectBuilder;
    CClass* replyObjectClass;
    CObjectBuilder* replyObjectBuilder;
    string transactionLabel;
    string transactionMethodName;

    for (CVariable* parameter : m_CurrentMethod->GetParameters()->GetParameters()) {
        if (formatType.IsObject(parameter->GetType()->GetName())) {
            parameter->GetType()->SetVariableType(VariableTypePointer);
        }
    }

    CBinderRewriter binderRewriter(m_CurrentMethod->GetParameters(), &m_InterfaceMethodMap,
        &m_ImplementationMethodMap, this, m_InterfaceName, m_InterfaceDescriptor,
        m_PythonInterfaceDocument, m_PythonPlugInModuleName);

    binderRewriter.BuildExtractionCase(CaseStatement, TransactionCaseStatement,
        ReplyCaseStatement, &transactionObjectClass, &transactionObjectBuilder,
        &replyObjectClass, &replyObjectBuilder, &transactionLabel, &transactionMethodName);

    ASSERT(transactionObjectClass != nullptr, "Transaction class can't be null\n");
    ASSERT(replyObjectClass != nullptr, "Reply class can't be null\n");

    m_TransactionObjectClasses.push_back(transactionObjectClass);

    m_ReplyObjectClasses.push_back(replyObjectClass);

    if (m_CurrentPass == BuildExtractorPass) {
        AnalyzeEdges(m_CurrentMethod->GetParameters(), transactionObjectClass);
        AnalyzeEdges(m_CurrentMethod->GetParameters(), replyObjectClass);
        AnalyzeEdges(m_CurrentMethod->GetParameters(), *TransactionCaseStatement);
        AnalyzeEdges(m_CurrentMethod->GetParameters(), *ReplyCaseStatement);
    }

    transactionMethodName[0] = (char)toupper(transactionMethodName[0]);

    m_TransactionToMethodPairs.push_back(make_pair(transactionLabel, transactionMethodName));
    m_TransactionObjectBuilders.push_back(transactionObjectBuilder);
    m_ReplyObjectBuilders.push_back(replyObjectBuilder);
}
```

Listing A.22: CInterfaceTranslator::ExtractCase method which analyses a specific transaction case statement and produces the necessary code

```
void CInterfaceTranslator::AnalyzeEdges(CParameters* Parameters, CASTNode* Node)
{
    CEdgeAnalyzer edgeAnalyzer(Parameters, m_TypeDatabase);

    Node->Accept(&edgeAnalyzer, nullptr);
    edgeAnalyzer.SetPass(LocateEdges);

    Node->Accept(&edgeAnalyzer, nullptr);
    for (int nodeId : edgeAnalyzer.GetNodesAccessed()) {
        InsertAccessedNode(nodeId);
    }
}
```

Listing A.23: CInterfaceTranslator::AnalyzeEdges method

```
void CInterfaceTranslator::Commit()
{
    CLabelStatement* label;
    vector<int> startingNodes;

    m_TypeDatabase->BeginTransaction();

    for (pair<int, bool> node : m_TraversedNodesMap) {
        startingNodes.push_back(node.first);
    }

    for (int nodeId : startingNodes) {
        TraverseEdges(nodeId);
    }

    for (pair<int, bool> node : m_TraversedNodesMap) {
        m_TypeDatabase->MarkNodeAccessed(node.first);
    }

    m_TypeDatabase->CommitTransaction();

    for (CClass* transactionObjectClass : m_TransactionObjectClasses) {
        transactionObjectClass->Write(m_IndentedOutput);
        m_IndentedOutput->Print("\n");
    }

    for (CClass* replyObjectClass : m_ReplyObjectClasses) {
        replyObjectClass->Write(m_IndentedOutput);
        m_IndentedOutput->Print("\n");
    }

    label = new CLabelStatement;
    label->SetLabel("private");
    m_ClassStatementBlock->Insert(label);

    for (CConstant* constant : m_InterfaceConstants) {
        m_ClassStatementBlock->Insert(constant);
    }

    m_Class->Write(m_IndentedOutput);

    m_IndentedOutput->Print("\n");
    m_IndentedOutput->ApplyIndentation(); // Auto-indent code for readability.
    m_IndentedOutput->Write(); // Output the source file for the current interface.
    ... // Python and C++ plug-in code generation.
}
```

Listing A.24: CInterfaceTranslator::Commit method for generating interface marshaling code (plug-in code generation excluded for brevity)

```
class Location : public Parcelable
{
public:
    Location(javaString provider);
    ...
    virtual string AsString();
    virtual bool InstanceOf(uint64_t ClassInstanceId);
    ...
    double GetLatitude();
    void SetLatitude(double LatitudeValue);
    double GetLongitude();
    void SetLongitude(double LongitudeValue);
    ...
    double mLatitude;
    double mLongitude;
    ...
};
```

Listing A.25: Android Location Object class declaration

```
...
double Location::GetLatitude()
{
    return mLatitude;
}

void Location::SetLatitude(double LatitudeValue)
{
    mLatitude = LatitudeValue;
}

double Location::GetLongitude()
{
    return mLongitude;
}

void Location::SetLongitude(double LongitudeValue)
{
    mLongitude = LongitudeValue;
}

string Location::AsString()
{
    StringFormat objectString;

    objectString.Format("{ \"mProvider\" : \"%s\", \"mTime\" : %ld, \"mElapsedRealtimeNanos\"
         : %ld, \"mLatitude\" : %f, \"mLongitude\" : %f, \"mHasAltitude\" : %s, \"mAltitude
        \" : %f, \"mHasSpeed\" : %s, \"mSpeed\" : %f, \"mHasBearing\" : %s, \"mBearing\" : %
        f, \"mHasAccuracy\" : %s, \"mAccuracy\" : %f, \"mExtras\" : %s, \"
        mIsFromMockProvider\" : %s }", mProvider.c_str(), mTime, mElapsedRealtimeNanos,
        mLatitude, mLongitude, CBoolean(mHasAltitude).AsString().c_str(), mAltitude,
        CBoolean(mHasSpeed).AsString().c_str(), mSpeed, CBoolean(mHasBearing).AsString().
        c_str(), mBearing, CBoolean(mHasAccuracy).AsString().c_str(), mAccuracy, (mExtras !=
         nullptr) ? mExtras->AsString().c_str() : "null", CBoolean(mIsFromMockProvider).
        AsString().c_str());

    return objectString.GetData();
}
...
```

Listing A.26: Android Location Object class implementation

```
public enum SupplicantState implements Parcelable {
    DISCONNECTED,
    ...
    ASSOCIATED,
    ...
    UNINITIALIZED,

    public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(name());
    }

    public static final Creator<SupplicantState> CREATOR =
        new Creator<SupplicantState>() {
            public SupplicantState createFromParcel(Parcel in) {
                return SupplicantState.valueOf(in.readString());
            }

            public SupplicantState[] newArray(int size) {
                return new SupplicantState[size];
            }
        };
}
```

Listing A.27: `SupplicantState Java` enum with methods

```
class SupplicantState : public Parcelable
{
public:
    enum  {
        DISCONNECTED,
        ...
        ASSOCIATED,
        ...
        UNINITIALIZED,
        ...
    };
    ...
    static int valueOf(javaString EnumName)
    {
        if (EnumName == "DISCONNECTED") {
            return DISCONNECTED;
        }
        ...
        if (EnumName == "ASSOCIATED") {
            return ASSOCIATED;
        }
        ...
        if (EnumName == "UNINITIALIZED") {
            return UNINITIALIZED;
        }
        ...
        return -1;
    }
    javaString name()
    {
        switch (mValue) {
            case DISCONNECTED:
            {
                return "DISCONNECTED";
            }
            ...
            case ASSOCIATED:
            {
                return "ASSOCIATED";
            }
            ...
            case UNINITIALIZED:
            {
                return "UNINITIALIZED";
            }
        }
        return "";
    }
    ...
    void writeToParcel(Parcel* dest, int flags)
    {
        dest->writeString(name());
    }
    static SupplicantState createFromParcel(Parcel* in)
    {
        return SupplicantState::valueOf(in->readString());
    }

    int mValue;
};
```

Listing A.28: Equivalent `SupplicantState` C++ class with enum and code

```
CMethod* CObjectBuilder::BuildValueOfImplicitMethod(CEnumConstants* EnumConstants)
{
    CMethod* method = new CMethod();
    CStatementBlock* statementBlock = new CStatementBlock;
    CParameters* parameters = new CParameters;

    method->SetName("valueOf");
    method->SetModifier(new CModifier(TOKEN_STATIC));

    CVariable* variable = new CVariable(new CType("javaString"), "EnumName");

    parameters->AddParameter(variable);
    method->SetParameters(parameters);

    for (CEnumConstant* enumConstant : EnumConstants->GetConstants()) {
        CIfStatement* ifStatement;
        CParentheticExpression* parentheticExpression;
        CComparisonExpression* comparisonExpression;
        CStatementBlock* ifStatementBlock = new CStatementBlock;
        CReturnStatement* ifReturnStatement = new CReturnStatement;

        ifStatement = new CIfStatement();
        comparisonExpression = new CComparisonExpression;

        comparisonExpression->SetLeftValue(new CIdentifierExpression("EnumName"));
        comparisonExpression->SetOperator(TOKEN_EQUAL);
        comparisonExpression->SetRightValue(new CStringLiteralExpression(enumConstant->
            GetName()));

        parentheticExpression = new CParentheticExpression;
        parentheticExpression->SetExpression(comparisonExpression);

        ifStatement->SetExpression(parentheticExpression);
        ifReturnStatement->SetExpression(new CIdentifierExpression(enumConstant->GetName()));
        ifStatementBlock->Insert(ifReturnStatement);
        ifStatement->SetStatement(ifStatementBlock);
        statementBlock->Insert(ifStatement);
        statementBlock->Insert(new CBlankLineStatement);
    }

    CReturnStatement* returnStatement = new CReturnStatement;
    returnStatement->SetExpression(new CIntegerLiteralExpression((int64_t)-1));
    statementBlock->Insert(returnStatement);

    method->SetReturnType(new CType("int"));
    method->SetParameters(parameters);
    method->SetStatementBlock(statementBlock);

    return method;
}
```

Listing A.29: Code that builds the valueOf method

```
uint64_t CObjectBuilder::GenerateTypeHash()
{
    string fullyQualifiedName = "";
    uint64_t identifier = SEED_VALUE;

    for (CObjectBuilder* parentObject : m_ParentObjects) {
        fullyQualifiedName += parentObject->m_ClassName;
        fullyQualifiedName += "::";
    }

    fullyQualifiedName += m_ClassName;

    for (size_t round = 0; round < 10; round++) {
        for (size_t index = 0; index < fullyQualifiedName.length(); index++) {
            identifier = 37 * identifier + fullyQualifiedName[index];
        }
    }

    return identifier;
}
```

Listing A.30: CObjectBuilder::GenerateTypeHash() method for generating the class ID for a given class

```
bool ComponentInfo::InstanceOf(uint64_t ClassInstanceId)
{
    return (ClassInstanceId == CLASS_INSTANCE_ID || PackageItemInfo::InstanceOf(
        ClassInstanceId));
}
```

Listing A.31: InstanceOf example for the ComponentInfo class

```
void CObjectTranslator::BuildObjects()
{
    vector<CClass*> classes;
    ...
    OrderClasses();
    ...
    for (CObjectBuilder* objectBuilder : m_FlattenedObjectBuilders) {
        if (objectBuilder->IsReferenced()) {
            BuildObject(objectBuilder);
        }
    }

    // Generate C++ source file

    for (CObjectBuilder* objectBuilder : m_FlattenedObjectBuilders) {
        CClass* objectClass;

        if (objectBuilder->IsReferenced()) {
            objectClass = objectBuilder->BuildClass(true, m_SourceDocument != nullptr);

            if (m_SourceDocument != nullptr) {
                for (CFieldInitialization* fieldInitialization : objectBuilder->
                    GetFieldInitializations()) {
                    m_SourceDocument->Insert(fieldInitialization);
                }

                m_SourceDocument->Insert(new CBlankLineStatement);
                for (CMethodDefinition* methodDefinition : objectBuilder->
                    GetMethodDefinitions()) {
                    m_SourceDocument->Insert(methodDefinition);
                    m_SourceDocument->Insert(new CBlankLineStatement);
                }
            }

            // Construct the Python bindings for the current object class and insert them in
               the AST

            if (m_PythonInterfaceDocument != nullptr) {
                CPythonInterface pythonInterface(m_PythonInterfaceDocument,
                    m_PythonPlugInModuleName);

                pythonInterface.BuildInterface(objectBuilder);
            }

            classes.push_back(objectClass);
        }
    }

    // Generate C++ header file

    m_HeaderDocument->Insert(new CBlankLineStatement);

    for (CClass* currentClass : classes) {
        m_HeaderDocument->Insert(currentClass);
        m_HeaderDocument->Insert(new CBlankLineStatement);
    }

    m_HeaderDocument->Write(m_IndentedHeaderOutput);
    m_IndentedHeaderOutput->ApplyIndentation();
    m_IndentedHeaderOutput->Write();

    if (m_SourceDocument != nullptr) {
        m_SourceDocument->Write(m_IndentedSourceOutput);
        m_IndentedSourceOutput->ApplyIndentation();
        m_IndentedSourceOutput->Write();
    }

    if (m_PythonInterfaceDocument != nullptr) {
        m_PythonInterfaceDocument->Write(m_IndentedPythonInterfaceOutput);
        m_IndentedPythonInterfaceOutput->Write();
    }
}
```

Listing A.32: CObjectTranslator::BuildObjects method for generating the header and source files for a given Java source file

```
class javaString : public string
{
public:
    javaString() : m_IsNull(false), string("");
    javaString(string Value) : m_IsNull(false), string(Value);
    ...
    javaString(nullptr_t NullPointer) : m_IsNull(true), string("");
    ...
    javaString& operator=(string Value);
    javaString& operator=(nullptr_t NullPointer);
    bool operator==(nullptr_t NullPointer);
    bool operator!=(nullptr_t NullPointer);
    ...
    bool operator!=(javaString Other);
    ...
    bool operator==(javaString Other);
    ...
    bool operator==(string StringValue);
    ...
    bool operator!=(string StringValue);
    ...
private:
    bool m_IsNull;
};
```

Listing A.33: Truncated javaString class definition

A.2.3 Chapter 5 Source Code Listings

```
template <class T> class TypedArray : public vector<T> {
public:
    TypedArray() : m_IsNull(false), m_ReservedElementCount(0), vector<T>::vector()
    {
    }
    ...
    virtual ~TypedArray()
    {
    }
    ...
    virtual string AsString()
    {
        string result;
        stringstream stream;
        size_t index = 0;

        stream << "[";

        for (T t : *this) {
            stream << " ";
            stream << t;

            if (++index < this->size()) {
                stream << ", ";
            }

        }

        stream << " ]";

        return stream.str();
    }
    ...
};
```

Listing A.34: TypedArray class

```
template <class T> class TypedObjectArray : public TypedArray<T> {
public:
    TypedObjectArray() : m_IsNull(false)
    {
    }
    ...
    virtual ~TypedObjectArray()
    {
        clear();
    }
    ...
    string AsString()
    {
        string result;
        stringstream stream;
        size_t index = 0;

        stream << "[";

        for (T t : *this) {
            stream << " ";
            stream << t->AsString();

            if (++index < this->size()) {
                stream << ", ";
            }
        }

        stream << " ]";

        return stream.str();
    }
    ...
    void clear()
    {
        while (this->empty() == false) {
            T value = this->back();

            value->Dereference();

            this->pop_back();
        }
    }
    ...
};
```

Listing A.35: TypedObjectArray class

```
PYBIND11_PLUGIN(CopperDroid) {
    module CopperDroid("CopperDroid", "CopperDroid definitions");
    ...
    py::class_<CActivityManagerGetIntentSenderTransaction>(CopperDroid, "
        CActivityManagerGetIntentSenderTransaction")
        .def("__repr__", &CActivityManagerGetIntentSenderTransaction::AsString)
        .def("GetType", &CActivityManagerGetIntentSenderTransaction::GetType)
        .def("SetType", &CActivityManagerGetIntentSenderTransaction::SetType)
        .def("GetPackageName", &CActivityManagerGetIntentSenderTransaction::GetPackageName)
        .def("SetPackageName", &CActivityManagerGetIntentSenderTransaction::SetPackageName)
        .def("GetToken", &CActivityManagerGetIntentSenderTransaction::GetToken,
            return_value_policy::reference)
        ...
        .def("GetIntents", &CActivityManagerGetIntentSenderTransaction::GetIntents)
        .def("SetIntents", &CActivityManagerGetIntentSenderTransaction::SetIntents)
        .def("GetResolvedTypes", &CActivityManagerGetIntentSenderTransaction::
            GetResolvedTypes)
        .def("SetResolvedTypes", &CActivityManagerGetIntentSenderTransaction::
            SetResolvedTypes)
        ...

    py::class_<CActivityManagerGetIntentSenderReply>(CopperDroid, "
        CActivityManagerGetIntentSenderReply")
        .def("__repr__", &CActivityManagerGetIntentSenderReply::AsString)
        .def("GetRes", &CActivityManagerGetIntentSenderReply::GetRes, return_value_policy::
            reference)
        .def("SetRes", &CActivityManagerGetIntentSenderReply::SetRes)
        .def("GetExceptionCode", &CActivityManagerGetIntentSenderReply::GetExceptionCode)
        .def("SetExceptionCode", &CActivityManagerGetIntentSenderReply::SetExceptionCode)
        .def("GetExceptionMessage", &CActivityManagerGetIntentSenderReply::
            GetExceptionMessage)
        .def("SetExceptionMessage", &CActivityManagerGetIntentSenderReply::
            SetExceptionMessage);
        ...

    return CopperDroid.ptr();
}
```

Listing A.36: Interface method generated class pybind11 bindings

```
import CopperDroid
import activitymanagerroutines

Transactions = {
    ...
    "GET_INTENT_SENDER_TRANSACTION" : activitymanagerroutines.GetIntentSenderTransaction,
    ...
    "GET_MEMORY_INFO_TRANSACTION" : activitymanagerroutines.GetMemoryInfoTransaction,
    ...
    }

Replies = {
    ...
    "GET_INTENT_SENDER_TRANSACTION" : activitymanagerroutines.GetIntentSenderReply,
    ...
    "GET_MEMORY_INFO_TRANSACTION" : activitymanagerroutines.GetMemoryInfoReply,
    ...
    }

def Invoke(Task, Type, Method, Object):

    if Type == CopperDroid.Operation.TypeTransaction:
        if Method in Transactions.keys():
            Transactions[Method](Task, Object)

    if Type == CopperDroid.Operation.TypeReply:
        if Method in Replies.keys():
            Replies[Method](Task, Object)
```

Listing A.37: Generated Python plug-in code for invoking IActivityManager methods

```
import CopperDroid
import systemcallplugin
...
import activitymanagerplugin
...

InterfacePlugIns = {
        ...
        'android.app.IActivityManager' : activitymanagerplugin.Invoke,
        ...
    }

def InitializePlugIn(ConfigFileName):
    pass

def SetPackageName(PackageName):
    pass

def UninitializePlugIn():
    pass

def InvokeInterfacePlugIn(Task, InterfaceName, Type, Method, Object):

    if InterfaceName in InterfacePlugIns.keys():
        InterfacePlugIns[InterfaceName](Task, Type, Method, Object)

def InvokeSystemCallPlugIn(Entry, Task, SystemCall, Object):

    systemcallplugin.Invoke(Entry, Task, SystemCall, Object)

def UpdateConfig(ConfigFileName):
    pass

def UninitializePlugIn():
    pass
```

Listing A.38: Generated Python plug-in code

```
import CopperDroid
import activitymanagerroutines

Transactions = {
    ...
    "GET_INTENT_SENDER_TRANSACTION" : activitymanagerroutines.GetIntentSenderTransaction,
    ...
    "GET_MEMORY_INFO_TRANSACTION" : activitymanagerroutines.GetMemoryInfoTransaction,
    ...
    }

Replies = {
    ...
    "GET_INTENT_SENDER_TRANSACTION" : activitymanagerroutines.GetIntentSenderReply,
    ...
    "GET_MEMORY_INFO_TRANSACTION" : activitymanagerroutines.GetMemoryInfoReply,
    ...
    }

def Invoke(Task, Type, Method, Object):

    if Type == CopperDroid.Operation.TypeTransaction:
        if Method in Transactions.keys():
            Transactions[Method](Task, Object)

    if Type == CopperDroid.Operation.TypeReply:
        if Method in Replies.keys():
            Replies[Method](Task, Object)
```

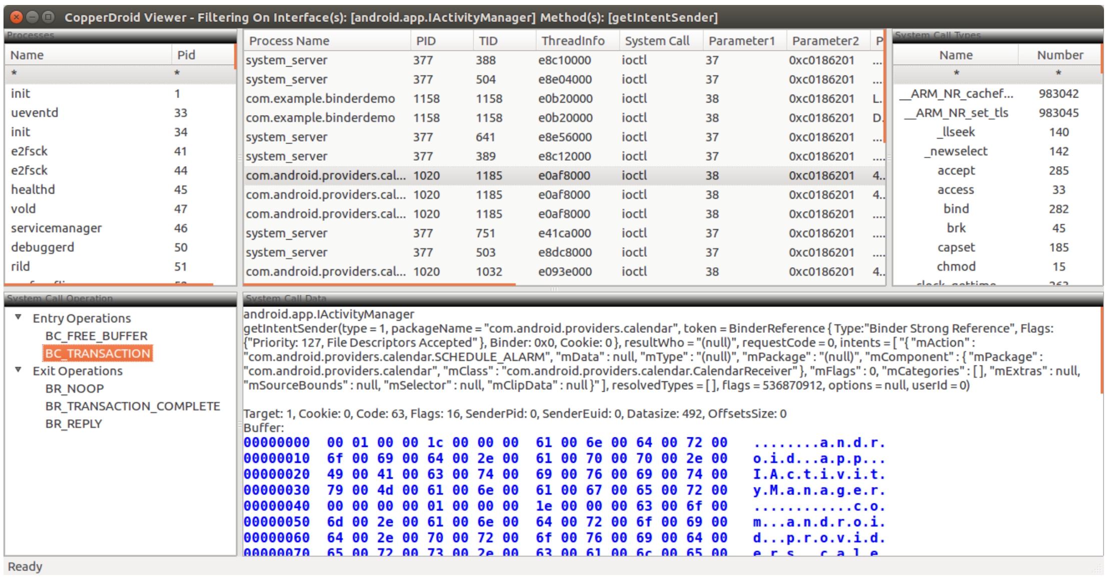
Listing A.39: Generated Python plug-in code for invoking IActivityManager methods

```
import CopperDroid

def GetIntentSenderTransaction(Task, Object):

    # Valid methods for Task
    # GetProcessIdentifier
    # GetThreadIdentifier
    # GetProcessName

    # Valid methods for Object of type: CActivityManagerGetIntentSenderTransaction
    # GetType()
    # SetType(TypeValue)
    # GetPackageName()
    # SetPackageName(PackageNameValue)
    ...
    # GetIntents
    # SetIntents
    # GetResolvedTypes
    # SetResolvedTypes
    ...

    pass

def GetMemoryInfoTransaction(Task, Object):

    # Valid methods for Task
    # GetProcessIdentifier
    # GetThreadIdentifier
    # GetProcessName

    # Valid methods for Object of type: CActivityManagerGetMemoryInfoTransaction

    pass

def GetIntentSenderReply(Task, Object):

    # Valid methods for Task
    # GetProcessIdentifier
    # GetThreadIdentifier
    # GetProcessName

    # Valid methods for Object of type: CActivityManagerGetIntentSenderReply
    # GetRes()
    # SetRes(ResValue)
    ...

    pass

def GetMemoryInfoReply(Task, Object):

    # Valid methods for Task
    # GetProcessIdentifier
    # GetThreadIdentifier
    # GetProcessName

    # Valid methods for Object of type: CActivityManagerGetMemoryInfoReply
    # GetOutInfo()
    # SetOutInfo(OutInfoValue)
    ...

    pass
```

Listing A.40: Generated Python plug-in routines for interface methods

Appendix B

Additional Figures



CopperDroid Viewer - Filtering On Interface(s): [android.app.IActivityManager] Method(s): [getIntentSender]

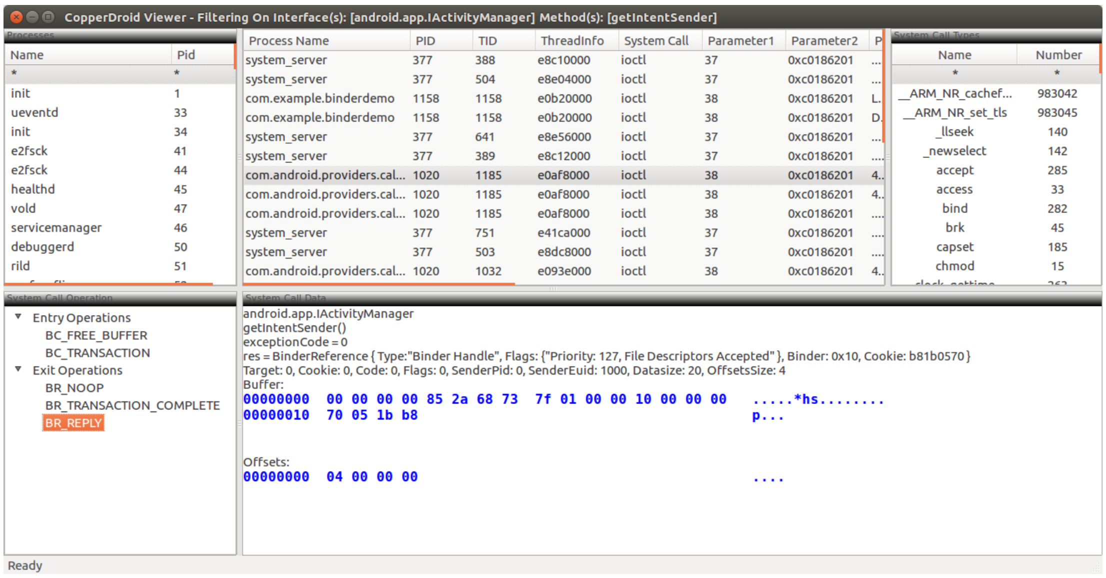
Name	Pid
*	*
init	1
ueventd	33
init	34
e2fsck	41
e2fsck	44
healthd	45
vold	47
servicemanager	46
debuggerd	50
rild	51

Process Name	PID	TID	ThreadInfo	System Call	Parameter1	Parameter2	P
system_server	377	388	e8c10000	ioctl	37	0xc0186201	...
system_server	377	504	e8e04000	ioctl	37	0xc0186201	...
com.example.binderdemo	1158	1158	e0b20000	ioctl	38	0xc0186201	L.
com.example.binderdemo	1158	1158	e0b20000	ioctl	38	0xc0186201	D.
system_server	377	641	e8e56000	ioctl	37	0xc0186201	...
system_server	377	389	e8c12000	ioctl	37	0xc0186201	....
com.android.providers.cal...	1020	1185	e0af8000	ioctl	38	0xc0186201	4..
com.android.providers.cal...	1020	1185	e0af8000	ioctl	38	0xc0186201	4..
com.android.providers.cal...	1020	1185	e0af8000	ioctl	38	0xc0186201	....
system_server	377	751	e41ca000	ioctl	37	0xc0186201	....
system_server	377	503	e8dc8000	ioctl	37	0xc0186201	....
com.android.providers.cal...	1020	1032	e093e000	ioctl	38	0xc0186201	4..

Name	Number
*	*
__ARM_NR_cachef...	983042
__ARM_NR_set_tls	983045
_llseek	140
_newselect	142
accept	285
access	33
bind	282
brk	45
capset	185
chmod	15

```
android.app.IActivityManager
getIntentSender(type = 1, packageName = "com.android.providers.calendar", token = BinderReference { Type:"Binder Strong Reference", Flags:
{"Priority: 127, File Descriptors Accepted" }, Binder: 0x0, Cookie: 0 }, resultWho = "(null)", requestCode = 0, intents = [ "{ "mAction" :
"com.android.providers.calendar.SCHEDULE_ALARM", "mData" : null, "mType" : "(null)", "mPackage" : "(null)", "mComponent" : { "mPackage" :
"com.android.providers.calendar", "mClass" : "com.android.providers.calendar.CalendarReceiver" }, "mFlags" : 0, "mCategories" : [ ], "mExtras" : null,
"mSourceBounds" : null, "mSelector" : null, "mClipData" : null }" ], resolvedTypes = [ ], flags = 536870912, options = null, userId = 0)

Target: 1, Cookie: 0, Code: 63, Flags: 16, SenderPid: 0, SenderEuid: 0, Datasize: 492, OffsetsSize: 0
Buffer:
00000000  00 01 00 00 1c 00 00 00  61 00 6e 00 64 00 72 00   ........a.n.d.r.
00000010  6f 00 69 00 64 00 2e 00  61 00 70 00 70 00 2e 00   o.i.d...a.p.p...
00000020  49 00 41 00 63 00 74 00  69 00 76 00 69 00 74 00   I.A.c.t.i.v.i.t.
00000030  79 00 4d 00 61 00 6e 00  61 00 67 00 65 00 72 00   y.M.a.n.a.g.e.r.
00000040  00 00 00 00 01 00 00 00  1e 00 00 00 63 00 6f 00   ............c.o.
00000050  6d 00 2e 00 61 00 6e 00  64 00 72 00 6f 00 69 00   m...a.n.d.r.o.i.
00000060  64 00 2e 00 70 00 72 00  6f 00 76 00 69 00 64 00   d...p.r.o.v.i.d.
00000070  65 00 72 00 73 00 2e 00  63 00 61 00 6c 00 65 00   e.r.s...c.a.l.e.
```

Figure B.1: Binder Transaction with inline reply


CopperDroid Viewer - Filtering On Interface(s): [android.app.IActivityManager] Method(s): [getIntentSender]
Processes
System Call Types
System Call Operation
System Call Data
Ready

Name	Pid
*	*
init	1
ueventd	33
init	34
e2fsck	41
e2fsck	44
healthd	45
vold	47
servicemanager	46
debuggerd	50
rild	51

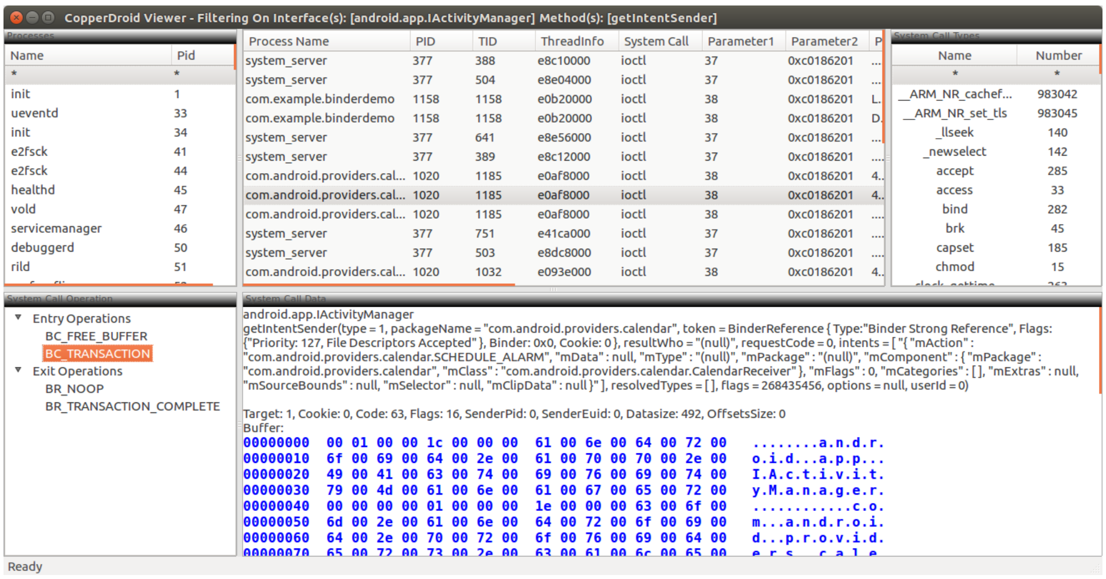
Process Name	PID	TID	ThreadInfo	System Call	Parameter1	Parameter2	P
system_server	377	388	e8c10000	ioctl	37	0xc0186201	...
system_server	377	504	e8e04000	ioctl	37	0xc0186201	...
com.example.binderdemo	1158	1158	e0b20000	ioctl	38	0xc0186201	L.
com.example.binderdemo	1158	1158	e0b20000	ioctl	38	0xc0186201	D.
system_server	377	641	e8e56000	ioctl	37	0xc0186201	...
system_server	377	389	e8c12000	ioctl	37	0xc0186201	....
com.android.providers.cal...	1020	1185	e0af8000	ioctl	38	0xc0186201	4..
com.android.providers.cal...	1020	1185	e0af8000	ioctl	38	0xc0186201	4..
com.android.providers.cal...	1020	1185	e0af8000	ioctl	38	0xc0186201	....
system_server	377	751	e41ca000	ioctl	37	0xc0186201	....
system_server	377	503	e8dc8000	ioctl	37	0xc0186201	....
com.android.providers.cal...	1020	1032	e093e000	ioctl	38	0xc0186201	4..

Name	Number
*	*
__ARM_NR_cachef...	983042
__ARM_NR_set_tls	983045
_llseek	140
_newselect	142
accept	285
access	33
bind	282
brk	45
capset	185
chmod	15

- Entry Operations
  - BC_FREE_BUFFER
  - BC_TRANSACTION
- Exit Operations
  - BR_NOOP
  - BR_TRANSACTION_COMPLETE
  - BR_REPLY

```
android.app.IActivityManager
getIntentSender()
exceptionCode = 0
res = BinderReference { Type:"Binder Handle", Flags: {"Priority: 127, File Descriptors Accepted" }, Binder: 0x10, Cookie: b81b0570 }
Target: 0, Cookie: 0, Code: 0, Flags: 0, SenderPid: 0, SenderEuid: 1000, Datasize: 20, OffsetsSize: 4
Buffer:
00000000  00 00 00 00 85 2a 68 73  7f 01 00 00 10 00 00 00   .....*hs........
00000010  70 05 1b b8                                        p...


Offsets:
00000000  04 00 00 00                                        ....
```

Figure B.2: Binder inline reply


CopperDroid Viewer - Filtering On Interface(s): [android.app.IActivityManager] Method(s): [getIntentSender]
Processes
System Call Types
System Call Operation
System Call Data
Ready

Name	Pid
*	*
init	1
ueventd	33
init	34
e2fsck	41
e2fsck	44
healthd	45
vold	47
servicemanager	46
debuggerd	50
rild	51

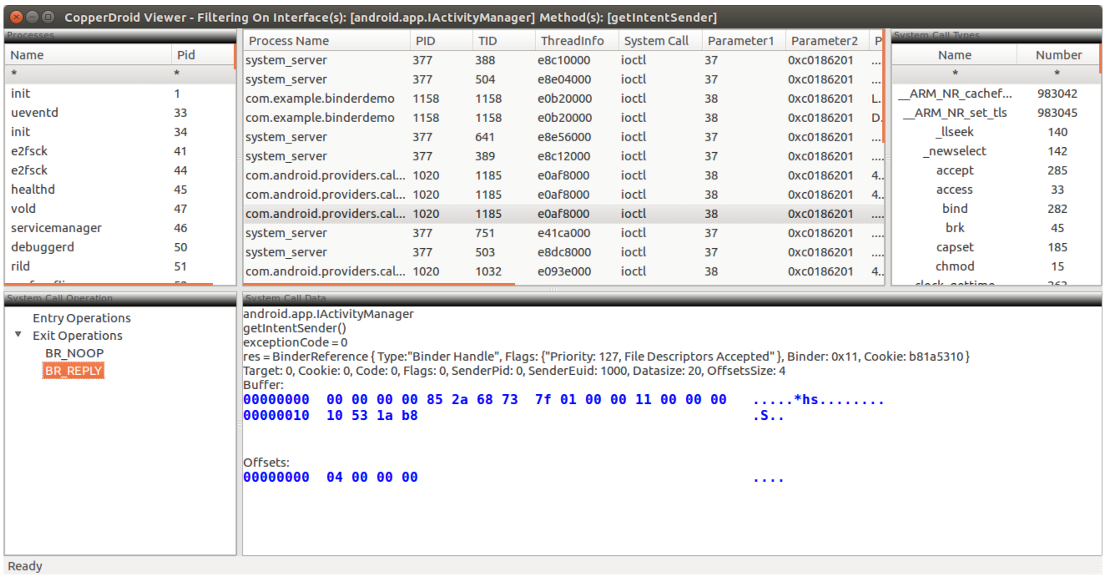
Process Name	PID	TID	ThreadInfo	System Call	Parameter1	Parameter2	P
system_server	377	388	e8c10000	ioctl	37	0xc0186201	...
system_server	377	504	e8e04000	ioctl	37	0xc0186201	...
com.example.binderdemo	1158	1158	e0b20000	ioctl	38	0xc0186201	L.
com.example.binderdemo	1158	1158	e0b20000	ioctl	38	0xc0186201	D.
system_server	377	641	e8e56000	ioctl	37	0xc0186201	...
system_server	377	389	e8c12000	ioctl	37	0xc0186201	....
com.android.providers.cal...	1020	1185	e0af8000	ioctl	38	0xc0186201	4..
com.android.providers.cal...	1020	1185	e0af8000	ioctl	38	0xc0186201	4..
com.android.providers.cal...	1020	1185	e0af8000	ioctl	38	0xc0186201	....
system_server	377	751	e41ca000	ioctl	37	0xc0186201	....
system_server	377	503	e8dc8000	ioctl	37	0xc0186201	....
com.android.providers.cal...	1020	1032	e093e000	ioctl	38	0xc0186201	4..

Name	Number
*	*
__ARM_NR_cachef...	983042
__ARM_NR_set_tls	983045
_llseek	140
_newselect	142
accept	285
access	33
bind	282
brk	45
capset	185
chmod	15

- Entry Operations
  - BC_FREE_BUFFER
  - BC_TRANSACTION
- Exit Operations
  - BR_NOOP
  - BR_TRANSACTION_COMPLETE

```
android.app.IActivityManager
getIntentSender(type = 1, packageName = "com.android.providers.calendar", token = BinderReference { Type:"Binder Strong Reference", Flags:
{"Priority: 127, File Descriptors Accepted" }, Binder: 0x0, Cookie: 0 }, resultWho = "(null)", requestCode = 0, intents = [ "{ "mAction" :
"com.android.providers.calendar.SCHEDULE_ALARM", "mData" : null, "mType" : "(null)", "mPackage" : "(null)", "mComponent" : { "mPackage" :
"com.android.providers.calendar", "mClass" : "com.android.providers.calendar.CalendarReceiver" }, "mFlags" : 0, "mCategories" : [ ], "mExtras" : null,
"mSourceBounds" : null, "mSelector" : null, "mClipData" : null }" ], resolvedTypes = [ ], flags = 268435456, options = null, userId = 0)

Target: 1, Cookie: 0, Code: 63, Flags: 16, SenderPid: 0, SenderEuid: 0, Datasize: 492, OffsetsSize: 0
Buffer:
00000000  00 01 00 00 1c 00 00 00  61 00 6e 00 64 00 72 00   ........a.n.d.r.
00000010  6f 00 69 00 64 00 2e 00  61 00 70 00 70 00 2e 00   o.i.d...a.p.p...
00000020  49 00 41 00 63 00 74 00  69 00 76 00 69 00 74 00   I.A.c.t.i.v.i.t.
00000030  79 00 4d 00 61 00 6e 00  61 00 67 00 65 00 72 00   y.M.a.n.a.g.e.r.
00000040  00 00 00 00 01 00 00 00  1e 00 00 00 63 00 6f 00   ............c.o.
00000050  6d 00 2e 00 61 00 6e 00  64 00 72 00 6f 00 69 00   m...a.n.d.r.o.i.
00000060  64 00 2e 00 70 00 72 00  6f 00 76 00 69 00 64 00   d...p.r.o.v.i.d.
00000070  65 00 72 00 73 00 2e 00  63 00 61 00 6c 00 65 00   e.r.s...c.a.l.e.
```

Figure B.3: Binder Transaction with out-of-line reply


CopperDroid Viewer - Filtering On Interface(s): [android.app.IActivityManager] Method(s): [getIntentSender]

Processes

Name	Pid
*	*
init	1
ueventd	33
init	34
e2fsck	41
e2fsck	44
healthd	45
vold	47
servicemanager	46
debuggerd	50
rild	51

Process Name	PID	TID	ThreadInfo	System Call	Parameter1	Parameter2	P
system_server	377	388	e8c10000	ioctl	37	0xc0186201	...
system_server	377	504	e8e04000	ioctl	37	0xc0186201	...
com.example.binderdemo	1158	1158	e0b20000	ioctl	38	0xc0186201	L.
com.example.binderdemo	1158	1158	e0b20000	ioctl	38	0xc0186201	D.
system_server	377	641	e8e56000	ioctl	37	0xc0186201	...
system_server	377	389	e8c12000	ioctl	37	0xc0186201	....
com.android.providers.cal...	1020	1185	e0af8000	ioctl	38	0xc0186201	4..
com.android.providers.cal...	1020	1185	e0af8000	ioctl	38	0xc0186201	4..
com.android.providers.cal...	1020	1185	e0af8000	ioctl	38	0xc0186201	....
system_server	377	751	e41ca000	ioctl	37	0xc0186201	....
system_server	377	503	e8dc8000	ioctl	37	0xc0186201	....
com.android.providers.cal...	1020	1032	e093e000	ioctl	38	0xc0186201	4..

System Call Types

Name	Number
*	*
__ARM_NR_cachef...	983042
__ARM_NR_set_tls	983045
_llseek	140
_newselect	142
accept	285
access	33
bind	282
brk	45
capset	185
chmod	15

System Call Operation

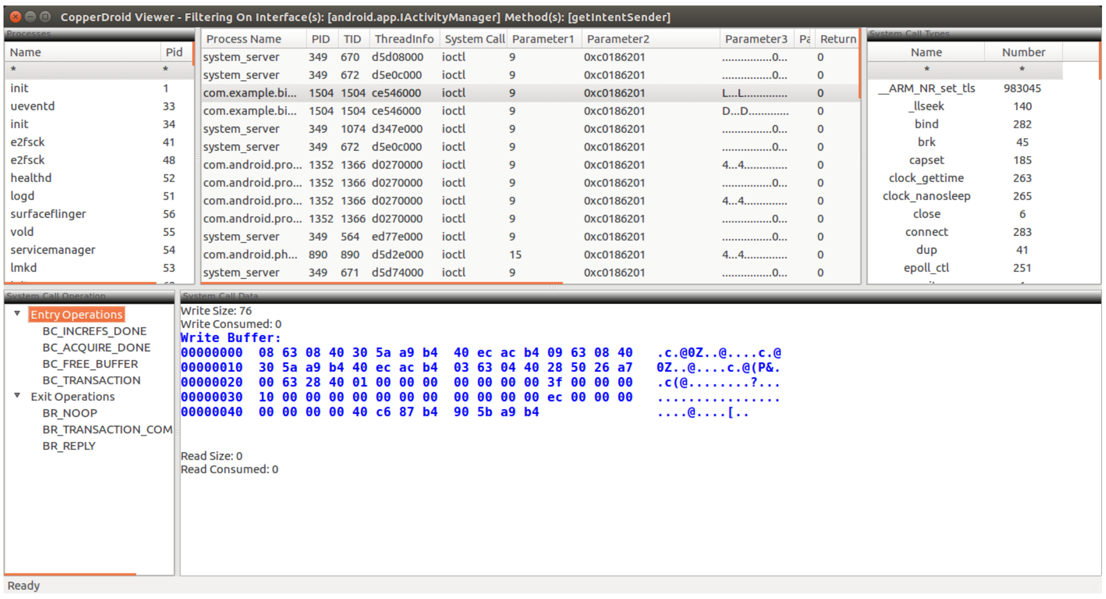
- Entry Operations
- Exit Operations
  - BR_NOOP
  - BR_REPLY

System Call Data

```
android.app.IActivityManager
getIntentSender()
exceptionCode = 0
res = BinderReference { Type:"Binder Handle", Flags: {"Priority: 127, File Descriptors Accepted" }, Binder: 0x11, Cookie: b81a5310 }
Target: 0, Cookie: 0, Code: 0, Flags: 0, SenderPid: 0, SenderEuid: 1000, Datasize: 20, OffsetsSize: 4
Buffer:
00000000  00 00 00 00 85 2a 68 73  7f 01 00 00 11 00 00 00   .....*hs........
00000010  10 53 1a b8                                         .S..


Offsets:
00000000  04 00 00 00                                         ....
```

Ready

Figure B.4: Binder out-of-line reply


CopperDroid Viewer - Filtering On Interface(s): [android.app.IActivityManager] Method(s): [getIntentSender]

Processes

Name	Pid
*	*
init	1
ueventd	33
init	34
e2fsck	41
e2fsck	48
healthd	52
logd	51
surfaceflinger	56
vold	55
servicemanager	54
lmkd	53

Process Name	PID	TID	ThreadInfo	System Call	Parameter1	Parameter2	Parameter3	Pa	Return
system_server	349	670	d5d08000	ioctl	9	0xc0186201	................0...		0
system_server	349	672	d5e0c000	ioctl	9	0xc0186201	................0...		0
com.example.bi...	1504	1504	ce546000	ioctl	9	0xc0186201	L...L..............		0
com.example.bi...	1504	1504	ce546000	ioctl	9	0xc0186201	D...D.............		0
system_server	349	1074	d347e000	ioctl	9	0xc0186201	................0...		0
system_server	349	672	d5e0c000	ioctl	9	0xc0186201	................0...		0
com.android.pro...	1352	1366	d0270000	ioctl	9	0xc0186201	4...4..............		0
com.android.pro...	1352	1366	d0270000	ioctl	9	0xc0186201	................0...		0
com.android.pro...	1352	1366	d0270000	ioctl	9	0xc0186201	4...4..............		0
com.android.pro...	1352	1366	d0270000	ioctl	9	0xc0186201	................0...		0
system_server	349	564	ed77e000	ioctl	9	0xc0186201	................0...		0
com.android.ph...	890	890	d5d2e000	ioctl	15	0xc0186201	4...4..............		0
system_server	349	671	d5d74000	ioctl	9	0xc0186201	................0...		0

System Call Types

Name	Number
*	*
__ARM_NR_set_tls	983045
_llseek	140
bind	282
brk	45
capset	185
clock_gettime	263
clock_nanosleep	265
close	6
connect	283
dup	41
epoll_ctl	251

System Call Operation

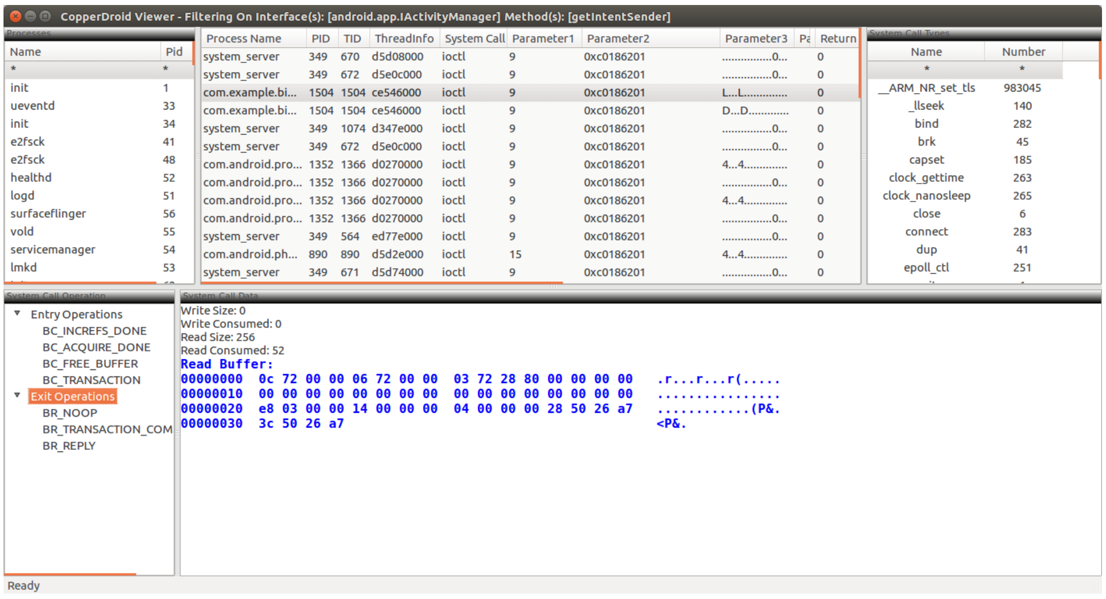
- Entry Operations
  - BC_INCREFS_DONE
  - BC_ACQUIRE_DONE
  - BC_FREE_BUFFER
  - BC_TRANSACTION
- Exit Operations
  - BR_NOOP
  - BR_TRANSACTION_COM
  - BR_REPLY

System Call Data

```
Write Size: 76
Write Consumed: 0
Write Buffer:
00000000  08 63 08 40 30 5a a9 b4  40 ec ac b4 09 63 08 40   .c.@0Z..@....c.@
00000010  30 5a a9 b4 40 ec ac b4  03 63 04 40 28 50 26 a7   0Z..@....c.@(P&.
00000020  00 63 28 40 01 00 00 00  00 00 00 00 3f 00 00 00   .c(@........?...
00000030  10 00 00 00 00 00 00 00  00 00 00 00 ec 00 00 00   ................
00000040  00 00 00 00 40 c6 87 b4  90 5b a9 b4               ....@....[..

Read Size: 0
Read Consumed: 0
```

Ready

Figure B.5: Command operations with `write_size` and `write_consumed`


CopperDroid Viewer - Filtering On Interface(s): [android.app.IActivityManager] Method(s): [getIntentSender]

Processes

Name	Pid
*	*
init	1
ueventd	33
init	34
e2fsck	41
e2fsck	48
healthd	52
logd	51
surfaceflinger	56
vold	55
servicemanager	54
lmkd	53

Process Name	PID	TID	ThreadInfo	System Call	Parameter1	Parameter2	Parameter3	Pa	Return
system_server	349	670	d5d08000	ioctl	9	0xc0186201	................0...		0
system_server	349	672	d5e0c000	ioctl	9	0xc0186201	................0...		0
com.example.bi...	1504	1504	ce546000	ioctl	9	0xc0186201	L...L..............		0
com.example.bi...	1504	1504	ce546000	ioctl	9	0xc0186201	D...D.............		0
system_server	349	1074	d347e000	ioctl	9	0xc0186201	................0...		0
system_server	349	672	d5e0c000	ioctl	9	0xc0186201	................0...		0
com.android.pro...	1352	1366	d0270000	ioctl	9	0xc0186201	4...4..............		0
com.android.pro...	1352	1366	d0270000	ioctl	9	0xc0186201	................0...		0
com.android.pro...	1352	1366	d0270000	ioctl	9	0xc0186201	4...4..............		0
com.android.pro...	1352	1366	d0270000	ioctl	9	0xc0186201	................0...		0
system_server	349	564	ed77e000	ioctl	9	0xc0186201	................0...		0
com.android.ph...	890	890	d5d2e000	ioctl	15	0xc0186201	4...4..............		0
system_server	349	671	d5d74000	ioctl	9	0xc0186201	................0...		0

System Call Types

Name	Number
*	*
__ARM_NR_set_tls	983045
_llseek	140
bind	282
brk	45
capset	185
clock_gettime	263
clock_nanosleep	265
close	6
connect	283
dup	41
epoll_ctl	251

System Call Operation

- Entry Operations
  - BC_INCREFS_DONE
  - BC_ACQUIRE_DONE
  - BC_FREE_BUFFER
  - BC_TRANSACTION
- Exit Operations
  - BR_NOOP
  - BR_TRANSACTION_COM
  - BR_REPLY

System Call Data

```
Write Size: 0
Write Consumed: 0
Read Size: 256
Read Consumed: 52
Read Buffer:
00000000  0c 72 00 00 06 72 00 00  03 72 28 80 00 00 00 00   .r...r...r(.....
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000020  e8 03 00 00 14 00 00 00  04 00 00 00 28 50 26 a7   ............(P&.
00000030  3c 50 26 a7                                        <P&.
```

Ready

Figure B.6: Return operations with `read_size` and `read_consumed`

Bibliography

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

[2] Android. Android developer reference. `https://developer.android.com/reference/packages.html`.

[3] Andrew W. Appel. Modern Compiler Implementation in C. Cambridge University Press, 1998.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[5] Giuseppe Attardi, Antonio Cisternino, and Andrew Kennedy. Codebricks: code fragments as building blocks. In PEPM '03 Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation, volume 38, pages 66–74, 2003.

[6] Android Authority. Google removed 700k apps from the play store in 2017 for violating policies. `https://www.androidauthority.com/google-play-store-removed-700000-apps-2017-violating-policies-834031/`.

[7] U. Bayer, A. Moser, C. Kruegel, and E. Kirda. Dynamic analysis of malicious code.

[8] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In USENIX Annual Technical Conference, FREENIX Track, volume 41, page 46, 2005.

[9] David W. Binkley and Keith Brian Gallagher. Program slicing. In Marvin V. Zelkowitz, editor, Science Direct, volume 43 of Advances in Computers, pages 1 – 50. Elsevier, 1996.

[10] Oberheide J. Miller C. Dissecting the android bouncer. `https://jon.oberheide.org/files/summercon12-bouncer.pdf`.

[11] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In NDSS, 2015.

[12] Catalin Cimpanu. Hummingbad android malware found in 20 google play store apps, 2017. `https://www.bleepingcomputer.com/news/security/hummingbad-android-malware-found-in-20-google-play-store-apps/`.

[13] Federal Trade Commission. Ftc approves final order settling charges against flashlight app creator, 2014. https://www.ftc.gov/news-events/press-releases/2014/04/ftc-approves-final-order-settling-charges-against-flashlight-app.

[14] Federal Trade Commission. Ftc settles with two companies falsely claiming to comply with international safe harbor privacy framework, 2015. https://www.ftc.gov/news-events/press-releases/2015/04/ftc-settles-two-companies-falsely-claiming-comply-international.

[15] Federal Trade Commission. Tech company settles ftc charges it unfairly installed apps on android mobile devices without users' permission, 2016. https://www.ftc.gov/news-events/press-releases/2016/02/tech-company-settles-ftc-charges-it-unfairly-installed-apps.

[16] Shuaifu Dai Guofei Gu Xiaorui Gong Xinhui Han Cong Zheng, Shixiong Zhu and Wei Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In ACM Security and privacy in smartphones and mobile devices (SPSM), 2012.

[17] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS), pages 1–16, 2017.

[18] Keith D. Cooper and Linda Torczon. Engineering a Compiler. Morgan Kaufmann, 2012.

[19] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. IEEE Transactions on Software Engineering, 35(5):684–702, Sept 2009.

[20] Tech Crunch. App revenue climbed 35 percent to usd 60 billion in 2017. https://techcrunch.com/2018/01/05/app-revenue-climbed-35-percent-to-60-billion-in-2017/.

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

[22] Nikolay Elenkov. Using app encryption in jelly bean. https://nelenkov.blogspot.com/2012/07/using-app-encryption-in-jelly-bean.html.

[23] W. Enck, P. Gilbert, B.G. Chun, L.P. Cox, J. Jung, P. McDaniel, and A.N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In USENIX OSDI, 2010.

[24] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[25] F-Secure. Brain. https://www.f-secure.com/v-descs/brain.shtml.

[26] Huan Feng and Kang G. Shin. Understanding and defending the binder attack surface in android. In Proceedings of the 32Nd Annual Conference on Computer Security Applications, ACSAC '16, pages 398–409, New York, NY, USA, 2016. ACM.

[27] Forbes. What is general data protection regulation?, 2018. `https://www.forbes.com/sites/quora/2018/02/14/what-is-general-data-protection-regulation/#510bc50d62dd`.

[28] Thomas Fox-Brewster. Google is fighting a massive android malware outbreak – up to 21 million victims, 2017. ``.

[29] Jyoti Gajrani, Jitendra Sarswat, Meenakshi Tripathi, Vijay Laxmi, Manoj Singh Gaur, and Mauro Conti. A robust dynamic analysis system preventing sandbox detection by android malware. In Proceedings of the 8th International Conference on Security of Information and Networks, pages 290–295. ACM, 2015.

[30] Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In Ndss, volume 3, pages 191–206, 2003.

[31] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In International Conference on Trust and Trustworthy Computing, pages 291–307. Springer, 2012.

[32] Google. Android bionic. `https://android.googlesource.com/platform/bionic/`.

[33] Google. Android emulator. `https://developer.android.com/studio/run/emulator`.

[34] Google. Android interface definition language (aidl). `https://developer.android.com/guide/components/aidl`.

[35] Google. Android lollipop gensyscalls.py source code. `https://android.googlesource.com/platform/bionic/+/lollipop-release/libc/tools/gensyscalls.py`.

[36] Google. Android software development kit. `https://developer.android.com/studio/releases/platforms`.

[37] Google. Extending the kernel with ebpf. `https://source.android.com/devices/architecture/kernel/bpf`.

[38] Google. The google play store. `https://developer.android.com/distribute/google-play/`.

[39] Google. Google protocol buffers. `https://developers.google.com/protocol-buffers/`.

[40] Google. Jelly bean. `https://developer.android.com/about/versions/jelly-bean`.

[41] GuardSquare. Dexguard: Protecting android applications and sdks against reverse engineering and hacking. https://www.guardsquare.com/en/products/dexguard.

[42] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11, pages 639–652, New York, NY, USA, 2011. ACM.

[43] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 – seamless operability between c++11 and python, 2017. https://github.com/pybind/pybind11.

[44] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. Privacy oracle: a system for finding application leaks with black box differential testing. In Proceedings of the 15th ACM conference on Computer and communications security, pages 279–288. ACM, 2008.

[45] YUKIYOSHI KAMEYAMA, OLEG KISELYOV, and CHUNG-CHIEH SHAN. Shifting the stage: Staging with delimited control. Journal of Functional Programming, 21(6):617–662, 2011.

[46] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. arXiv preprint arXiv:1801.01203, 2018.

[47] Oren Koriat. A whale of a tale: Hummingbad returns, 2017. https://blog.checkpoint.com/2017/01/23/hummingbad-returns/.

[48] Kaspersky Lab. Carberp-in-the-mobile. https://securelist.com/carberp-in-the-mobile/57658/.

[49] F-Secure Labs. Trojan:android/pincer.a, 2013. https://www.f-secure.com/weblog/archives/00002538.html.

[50] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, page 75. IEEE Computer Society, 2004.

[51] Anh Le, Janus Varmarken, Simon Langhoff, Anastasia Shuba, Minas Gjoka, and Athina Markopoulou. Antmonitor: A system for monitoring from mobile devices. In Proceedings of the 2015 ACM SIGCOMM Workshop on Crowdsourcing and Crowdsharing of Big (Internet) Data, C2B(1)D '15, pages 15–20, New York, NY, USA, 2015. ACM.

[52] John R. Levine. Linkers and Loaders. Morgan Kaufmann Publishers, 200.

[53] Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graphs. In Software Maintenance, 1998. Proceedings., International Conference on, pages 358–367. IEEE, 1998.

[54] Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graphs. In ICSM '98 Proceedings of the International Conference on Software Maintenance. IEEE Computer Society, 1998.

[55] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware behaviours. In Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), 2014.

[56] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In 27th {USENIX} Security Symposium ({USENIX} Security 18), pages 973–990, 2018.

[57] Yabing Liu, Han Hee Song, Ignacio Bermudez, Alan Mislove, Mario Baldi, and Alok Tongaonkar. Identifying personal information in internet traffic. In Proceedings of the 2015 ACM on Conference on Online Social Networks, pages 59–70. ACM, 2015.

[58] Hiroshi Lockheimer. Bouncer. `http://googlemobile.blogspot.it/2012/02/android-and-security.html`.

[59] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In USENIX winter, volume 46, 1993.

[60] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual, pages 421–430. IEEE, 2007.

[61] The Hacker News. Nasty android malware that infected millions returns to google play store. `https://thehackernews.com/2017/01/hummingbad-android-malware.html`.

[62] Idan Revivo Nitay Artenstein. Man in the binder: He who controls IPC, controls the droid. `https://blog.checkpoint.com/wp-content/uploads/2015/02/Man-In-The-Binder-He-Who-Controls-IPC-Controls-The-Droid-wp.pdf`.

[63] Oracle. randomuuid, 2018. `https://docs.oracle.com/javase/7/docs/api/java/util/UUID.html#randomUUID()`.

[64] Terence Parr. Language Implementation Patters. Pragmatic Bookshelf, 2010.

[65] Terence Parr and Kathleen Fisher. Ll(*): The foundation of the antlr parser generator. In Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, pages 425–436, New York, NY, USA, 2011. ACM.

[66] PassMark Software. PassMark Android. `http://www.passmark.com/`.

[67] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In Proceedings of the Seventh European Workshop on System Security, page 5. ACM, 2014.

[68] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting system emulators. In Juan A. Garay, Arjen K. Lenstra, Masahiro Mambo, and René Peralta, editors, Information Security, pages 1–18, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[69] Vaibhav Rastogi, Yan Chen, and William Enck. Appsplayground: automatic security analysis of smartphone applications. In CODASPY'13 Proceedings of the third ACM conference on Data and application security and privacy, 2013.

[70] Abbas Razaghpanah, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Christian Kreibich, Phillipa Gill, Mark Allman, and Vern Paxson. Haystack: In situ mobile traffic analysis in user space. ArXiv e-prints, 2015.

[71] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviours. In Proceedings of the 6th European Workshop on System Security (EUROSEC).

[72] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. Recon: Revealing and controlling pii leaks in mobile network traffic. In Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '16, pages 361–374, New York, NY, USA, 2016. ACM.

[73] Sarker T Ahmed Rumee, Donggang Liu, and Yu Lei. Mirrordroid: A framework to detect sensitive information leakage in android by duplicate program execution. In 2017 51st Annual Conference on Information Sciences and Systems (CISS), pages 1–6. IEEE, 2017.

[74] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In NDSS, 2008.

[75] Yihang Song and Urs Hengartner. Privacyguard: A vpn-based platform to detect information leakage on android devices. In Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM '15, pages 15–26, New York, NY, USA, 2015. ACM.

[76] Statista. Number of smartphone users worldwide from 2014 to 2020 (in billions). `https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/`.

[77] Mingshen Sun, Tao Wei, and John C.S. Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In ACM CCS 2016. ACM, 2016.

[78] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware behaviours. In NDSS Symposium 2015. Internet Society, 2015.

[79] Fred Touchette. The evolution of malware. Network Security, 2016(1):11–14, 2016.

[80] Omer Tripp and Julia Rubin. A bayesian approach to privacy enforcement in smartphones. In USENIX Security Symposium, pages 175–190, 2014.

[81] Unknown. Dendroid malware source. `https://github.com/mwsrc/Dendroid-HTTP-RAT`.

[82] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In CASCON First Decade High Impact Papers, CASCON '10, pages 214–224, Riverton, NJ, USA, 2010. IBM Corp.

[83] M. L. Van De Vanter. Preserving the documentary structure of source code in language-based transformation tools. In Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation, pages 131–141, 2001.

[84] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In ASIA CCS '14 Proceedings of the 9th ACM symposium on Information, computer and communications security, Pages 447-458. ACM New York, NY, USA 2014, 2014.

[85] James Vincent. 99.6 percent of new smartphones run android or ios. `https://www.theverge.com/2017/2/16/14634656/android-ios-market-share-blackberry-2016`.

[86] Mark Weiser. Program slicing. In IEEE Transactions on Software Engineering, volume 10, pages 352–357, 1984.

[87] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. ACM SIGSOFT Software Engineering Notes, 30(2):1–36, 2005.

[88] L.-K. Yan and H. Yin. Droidscope: Seamlessly reconstructing os and dalvik semantic views for dynamic android malware analysis. In USENIX Security 2012. USENIX, 2012.