Targeted Security Analysis of Android Applications with Hybrid Program Analysis

by

Michelle Yan Yi Wong

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Targeted Security Analysis of Android Applications with Hybrid Program Analysis

Michelle Yan Yi Wong
Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto
2021

Mobile devices are prevalent in everyday society and the installation of third-party applications provide a variety of services, such as location tracking, messaging, and financial management. The trove of sensitive information and functionality on these devices and their large user base attract malware developers who want to exploit this functionality for monetary gain or to cause harm. To protect the security and privacy of mobile device users, we wish to analyze applications to extract the types of actions they perform and to determine whether they can be trusted. Program analysis techniques have commonly been used to perform such analysis and are primarily static or dynamic in nature. Static analysis operates on the code of the application and provides good analysis coverage, but is imprecise due to the lack of run-time information. Dynamic analysis operates as the application is executing and is more precise due to the availability of the execution trace, but is often limited by low code coverage since only the parts of the application that are actually executed can be analyzed.

In this thesis, we explore the use of hybrid program analysis techniques that use the strengths of both static and dynamic analysis to achieve more effective security analysis of applications on the Android mobile platform. We propose and develop the idea of targeted execution, in which analysis resources are focused on the specific code locations that are of interest to a security analyzer. We dynamically execute the application at these locations to enable precise security analysis of the behaviors. To target the locations, we preface the dynamic analysis with a static phase that performs a conservative search for potential behaviors of interest and extracts the code paths that lead to them. It then determines how these code paths can be executed such that the target behavior can be analyzed. We show how the use of both static and dynamic analysis can enable more effective execution and analysis of applications than the existing state-of-the-art techniques. We further show how hybrid program analysis can enable the deobfuscation of applications, a challenge that often plagues security analysis tools.

Acknowledgements

I would first like to thank my advisor, Professor David Lie, for all of his guidance and mentorship throughout my graduate school career. His advice and feedback have been invaluable, not only in solving technical issues or writing papers, but also in navigating the ups and downs of graduate school, providing career opportunities, and more recently, supporting his students through the challenges of COVID-19 and the lockdown. I hope to carry the skills that I have learned from him throughout my career. To me, he will always be a role model of what makes a good leader and mentor.

I would also like to thank my husband, Yan, for all of his support throughout my years in graduate school. When I think back to what made me successful in my PhD, while hard work was of course important, a key component was the external support I received during those times when a paper deadline was looming or when my experiments produced negative results and it felt like what I had designed would never work. Perhaps even more crucial was Yan's insistence on maintaining a healthy balance between work and personal life, which ultimately made me more creative, focused, and resilient when facing challenges at work.

I would like to thank the other graduate students in my research group, especially Mariana, James, Ben and Shawn, for all of their help and encouragement during meetings and in the office. I appreciate all of our discussions and chats. It made graduate school much more enjoyable to have a group of supportive peers that buoy each other up and work together toward mutual success.

I would also like to extend my thanks to my PhD supervisory committee, Professor Ashvin Goel and Professor Tarek Abdelrahman, for their feedback and encouragement. They have always strived to ensure my work was the best it could be. I would further like to thank my other thesis examiners, Professor Michael Stumm and Professor Patrick McDaniel, for their very helpful feedback and suggestions.

I would like to thank my family and friends for their support over the years. They have always believed in me and encouraged me throughout my academic career.

Finally, I would like to thank the University of Toronto and the Natural Sciences and Engineering Research Council of Canada (NSERC) for their financial support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Society increasingly relies on computers to perform daily tasks, either in the workplace or in personal life. While most computers are bundled with software from the original manufacturer, they are commonly designed to allow the installation of code or programs from other parties to extend their functionality. The development of third-party software has grown dramatically in the past several decades and has produced applications that touch all aspects of our lives including our health, finances, household, leisure, and work/business management.

The use of computing devices increased significantly with the development of mobile devices. The rise of smartphones, in particular, has driven the adoption of mobile devices as an everyday item carried by most people. Smartphones provide a variety of services that extend beyond the telecommunication capabilities of their predecessors; in particular, the numerous hardware sensors now included within the devices and the ability to install third-party applications to make use of this hardware has extended their functionality to that of a personal assistant that can take notes, manage users' messages and contacts, track locations and route travel, provide health information, and a number of other services. There are a variety of mobile device providers and several operating systems that are targeted for such devices. As of 2020, the most popular mobile operating system is Android, which is open-sourced and currently maintained by Google. Android is now used by 87.4% of mobile devices worldwide [95].

The creation and adoption of third-party applications have made many tasks easier but bring a new vector of attack for those who wish to subvert their normal processes for nefarious purposes, such as for monetary gain or to cause harm to the users of the devices. In particular, due to the plethora of functionality in mobile devices and their use in everyday lives, they are a trove of sensitive personal information and a controller of a multitude of sensitive actions that can be performed on behalf of their users. The services provided by third-party applications can cause great damage to the user if they were subverted by a malicious actor; for instance, a bank application might be exploited such that money is stolen from the user or a household management application subverted to manipulate an appliance in a dangerous way. As such, there is a large motivation to check or vet third-party software to ensure that they do not contain vulnerabilities that can be exploited to enable such attacks and that they perform the functions expected of them (i.e. they are not malicious in nature and do not harm the user). In particular, application marketplaces that promote and facilitate the installation of third-party software endeavor to remove malware from their offerings to maintain the security of their user base. The Google Play Store, the primary distributor of applications for Android, hosts an estimated 3 million

applications [94] and must efficiently and effectively vet the submitted applications to determine whether they perform any malicious or harmful behavior.

1.1 Program analysis

One method of performing security analysis of applications is through program analysis techniques, which analyzes the semantics of an application's code, either in storage or in execution, to check whether it contains vulnerabilities or malicious behavior. Program analysis generally falls into two categories: static or dynamic. Static techniques operate on the application's bytecode while dynamic techniques use information or data collected as an application is running. Both have their advantages and their drawbacks. Static analysis allows one to reason about the entire code base of the application as a whole, but must be imprecise since it does not have access to information that may only be available during execution. On the other hand, dynamic analysis techniques are very precise and suffer from very few false positives (or no false positives), since any interesting or malicious behavior that is detected must have been executed and its exact trace is available. However, since dynamic analysis can only report behavior that has actually been executed, it often suffers from poor code coverage since it is difficult to know exactly how to execute behavior that is hidden behind complex logic and only triggered under specific conditions.

Malicious application developers often take advantage of these drawbacks to evade detection. For instance, code obfuscation is a long established technique to hide malicious code from antivirus programs and anti-malware scanners. Obfuscation techniques can include hiding the code that is executed and loading it only at run-time, making any static analysis incomplete due to its lack of dynamic execution traces or loaded code. Obfuscation can also affect dynamic analysis—malware developers may try to hide the malicious activity within a seemingly benign application and only trigger the malicious actions under very specific conditions or in a particular (likely non-analysis or non-testing) environment. This affects the code coverage of the dynamic analysis and the completeness of the results.

A number of different techniques have been proposed to combat the issue of code coverage in dynamic analysis. One family of techniques is random fuzzing [99, 137], which mutates inputs and injects them into the application in an effort to trigger previously unexplored code. Another common approach is symbolic execution [5,27,28], which tracks how inputs into the application are used at conditional branch instructions in order to represent the execution path through constraints placed on the input values. It tries to reach new code in the application by manipulating these constraints to generate new input values that can trigger different branch outcomes. For Android devices, GUI exercising [55,71,81,112] is another technique in which different screens/widgets in the application are explored either randomly or by searching through a constructed model of the application's user interaction (UI) flow. On the whole, these dynamic techniques focus on achieving maximum code coverage—that is, trying to execute all of the code in an application. They are often attached to a separate analyzer or detector that will then perform the actual analysis of interesting behaviors; for instance, a malware detection tool that checks for leakage of private data may attach a fuzzer to a dynamic data-flow tracker (e.g. a taint tracker) and attempt to execute all of the code in the application to verify whether any data sent to the network contains private user information. By combining a code exploration or coverage tool with the dynamic analyzer, any interesting or malicious behavior in an application should be eventually executed and detected.

1.2 Targeted execution: an alternative approach

While coverage-based execution driving is useful for some dynamic tools, blanket code coverage is not necessarily the optimal metric for all types of analyses. The goal of security analysis is to determine whether the application performs specific interesting or malicious behaviors, which often occur in only a few locations. For instance, when detecting sensitive information leakage through taint analysis [40], the analyzer is focused on locations in an application where sensitive information is read and locations where information may be sent off of the device (i.e. the sources and sinks of the sensitive data flow). There may be a large amount of other irrelevant code in the application, such as code managing the manipulation of the user interface or storing user settings. Attempting to execute all the code in an application to reach those specific locations is a waste of resources and may not yield the best results compared to an analysis that is more focused and makes better use of these resources [29].

In an ideal dynamic analysis system, a code exploration or execution driving tool would execute only the parts of the application that the analyzer is interested in. This is, of course, unrealistic. If the exploration tool knew exactly what behavior in an application the analyzer would be interested in, the analyzer would be redundant—if we knew how to identify and execute these interesting behaviors, we technically already know if the application contains the behaviors and whether they are malicious actions or vulnerabilities. More realistically, the exploration tool would likely execute an over-approximation of behaviors that the analyzer *may* be interested in. The analyzer, upon the execution of these behaviors, can then determine if the behavior of interest is actually present. In this scenario, the goal would not be complete code coverage but coverage of behaviors that the analyzer is geared toward.

We name this approach *targeted execution*, where the code exploration or driving tool uses knowledge of the analyzer's goals to execute specific parts of the application that are of interest. By shifting focus from full code coverage to specific targets, we can be more efficient by executing less code and more precise in triggering behavior that is difficult to reach, thereby increasing the effectiveness of the analyzer. We do not claim that targeted execution can supplant fuzzing, symbolic execution, or other coverage-based techniques. Targeted execution can work well in situations where the analyzer is focused on detecting very specific behaviors and wants to perform this detection accurately by triggering the behavior dynamically, such as detecting a specific a type of malicious activity or a particular vulnerable sequence of instructions. However, if the behavior of interest is prevalent throughout the code, a coverage-based tool would likely be more suited to the task. For instance, detecting crashes in Android Java applications (e.g. for permission-related exceptions) would require analyzing all of the instructions in an application to determine if they might cause a memory or runtime error and whether they are protected by a catch block that handles all of the possible exceptions that may occur. Dynamic analysis for this goal would require triggering most of the application's code base and a coverage-based tool would be more effective. However, targeted execution is suited toward security tools where the behavior of interest and its over-approximation only occurs in a few places in the code (e.g. access to sensitive information) and is often hidden or obfuscated, requiring more sophisticated dynamic triggering techniques.

In this thesis, we demonstrate the feasibility and effectiveness of targeted execution in a security setting. In particular, we show how novel combinations of both static and dynamic program analysis techniques can overcome the weaknesses of each to achieve targeted security analysis of mobile applications. Static analysis, while providing good code coverage, must trade off precision with scalability. We find that it can provide the required over-approximation of target behaviors through an imprecise search through the application's entire code base. Once the over-approximation is computed, precision

can be added as we analyze each instance of the target behavior; this is scalable as we have filtered the application's code base to specific locations that we are interested in. To ultimately trigger each behavior instance at run-time, we use this precision to compute path constraints defining the input values that when injected, trigger the location where the target behavior is located. As execution is now restricted to the target paths and locations of interest, we further show how dependencies on other parts of the application can be resolved in order to execute these specific target code locations. Finally, information gathered during the dynamic driving can then be fed back into static analysis, enabling greater precision and completeness through the addition of precise dynamic data. This process can iteratively deobfuscate an application to achieve more complete and effective security analysis of applications.

1.3 Contributions

We make three key contributions in this thesis, which form the basis of Chapters 3–5.

- We propose the idea of targeted execution and show how it can improve the effectiveness of dynamic analysis tools, particularly those performing security analysis. We design and implement IntelliDroid [123], a targeted execution framework for Android applications using hybrid program analysis techniques to over-approximate and refine detection of interesting code behaviors. Using IntelliDroid, we show that targeted execution can improve the effectiveness of existing Android analyses, such as privacy leakage detection.
- We describe the challenges of restricting execution to target code paths and locations of interest and show how dependencies on other parts of the application can be resolved through the use of contexts, which represent the constraints a path imposes on the program state it is dependent upon. We design and implement Car [125], which approximates and refines path contexts to enable more effective execution of target code. We show that Car is able to trigger deep, difficult-to-reach behavior in a variety of large and complex applications popular on the Android platform, further extending the capabilities of targeted execution.
- We extend targeted execution into a fully iterative hybrid framework, Tiro [124], where targeted execution drives dynamic analysis that then feeds information back into static analysis, incrementally refining the analysis results. We use our hybrid framework to investigate obfuscation in Android applications and iteratively deobfuscate certain forms of obfuscation. Through the analysis performed with Tiro, we discover a new form of obfuscation in Android called runtime-based obfuscation, and find that it is prevalent in current malware.

1.4 Outline of thesis

We begin in Chapter 3 by exploring the concept of targeted execution and its uses in security analysis. We present IntelliDroid [123], our initial work into hybrid targeted execution of Android applications. Through IntelliDroid, we show that targeted execution can be achieved using a combination of static and dynamic analysis techniques. We also show that through targeted execution, we can achieve greater effectiveness in detecting malicious behavior in applications, such as privacy leaks.

In Chapter 4, we refine how targeted execution is performed and address the challenges of restricting execution to specific target paths in large, complex applications where there are numerous dependencies

between different parts of the application. We describe how these dependencies can be resolved through the use of contexts, which represent the program state required by a path as it is executing. We present Car [125], which approximates and refines a target path's context to increase coverage of target code while maintaining reasonable soundness in the paths that are executed. We show that Car is able to greatly increase the effectiveness of targeted execution and trigger complex code paths to uncover sensitive behaviors in a variety of benign and malicious applications, enabling widespread analysis of all applications.

Finally, to complement the flow of static information to guide dynamic analysis, we show how dynamic information can be fed back into static analysis to improve its precision and completeness, resulting in more effective targeting. We present Tiro [124] in Chapter 5, a framework that extends the ideas in IntelliDroid and Car to create an iterative hybrid target execution tool through a novel combination of static and dynamic techniques. We show how information from both static and dynamic analysis can be used to incrementally refine the capabilities of both. We use Tiro to tackle the problem of obfuscation and show how it is able to deobfuscate sveral common forms of obfuscation used in Android malware. We also use Tiro to uncover a new type of obfuscation that subverts the runtime on which applications are executed and hides malicious code from security analysis tools.

We begin with Chapter 2, which provides background information on the Android system and common program analysis techniques. We also describe the current state-of-the-art research in the security analysis of Android applications.

Chapter 2

Background

In this chapter, we provide an introduction to the Android operating system and how Android applications differ from traditional programs on desktop machines. We also present a background on program analysis techniques, including static and dynamic analysis. Finally, we describe how prior research use these techniques for the security analysis of Android applications.

2.1 Android operating system

The Android operating system is built on top of the Linux kernel. Applications are written in Java or Kotlin [1], a programming language similar to Java, and compiled into a custom DEX bytecode format. Applications can also include native code that is invoked through the APIs provided through the Java-Native Interface (JNI). Each application runs in its own process and relies on Linux processes and user permissions to securely separate different applications. In addition to the application and underlying system, there is an Android framework layer that facilitates the interaction between the application and the hardware sensors included on the device. Hardware events are propagated (and sometimes stored) in the framework between the sensors, the system/kernel, and the application. In Figure 2.1, we show the execution of an application in the context of the Android framework, JNI interface, and the underlying system.

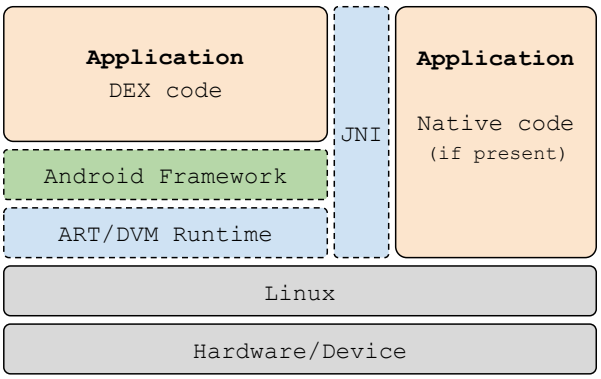


Figure 2.1: System diagram for the execution of an application on an Android device

[1] Kotlin homepage: https://kotlinlang.org/

2.1.1 Event-driven architecture

Android applications are event-driven. Unlike traditional desktop programs, applications are developed with a set of *entry-point* methods where the application receives events from the Android framework and can react to them. Applications make heavy use of these event callback/handler methods, which must be registered by the application with the Android framework in a number of different ways specific to the type of event (e.g. callbacks for location, SMS, and the various types of UI events are registered with the framework differently). There are several categories of events that can be received by an application, described below:

Lifecycle: Android applications consist of four types of core components: activities, services, receivers, and content providers. The main component is the activity, which programmatically represents a screen in the application's user interaction (UI) flow. Each application contains a main activity that is declared in the application's manifest file (included in the APK package containing the application's binary) and is triggered when the user starts the application. A series of events is delivered to the application when activities are started for different phases of the activity's start and termination processes (e.g. `onCreate`, `onStart`, `onStop`, etc.). Activities can start other components through *intents*, which are the main form of interprocess communication (IPC) facilitated by the framework. Service components are long-running processes that provide background functionality while content provider components respond to queries for specific types of data that the application can provide (similar to a custom database). Each type of component has a list of similar lifecycle events that are triggered by the framework and received by the application. Components can also be started by other applications if they are declared as `exported` in the manifest file.

Receiver: Broadcast receiver components are triggered by the framework when device events occur, such as the receipt of an SMS or Bluetooth message. Receivers must be explicitly registered with the framework so that the framework is aware of the events for which an application wishes to be notified. The registration can occur statically by declaring the receiver in the manifest file or dynamically through the `registerReciever()` API method. A receiver component can determine the exact event that triggered it through an intent action string that is passed as an input to the receiver. Receiver components can be triggered even if the application is not currently running on the device.

Framework callback: Framework callback methods are another means by which the Android framework can notify the application about hardware events, though they are only delivered for a running application. Applications must register the callback method programmatically with the framework using the APIs provided. These registration API methods and the callback objects/methods that are triggered on an event vary for different types of events (e.g. location, telephony, etc.).

User interaction (UI): UI callback methods are technically a type of framework callback but we treat them separately due to the relationship they have with on-screen elements and the separate event processing flow they follow in the framework. When an on-screen event occurs as a result of a user action, such as a click or a swipe, the framework processes the screen inputs to determine the precise action that occurred and the event that should be generated as a result of the sequence of screen inputs (e.g. click, long click, swipe, etc.). The framework also determines the screen

element to which the inputs were directed using a hierarchy of UI elements it maintains. It then delivers the UI event to the application enclosing that screen element. UI callback methods must be registered with the framework, which can occur statically in a UI layout file in the APK package or dynamically through UI API methods. Layout files, in turn, are dynamically associated with activities, which load them when the activities are created to display the appropriate (possibly region- or language-specific) screen to the user.

Applications may contain any number of components and event handlers. Core components, such as the main activity that is started when the application is loaded, are declared in the application's manifest file included with the application's bytecode. The manifest file, bytecode, and other resources required by the application are packaged into an APK file, which are distributed and installed on user devices. Other components and event handlers in the application can be declared and registered dynamically.

2.1.2 Permissions

Access to most non-trivial functionality provided by the Android framework are controlled through Android's permission system. Applications declare the permissions required by their application in their manifest file, which allow them to request sensitive user information (such as the user's location or a telephony identifier) and sensitive device functionality (such as the camera or file storage system). They cannot access the sensitive data or functionality unless the user grants the associated permission. A permission controller service in the framework enforces these accesses. If the application tries to access functionality for which they have not been granted permission, a permission exception will occur. The documentation of the different permissions is piecemeal and spread throughout the documentation for different Android services. Several past works [2, 11, 16, 42] have performed static and/or dynamic analysis of the framework to automatically extract a mapping of permissions to the framework APIs that require them.

Starting from Android 6.0, a run-time permission system was introduced to provide context to the user for the permission request. Prior versions required users to grant all request permissions when an application is installed. The run-time permission system requires applications to request permissions when the application is running, which can provide users with a better understanding of how the associated sensitive data or functionality will be used. Users can also revoke permissions later if they no longer wish for an application to access a particular device resource.

2.1.3 Android runtime

Each application runs within the Android runtime, which is similar to a Java virtual machine (JVM). Most applications are written in Java and compiled to bytecode in the Dalvik Execution Format (DEX) [35]. Early versions of Android used the Dalvik Virtual Machine (DVM) to interpret and execute DEX bytecode. An alternative Android RunTime (ART) was introduced in Android 4.4 and became the default runtime environment starting in Android 5.0. DEX bytecode in ART can be precompiled into native code (either ahead-of-time or just-in-time) for performance optimization. When executing an application, ART may be interpreting the application's DEX bytecode ("interpreter" mode) or executing precompiled native code ("quick" mode). Applications can also include native code libraries and ART facilitates the transition through the Java Native Interface (JNI), which provides a set of APIs for transitioning between bytecode originating from Java source code and the native library.

ART maintains all of the state for the DEX code files, classes, methods, and DEX/native code pointers for an application. The runtime environment is not secure, as any native code in the application is executed outside of the runtime environment and can modify runtime state (Figure 2.1). Runtime API methods (e.g. for reflection, a feature of the Java language), can also allow applications to access and modify the code elements that are currently loaded and/or executing.

2.2 Program analysis

Program analysis techniques have been used extensively in the past to perform security analysis of applications. This analysis includes detection of vulnerabilities and analysis of malicious or suspicious activity.

2.2.1 Static analysis

Static analysis is performed on the bytecode of the application. The simplest form of analysis may search for specific markers in the code; for instance, because sensitive functionality in Android is accessed through framework API invocations, one may search an application's code for invoke instructions to a set of known framework methods. However, application binaries may sometimes contain code that is never executed dynamically—this normally occurs when the application includes a third-party library and uses only a portion of the library's functionality. The unused code is called *dead code* and can result in false positives if it contains the behavior being analyzed but is not actually part of the application's execution. To avoid this issue, a model of the control flow of the application can be extracted to determine whether a piece of code is reachable during the application's execution.

Control-flow analysis requires an *entry-point* location at which the flow analysis should begin. For a method, the entry-point is the first instruction. For an application, the entry-point is the location at which the underlying system begins executing application code. In desktop applications, this is the `main` method. However, because Android applications are event-driven, there are a number of methods where the Android framework may begin execution and when analyzing an application, the paths stemming from these entry methods may appear unrelated. We consider all Android event types (lifecycle, receiver, callback, UI) implemented by an application to be entry-points if they are registered with the framework at some point during its execution.

The control flow from the entry-point location(s) in an application is traditionally represented by a graph. For a method, an intraprocedural control-flow graph is usually generated in which nodes are basic blocks (i.e. a continuous straight-line sequence of instructions with one entry and one exit point) and edges are transitions between basic blocks that might occur during execution. To represent the control flow of the application as a whole, methods can be connected through a call-graph, which models the method invocations an application may execute. Nodes represent methods and edges represent invocations between methods.

Construction of the call-graph of a Java application requires parsing method invocation instructions to determine the target method of the invocation. We refer to the components of an invocation instruction as follows:

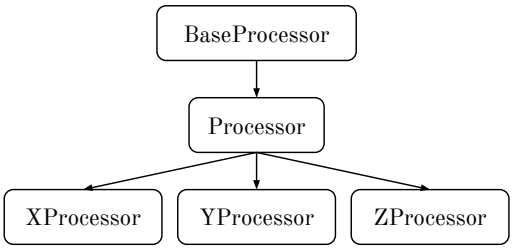
`invoke <declared invocation method> <receiver (for non-static invocations)> [arguments]`

The process of determining the target method (i.e. the method that is actually invoked and executed)

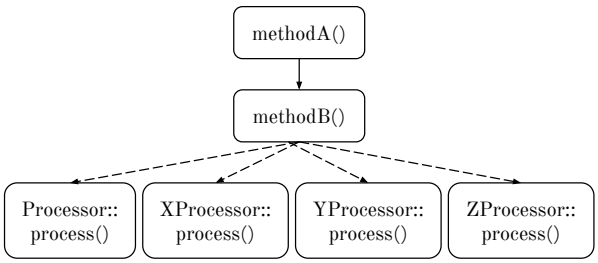
from the declared invocation method may be imprecise due to the use of polymorphism in applications. For instance, an invocation to a virtual method of an abstract class or interface can potentially be received by any subclass or implementer object. Furthermore, if a subclass overrides the method, the invocation can technically be received by any one of a number of methods. For completeness, a call-graph can be conservatively constructed such that edges are created for all possible targets of an invocation, even though some targets may never occur dynamically for that specific invocation instruction. Greater precision in the determination of the target method can be achieved by analyzing the data-flow for the receiver object in non-static invocation instructions (i.e. the `this` parameter passed to a class instance method). Type analysis of the register holding the receiver object can filter out some of the target possibilities (for instance, an invocation instruction might only specify a virtual/abstract method and class target but the receiver variable could have been declared with a more specific subclass type).

```
void methodA() {
  Processor xp = new XProcessor();
  methodB(xp);
}
void methodB(BaseProcessor p) {
  p.process()
}
void otherMethod() {
   Processor zp = new ZProcessor();
   ...
}
```

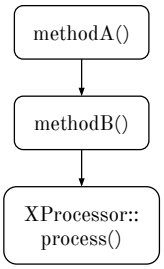
(a) Code example



(b) Class inheritance hierarchy



(c) Conservative call-graph from the class hierarchy



(d) Call-graph from points-to analysis

Figure 2.2: Effect of type and points-to analysis on generating call-graph edges

Type analysis can be performed in several ways. In Figure 2.2, we show how these techniques can affect the generation of the call-graph. A common technique is class hierarchy analysis (CHA), which represents all of the classes in an application with an inheritance tree. For a given class, it is trivial to determine its parent, implemented interfaces, ancestors, and direct and indirect subclasses. For a variable declared with a specific type, class hierarchy information can be used to determine the types of objects that can populate the variable (Figure 2.2c) A more precise form of type analysis can also be obtained through points-to analysis, which models the objects referred to by pointers and references in the application's code. Such a mapping of pointers to objects can help de-alias data when performing static analysis; for instance, if a pointer definitively refers to two different objects at two locations in the application, the analysis will know that there is no data flow between those locations, which can narrow down possible invocation targets (Figure 2.2d) A common method of performing precise

points-to analysis is to represent each allocation site (i.e. `new` instruction) with a modeled object and create edges from pointers to objects to represent the flow of data in the application's code (e.g. through assignment instructions). Because the flow of data across methods depends on the invocations performed by an application, the points-to analysis may rely on the call-graph. In addition, given an invocation instruction, the points-to data about an invocation's receiver pointer (i.e. the possible receiver objects) can narrow down the targets of the invocation. If points-to type information is used in the construction of the call-graph, the call-graph generation and points-to analysis can be iterative to achieve complete analysis, as in the Spark module [68] in the Soot static analysis framework [116].

The precision of a call-graph is also affected by the modeling and potential aliasing of methods. The control flow taken by a method may depend on how it is invoked. Two different invocation sites may trigger different paths in the same method, which in turn may subsequently invoke different methods. The representation of these method invocations within the call-graph can change the structure of the graph. If each invocation to a method was represented separately for each code path, the resulting call-graph would be exponential in size with respect to the number of methods in the application, which is infeasible for the analysis of most non-trivial applications. Instead, nodes in a call-graph represent a method within a particular invocation context. If no context is used (i.e. a context-insensitive call-graph), each method is represented by a single call-graph node and all invocations to that method will be represented by an edge to that node. If there are "bottleneck" methods used by many different paths in an application (e.g. a dispatching method, as shown in Figure 2.3), the resulting call-graph may alias different control and data flows, and incorrectly show many possible intersecting code paths when they are actually separate. Using one level of context will differentiate methods based on their caller method; for instance, if method `handleEvent()` was invoked by `methodC()` in one location and by `methodD()` in another, the two invocation contexts will be represented by different call-graph nodes, allowing the control-flow paths for each to be differentiated and separated in the resulting call-graph. Subsequent levels of context will further differentiate between the calling paths of each method. Greater context sensitivity in the call-graph will increase precision, as different code paths in the application can be represented individually, but will exponentially increase the memory requirements (to store the call-graph nodes) and analysis time (to analyze the larger call-graph).

The primary drawback of static analysis is the trade-off between imprecision and performance, as some information may only be known precisely at runtime. Furthermore, modeling the application's execution with a high degree of precision is resource intensive. When imprecision occurs, the analysis must choose to be complete (i.e. model all possible paths) and imprecise, or precise and incomplete (i.e. model only known paths). The trade-off between performance, precision, and completeness that must be made will depend on the ultimate goal of the static analysis and its operational constraints.

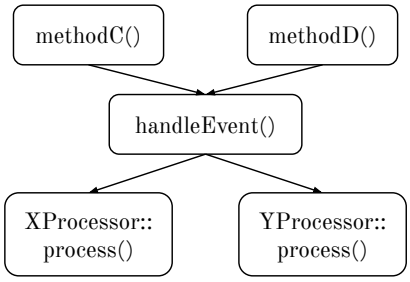
2.2.2 Dynamic analysis

Dynamic analysis is performed as the application is executing using data from its execution environment. This may include analysis of the system calls or framework methods invoked by the application, or the resources (e.g. files, network accesses, etc.) used by the application. Because dynamic analysis tools have access to run-time information from the application's execution, it is generally more precise than static analysis as it analyzes instructions and paths that are known to be executable and reachable in the application.

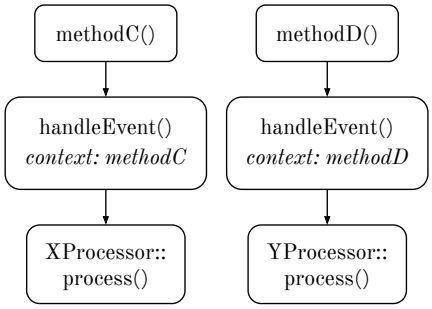
The main weakness of dynamic analysis is that only code that has been executed can be analyzed.

```
void methodC(XProcessor xp) {
  handleEvent(xp);
}
void methodD(YProcessor yp) {
  handleEvent(yp);
}
void handleEvent(BaseProcessor p) {
  ...
  p.process();
}
```

(a) Code example



(b) Call-graph with no context sensitivity



(c) Call-graph with one level of context sensitivity

Figure 2.3: Aliasing of interprocedural control and data flows due to context sensitivity

It can be difficult to automatically explore all of the code in an application. Unexplored code can lead to false negatives, where the analysis misses behavior or actions taken by the application, resulting in incomplete results. As such, automated dynamic code exploration and test generation is an active field of research.

2.2.3 Dynamic code exploration

The two primary dynamic code exploration techniques are fuzzing and symbolic execution. Fuzzing tools, such as AFL [137], inject mutated inputs to trigger different code paths in an application. Inputs that trigger new code (usually determined by measuring the amount of new code coverage an input discovers) are retained and mutated further to try to reach more code. Fuzzing achieves good coverage of surface code in an application but has trouble penetrating paths where specific values are required for certain branches. While fuzzing explores new paths randomly, various techniques have been proposed to direct fuzzing toward branches of interest through some form of auxiliary analysis, such as taint tracking to determine how the injected inputs influence the execution of different paths [29, 47, 57, 72, 100, 118]. This guiding is not explicit targeting but acts more as hints for prioritization of which paths to explore and which input bytes to mutate based on features in the application's code or execution.

In contrast to the randomness of fuzzing, symbolic and concolic execution [5, 28, 52, 106] are alternate techniques for code exploration and test generation that systematically explore each branch in an application. They track how inputs are manipulated and used in conditional branch instructions to extract a set of constraints on input values that represent a path's execution. To explore untaken branches, specific constraints are reversed and solved to generate a new input value that will trigger a new branch

outcome. When a particular behavior of interest is found, such as a crash, the inputs used to trigger the path serve as a test case to reproduce the behavior. In general, symbolic execution is extremely expensive to perform as input values and heap state must be tracked symbolically to extract the full conditional branch constraints. Furthermore, symbolic execution suffers from the path explosion problem, in which the number of branches and paths to be explored grows exponentially as more code is discovered. For long execution paths, the large number of constraints that are extracted is also resource intensive for the constraint solving process, which is exponential with the number of constraints (i.e. size of the search space). Concolic execution can alleviate some of this burden by substituting some of the symbolic state with concrete values, thereby reducing the search space.

To address the performance issues of symbolic execution, a variety of techniques have been proposed to filter the paths that should be explored. These techniques include skipping the symbolic execution of parts of the paths, such as omitting the callers of arbitrary methods in under-constrained symbolic execution [96] or bypassing the execution of seemingly irrelevant functions in chopped symbolic execution [115]. Other works try to guide the symbolically executed path using information from a pre-computed static analysis [15] that extracts interesting paths. Another technique to mitigate the weaknesses of both fuzzing and symbolic execution is to run them in conjunction, such as in Driller [110], which fuzzes applications to explore many paths quickly within a section of the application. When fuzzing cannot penetrate a particular input requirement and there is no new code coverage, symbolic execution is used to extract the input constraints required for further progress and to generate the inputs to explore a different section of the application.

2.3 Android application security analysis

On the Android platform, a variety of static and dynamic tools have been developed to analyze the actions taken by an application and detect whether the application exhibits undesired or malicious behavior. Security analysis on Android differs from previous work on x86 and desktop applications in several ways:

- Most Android applications are implemented in Java and compiled into DEX bytecode, which is executed by the Dalvik Virtual Machine (DVM) or the Android RunTime (ART). While precise static analysis of x86 binaries can be fairly challenging, the contextual information and structure of DEX bytecode helps static analyzers reason about the code more easily and enables more complex analysis to be performed. A number of existing Android analysis tools analyze the bytecode of applications directly (i.e. without requiring source code) using existing support for DEX within the Soot static analysis framework [116] or by converting the bytecode in Java bytecode [37, 85] and analyzing the Java bytecode using frameworks such as WALA [120]. A handful of analysis tools for Android also support the analysis of application native code in conjunction with DEX bytecode [114, 121], though most do not.

- Android applications are event-driven and code is executed in response to events generated by the Android framework when user input or sensor events are detected. Unlike traditional applications that have a single `main()` method where execution begins, each callback method is an entry-point into the application's execution. Effective analysis of an application must find and analyze each entry-point to achieve sufficient code coverage.

- Users trust their mobile devices with a great deal of sensitive information, such as contact information, private messages, and location data from the GPS sensor. A particular concern in the Android security space is how this sensitive data is used once it has been accessed by the application and whether private information is being sent to third parties without user knowledge or intention.

2.3.1 Malware detection and classification

Malware targeting Android devices are mainly geared toward exploitation of the services provided by the device hardware, though some forms of malware (such as command-and-control malware and ransomware) are similar to that seen for traditional computer systems. A comprehensive manual analysis of Android malware was performed by the Android Malware Genome project [144], which collected and characterized 1200 malware samples, sorting them into 49 malware families. The dataset is skewed toward SMS-based malware, where SMS messages are used for command-and-control schemes and for monetary gain (by sending unintended messages to paid SMS services). The authors also found that malicious code can be repackaged with benign applications to trick users into installing them. This dataset was heavily used in subsequent research in Android security to evaluate the efficacy of malware detection and analysis tools. Other manually curated and classified malware datasets include the Contagio [90] dataset, which includes a variety of malicious activity, including ransomware, and the Creepware dataset [103], which focuses on applications used in interpersonal attacks. Furthermore, the VirusTotal collection [117] includes malicious applications that have been submitted and scanned by a number of existing antivirus tools.

Early works in automated Android malware detection include Kirin [41], which enforces security policies based on the combination of permissions requested and used by applications. Other tools determine malicious activity in applications based on a variety of features, such as permission usage [140], system call invocations [25], system resource usage [107], and external accesses [66]. TriggerScope uses the existence of "logic bombs" as an indication of malicious behavior, where logic bombs are complex constraints or conditions on input values that must be met before malicious activity is triggered (for example, the comparison of specific command strings from a controller server sent to a botnet of mobile devices). It employs static analysis to detect complex conditional constraints that dominate access to sensitive data or functionality.

The use of machine learning to classify malware based on code features was proposed in Drebin [9]. Drebin uses statically derived features (such as application components, requested permissions, invoked APIs, network addresses, etc.) to classify malicious applications. It achieves good performance, although its false positive rate of 1% still poses issues for large marketplaces where thousands of applications would still have to be manually analyzed for malicious activity. Subsequent works in using machine learning include MaMaDroid [88], whose classification is based on static sequences of API calls, and DroidEvolver [130], which proposes the use of online learning to update the classifier in the face of evolving malware and concept drift [62]. While machine learning is a promising technique for identifying malicious activity and malware, their effectiveness relies on the quality and completeness of their classification features, which are extracted either through static or dynamic analysis of applications.

More general analysis tools have also been proposed, with the goal of detecting and deciphering application activity, which can help in determining whether that activity is malicious or benign. DroidScope [132] uses a QEMU-based emulator and API hooking techniques to provide a dynamic analysis

framework that can support a variety of different analysis goals. CopperDroid [114] is a similar emulator-based tool that analyzes the system call trace of an application to reconstruct a higher-level semantic view of the application and its interprocess communication with the Android framework. Because it relies on the low-level system call trace, it can analyze both Java and native code executed by the application.

2.3.2 Sensitive information leakage

Due to the volume and sensitivity of data stored on mobile devices, a common target of Android security analysis is the transmission of private or sensitive data to third parties, which may be unintended and/or unknown to the user of the application. The use of permissions, which guards access to this data, has been studied by tools such as Stowaway [42], PScout [11], Axplorer [16], and Arcade [2], which use static or dynamic analysis of the Android framework to map API methods to the permissions they require. Similarly, AndroidLeaks [50] uses a statically extracted permission-API mapping and static taint tracking analysis of applications (i.e. tracking of data from sources of sensitive information to locations where the data may leave the device) to detect leakage of private data obtained from sensitive APIs. FlowDroid [10] is another static taint-tracking tool that reports leakage of private information using a combination of forward and backward propagation to track data flow in a scalable manner. To identify sources and sinks of sensitive data, it uses a pre-configured list of API methods where sensitive data is obtained and where data is sent from the device. Related work such as Epicc [86] and Amandroid [122] propose techniques to perform static data-flow analysis across application components, which require special support since most analysis is performed on application code and miss the flow of execution through the framework code that triggers different components. Enabling intercomponent analysis results in more complete taint and data tracking (e.g. IccTA [69]). Similarly, Apposcopy [43] uses static semantic-based signatures of control and data flow in applications to classify malware and detect sensitive information leakage. To further improve the completeness of static taint tracking, JN-SAF [121] extends the propagation of taint through JNI method invocations and into native code in the application.

On the dynamic side, TaintDroid [40] and TaintART [113] provide taint-tracking on the DVM and ART runtimes, respectively, by instrumenting source and sink methods and reporting instances where tainted sensitive information reaches a sink method. Hybrid tools such as AppIntent [134] and AppAudit [127] use a combination of static analysis, symbolic execution and/or approximate execution to find and verify privacy leaks. AppIntent also analyzes a potential privacy leak within the context of the event(s) that triggered the information flow, with the goal of identifying malware that leak sensitive information without user intent.

2.3.3 Vulnerability detection

A number of tools have also been developed to detect vulnerabilities and avenues of attack in Android applications. Although sensitive functionality is protected by the permission system, applications that have been granted a permission may not protect this access sufficiently or may contain vulnerabilities that expose them to confused deputy attacks. CHEX [74] statically analyzes application components to detect whether they are vulnerable to hijacking through Android's intent system, which passes messages between application components and between different application processes. ContentScope [61] performs a similar analysis to detect misuse of content provider components, using a combination of static analysis

to find vulnerabilities and dynamic analysis to verify them. Brahmastra [20] uses hybrid techniques to drive the application's UI to trigger third-party application components (i.e. libraries) to analyze possible vulnerabilities within third-party code, which is granted the same permissions as its enclosing application.

2.3.4 Obfuscation

Malware development and security analysis are two opposing parties in an arms race: when a new exploit is developed, security analyzers eventually notice and develop tools to detect the malware to protect users; conversely, as the analysis tools become increasingly better at detecting the malware, the malware developers try new ways of hiding the actions taken by their code. Malicious code in applications is commonly obfuscated in an effort to hide it from automated or manual analysis so that it can have as much impact as possible on its intended target before being detected. Obfuscation occurs in both desktop and Android applications and can limit the effectiveness of both static and dynamic analysis techniques.

To combat static analysis, malware can try to exacerbate static imprecision by making it difficult resolve a model of the application without dynamic information. Techniques that hide code (e.g. through dynamic code loading) or aspects of the code (such as reflection to hide method invocation targets) can limit the completeness or precision of the static call-graph, resulting in false negatives and missed potential malicious activity. For instance, static taint tracking tools that cannot propagate taint across a reflected invocation may incorrectly report that no privacy leakage occurs. Privacy enforcement based on this analysis would also be insecure.

For dynamic analysis, code coverage is often the main cause of false negatives since malicious behavior cannot be analyzed if it is not first executed. Malware may try to prevent the execution of certain actions during analysis and only enable them when an unsuspecting user is operating the application, preventing the detection of the malicious behavior. These behaviors can be hidden or obfuscated by guarding them behind complex constraints on the input values that can trigger the behavior.

To address the issue of obfuscation in Android applications, there have been recent works exploring obfuscation-resilient analysis and malware detection. RevealDroid [48], uses machine learning techniques to train and select features that can classify malware correctly despite the use of obfuscation, with some success on their limited dataset. Agrigento [33] uses black-box differential analysis to detect whether outgoing network messages contain private information (i.e. privacy leakage). However, as with other dynamic analysis tools, it is limited by poor code coverage and it cannot detect other forms of malicious activity.

2.3.4.1 Types of obfuscation in Android

Before describing recent works in deobfuscating Android applications, we first describe some of the obfuscation techniques commonly used to hide malicious activity from security analysis. The simplest form of obfuscation is the renaming of classes, methods, and fields through compilation tools such as ProGuard [56]. While this makes manual analysis of applications more difficult, it does not hinder program analysis, which is based on the semantics of the application code. We focus primarily on obfuscation that affect static or dynamic analysis of Android applications.

Reflection is a feature in the Java language allowing introspection into the class, method, and field

objects as the application is running. Its use can cause imprecision in static analysis since reflective method calls can be made by specifying a dynamically resolved target method name that may not be available statically, making it difficult to determine the invocation target and leading to missing edges in the static call-graph. Similarly, class and field objects can be obtained reflexively using a target name, causing imprecision in any type-based analyses and in points-to analysis. Reflection can also be used legitimately, and it has been found that reflection is used by benign applications for code generality, backwards compatibility, and protection of internal code and intellectual property [70].

While native code in Java applications is not necessarily a form of obfuscation (for instance, native code is often used in performance-critical code), its use in Android applications foils most existing static analysis tools as they are generally built on frameworks that support the analysis of only Java or DEX bytecode; therefore, while invocations to native methods are visible, the actions taken by native code are not within the analysis scope. Even if a separate analysis were to operate on the native portions of an application (using existing x86 or ARM analysis tools), special support must be added to analyze the flow of data between the Java and native components over the JNI interface. Native code usage in Android applications has been studied [4] and a statistically derived sandbox was proposed to limit system call capabilities of application native code while incurring minimum impact on benign applications.

Code packing has long been used by x86 malware to hide code from analysis. The unpacking process normally begins by retrieving an encoded binary file from the file system or from a third-party server, decoding the file to generate valid code, and dynamically executing this previously hidden code. Polymorphism can also be incorporated into the encoding/decoding process such that the unpacked code is different (but functionally equivalent) each time it is unpacked. On Android, the runtime allows applications to dynamically load code through the DEX class loader [141,142]. Applications can use this feature to apply updates without resubmitting their application through a separate distribution channel (e.g. the Google Play application marketplace). However, when performing static security analysis, because the code is dynamically loaded, it is not visible during analysis and any malicious activity performed within it would result in false negatives. Furthermore, if the code is packed or encrypted (and unpacked before dynamic loading), it can be difficult to retrieve the valid bytecode as it is often deleted from the file system immediately after it is loaded.

Another obfuscation technique is method hooking [34,138], which obfuscates an application's execution by redirecting method invocations and executing an unexpected set of instructions on these method calls. Method hooking can be achieved by modifying metadata in the method objects stored within the runtime, as with the ZHookLib framework [138], or by modifying the virtual method table in class objects, as in ARTDroid [34]. These techniques allow fine-grained instrumentation of specific methods, which can help implement dynamic analysis of applications. However, they can also be used to obfuscate method invocation patterns.

The previously listed obfuscation techniques primarily affect the completeness of static code analysis but application behavior can also be hidden from dynamic analysis by cloaking them under specific execution conditions that are only triggered in certain circumstances. This was the motivation of TriggerScope [44] in using "logic bombs" to identify possible malicious activity; however, since TriggerScope is a static tool, its ability to detect these complex triggering conditions is hindered by the static obfuscation techniques mentioned previously. The dynamic conditions enforced by malicious applications can form anti-analysis checks that ensure that any offending malicious activity only occur on user devices and not when an application is under scrutiny by a security analyzer (such as when applications

are being vetted by an application marketplace). In addition to the trigger conditions, they may also try to fingerprint their execution environment to determine if they are running on an emulated device (common for security analysis) or whether their binaries have been modified (e.g. instrumented to aid code analysis). The code coverage issue in Android dynamic analysis is explored by some works, such as FuzzDroid [99], which incorporates multiple fuzzing and input injection techniques, and applies machine learning to determine which technique should be used in a given context. However, many of its input injection techniques rely on static analysis that does not handle obfuscation.

Most of the obfuscation techniques seen in Android applications are derived from obfuscation techniques in x86 and Java. The effect of obfuscation against program analysis has been explored for x86 and Java applications [31,32], with the goal of hardening a program against intellectual property theft. The impact of obfuscation on security analysis and its effectiveness against traditional signature detection has also been explored [87] and it has been shown that many commercial static analysis tools fail to handle obfuscated malware [84]. Following works that analyze the effects of obfuscation have found that while obfuscation is effective against most program analysis tools, code transformations that preserve program semantics is less effective against symbolic execution [17]. However, symbolic execution is expensive and there does not exist a full implementation of symbolic execution on Android as of yet. Furthermore, research into dynamic analysis obfuscation through the use of complex conditionals involving hashed or encrypted values (i.e. "secure triggers" [46]) has shown that symbolic execution analysis suffers when trigger-based obfuscation is employed [108].

2.3.4.2 Deobfuscation

Deobfuscation tools aim to reconstruct the actions taken by an obfuscated application to improve manual or automated analysis. Often, deobfuscation involves retrieving the code executed by an application, either in its static form or as an instruction trace. Some obfuscation techniques hide data values through encryption or obfuscated method invocations, so deobfuscation may also require reporting values that are only available at run-time.

Several tools have been proposed for the deobfuscation of Android applications, with most focusing on reflection and dynamic code loading. DroidRA [70] uses static analysis to deobfuscate reflection targets through constant propagation of string values. While it handles very simple forms of dynamic code loading (where the loaded code is stored unencrypted in the application package), this support is insufficient for most modern cases of code packing. In addition, encryption of the target string for a reflective call will make static resolution infeasible. Harvester [98] uses static code slicing to identify paths leading to specific code locations, such as reflection invocations. These slices are then executed dynamically by instrumenting the application to force the branch outcomes required for the path and triggering the obfuscation location, where the deobfuscated value is then logged. StaDynA [142] uses a hybrid iterative approach to deobfuscate reflection and retrieve dynamically loaded code for further static analysis, using instrumentation of specific reflection and dynamic code loading APIs. Some Android unpackers, such as DexHunter [141] and Android-unpacker [111], handle certain cases of manipulation of DEX files and DEX bytecode, but use special packer-specific values to identify the code that must be extracted, making them easily circumventable. They also do not handle any other form of obfuscation, which makes it difficult to analyze the retrieved code if it is further obfuscated in another way. Others, such as PackerGrind [131] and AppSpear [60] have a more general design but their ability to extract hidden DEX bytecode is limited to their instrumentation of specific API methods they expect

obfuscation code to use. DroidUnpack [39] uses full system emulation to dynamically extract packed code by instrumenting the runtime to detect execution of previously written sections of memory, similar to Renovo's [63] approach on x86. DeGuard [21] focuses on a different form of obfuscation than the previously described work—namely, name obfuscation for class, method, and field names, such as the transformations performed by the ProGuard [56] tool included with the Android SDK. DeGuard uses a statistical model to determine the likely names used in the original, pre-ProGuard version of the code, based on the actions the code performs and its interactions with library methods.

Because most Android applications are implemented in Java, general deobfuscators for the Java runtime can also apply for the Android runtime. TamiFlex [23] deobfuscates reflection by instrumenting the reflection classes loaded by the Java runtime, but does not handle other forms of obfuscation. It modifies the class loader in the runtime to instrument reflection-related classes and report target strings for reflective calls. Similarly, Ripple [139] also targets reflection but does so through static target string resolution (similar to DroidRA [70]), which is less precise and ineffective against target string encryption.

2.4 Context of this thesis

In this thesis, we address the difficulty of performing security analysis of Android applications in the face of obfuscation and anti-analysis techniques. We make use of both static and dynamic program analysis techniques in order to address their respective weaknesses. We propose the use of targeted execution to perform dynamic analysis of applications that focuses on locations of interest to a security analyzer. We show how static techniques can be used to guide targeted execution to bypass the complex constraints and triggering conditions a malicious application may employ to hide their activity from analysis. We then show how targeted dynamic analysis can also retrieve information to address static obfuscation. Similar to prior deobfuscation tools, our resulting hybrid analysis can uncover hidden application behavior and enable more effective security analysis and malware detection.

Chapter 3

IntelliDroid: Targeted execution for Android

In this chapter, we introduce the core idea of targeted execution, in which execution of an application is guided toward locations of interest. Rather than striving for full coverage of the application during dynamic analysis, our goal is to reach specific target locations within the application. These locations depend on the type of analysis being performed, with the goal that focusing resources on reaching these locations enables more effective analysis of the application. For example, for dynamic taint analysis to detect leakage of private user data, it would be worthwhile to target execution of the application to locations where private data is obtained by the application and locations where data is sent.

The motivation for targeted execution is that aiming for blanket code coverage wastes analysis resources that might be better used for the particular dynamic analysis being performed. This is in contrast to existing approaches that try to execute all of the code in the application, which we call coverage-based exploration. There are several common strategies for such coverage-based exploration. The simplest, but least effective, is to have a predefined script of common inputs that will be executed on the application under analysis. This not only has a very low chance of triggering the malicious behavior, but can be easily evaded by a knowledgeable adversary. A more sophisticated approach is random fuzzing [75, 135, 137], which applies randomly generated inputs on the application in an effort to trigger as many behaviors as possible. However, random fuzzing is inefficient, as it may generate many inputs that trigger the same code and behavior in the application. To address this, a recent and more effective technique is concolic testing [5, 51], which uses symbolically derived path constraints to exercise different paths in the application for each generated input.

However, none of these methods are ideal for triggering malicious behavior in applications because they are blind to distinguishing between code that performs the behavior that the dynamic analysis is trying to detect and code that does not. Thus, while concolic testing can efficiently achieve high code coverage, it will still waste many compute cycles by having the dynamic analysis analyze irrelevant parts of the application. Instead, we propose *targeted analysis*, which uses information about the dynamic tool in combination with static analysis of the application to generate a reasonably small set of inputs that will trigger the malicious behavior to be detected by the dynamic analysis. We implement and evaluate the concept of targeted analysis for detecting Android malware in a prototype we call *IntelliDroid*.

Naturally, it would be very difficult for a static analysis tool to generate only the exact inputs that are

needed to trigger malicious activity in an application, as this would imply that the static analysis tool is as precise as the dynamic tool at detecting malicious behavior. Instead, we need a reasonably accurate over-approximation of the behaviors that will be analyzed by the dynamic tool. IntelliDroid can then generate a small set of inputs that will trigger all of the code matching the over-approximation and allow the dynamic analysis to decide if it is actually malicious or not. For this approximation, IntelliDroid uses a set of "target" code locations that is specific to the dynamic analysis tool; in particular, we focus on triggering target API invocations and show that this can enable the targeted analysis of most existing dynamic tools. This design decision is motivated by the observation that most malicious behaviors, such as sending and intercepting SMS messages, leaking private information, or making malicious network requests, require the use of APIs [144] [1]. In addition, malware may obfuscate malicious activity using reflection or dynamic class loading. IntelliDroid can trigger the reflection and class loading APIs so that the dynamic tool can observe the resolution of the reflected calls or the behavior of the dynamically loaded classes.

It is crucial that IntelliDroid can generate inputs that trigger all target code locations. Most tools that combine static and dynamic techniques use the dynamic analysis to prune false positives generated by the static analysis [49, 61]; thus, if the dynamic phase is unable to execute a particular path, it only increases the number of false positives. However, if IntelliDroid fails to trigger a malicious behavior, this will result a false negative, with more serious security consequences.

IntelliDroid introduces two new input generation and injection techniques that enable it to trigger code paths on which previous Android input generation techniques would fail [20, 49, 61, 109, 134, 143]. First, Android applications do not have a single entry-point, but are instead composed of a collection of event handlers. It can be insufficient to call just the event handler that contains a particular API invocation. Instead, the event handlers need to be triggered in a particular order, and in some cases a "chain" of several handlers needs to be triggered with specific inputs. IntelliDroid iteratively detects such *event chains* and computes the appropriate inputs to inject, as well as the order in which to inject them.

Second, while previous work injects inputs at the application boundary [7, 61, 102, 134], this low-fidelity injection can lead to false application behavior because the application state is inconsistent with the Android system state. For example, to hide the presence of SMS messages from the user, an Android malware program could register an event handler for an incoming SMS, and then access and search the SMS content provider to delete the received message. Simply injecting the SMS notification at the application boundary will result in inconsistent behavior because the application expects the message to be in the SMS content provider database, but the Android framework itself has received no such message. IntelliDroid maintains environment consistency after input injection by injecting inputs at the lower-level *device-framework interface*, allowing all state in the Android framework to be automatically changed consistently. This high-fidelity input injection means that IntelliDroid can be integrated with essentially any dynamic analysis tool, including full system analysis tools such as TaintDroid [40].

IntelliDroid and its base hybrid program analysis techniques were initially introduced in a earlier Master's thesis [126] by the same author as this thesis. In the earlier work, the overarching design of using static analysis to guide dynamic execution was described and implemented. In this chapter, we present an extension of IntelliDroid that demonstrates why hybrid targeted analysis is an improvement on existing static-only, dynamic-only, or symbolic code analysis techniques. We begin by describing the

[1] Our own analysis shows this is true of more recent malware as well as the older malware in the cited study.

initial design of IntelliDroid and how we use static analysis to guide execution. We then describe how to identify target behaviors and apply targeted execution to different types of dynamic analyses, showing the versatility of targeted analysis for general analysis of applications. We further integrate IntelliDroid with an existing dynamic analysis tool that operates on concrete execution of the application and empirically demonstrate the value of targeted analysis against static- or dynamic-only techniques.

In summary, we make three main contributions in this chapter:

1. We present the design and implementation of IntelliDroid, whose initial design was proposed in a Masters thesis by the same author. IntelliDroid is an input generator that takes into account the malicious behavior a dynamic analysis tool is trying to detect. From a set of target locations in the application's code where these behaviors occur, IntelliDroid generates inputs that trigger these behaviors at run-time. We describe two novel techniques that enable IntelliDroid to trigger target locations with high execution fidelity: detecting event chains and device-framework interface input injection. These techniques enable it to effectively generate inputs that trigger 70/75 targets in a corpus of malware.

2. As an extension to the earlier Master's thesis work, we motivate the use of hybrid program analysis techniques and demonstrate the feasibility and utility of targeted execution in real-world malware analysis. We show how API invocations can serve as an over-approximation for malicious behaviors. This abstraction, and the ability to target concrete execution, makes IntelliDroid easy to use with a dynamic analysis tool. We demonstrate this by integrating it with the TaintDroid dynamic analysis tool. When run on a corpus of malware, we show that the combination of IntelliDroid with TaintDroid can trigger and detect all privacy leaks while obtaining better precision than a purely static privacy leakage detector and better coverage than untargeted dynamic-only techniques.

3. We show that IntelliDroid is cheap and fast, requiring only 138.4 seconds of analysis time on average to successfully generate inputs to trigger target code on a corpus of malicious and benign applications. We also show that IntelliDroid is able to avoid running approximately 95% of an application during dynamic analysis while still detecting all malicious behaviors.

3.1 Background on the Android framework

As we described in Chapter 2, the Android operating system consist of applications running within a runtime (i.e. the DVM or ART) on top of a custom Linux kernel. The Android framework forms the middle layer between applications and kernel, facilitating application access to resources on the device. It consists of a set of system services (i.e. background processes) that communicate with the device's hardware components, as well as classes that implement application programming interface (API) methods that third-party applications invoke. When an application is launched, a "zygote" process with a pre-initialized instance of the runtime and framework is cloned and execution is passed to the framework, which loads the application components and manages their lifecycle events. As the application is executing, it may register for notifications about certain hardware events, such as location events from the GPS sensor or SMS events from the cellular chip; the framework polls for these events and notifies the application through the registered event handler methods, which serve as entry-points into the application's code. A diagram of this process is shown in Figure 3.1.

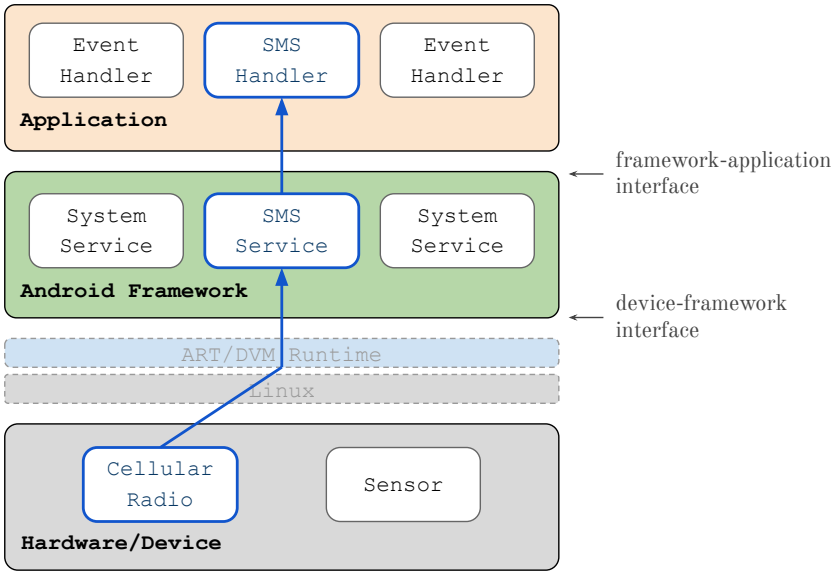


Figure 3.1: Hardware events delivered to application handlers through the Android framework

Together with the APIs that applications can invoke, these entry-point methods form the *framework-application interface* that divides the Android framework from code in third-party applications. At the other end of the framework is the interface between the framework and the underlying devices from which external events are generated, which we call the *device-framework interface*. When a device sensor receives new data, it notifies the framework so that the event data can be processed and disseminated to applications by calling the corresponding entry-points in the framework-application interface. The interface between device sensors and the framework differs depending on the type of sensor and may involve a direct invocation of a public framework method or a polling loop in a system service that waits for the event to occur.

In addition to invoking application event handlers, system services within the framework also store information about the event as it is processed. This allows applications to refer to the event and obtain extra information at a later time. Some services, such as the SMS service, store all past event information in a content provider database to be queried and modified by applications with sufficient permissions. Other services, such as location, store only the last event received. In both cases, the handler invocation in the application and the event information stored in the framework must be kept consistent for correct execution.

3.2 Initial design of IntelliDroid

We begin by providing a description of our initial design of IntelliDroid. A large portion of this section overlaps with our earlier work in the Master's thesis [126] that first introduced IntelliDroid. Since our extensions to IntelliDroid, as well as the work in the subsequent chapters of this thesis, build upon this design of targeted execution, this section serves as a background for our approach and provides design details that we will reference later.

3.2.1 Overview

IntelliDroid's goal is to generate inputs for a dynamic analysis tool that operates on the execution of an Android application. Given a set of target locations in the application's code that represent the behaviors that are detected and analyzed by the dynamic tool, IntelliDroid will find instances of these target locations in the application and generate inputs to trigger them. It aims to inject these inputs into an actual Android system, allowing IntelliDroid to be integrated with any dynamic analysis tool, including those that monitor application execution from an instrumented OS [40] or from a virtual machine emulator [114]. To accomplish this task, IntelliDroid takes the following steps, which are illustrated in the system diagram in Figure 3.2:

1. **Target path extraction**: IntelliDroid begins by identifying locations of target behaviors and for each location, it identifies the event handlers where execution in the application code can begin. From these entry-point event handlers, it finds the *target call paths* that lead to the target locations.

2. **Symbolic constraint analysis**: For each call path, it extracts symbolic path constraints that determine the input variable values required for the path to be executed.

3. **Dependency extraction and tracking**: In cases where the path constraints access and depend on variables that are external to the path, it determines the dependent application paths that would set these variables to the necessary values, extracts the constraints for these paths, and generates the *event chain* (i.e. an ordered set of dependent paths) that is required to resolve the dependencies for the execution of the target path.

4. **Constraint solving with run-time data**: For each dependent and target path, IntelliDroid employs an off-the-shelf constraint solver to solve their constraints to determine the necessary input values that will trigger the path dynamically. For variables that are external to the target path, such as responses to network requests, IntelliDroid dynamically extracts the concrete values for these external dependencies during execution.

5. **Input injection**: Finally, IntelliDroid applies the computed input values in the appropriate order to the *device-framework interface*. This will consistently execute the appropriate paths required for the target location to be executed. IntelliDroid contains a modified Android OS that can inject inputs at the device-framework boundary.

3.2.2 Identifying paths to target locations

For a given application and set of target behaviors, IntelliDroid first performs static analysis to identify the locations of these target behaviors and the paths leading to them. We defer the discussion of how to extract and determine these targets to Section 3.3, where we describe how targeted execution can be applied for existing dynamic analyses. Instead, in the following sections, we use the abstract notion of a target to refer to a location in the application's code that we wish to trigger and analyze when executing the application.

Because Android is event-driven, an application may contain several entry-point methods where the Android framework can transfer execution to the application. These methods are normally event handlers that receive various system events, such as callbacks to control a component's lifecycle (e.g. on

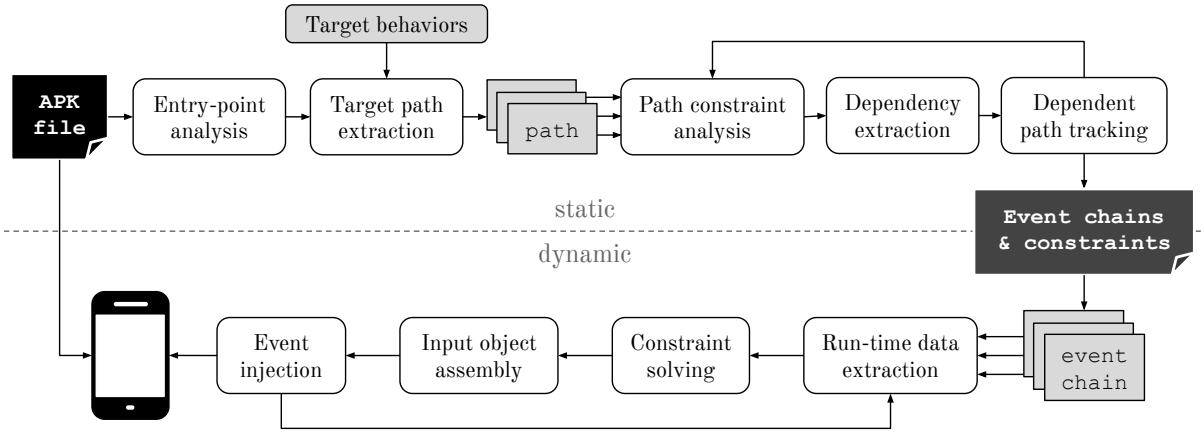


Figure 3.2: IntelliDroid system diagram

the starting or stopping of a component), process hardware sensor inputs, or respond to UI events. To determine the entry-points of an application, we use an entry-point analysis similar to FlowDroid's [10], presented in the pseudocode in Algorithm 1. We begin with the statically registered event handlers that are declared in the application's manifest and UI layout files, which are provided in the application's binary package (i.e. APK file). A partial call-graph is generated from these entry-points and is used to search for instantiations and registrations of dynamically registered event handlers, such as Android callback listeners. These new event handlers are added to the list of application entry-points and used to generate a new, more complete call-graph in which further registrations might be found. This process is repeated iteratively until no new event handler registrations are detected.

A final call-graph used to model the application's method invocation patterns is generated using the event handler entry-points as starting points for the code traversal. The construction of this call-graph uses no context or path sensitivity when resolving method invocation paths, which is imprecise but enables the analysis to scale for common Android applications. Furthermore, to track the flow of data across heap variables, a type-based heap model is constructed, resulting in a relatively imprecise points-to analysis in which unrelated heap variables can be incorrectly aliased (i.e. an overly conservative data flow analysis). Unlike a call-graph for a regular Java application, we also augment the construction of this call-graph to account for control and data flows that are specific to the Android environment, such as the Android `intent` mechanism for message passing between components (this is described further when we provide implementation details in Section 3.4.1).

A search through each reachable method in the application's call-graph is performed to find the target code locations (function ComputeTargetLocations in Algorithm 1). A traversal of the call-graph is then performed between the target locations and the event handler entry-points to extract a *target call path* for each target (Line 6). This path contains the sequence of method invocations from an event handler entry-point to the target location. Recursive call patterns are avoided during the traversal by checking whether a method is already in the currently constructed path.

To illustrate this process, we use the code provided in Figure 3.3 as an example throughout the rest of this section. This code is derived from several malicious applications and is representative of malware that intercepts and automatically responds to SMS messages received from a malicious party using the Android APIs `BroadcastReceiver.abortBroadcast` and `SmsManager.sendTextMessage`, respectively.

Algorithm 1 Pseudocode of entry-point analysis and target path extraction

```
1: function ExtractTargetPaths(apk)
2:     entryPoints ← ComputeEntryPoints(apk)
3:     cg ← BuildCallGraph(apk, entryPoints)
4:     targetLocations ← ComputeTargetLocations(cg)
5:     targetPaths ← {}
6:     for each target ∈ targetLocations do
7:         currentMethod ← GetMethod(target)
8:         path ← [currentMethod]
9:         while currentMethod ∉ entryPoints do          ▷ Backward traversal from target to entry-point
10:            callers ← GetCallers(currentMethod) − path   ▷ Traverse only unexplored methods to avoid recursion
11:            currentMethod ← callers[0]
12:            path ← [currentMethod] || path
13:        end while
14:        targetPaths ← path ∪ targetPaths
15:    end for
16:    return targetPaths
17: end function
18:
19: function ComputeEntryPoints(apk)
20:    entryPoints ← GetManifestEntryPoints(apk)          ▷ Begin with statically registered entry-points
21:    changed ← True
22:    while changed = True do                            ▷ Iterate until all entry-points are found
23:        changed ← False
24:        cg ← BuildCallGraph(apk, entryPoints)          ▷ Build partial call-graph from known entry-points
25:        newEntryPoints ← FindNewEntryPointRegistrations(cg)   ▷ Find new dynamically registered entry-points
26:        if newEntryPoints ≠ ∅ then
27:            changed ← True
28:            entryPoints ← entryPoints ∪ newEntryPoints
29:        end if
30:    end while
31:    return entryPoints
32: end function
33:
34: function ComputeTargetLocations(cg)
35:    targetLocations ← {}
36:    for each method ∈ cg do                            ▷ Search for targets in all reachable methods
37:        for each instruction ∈ method do
38:            if IsTarget(instruction) then
39:                targetLocations ← targetLocations ∪ instruction
40:            end if
41:        end for
42:    end for
43:    return targetLocations
44: end function
```

We select the invocations to these APIs as the target locations for IntelliDroid's analysis and extract target paths to them that would be triggered for a hypothetical dynamic analysis of malicious SMS usage. IntelliDroid will begin analyzing the example by identifying `SmsReceiver.onReceive()` at Line 4 as an event handler entry-point and the calls to `sendTextMessage()` and `abortBroadcast()` at Lines 22 and 26 as targets. IntelliDroid then identifies the target call paths from the event handler to each of the target locations, which are the paths through the method invocations on lines 4→11→22 and 4→11→26.

3.2.3 Extracting call path constraints

To actually trigger and execute a target call path, the appropriate inputs must be injected into the application such that the control flow of the path is dynamically realized. For each method in the target call path, the invocation of the next method in the path may be control-dependent on conditional branches within the method body. To extract these control dependencies, a forward control- and data-flow analysis is performed on the control flow graph (CFG) of each method. This is essentially a path-sensitive symbolic constraint analysis on the inputs and variables accessed by the target path and used to determine the control flow path that must be taken.

When performing this constraint analysis, IntelliDroid represents inputs to the target path's entry-point and accesses to heap variables symbolically. Beginning with the first method in the target call path (i.e. the entry-point method), we construct the method's CFG and perform a forward traversal of the graph. The initial input symbols are propagated based on the operations performed upon them, such as arithmetic computations or assignments to other variables. When the analysis encounters a split in the CFG due to a conditional branch instruction, IntelliDroid represents the logical condition in terms of the symbolic variables being compared and extracts a constraint on the variables' values that determines the outcome of the branch (i.e. the out edge that is taken from the basic block containing the branch instruction). Each outcome is analyzed separately by propagating the corresponding constraint for that outcome, which is either the condition extracted from the branch instruction or its inverse. At locations where multiple paths in the CFG converge, the analysis combines the propagated symbolic variables and constraints of the joining paths with a logical OR ($\vee$), indicating that any of the joining paths (and thus, any of their propagated symbolic states) can be used to reach that location. The propagation of symbolic data is iterative and performed over the CFG until it converges, which can require multiple iterations due to backedges in the CFG. Loops are heuristically handled by limiting the number of iterations the analysis is performed for a given instruction.

To propagate symbolic information across target path methods, the symbolic variable and constraint information in the caller method is extracted at the invocation site to the next path method. This data passed as the initial symbolic state for the analysis of the next method, with the argument variables used as inputs to the method. Each method along the path is analyzed sequentially until we reach the target location. This interprocedural constraint analysis is fully context-sensitive, since we perform the constraint analysis for each target path separately and the symbolic state passed for the analysis of a method is determined solely from that single target call path. This is scalable, as constraints are extracted only for target paths and not for the entire code base of the application.

As an example, consider the execution path in Figure 3.3 ending in the `abortBroadcast()` invocation at Line 26. We represent the inputs to this path (`c` and `i` in Line 4) as the symbols c_{sym} and i_{sym}. The target call path generated by IntelliDroid includes an invocation of `handleSms()` in the `onReceive()` method, which is dependent on the intent action string; therefore, the constraint (i_{sym}`.getAction()`

```
1  class SmsReceiver extends BroadcastReceiver {
2    private String sNum = null;
3
4    void onReceive(Context c, Intent i) {
5      if (i.getAction() == "SMS_RECEIVED") {
6        handleSms(i);
7      } else if (i.getAction() == "BOOT_COMPLETED") {
8        this.sNum = "99765";
9      }
10   }
11   void handleSms(Intent i) {
12     Bundle b = i.getExtras();
13     Object[] pdus = (Object[])b.get("pdus");
14
15     for (int x = 0; x < pdus.length; x++) {
16       SmsMessage msg = SmsMessage.createFromPdu(pdus[x]);
17       String addr = msg.getOriginatingAddress();
18       String body = msg.getMessageBody();
19       // Constraint depends on local function
20       if (needsReply(addr, body)) {
21         SmsManager sm = SmsManager.getDefault();
22         sm.sendTextMessage(addr, null, "YES", null, null);
23       }
24       // Constraint depends on heap variable
25       if (addr.equals(this.sNum)) {
26         abortBroadcast();
27       }
28     }
29   }
30   boolean needsReply(String addr, String body) {
31     if ((addr.startsWith("10658") && body.contains("RESPOND"))
32         || (addr.startsWith("10086") && body.contains("REPLY"))) {
33       return true;
34     }
35     return false;
36   }
37 }
```

Figure 3.3: Code example of target paths in an Android application

= "SMS_RECEIVED") would be extracted. IntelliDroid's context-sensitive interprocedural analysis would indicate that when `handleSms()` is called along the target path, this constraint on the input i_{sym} holds. Analysis through `handleSms()` shows that the execution of the target location is also dependent on the length of the PDU array and the originating address of the SMS message received. The conditional statement in Line 20 is on the execution path, but since the target can be reached regardless of the branch outcome, it has essentially no effect—when processing the control flow, the constraints extracted from both sides of this branch would be combined with the OR operator.

In some cases, the variables extracted for the constraints are return values from other method invocations. Although these methods are not part of the target call path, their return values affect the execution of the path and the constraints they impose must be extracted. An example of such an *auxiliary method* is `needsReply()` in Figure 3.3. To handle these cases, IntelliDroid extracts constraints for the return values and the paths leading to the return sites within the auxiliary method. These auxiliary constraints are combined with the main path constraints with a logical AND ($\wedge$) to enforce a specific return value and return path through the auxiliary method. For performance reasons, only one level of auxiliary methods are analyzed during the extraction of path constraints, though this can be configured.

For some situations, IntelliDroid also inserts library constraints manually extracted from Android API calls to pure functions—i.e., functions whose result depends only on their arguments, with no side-effects. For example, `addr.equals()` on Line 25 is an invocation to a pure function and IntelliDroid will convert this to the constraint `(addr = this.sNum)`. In some cases, the API method invoked would generate constraints that are too large or complex for the constraint solver; this is the case for `createFromPdu()` on Line 16, which performs bytewise operations on the bytecode of the SMS message. In these cases, rather than rely on the constraint solver, we provide IntelliDroid with a manually implemented function that inverts the computations within `createFromPdu()`, thus allowing IntelliDroid to generate an appropriate input. This is conceptually equivalent to "stitching", which is used to solve constraints for similarly complex functions, such as SHA1 and MD5, in BitFuzz [26]. Android API methods that are not pure functions must be handled dynamically at run-time by either monitoring or controlling them, as described below in Section 3.2.5.

3.2.4 Extracting event chains

For each target path, the constraints extracted for the target location contains a boolean expression of variables and concrete values. Ideally, all of the variables should be dependent on the input parameters of the path's entry-point method. In such a case, solving the constraints for these variables and injecting the solved input values to the entry-point will execute the desired path. However, there may be cases where the constraints depend on heap variables that cannot be set to the correct values using only the arguments to the entry-point method. In these cases, IntelliDroid must find a definition for these variables that can be executed to set the heap variables to the required values.

An example of such a heap variables in Figure 3.3 is `SmsReceiver.sNum` for the target path to `abortBroadcast()`. This variable is used in a constraint imposed by the conditional branch in Line 25, but is defined in another invocation of `onReceive()` where the intent action string is `BOOT_COMPLETED`. To complete the constraints and execute the target path, two actions must be completed: (i) any additional constraints on the heap variable imposed by the other path must be extracted and added to the current path constraints; and (ii) the heap value required by the target constraints is actually stored in the heap variable prior to executing the target path. Thus, when the constraints contain a heap

dependence, IntelliDroid searches for statements where the heap variable is defined/stored and finds the event handlers leading to these definitions. The path from the event handler to the store instruction becomes a *supporting call path* and IntelliDroid extracts *supporting constraints* for this path in the same manner used for the target path. Later, when solving the constraints, a concrete value will be assigned to the heap variable and used to solve the target path constraint.

For `sNum` in Figure 3.3, the supporting call path would begin at `onReceive()` and the supporting constraints would include (i_{sym}`.getAction() = BOOT_COMPLETED`). The main target path constraints would be appended with the extra constraint (`sNum = 99765`), which is extracted from the supporting path and ensures that the SMS originating address is properly constrained. The supporting path and the main target path form an *event chain* that results in the execution of the target location. In the run-time component of IntelliDroid, this event chain will result in multiple input injections, one for each supporting and target call path. In cases of multiple dependencies, the process is performed iteratively, as shown by the topmost backedge in Figure 3.2. Dependencies may also be recursive, as the constraints for a supporting path might contain further dependencies on other heap variables. The analysis forms an event chain ordered by the data-flow dependency between the variables used in the paths.

Event chains are also used to handle control dependencies, which are imposed by the registration of event handlers with the Android framework. In the Android system, some event handlers are known to the system (e.g. lifecycle handlers), some are declared in the application's manifest, and some are registered dynamically within the execution path of a previous event handler. For those that are registered dynamically, the registration process may require parameters specifying how and when the event handler is to be called. For instance, registering location callbacks requires that the application specify the frequency and minimum distance between consecutive callback invocations. These values are added to the constraints to ensure that the injected event abides by these parameters in the same way the Android framework would in normal execution. The supporting call path leading to the event handler registration is added to the event chain due to the control-flow dependency between it and the target call path.

3.2.5 Run-time data

For the simple case where all constraint variables are input-dependent or can be concretized through the execution of supporting paths, a path's constraints can be solved statically and the run-time system merely has to inject the input values. However, there may be cases where the values of constraint variables cannot be determined statically. This may occur for heap variables where the points-to analysis is imprecise or incomplete (e.g. for reflected object accesses) or for values obtained from Android API methods that cannot be modeled statically. A purely static constraint extraction approach would either be unable to extract all the path constraints or would need a considerably more precise and expensive static analysis to do so. However, IntelliDroid's hybrid static-dynamic design sidesteps this dilemma by obtaining the values during run-time by performing the constraint solving step immediately prior to event injection. That is, although the constraints are extracted during static analysis, they are solved at run-time so that any statically unresolved variables can be resolved prior to the constraint solver step.

The delayed constraint solving gives the user a choice of how to use IntelliDroid to handle interactions between the Android application and its environment. Variables whose values depend on these interactions can either be *monitored*, indicating that the interaction is allowed to proceed without modification and IntelliDroid merely eavesdrops on the interaction, or *controlled*, where the IntelliDroid intercepts

and replaces these interactions with values it determines will exercise target paths. Variables that depend on input from a possibly malicious external component can be either monitored to understand the interaction of the malware with the external component, or controlled to understand what potential capabilities the malware may give to an adversary in control of the external component. On the other hand, variables that are derived from the Android framework and OS, which are trusted, would generally be controlled to take a value that, together with the other constrained variables, satisfies the target path constraints and enables the target to be executed.

For monitored variables, the external input is often derived from a control server that sends commands to the application. In some cases, the application may request data from the server and use this data to perform malicious activities. A common example from the Android Malware Genome dataset [144] involves applications that download a list of premium SMS numbers from the network and intercept messages to/from these numbers such that the user is unaware of the premium fees. Although the server input cannot be determined statically, network monitoring can extract the values returned and add these values to the target path constraints. Constraints that occur on the server side are not captured by IntelliDroid, although it is possible to set up a fake server that sends the necessary replies to the application when it makes network requests. However, because the fake replies can affect the malicious activity that IntelliDroid aims to analyze, these external variables are instead monitored to determine the real values that the application expects and how it behaves when given these values.

For controlled variables, they are unresolved due to their dependence on the device state. For instance, malicious behavior may only manifest during a certain time or date, and this is reflected by constraints that contain the system time/date as variables. These variables can be resolved by setting the device state (e.g. setting the time) prior to injecting the main event. The actual value used is determined by the statically extracted constraints that depend on the external variable. For instance, if a constraint is extracted stating that a particular target location is only triggered only when the system time is set to "1:00", the device time will be set to this value before injecting the inputs to trigger the target API. This is essentially another form of event chain extraction, where supporting events must be injected prior to executing the main target call path.

3.2.6 Input injection for target path

Once the constraints are generated and all run-time values are obtained, IntelliDroid can trigger the desired target call path by obtaining the input parameters that fulfill the constraints. As previously discussed, the task of solving the constraints is placed on the dynamic component of IntelliDroid; thus, the input parameters are solved for and generated immediately prior to executing the target path.

The dynamic component of IntelliDroid consists of a client program running on a computer attached to the device. Communication between this program and the device is facilitated by a newly constructed Android service (`IntelliDroidService`) that serves as a gateway for the tool. As motivated earlier, IntelliDroid must inject inputs at the device-framework interface, rather than the application event handler, to ensure that state in the Android framework is consistent with application state. When the static component specifies inputs for the execution of the target location, the gateway service is responsible for injecting that input into the device-framework interface of the Android OS on which the application is running. To do this, it must perform two tasks: first, it must identify a suitable injection point; second, it must format the input values for injection into the Android OS.

To identify a suitable injection point, IntelliDroid must identify a method at the device-framework

interface that: (1) is called when the corresponding external event occurs; and (2) directly calls the desired application handler when it is invoked. Further, such input injection points must have a one-to-one relationship with the event handler of interest, so that inputs thus injected will only result in the invocation of the desired application event handler and no other handlers. For instance, SMS events are received by the framework via a socket, which is monitored by a long-running process. When an SMS message arrives on the socket, a device-framework interface is invoked by the process, which eventually calls `PhoneBase.sendMessage()`, the desired device-framework interface handler.

To find suitable injection points, we perform static analysis of the Android framework, using a backward call-graph traversal starting from the event handlers of interest to find candidate injection points. Alternatively, since these injection points are often located in Android service classes and these service classes are well-known, IntelliDroid can be given a list of classes where injection should occur and it will automatically generate paths between methods within these classes and the event handlers to be triggered. Because invoking such injection methods will often require interprocess communication, IntelliDroid preferentially selects RPC methods (i.e. remote procedure calls, a generated interface for interprocess communication when compiling the Android OS) as input injection points as they present a cleaner interface.

To properly format inputs values for injection, the input constraints for the application event handler (extracted by the static phase of IntelliDroid) must be transformed into constraints at the input injection point and then solved. As a result, constraints imposed on the *injection path* between the injection method and the application event handler are extracted using the same analysis that IntelliDroid performs on applications. In some cases, injection paths may have dependencies on other paths in the framework, requiring a chain of device-framework events to be injected to properly invoke the application event handler.

Since the Android framework is the same for every application, IntelliDroid extracts the injection points and injection path constraints for supported application event handlers once and stores them in a library for use at run-time. At run-time, injection path constraints are combined with target call path constraints in the application using a logical AND. In addition, IntelliDroid appends extra constraints specifying how the injection method parameters are related to the event handler parameters. Finally, the inputs may need to be formatted by initializing the fields of a specific input object (for instance, a `Location` object for a location event) to the desired value. While the constraint solver can automatically generate the appropriate values for the fields, the code to populate them in the object is manually implemented. We have performed the framework injection point analysis and implemented the input object reconstruction for several types of events: component lifecycle events, intents for inter-component messaging, SMS, location, and a small subset of UI events for the click action.

3.3 Extensions to IntelliDroid

The primary intellectual extensions to IntelliDroid in this thesis lies in the motivation for targeted analysis in the face of existing static, dynamic, and symbolic analysis techniques. A key component is the versatility of applying targeted execution for concrete dynamic analysis. In the prior section, we described the core hybrid techniques used to guide execution to locations of target behaviors. We now describe how these target behaviors are found and how they can be applied to existing dynamic analysis tools for Android. In the evaluation section, we also present a further extension to IntelliDroid

through an integration with the TaintDroid [40] dynamic taint tracking tool. We show how we can use IntelliDroid's targeted concrete execution with TaintDroid to trigger and detect privacy leaks in applications more effectively than static-only or dynamic-only techniques.

3.3.1 Specifying target APIs

The primary contribution of targeted execution is that it guides concrete execution of an application toward specific target locations; as such, this concrete execution can be combined with general dynamic analysis tools to target behaviors of interest for their analyses. Because we want IntelliDroid to be applicable to as wide a range of dynamic analysis tools as possible, we need to select a suitable abstraction that over-approximates the types of behavior that various dynamic analysis tools are trying to detect. We perform a survey of recent Android malware dynamic analysis techniques that have been proposed in the literature.

Table 3.1: Existing Android dynamic analysis tools and the features used for malware detection

Dynamic Tool	Goal	Features for Analysis
AASandbox [22]	Monitor behavior via tracking of system calls	System calls
Andromaly [107]	Malware detection via system resource usage	Low-level device features (e.g. battery usage, CPU load)
CopperDroid [114]	Monitor behavior via system call tracking	System calls
Crowdroid [25]	Monitor behavior via tracking of system calls	System calls
DroidBox [66]	Sandbox to monitor external accesses	Sink API methods
DroidRanger [145]	Detect malware using pre-specified behavioral footprints and heuristics	Sequence of API method invocations and parameters
DroidScope [132]	Plugins for API tracking, instruction tracing, and taint tracking	API methods; source/sink API methods
RiskRanker [53]	Detect malware using known vulnerability signatures	Sequence of API method invocations
TaintDroid [40]	Detect privacy leakage	Source/sink API methods
VetDroid [140]	Malware detection via permission use behavior	Permission requests (can be mapped to API methods)

From our results in Table 3.1, we show that dynamic analysis tools for Android malware detection can be separated into three categories depending on their operation: (1) by analyzing invocations to certain API methods; (2) by analyzing invocations to system calls; and (3) by analyzing low-level side effects of the application, such as CPU load or battery usage. We find that specifying behaviors as API methods allows IntelliDroid to cover most of the current dynamic tools. We elaborate on our reasoning below.

3.3.1.1 Analyzing API methods

The vast majority of dynamic analysis tools analyze API method invocations and the target methods for IntelliDroid can be determined by analyzing the specific API methods used to configure the tool. For instance, TaintDroid [40] performs taint tracking by adding taint tags in locations where sensitive information is obtained by the application (i.e., sources) and reading taint tags in locations where information leaves the application (i.e., sinks). By referring to the documentation or searching through the source code, these methods can be found and used as target methods. Sandboxing tools such as DroidBox [66] track locations where data leaves the application and target methods can be determined by finding the instrumented API methods. Other tools such as DroidScope [132] allow the user to trace specific API method invocations; these API methods would serve as target methods.

Some dynamic tools, such as VetDroid [140], detect malware by dynamically analyzing an application's permission usage. Although the tool does not trace API methods, the mapping between permission use and API methods has been well-studied and can be obtained from a number of works that automatically extract this mapping from the Android framework [2, 11, 16, 42]. IntelliDroid can therefore be configured with target methods that map to the permissions of interest. Since the majority of dynamic analysis tools analyze API calls, using API calls as our abstraction would enable IntelliDroid to generate inputs for most of the dynamic analysis tools.

3.3.1.2 Analyzing system calls

The next most common method used by dynamic analysis tools is to analyze the use of system calls. Such tools include CopperDroid [114], AASandbox [22] and Crowdroid [25]. In these cases, the identification of the target methods requires a mapping between the system call method and API methods that use the system call. If only specific system calls are traced (e.g. file access), we can use Android's documentation to find API methods that use the system call's functionality and generate the mapping manually. In general, however, it can be difficult to map every system call to API methods in this manner; therefore, we may need to perform a one-time static analysis of the Android framework. A backward traversal of the framework's call-graph from the invocations of system calls to public API methods should provide the necessary mapping, which can then be used to obtain the target methods. As a result, we believe IntelliDroid would be able to generate inputs for dynamic analysis tools that analyze system calls, albeit with more effort required for the configuration of targets.

3.3.1.3 Analyzing low-level events

A few tools focus on analyzing low-level events on the device. Andromaly [107] is one such dynamic tool that tries to infer malicious activity by detecting anomalies in CPU load and battery usage during the application's execution. The ability to attach IntelliDroid to such analysis tools depends on how the features are being traced. If the tool merely detects single instances of usage, it may be possible to use IntelliDroid to trigger API methods that correspond to those resources, such as those that invoke the camera or GPS. However, IntelliDroid is not an appropriate input generator for analysis tools that profile anomalies in resource usage over time, as the IntelliDroid does not seek to mimic realistic usage. In such cases, it would be more effective to use a tool that aims to replicate normal use or have a user manually execute the application. We note that other common code exploration techniques, such as fuzzing or symbolic execution, would likely suffer from the same limitation since random or systematic

triggering of each application path would not resemble normal usage by a user.

3.3.2 Identification of targets for general security analysis

While the specification of the targets for a dynamic analysis is manual, it is not overly onerous. We demonstrate this by extracting the target APIs for TaintDroid, for which we further discuss the associated effort and effectiveness in Section 3.5. For the general analysis and targeting of security-related or "dangerous" behaviors in an application, the identification of targets can be automated by relying on the relationship between sensitive behaviors and permissions in the Android system. Most sensitive actions on an Android device are accessed through API methods in the Android framework and protected by a permission that must be explicitly granted by the user. Therefore, the identification of security-related targets can be translated into the identification of framework APIs that require a permission. Previous works in permission analysis [2, 11, 16, 42] can generate a mapping of APIs and the permissions they require through automated analysis of framework code. IntelliDroid's targeting can use this mapping to target invocations to permission-guarded APIs as a proxy for targeting security-related application behaviors.

IntelliDroid's design also allows other forms of targets to be specified. In general, if the user can determine a point in the code to which execution is desired, this information can be given to IntelliDroid, which will extract the call paths and path constraints to the specified code location. This location can be as simple as a method invocation, or can be derived from some other analysis. For instance, to direct execution for a dynamic tool that focuses on native code usage, IntelliDroid can be configured to extract paths and constraints for invocations to native methods. Alternatively, to target malicious behaviors that cannot be defined by a single API invocation, a signature of the malicious behavior could be developed from known malware samples and identified during IntelliDroid's target analysis using lightweight pattern matching.

3.4 Implementation

IntelliDroid's design contains a static and dynamic component. For the static analysis, our prototype uses the WALA [120] static analysis framework. For the dynamic component, we instrument the Android operating system (AOSP) to facilitate the injection of inputs into a full Android system and we use Z3 [36] to solve constraints when determining the injected input values.

3.4.1 Static component

For versatility, IntelliDroid performs its analysis on compiled Android applications and does not require source code. Because they are packaged in APK files and stored as DEX bytecode, the applications must be unpacked and converted to Java bytecode prior to analysis, using tools such as Dare [85] and APKParser [8]. The converted files are then passed to IntelliDroid's static component, which uses the WALA static analysis libraries [120]. WALA provides support for basic static analysis, such as call-graph generation, data flow analysis, alias analysis, and an intermediate representation based on SSA.

Part of the static analysis performed by IntelliDroid includes the construction of a call-graph to model the application's invocation patterns. This call-graph construction is somewhat complicated by the Android platform, which provides facilities, such as `Intents`, `Threads`, `Executors`, IPCs, RPCs, and

`AsyncTasks`, that allow applications to transfer execution between event handlers without an explicit method invocation. When generating the call-graph within WALA, we augment the call-graph construction such that these Android-specific edges are automatically added, ensuring that the call-graph can give an accurate representation of how execution flows between methods in the application. The call edges are conservatively patched based on the documented behavior of the invoked method and on the parameters or constant values used in the invocation.

There are certain cases where framework API method invocations must be treated differently. For instance, when the constraint extraction encounters API methods that obtain information from external sources (such as the network or a file), it must note whether the returned values can be controlled or monitored. This distinction is currently made on a per-method case and is determined by whether the source of the data is controlled by the third-party application developer. Any data originating from an external source other than the device, Android framework, or Android OS is considered potentially malicious and the value is monitored. Other framework methods may also be modeled due to the limitations of the constraint solver. For instance, string methods are modeled internally as well as trigonometric operations, since the constraint solver does not support such functionality. In general, processing the invocation of a framework method depends on whether it introduces externally-obtained data and whether the constraint solver supports the operations performed.

The constraints generated by IntelliDroid are placed into an application-specific file. When the static phase has completed, this file will contain all target call paths found in the application, along with information detailing how the dynamic component can trigger them. For a given application, only one execution of the static component is needed, since this file will contain all of the information that the dynamic component requires.

3.4.2 Dynamic component

The dynamic component of IntelliDroid consists of a client program running on a computer, connected to a device or emulator with a custom version of Android. The dynamic client program is implemented using Python and acts as the controller that determines the target call path to execute. It also interfaces with the constraint solver used to generate the path inputs: the Z3 constraint solver [36] with the Python API (Z3-py). Communication between this program and the device is facilitated via sockets, using the device port-forwarding feature of the Android Debug Bridge (ADB) [2]. The other endpoint of the socket is located in the gateway Android service, `IntelliDroidService`. This is implemented as a long-running system service that is instantiated upon device boot. On receipt of messages from the client program, this service can obtain information about event handlers, assemble an input object using values that the client program sends, and trigger an event with the input object.

In certain cases, run-time values for constraints in the injection path are needed. For instance, the `onLocationChange()` event handler is called only when there is a minimum distance from the last location sent to the application. The constraint modeling this relationship would require the value of the last location that the event handler received, as well as the minimum distance parameter stored in the framework. IntelliDroid extracts these values during run-time, by instrumenting the system services handling these events to send event handler information when requested by `IntelliDroidService`. Although such run-time extraction is not strictly necessary, it can provide an advantage over static

[2] ADB documentation: http://developer.android.com/tools/help/adb.html

extraction in cases where the event handler registration parameters are not explicit within the application code.

Because IntelliDroid is currently using the Python API for the Z3 solver [36], the Z3 string library is not available. Therefore, string functions such as `equals()`, `contains()`, or `startsWith()` must be modeled and string variable types are handled by the dynamic component as a special case. Due to the heuristics used when modeling such functions, there can be cases where complex string manipulation may not be represented precisely by the extracted constraints.

3.5 Evaluation

Our IntelliDroid prototype is implemented for the Android 4.3 operating system (AOSP) and evaluated on an Intel i7-2600 CPU at 3.40 GHz with 16GB of memory. In the evaluation, we aim to answer the following questions:

- *How effective is targeting API calls derived from a real dynamic analysis tool, and can this technique trigger all of the malicious behavior that the dynamic analysis tool can detect?*

  We integrate IntelliDroid with TaintDroid [40], a dynamic taint-tracking tool and we demonstrate that the combination is able to detect all sensitive data leaks in a corpus of privacy infringing malware.

- *Given a target API, how effective is IntelliDroid at generating the inputs to trigger it?*

  We test IntelliDroid on a wider range of target APIs and malware, and evaluate whether it can generate inputs to trigger all malicious behavior. We also discuss the effectiveness of the different techniques used by IntelliDroid, such as event chains and run-time data gathering.

- *What performance benefits can IntelliDroid's targeting potentially provide for a dynamic analysis tool? What are the run-time costs of the static and dynamic components?*

  We measure the time IntelliDroid takes to generate and inject inputs, the number of inputs required, and the amount of code that IntelliDroid is able to avoid executing.

3.5.1 Targeted execution with IntelliDroid-targeted TaintDroid

To demonstrate how targeted execution can be used in practice and its advantages over existing analysis techniques, we integrated IntelliDroid with TaintDroid [40], a dynamic taint-tracking system, to produce a combined system we call Intelli-TaintDroid. Integration with TaintDroid is straightforward and requires the merging of IntelliDroid's input injection component with TaintDroid's code base, which can be done with an automated patch. To derive the set of target APIs from TaintDroid, we analyze TaintDroid's documentation and source code to identify the instrumented methods that add and check taint tags. In cases where taint is assigned or checked in an internal framework method, we traced the call path back to an API method. Table 3.2 summarizes the number and types of APIs targeted. We found that specifying the target APIs for TaintDroid was fairly easy and took us on the order of 2-3 hours to produce the full set of target APIs.

We perform three experiments with Intelli-TaintDroid. First, we evaluate against a malware set for which we know the ground truth of all malicious behaviors. In this way we can evaluate the accuracy

Table 3.2: Target APIs for TaintDroid's analysis

API Type	Number of APIs
Read phone data	4
Read database	13
Read location	7
Read UI data	1
Read account data	1
Read media data	13
Write data to HTTP	8
Write data to SMS	4
Write data to file	11
Total	62

of Intelli-TaintDroid. Second, we compare against FlowDroid [10], a purely static analysis tool that also detects privacy leakage. Finally, we compare against TaintDroid driven by Monkey [81], a generic non-targeted fuzzer. For all of the experiments, we granted all of the permissions requested by the tested applications.

To perform a ground-truth evaluation of Intelli-TaintDroid, we need malware for which all known privacy leaking behaviors are known. To this end, we use 14 documented malware families from the Android Malware Genome dataset [144] that are known to leak sensitive information and supplement this with several recent samples from the Contagio project [90], which we manually analyzed to find all privacy leaking behaviors. Table 3.3 summarizes all of the behaviors that the malware is known to exhibit. The Intelli-TaintDroid combination is able to detect all of these behaviors with no false positives. IntelliDroid generates the appropriate inputs that trigger the privacy leakages and TaintDroid's dynamic tracking promptly reports it. In some cases, tainted data may flow through the heap and this would require executing intermediate paths that do not directly invoke the target API methods. IntelliDroid's event chain mechanism detects these flows and invokes the necessary intermediate events to complete the flow from taint source to taint sink.

We further compare Intelli-TaintDroid against FlowDroid [10], a purely static taint-tracking tool, on the same set of malware. Since FlowDroid uses a more sophisticated static analysis than IntelliDroid, we expect that it might be more complete than IntelliDroid. However, out of the 26 privacy leaks, FlowDroid is unable to precisely detect the leakage in 7 cases because it stops when the sensitive information is sent via intent to another application component (we note that in the intervening time since our experiments were performed, FlowDroid has been augmented with IccTA [69], which facilitates the propagation of taint across Android components). Since Intelli-TaintDroid executes the full system, it is able to detect that data sent to these intents is eventually leaked via SMS or HTTP. We also note that Intelli-TaintDroid has no false positives, though it does report extra leaks that FlowDroid does not, since TaintDroid also monitors system services while FlowDroid only analyzes the application. We manually confirmed these extra flows to be true privacy leaks.

To fully compare against FlowDroid, we also tested Intelli-TaintDroid with the DroidBench test suite used in FlowDroid's own evaluation. Although DroidBench was meant to evaluate static analysis tools,

Table 3.3: Privacy leaking malware

Malware	**Leakage Paths**	**Sensitive Data**
Backflash	SMS → SMS	SMS, IMEI
	SMS → HTTP	SMS, IMEI
	Lifecycle → HTTP	IMEI
	Boot → HTTP	IMEI
Bgserv	SMS → HTTP	phone number
	SMS → File	phone number
	SMS → HTTP	phone number
Cajino	Intent → HTTP	SMS, IMEI, contacts, files
CoinPirate	SMS → HTTP	SMS
Crusewin	SMS → HTTP	SMS
Endofday	SMS → File	phone number
GamblerSMS	SMS → SMS	SMS
GGTracker	SMS → HTTP	SMS, phone number
GoldDream	SMS → SMS	SMS
GPSSMSSpy	Location → SMS	location
NickyBot	SMS → SMS	IMEI
	Lifecycle → HTTP	IMEI
HeHe	SMS → HTTP	SMS, IMSI
	SMS → File	SMS
	Lifecycle → HTTP	IMEI, IMSI
NickySpy	Boot → SMS	IMEI
Pjapps	SMS → SMS	SMS
	Lifecycle → HTTP	IMEI
SMSReplicator	SMS → SMS	SMS
Spitmo	SMS → SMS	SMS
Zitmo	SMS → HTTP	SMS

this comparison shows the advantages of dynamic analysis when attached to a targeted execution tool such as IntelliDroid. Intelli-TaintDroid is able to detect all privacy leaks without any of the false positives of FlowDroid, due to the increased precision of dynamic taint-tracking.

Finally, we compare our Intelli-TaintDroid implementation against TaintDroid on its own being driven by Monkey [81], a testing tool included with the Android SDK. Monkey is a simplistic tool that performs fuzzing by activating random GUI and system events. There are more sophisticated dynamic code exploration tools for Android; however, we found that we were unable to integrate them with TaintDroid. We had attempted to compare with DynoDroid [75] (only available on Android 2.3), but we were unable to integrate it with TaintDroid successfully. We were also similarly unsuccessful with integrating the Android concolic testing system ACTEve [5] with TaintDroid.

We ran Monkey on each application for one hour, sending over 60K injections per application. Since Monkey is only capable of sending UI events and select system events, Monkey-TaintDroid missed 21 out of 26 cases of privacy leaks in our malware dataset, where the leaks require non-UI events such as location or SMS. Monkey was also unable to trigger leaks in cases such as *GPSSPSSpy*, where specific input strings must be injected to trigger the privacy leak. In comparison, Table 3.4 shows the number of input injections that Intelli-TaintDroid required to detect all malicious behavior in each application. Overall, IntelliDroid needs between 2 and 430 inputs (with an average of 72) to trigger all malicious behavior in any one of our malware samples. While we speculate that DynoDroid would likely have been able to detect more leaks because it can inject non-UI events, we do not believe that it would be able to guess the correct input strings needed to trigger the privacy leak either. ACTEve, being a concolic testing tool that performs static analysis, would likely be able to determine the correct inputs, but as a coverage tool, it seeks to execute each path only once and thus may miss malicious behaviors since it does not know the order in which to inject inputs. In contrast, IntelliDroid injects each input once and determines from static analysis the correct order to inject them.

3.5.2 Generating inputs to trigger target APIs

The previous section shows that IntelliDroid is effective in practice when integrated with a real dynamic analysis system. However, TaintDroid itself is only capable of detecting privacy leaks. We now seek to understand the limits of what types of inputs IntelliDroid can generate when tasked with triggering a larger variety of behaviors. To do this, we use 27 malware families from the Android Malware Genome [144] and Contagio datasets [90], and use a set of target APIs that would have been derived from a hypothetical tool that would be capable of detecting all known malicious behavior, given IntelliDroid's ability to trigger it. The malware in our dataset performs malicious actions that are typical of many types of malware, including SMS manipulation and monetization, receiving command and control messages via the network and SMS messages, sending stolen data over the network, and other malicious network requests. They also obfuscate their actions using techniques such as reflection and dynamic class loading, which are common among Java-based malware. In some cases, a malware sample can exhibit several malicious behaviors, giving the dataset a total of 75 malicious behaviors that IntelliDroid must trigger.

For each behavior, we describe both the target APIs that IntelliDroid uses (i.e., the static configuration), as well as how we confirm that IntelliDroid is able to successfully trigger the target API invocation.

Table 3.4: Number of injected inputs required by IntelliDroid to trigger malicious behavior

Malware	**Injections Required**
Backflash	41
Bgserv	91
Cajino	167
CoinPirate	85
Crusewin	2
Endofday	44
GamblerSMS	5
GGTracker	9
GoldDream	43
GPSSMSSpy	19
HeHe	430
NickyBot	104
NickySpy	107
Pjapps	64
SMSReplicator	7
Spitmo	5
Zitmo	3
Average	72

① **Premium SMS**: Trigger paths to `SmsManager.sendTextMessage`. Confirmed by checking that `sendTextMessage` is called with a premium number.

② **Blocking SMS**: Trigger paths to `BroadcastReceiver.abortBroadcast` from within an `onReceive` event handler. Confirmed by checking that `BroadcastReceiver.abortBroadcast` is invoked

③ **Deleting SMS**: Trigger paths to `ContentProvider.delete` where the URI is `content://sms`. Confirmed when a deletion occurs on the SMS content provider, where the deleted message was injected by IntelliDroid.

④ **Leaking information via SMS**: Trigger paths with calls to `sendTextMessage`. Confirmed by inspecting the content of messages sent by `sendTextMessage`.

⑤ **Network access**: Trigger paths to HTTP API methods. Confirmed by recording and inspecting the device's network traffic.

⑥ **Reflection and dynamic class loading**: Trigger paths to reflection and dynamic class loading API methods (e.g. `DexClassLoader.loadClass`). Confirmed by checking that the API methods are invoked.

In some cases, the malware constraints depend on values obtained from network requests to a remote control server. To resolve these constraints, IntelliDroid will monitor these network requests to extract the necessary values and solve the constraints to generate inputs that will match these requests. However, for the *CoinPirate*, *Crusewin*, and *Pjapps* malware, the third-party servers were no longer available and the network data values could not be extracted. To test these samples, we implemented an HTTP proxy server that imitates the original control server and responds to application requests with appropriately formatted replies.

Using the malicious dataset and the specification of target APIs, we measure the number of instances where IntelliDroid successfully generates inputs that trigger the target API. Many of the malware samples have multiple malicious invocations of the APIs, in which case they are tested once for each invocation. Table 3.5 provides detailed information about the target APIs found and triggered in each malware family. The specific target API (and their corresponding numbers) are described above and the table is organized with respect to the event handler that triggers the API. In the case of the *Jifake* malware, IntelliDroid extracted a path to a reflected method call and when dynamically executing this path, the reflected call triggered a malicious behavior that leaked sensitive information via SMS. Since the malicious behavior was triggered, a dynamic analysis tool would detect it in theory, even though IntelliDroid only generated inputs that triggered the path to the reflected call.

IntelliDroid was successful in triggering the target API in 70 out of the 75 instances. We found that IntelliDroid's ability to extract event chains, perform device-framework input injection, and solve constraints at run-time were significant in achieving targeted execution, which we discuss below.

Event chains: The event chains generated by IntelliDroid were instrumental in 6 cases of malicious behavior and ensured that all constraints imposed by the application were satisfied. For example, in the *Endofday* and *Zsone* malware, the malicious behavior was activated only when the injected event occurs on a certain date or only after the application has been running for certain amount of time. In these cases, simply injecting a single event would not have satisfied the multi-event

Table 3.5: IntelliDroid's effectiveness by malware family

Rows indicate the malware family and columns indicate the type of input(s) injected.
Numbers indicate the type(s) of malicious activity triggered/missed.

***Event* →**	**SMS**	**Intent msgs.**	**Location**	**UI (onClick)**	**Lifecycle**
AnserverBot	[2] [5] [6]			⑤ ⑥	⑥
Backflash	[1] ② ④ ⑤	⑤			⑤
Bgserv	② ④			①	① ⑤ ⑥
Cajino		① ⑤			⑤
CoinPirate	① ② ⑤				⑤
Crusewin	① ③ ⑤				⑤
DogWars		①			
Endofday	① ③ ④				
Fakemart	② ③				①
FakeNetflix				⑤	
FakePlayer					①
GamblerSMS	④				
GGTracker	① ② ⑤				⑤
GoldDream	[4]				⑤
GPSSMSSpy	④		④		
HeHe	① ② ⑤				⑤
Jifake					6 → 1
HippoSMS	① ②				
KMin	②			③	
NickyBot	③ ④				⑤
NickySpy		④			
Pjapps	② ④				⑤
RogueSPPush	① ② ③				⑤
SMSReplicator	① ④				
Spitmo	② ④				
Zitmo	⑤				
Zsone	① ②			①	

◯ Malicious behavior triggered [ ] Missed behavior

constraints that these applications impose. The event chains for these paths include a separate event to change the device's system time, which is triggered prior to the injection of the input that finally triggers the target API. In a different case, the *GPSSMSSpy* malware watches for a control SMS message that contains a certain string ("how are you?"). Once received, it saves the originating address of the message and begins listening for location updates. For each location update, it sends the location information to the saved SMS address, thereby leaking sensitive data to a third-party. IntelliDroid's event chain generation component successfully recognizes this data-flow dependency through the third-party address saved on the heap and accessed in the location event handler, thus ensuring that the SMS is injected prior to the location event.

Device-framework injection: Injection at the device-framework interface was necessary in 9 cases of malicious activity. For these cases, the malware would not have behaved realistically if IntelliDroid had not implemented consistent framework behavior by injecting inputs into the framework rather than at the application boundary. For example, the *GamblerSMS* malware receives new SMS notifications by registering a custom `ContentObserver` object to listen for SMS database changes. Merely injecting new SMS events at the framework-application boundary would not have triggered the malicious behavior, since the injected event would not have been entered into the framework's SMS database. The *CoinPirate* and *HippoSMS* malware also use a similar technique to receive new SMS notifications while avoiding traditional telephony APIs, which are commonly detected. For these applications (and any others that use `ContentObserver` for other databases), it is essential that the events are injected at the device-framework interface so that they are entered into the appropriate database. In addition, 5 cases were found where SMS entries are deleted from the database when the application detects that a new SMS message was received. Often, these deletions require a query into the database to obtain a handle (e.g. URI) on the message. If the SMS event was not entered into in the database, the query would have failed and the deletion would not have been executed. If this occurred while screening the applications for malware, it would have caused the screening tool to miss potential malicious activity.

Run-time constraint data: Run-time data was required for the extracted constraints of 22 malicious call paths. This data included controlled variables such as the device time or location, and monitored variables derived from third-party server replies to network requests. For instance, the *CoinPirate*, *Crusewin*, and *Pjapps* malware contained malicious call paths relying on values obtained from a third-party server. For the malicious activity to occur, the network reply values are compared against those from the injected event and thus, the extracted constraints depend on the run-time variables. In other cases, values are obtained from the application's `SharedPreferences` file or from the device state. Because IntelliDroid employs a hybrid system and performs the constraint solving in the dynamic component, the statically extracted constraints could be augmented with the run-time data prior to generating the input values. Without the run-time variables, the constraints would not have been as precise and the malicious call path would not have executed fully.

Of the five malicious behaviors that IntelliDroid could not trigger, three of them occurred for *AnserverBot*, which has path constraints that contain hash functions. The solving of constraints containing hash functions is beyond the capabilities of the Z3 constraint solver that IntelliDroid uses. Similarly, *Backflash* contained constraints that require Base64 decoding and string/array manipulation, which In-

telliDroid currently does not fully handle. The remaining case occurred in *GoldDream* and was the result of data flow through files. While IntelliDroid currently does not support flow through files, it would be possible to extend it to recognize file system dependencies in the same manner as heap dependencies.

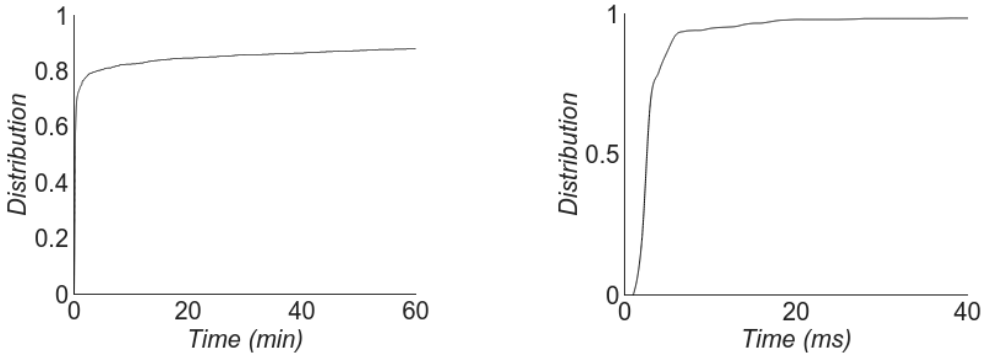
While other hybrid targeted or semi-targeted execution tools have been developed for Android, such as Harvester [98] or FuzzDroid [99], they were published after our work in IntelliDroid and were not tested in this evaluation. Harvester uses forced branching to target obfuscation locations in an application (e.g. to log decrypted values used in a reflected method invocation) but this can lead to the execution of many unsound paths due to inconsistent data usage along the forced paths. It is unclear how its techniques would perform for targeting execution for other types of dynamic analyses, such as the tracking of private data usage from the previous section. FuzzDroid only performs semi-targeted execution and does not handle dependencies between application paths; therefore, it is unlikely that it would have been able to trigger paths for which IntelliDroid's event chains were required. We provide a more detailed comparison of their techniques in Chapter 4.

3.5.3 Performance

We measured two quantitative performance aspects of IntelliDroid. The first is the reduction in analysis time IntelliDroid imparts by saving a dynamic analysis tool from having to exercise irrelevant portions of the application. For this, we measure the percentage of the application that IntelliDroid actually dynamically executes to allow TaintDroid to detect all privacy leaks. The second is the time IntelliDroid takes to generate and inject inputs, which has two distinct phases: (1) static extraction and analysis of path constraints; and (2) dynamic generation of inputs based on run-time state and constraint solving. We do not include the time to actually run the dynamic analysis as this is more of a function of the dynamic tool than of IntelliDroid.

Our previously described experiments with Intelli-TaintDroid give a glimpse of the reduction in analysis time that IntelliDroid can provide. While Monkey injected over 60K inputs, it was only able to trigger 7 of the 26 malicious behaviors that IntelliDroid could trigger with an average of 72 inputs. However, Monkey is a fairly simplistic tool and it was not possible to integrate more complex tools with TaintDroid. Thus, we measure the average percentage of application code that Intelli-TaintDroid must exercise and compare against the amount of code that an input generator based on random fuzzing or concolic testing might need to execute to achieve the same detection results. By measuring the total number of call-graph nodes and edges in each application and comparing with the number that IntelliDroid actually executes, we find that IntelliDroid need only execute less than 5% of the code on average in the applications we tested. On the other hand, both random fuzzing and concolic testing, which inject inputs without being aware of the goals of the dynamic analysis, might have to statistically execute 50% or more of the application before it has a better than 50% probability of trigger all the relevant behavior in an application. This conservatively suggests that IntelliDroid might cut execution time by as much as 90% against state-of-the-art input generation methods, and this estimate does not take into account that the number of paths (and thus inputs) is actually exponentially related to the size of the code. In addition, fuzzing and concolic testing do not actively determine the correct order in which inputs must be injected so they may have to try several permutations to achieve full coverage.

IntelliDroid's static analysis time and the number of inputs it must inject is heavily dependent on the number of target APIs specified and the number of target paths it must extract. Thus, to simulate a worst-case scenario with a very comprehensive dynamic analysis engine, we use an even larger set of



(a) Static analysis (b) Dynamic constraint Solving

Figure 3.4: Distribution of IntelliDroid's analysis time

target APIs than in our previous experiments by deriving them from the set of potentially malicious APIs identified by the FlowDroid static analysis tool [10]. The fact that FlowDroid uses static analysis is not relevant—we use this set mainly because it is a large collection of Android APIs that have been identified as potentially malicious. We note that this set of target APIs is a superset of both the target APIs used by TaintDroid and the target APIs used by the hypothetical malware detection tool in our experiments, containing a total of 228 API methods. The extra methods in the FlowDroid set include more conservative sources and sinks, such as those where data is sent via an intent or printed in a log message. These generic API methods are commonly used by both malicious and benign applications.

We measure the time IntelliDroid takes to find invocations of the target APIs and extract the constraints required for input generation using a combination of two datasets: 1260 malware samples from the Android Malware Genome project [144] and 1066 benign applications from the Android Observatory [18]. The Android Observatory dataset was obtained by filtering for applications that declare the permissions necessary for the set of target APIs used in this experiment. The static component, running with a time limit of 60 minutes (enforced for timeliness of results), took an average of 138.4 seconds per application and 88.1% completed analysis within the time limit, with the distribution shown in Figure 3.4a. The bigger set of target APIs and the larger applications in the benign dataset used in this experiment resulted in an average of 1760 inputs generated for each benign application. Despite this large number, the extracted paths still comprise less than 5% of the code in each application on average.

The analysis time of IntelliDroid is dominated by WALA's call-graph extraction and the search for target API invocations, which must be performed on the entire application and accounts for roughly 50% of the static analysis time. We found that the applications that required longer analysis times often used advertisement libraries. The extra code included with these libraries resulted in larger call-graphs and thus, more time was spent searching for target APIs.

Unlike the static component, the dynamic generation of input values must be extremely quick since it is performed for every injected input, of which there could be several thousand per application. Because the constraint solver component of IntelliDroid is completed during run-time, it is especially important that it runs efficiently. We measure the total time taken by the Z3 solve the constraints for all target paths in our dataset to be an average of 4.22 milliseconds, with the distribution show in Figure 3.4b. As a result, we expect the main run-time cost to be that of the dynamic analysis tool itself.

3.6 Limitations

3.6.1 Call-graph generation

Since IntelliDroid aims to generate inputs to event handlers that will trigger all targeted APIs, the tool needs an accurate call-graph to be extracted. Missing edges in the call-graph may cause IntelliDroid to incorrectly believe that a targeted API is not reachable and thus cannot be triggered by any inputs. This is particularly challenging for Android because there are many implicit paths that applications can take via intents, callbacks and other Android-specific facilities. This challenge is not limited to IntelliDroid, but is common to all tools that perform any sort of static analysis on Android applications. We currently model all Android-specific call edges that we are aware of with the exception of exceptions, although we can report exception handlers that invoke a target API as a case we cannot handle and flag it as suspicious. We did not encounter any such cases in our experiments.

Another limitation of our prototype, documented in Section 3.5, is that IntelliDroid currently only detects inter-event data dependencies if they occur through heap variables. Thus, IntelliDroid failed to generate inputs for data flow through a file. It is possible to extend dependency tracking through files, as well as other Android facilities such as content providers, in the same manner. A further limitation is the use of the type-based heap model in IntelliDroid's static analysis, which is imprecise and may not accurately chain paths through dependent heap variables (for instance, if heap variables in a set of dependent paths were determined to alias when they actually do not, the dependency will not be resolved at run-time when the event chain is executed). This incomplete dependency tracking and resolution is addressed in the following chapter (Chapter 4), which proposes a general framework for resolving dependencies to execute target paths in applications.

3.6.2 Extracting and solving constraints

Most technical limitations are the result of limitations of the constraint solver used. In some cases, the extracted constraints contain theories that are undecidable with current solvers, such as trigonometric functions to compute location changes. In other cases, the extracted constraints are too complex for the constraint solver, such as the functions that convert the SMS PDU bytecode format [1] used by the hardware for the SMS message format. IntelliDroid mitigates such shortcomings by extracting the necessary information at run-time and solving for inputs dynamically, but this currently still requires manual instrumentation of the Android framework. Fortunately, this instrumentation need only be done once for each Android version.

Another inherent limitation of inputs generated through constraint solving is that they are not necessarily realistic and thus might not happen in practice. For example, IntelliDroid can manipulate time and other inputs in such a way that it injects a sequence of inputs that is not physically possible, resulting in the detection of malicious behavior that cannot happen in reality. We see no reason why IntelliDroid cannot be enhanced to account for the constraints of the physical world when generating inputs. The main challenge would be enumerating what all the relevant physical constraints are.

3.6.3 Malicious obfuscation

More recent malware may try to obfuscate their behavior by using reflection and encryption. While IntelliDroid has been implemented with support for constant reflection targets during call-graph genera-

tion, in general, IntelliDroid can only compute inputs that will trigger the reflection. It cannot determine path constraints after the reflected call because the statically extracted call-graph is incomplete at that point. A possible solution is to feed dynamic information back to the static component to resolve such issues and build a more complete call-graph. In particular, IntelliDroid's targeted execution can be used to ensure that these statically unresolved method invocations are executed during run-time to obtain the dynamically resolved target.

A similar and related limitation is that IntelliDroid is unable to compute inputs that are processed by complex functions (e.g. encryption or hashing) in a path constraint. This is because constraint solvers are generally unable to determine the inputs to such functions, which are necessary to produce an output that would satisfy the path constraint. Again, in some cases, a system that uses dynamic feedback as described in the case of reflection might allow IntelliDroid to produce inputs in these situations.

Similar to obfuscation through reflection, malware developers may hide malicious behavior in packed applications or native code, which IntelliDroid does not support. IntelliDroid can be used to direct execution to locations in the code where these are used (e.g. `DexClassLoader`, JNI invocations); however, once again, any constraints that occur after these invocations will not be extracted by the static component. While not ideal, the ability to direct execution to these questionable parts of the application is still valuable and can help an attached dynamic tool analyze these portions more effectively. In Chapter 5, we describe how the hybrid techniques used in IntelliDroid can be extended such that when targeting and executing obfuscated locations in application, run-time information is fed back into static analysis to mitigate the effects of obfuscation and enable more complete targeted analysis.

3.6.4 Knowledgeable attacker

Given the above limitations, a suitably knowledgeable attacker has two main avenues for defeating IntelliDroid. First, they can exploit the technical difficulty of extracting a complete call-graph for Android applications by placing the malicious code in a section of code that appears to be disconnected from the rest of the call-graph (i.e., dead code). Since IntelliDroid cannot determine a path to the code, it cannot generate inputs. This can be mitigated by using a more precise model of Android call edges, as well as conservative over-approximation of call edges. The former requires more engineering effort, while the latter may result in IntelliDroid injecting inputs for paths that are not actually possible to execute.

Second, the attacker can process inputs in malicious code with complex functions such as encryption and cryptographic hashing that will defeat the current generation of constraint solvers (and likely many future ones as well). This is a fundamental limitation of targeted execution, as we rely on the extraction and solving of path constraints in order to determine the inputs and data required to execute a target path. In such cases, however, IntelliDroid will experience many constraint solver time-outs, which in itself is anomalous as none arose during our experiments. While not necessarily indicative of malicious behavior, they are infrequent enough to certainly warrant more attention and possibly manual analysis. An alternate method of mitigating the constraint-based limitation is to incorporate a random component to explore the state space of variables for which encryption, hashing, or other complex operations are involved. The combination of constraint solving and random fuzzing for different types of variables can yield greater dynamic coverage of target paths and is similar to the techniques used in hybrid fuzzing/symbolic tools, such as Driller [110].

3.7 Related work

Static analysis of Android applications has been widely used to detect malicious behavior or vulnerabilities [10,30,42,45,50,65,74]. IntelliDroid's static analysis is comparable to the techniques used in previous work, though in some cases it reduces precision for better scalability and analysis speed. However, its static results are then used to guide dynamic analysis, which generally provides better precision than purely static techniques.

IntelliDroid is designed to complement dynamic analysis tools to allow them to quickly identify and analyze paths that are likely to contain malicious behavior. There are a variety of dynamic analysis tools that IntelliDroid could be used with, such as TaintDroid [40], CopperDroid [114], DroidScope [132], VetDroid [140] or RiskRanker [53]. Similarly, IntelliDroid can also be used to aid reverse-engineering or manual analysis using sandboxing analysis tools such as DroidBox [66].

While IntelliDroid's extraction of path constraints is technically a form of symbolic execution, it is performed on a static abstraction of the program rather than on a concrete execution trace. As a result, it should generally provide faster performance than concolic test generation systems such as DART [51], EXE [28] and KLEE [27], which use concrete symbolic execution. In addition, IntelliDroid's main focus is on generating inputs to trigger a specific path rather than obtaining code coverage, making its goals fundamentally different from these systems, as well as more recent Android-focused concolic testing work, such as DynoDroid [75] and the ACTEve algorithm [5]. The work in [83] targets malicious code by exploring paths that branch on interesting input, although the input dependency tracking and constraint extraction is performed dynamically. Purely static constraint extraction and solving has been used in tools like Saturn for verification [128] and hybrid static/dynamic symbolic execution is used in MergePoint [13]. IntelliDroid is also similar to AEG [12], APEG [24], and DyTa [49] which generate malicious inputs to exercise vulnerabilities in program binaries. However, these systems do not target Android applications and thus, do not have to handle consistent input injection or event chains.

The work most closely related to IntelliDroid are hybrid static/dynamic analyses such as AppAudit [127], ContentScope [61], AppIntent [134], SmartDroid [143], Smv-Hunter [109] and Brahmastra [20]. The main difference between IntelliDroid and these systems is the level of fidelity of the injected inputs. IntelliDroid can inject inputs into an actual Android system, enabling integration with full system dynamic analysis tools such as TaintDroid [40]. To do this, it resolves dependencies between paths and between the application and the framework through event chains and device-framework input injection. In contrast, systems like AppAudit and ContentScope rely mainly on the static analysis to find vulnerabilities, and only use dynamic analysis to confirm the feasibility of the paths. Moreover, ContentScope focuses solely on content providers. In contrast, IntelliDroid's goal is to detect malware so it must support and analyze a wider range of behavior. AppIntent also uses static analysis to identify relevant sections of code to execute. However, while IntelliDroid targets specific paths and statically generates concrete inputs, AppIntent requires an exhaustive dynamic symbolic execution to fully explore all behaviors, similar to that used in concolic testing. In addition, AppIntent, SmartDroid, Brahmastra, and Smv-Hunter only handle UI events. In contrast, IntelliDroid is designed to support and trigger all types of events.

3.8 Summary

In this chapter, we introduced the idea of targeted execution and showed how targeted analysis of Android applications can enable more effect dynamic analysis by focusing resources on specific paths and locations of interest. To demonstrate the efficacy of targeted security analysis, we presented the design and implementation of IntelliDroid, a target input generator that specifically exercise code paths in an application that are relevant to a dynamic analysis tool. Through IntelliDroid, we propose several novel ideas that enables high-fidelity execution of target application paths, such as the use of target APIs as an abstraction for dynamic analysis techniques, event chain detection and input generation, and device-framework injection. IntelliDroid is able to identify and generate inputs to trigger the target behavior in applications in a reasonable amount of time (138.4 seconds on average) and use the inputs to dynamically trigger 70 out of 75 malicious behaviors in a set of malware, while saving the dynamic analysis from having to execute 95% of the application code. We further integrated IntelliDroid with a dynamic taint-tracking tool, TaintDroid [40], and show that IntelliDroid-targeted TaintDroid is able to offer better precision than fully static taint-tracking and triggers malicious paths more precisely than a standard off-the-shelf input fuzzer.

With our initial prototype of IntelliDroid, we focused on high-fidelity execution of applications through the event chain and framework injection mechanisms. However, while we showed that IntelliDroid is effective in triggering code paths in known malicious applications, we may also want to perform security analysis on benign or unclassified applications (for example, to determine how they use private data and whether they adhere to their privacy policy or public privacy legislation). During our experimentation, we found that the emphasis on sound execution in IntelliDroid's design requires a high degree of analysis precision and manual engineering, which detrimentally affects its ability to scale. This inhibits its ability to analyze general applications such as the large, commonly used applications from popular marketplaces. In the following chapter, we characterize the event chain and framework injection mechanisms as highly precise dependency resolution techniques and highlight their shortcomings in achieving scalable targeted execution. We describe the importance of dependency resolution for targeted execution and present an alternate approach that supplants the need for event chains, framework injection, and controlled/monitored run-time variables, all of which rely on high analysis precision and extensive manual implementation effort (to achieve complete support for all Android device functionality). Instead, we aim for greater coverage of target code in general applications while maintaining reasonable soundness in the target paths that are executed, enabling much more effective targeted security analysis.

Chapter 4

CAR: Driving execution with context-based dependency resolution

In the previous chapter, we introduced the idea of targeted execution as an alternative to coverage-based dynamic code exploration. We showed that for security analysis, where the goal is to target specific interesting or malicious behavior in an application, targeted execution can be effective in applying computing resources effectively for the dynamic analysis of such behavior. We further showed how our tool, IntelliDroid [123], was able to trigger malicious actions and uncover sensitive information flows in a set of known Android malware.

While we have shown that targeted execution is effective for analyzing Android malware, one may also want to perform security analysis on benign applications for a variety of reasons, such as investigating their usage of sensitive device functionality or to determine whether an unclassified application is benign or malicious (or perhaps questionable [6]). An application marketplace that wishes to analyze their submissions for malicious behavior must be able to handle all types of applications and effectively determine whether the application's functionality and behavior is consistent with the marketplace's policies. A particular challenge are large, popular applications, such as games or social media applications, which often include code from several third-party libraries, require a large number of resources for static analysis, and contain complex functionality. We find that the complexity of such applications poses a challenge for the hybrid techniques proposed in IntelliDroid.

A key challenge of targeted analysis is that unlike normal execution of the application, execution is now restricted to the paths that reach the target behaviors. However, as we found with IntelliDroid, code paths are interconnected through data dependencies on other paths in the application or on external system values. These dependencies on other parts of the program interfere with the goal of restricting the execution to just the paths that trigger the target locations. In small malicious applications, such as those from the Android Malware Genome [144] or Contagio [90] datasets, the simplicity of the applications usually results in fewer dependencies between code paths. However, for most popular applications, the dependencies between application paths significantly hinder the ability to restrict execution and perform targeted analysis. In this chapter, we take a closer look at the challenges of restricted targeted execution of an application and how it affects the analysis of general applications.

We unify our approach for dependencies in targeted analysis by representing a path's dependencies through its *context*, which we define as the constrained inputs and program state that satisfy these

dependencies and are required for the path to execute. This program state includes all types of data shared between application paths (e.g. heap variables) and between the application and any external entities (e.g. access to framework databases or network servers). Ensuring that an isolated path executes with its expected context is difficult. Our prior work in IntelliDroid used multiple approaches to recreate this context at run-time by: (1) tracking dependencies to other paths that can resolve them and executing these dependent paths in advance to set up the state required (i.e. event chains), (2) tracking dependencies on the framework and resolving them by injecting events such that the framework state is consistent (i.e. framework injection), and (3) tracking how run-time data from framework APIs or network servers is used and supplying the required data for a path through monitoring or controlling these values (i.e. run-time constraint data). These design elements result in high-fidelity execution that mimics what would normally happen for a user operating the application but the required level of precision for the static dependency tracking does not scale to large applications. Any lapse in the static dependency resolution, which is undecidable in the general case and incomplete due to the trade-offs made in the static tracking, leads to incomplete execution of the target path and low coverage of target code locations. Alternatively, prior works in guided symbolic execution [15, 91, 134] can also resolve path dependencies as they arise but requires expensive symbolic tracking of all program state, which can also result in low coverage due to scalability issues, and they cannot easily integrate with dynamic analysis tools that operate on concrete execution. Works that ignore context altogether by skipping the parts of the path that enforce the dependencies (e.g. through instrumented forced branching [98,119] or arbitrary invocation [96]) can lead to the execution of many unsound paths and result in a high rate of false positives.

In this chapter, we propose a different approach, **C**ontext **A**pproximation and **R**efinement (Car) [125], to achieve a balance between forced execution, which produces many unsound paths, and full dependency tracking, whose overhead ultimately results in low coverage of the target code. Rather than tracking dependent paths precisely or ignoring them altogether, we show how the combination of static constraint analysis and dynamic error recovery can approximate a context for a path such that execution is driven to the target code location. The approximation is constructed to reduce the amount of false positives, which we define as the execution of unsound or infeasible paths. Car achieves this by: (1) generating an initial context inferred from the desired control flow and (2) dynamically refining the incomplete context when any unresolved dependencies trigger errors during the path's execution. By relaxing the requirement for precise dependency tracking and handling the resulting errors with dynamic dependency recovery, we can achieve much greater coverage of target behaviors in an application while maintaining reasonable soundness in the paths that are executed.

We apply our approach to the targeted analysis of Android applications, which contain a variety of dependencies between event paths through heap variables, files, databases, user interface flow, Android framework callback registrations, etc. We integrate Car with our previous work, IntelliDroid [123], and show that the use of approximated contexts for targeted dynamic analysis enables much higher coverage of target code locations in applications, including accesses to sensitive device functionality for security analysis.

In summary, we make four main contributions in this chapter:

1. We describe Car, which effectively resolves dependencies for an isolated code path by approximating its expected context through hybrid static and dynamic context inference and refinement.

2. We design and implement Car for Android application analysis to show how context approximation can be used for targeted execution.

3. We evaluate Car on the most popular applications in Google Play, which are large and complex, and show that it is able to reach 3.1× more non-trivial target locations for an application than the existing state of the art, with a false detection rate of 9.0%.

4. We show that the use of approximated contexts for path driving can uncover a variety of security-sensitive behaviors in both benign and malicious applications.

4.1 Background on Android applications

As we first described in Chapter 2, Android applications are event-driven. As such, their execution can be described as a series of events received from the framework. Some of these events are triggered as a result of the application loading process and allow the application to set up or tear down state as it is loaded or unloaded. Others are triggered in response to a hardware sensors, such as a location change event from the GPS sensor or a message received event from the cellular chip. A subset of these hardware events are user interface (UI) events that are triggered when the user submits input through interaction with the device's screen and buttons. For all types of events, the application must register a callback method with the framework that is invoked when the event occurs. This registration can be done implicitly by overriding specific methods in framework subclasses (e.g. overriding the `onCreate` method in an Activity class) or explicitly by invoking a callback registration method (e.g. `View.setOnClickListener()`), which allows the application to specify the exact callback method that should be invoked for a particular event type. These registered callback methods serve as entry-point methods to the application, as the application executes only in response to an event received from the framework.

The different entry-points into an application and the code paths stemming from them can sometimes be independent; for instance, one can imagine that code handling the receipt of an SMS message is usually unrelated to the code that handles location changes. However, most event code paths are interconnected through shared program state, such as accesses to heap variables or files. These interconnections often impose constraints on the ordering of events and the execution of code paths within the application, forming dependencies between seemingly independent execution paths.

4.1.1 Types of Android dependencies

We define a path dependency as a constraint on a piece of non-input program state that can affect the control flow of a target code path and affect whether it is executed. While a path's execution can depend on a number of different factors, including the input arguments passed to the entry-point method, we are particularly interested in any dependent external state, as they are affected by the other paths in the application and require special handling when selectively executing application paths. We describe the different types of dependencies in Android applications below.

Heap: We refer to heap dependencies as non-input memory locations that are referenced and used in within a path. In Android applications (and general Java applications), these heap variables are static and instance class field members. When handling heap dependencies in static analysis, we

generally turn to points-to analysis, which maps heap variables, such as field members and local pointers, to the objects they might reference. Points-to analyses can vary in their precision. Some may track objects based on their type, where all heap variables of the same type are conservatively considered to be pointing to the same object (i.e. they are aliased). This was used in our initial prototype of IntelliDroid. More precise points-to analysis may differentiate objects based on their allocation site (i.e. the `new` instruction), resulting in more a fine-grained mapping between variables and objects. However, precise points-to analysis requires both time and memory to compute and store these alias mappings. Often, there is still some imprecision in the points-to results resulting from the lack of run-time data during static analysis.

Files: File dependencies refer to accesses to locations within persistent disk storage, either on the device file system or an attached storage device (e.g. SD card). Without application-specific knowledge, files must be treated as an unstructured data store, resulting in significant imprecision when tracking their accesses (for instance, it can be very difficult to determine whether a write access to a file followed by a read access are actually referencing the same location within the file). Luckily, many file accesses in Android applications are made through APIs in the framework that enforce a specific file format, namely database and shared preference accesses.

Databases: Database dependencies are similar to file dependencies but their accesses are made through a defined query/update interface. While any database can be used by Android applications, the framework provides APIs for SQLite databases [1].

Shared preferences: Android applications can read and store data using the provided *shared preferences* mechanism in the framework. Shared preferences are essentially files that store a list of key-value mappings and are accessed through specific API methods.

Framework/system APIs: Application paths may depend on specific values returned from framework or system API methods. These values can depend on the execution environment (e.g. the device, installed libraries or applications, user data, etc.) or on values that were set by previous calls to the API methods.

Interprocess communication (IPC): In Android, data can passed and shared between different application components using `intents`, a special serializable object that stores extra data in an attached `Bundle` object. The `Bundle` is essentially a heap object that enforces key-value access of the enclosed data. The intent mechanism enables one application component to start another by specifying the target of the intent. The intent object is passed to the framework, which starts or resumes the target component.

Framework callbacks: The entry-points in an application can only be invoked by the framework after they have been registered, either implicitly by overriding framework methods for Android application components or explicitly through the invocation of a registration API method. Therefore, when considering a code path from a particular entry-point, the path cannot actually be executed unless the entry-point has been registered, resulting in a callback dependency on the registration event. Similarly, asynchronous tasks and runnables in Java are also control dependent on the code that initializes or "registers" them in the Java runtime.

[1] SQLite homepage: https://www.sqlite.org/index.html

User interactions and flow (UI): UI entry-point methods are technically a type of callback method and has the same callback registration dependency. However, UI callbacks have an addition constraint in that they can only be triggered when their associated UI element is visible and interactive on the device display. Therefore, we can consider all code paths from UI callbacks to be control dependent on their corresponding UI elements and on the framework's UI hierarchy, which models the UI objects currently visible to the user.

4.2 Motivating example

In Figure 4.1, we present an example of a code path to a sensitive behavior in an Android application, in which the execution of the path depends on data from other parts of the application. While this is not extracted from a real application, it contains similar code patterns.

Figure 4.1a shows a sensitive action taken when a keyword is detected within the user's text input after the user presses the "enter" button. The key event handler (`EnterKeyListener`) is the entry-point of this path. It first checks if the key pressed is the "enter" key and if so, it calls `UserInputText`, a custom UI text element, to check the user's input text. It determines whether a keyword has been loaded, checks it against the user's input, and saves the input text if the keyword was detected. Finally, it performs a sensitive action if these conditions have been met (e.g. it may send the input text to a network server or access a device sensor).

Figure 4.1b shows how the non-input dependencies are resolved during normal execution by other application paths. When the target path is triggered, it expects that these other dependent paths have already been executed. Furthermore, there may be recursive constraints or dependencies in the dependent paths; for instance, the `detectionMode` and `keyword` fields are set on receipt of an SMS message from a certain address, which might be set in yet another dependent path. Likewise, the enclosing component, `InnerActivity`, is not the application's main activity and may require navigation through multiple screens in the UI flow before it can be started.

To execute the path in Figure 4.1a, there are several approaches to resolving the path's constraints and dependencies such that it reaches the target location (we do not focus on coverage-based approaches, such as fuzzing, since they aim to execute all application paths rather than focusing on the target path). A dynamic approach, such as guided symbolic execution [15, 91, 134], can resolve dependencies along the target path by symbolically tracking the input and non-input variables that are accessed as it executes (e.g. `detectionMode` and `keyword`). It uses the symbolic data to generate constraints that determine the values required for the path's control flow. However, this symbolic tracking is expensive and requires a symbolic execution environment that does not integrate easily with the numerous dynamic analysis tools are based on concrete execution [40, 53, 66, 113, 114, 132, 140].

A hybrid approach could instead use static analysis to resolve dependencies and guide concrete execution of the target path [98, 99, 119, 123]. There are several methods of performing this static guiding or targeting, which can be differentiated based on the execution abstraction used. A holistic abstraction that is faithful to the normal execution of an application (i.e. sound and with high fidelity) is to abstract at the level of the Android framework, which manages the execution of an application and facilitates its access to underlying hardware and system functionality. IntelliDroid [123] uses this framework abstraction and guides execution by injecting events into the framework to trigger a target path in the application. It performs static dependency tracking to determine exactly which framework

```
1  class EnterKeyListener implements View.OnKeyListener {
2    @Override public void onKey(View v, int code, KeyEvent event) {
3      if (code == KeyEvent.KEYCODE_ENTER) {
4        handleEnterKey(v, event);
5      }
6    }
7    private void handleEnterKey(View v, KeyEvent event) {
8      UserInputText textView = (UserInputText)v;
9      textView.checkText();
10   }
11 }
12 class UserInputText extends EditText {
13   static public String keyword = null;
14   int detectionMode = 0;
15   public List<String> detectedInput = null;
16
17   public void checkText() {
18     if (detectionMode == 2 && detectMode2()) {
19       recordInput(getText());
20       <target sensitive action>
21     } else { ... }
22   }
23   private void recordInput(String text) {
24     detectedInput.add(text);
25     write(<output file>, detectedInput);
26   }
27   private boolean detectMode2() { return hasKeyword() && keyword.equals(getText()); }
28   private boolean hasKeyword() { return keyword != null && !keyword.isEmpty(); }
29 }
```

(a) Target path that performs a sensitive action when a keyword is detected in the user's input after they press the "enter" key

```
30 /* Secondary activity in the application */
31 class InnerActivity extends Activity {
32   @Override public onResume(Bundle savedInstanceState) {
33     View view = new UserInputText(...);
34     View.OnKeyListener listen = new EnterKeyListener();
35     process(view, listen);
36
37     view = new OtherTextView(...);
38     listen = new OtherKeyListener();
39     process(view, listen);
40   }
41   private void process(View v, View.OnKeyListener k) {
42     /* Register a key event handler for the view */
43     v.setOnKeyListener(k)};
44   }
45
46   /* Called by an SMS receiver elsewhere (not shown) */
47   public void onSmsReceived(SmsMessage msg) {
48     if (msg.getOriginatingAddress().equals(ServerInfo.getAddress())) {
49       UserInputText view = findViewById(...);
50       view.detectionMode = extractMode(msg);
51       view.detectedInput = new ArrayList<String>();
52       UserInputText.keyword = extractKeyword(msg);
53     }
54   }
55 }
```

(b) Dependent code paths that register the key event handler with the framework and set the heap variables to their expected state

Figure 4.1: Example of targeting sensitive behavior in an Android application

events to inject such that the path's constraints and dependencies are resolved. For Figure 4.1, they might execute the following ordered chain of events to reach the target code:

i) UI or lifecycle event(s) to start `InnerActivity`
ii) Lifecycle event to register `EnterKeyListener`
iii) SMS event with an input message that sets the required values for `detectionMode` and `keyword`
iv) UI key event for the target path from `onKey()`

There are two drawbacks to this approach: (1) the surface area of the abstraction (i.e. the points at which to inject and manipulate input events to guide the path dynamically) is large and requires custom engineering to inject each type of event into the Android framework, which is numerous due to the number of hardware sensors available and the many ways in which UI elements can be manipulated; and (2) the abstraction area (i.e. the degree of analysis required between the injection point and the dependencies they resolve) is also large and requires precise static dependency tracking across the application and framework. This tracking is necessary to determine the framework events to inject that will set the dependent state required (e.g. a heap variable or file value) and it must be precise, as we need to know the exact input and non-input values to translate a static path into a dynamically executed one. Both drawbacks limit the scalability of this approach for large applications that have multiple dependencies between paths and access a variety of functionality from the framework. For instance, IntelliDroid does not support the injection of UI events, which precludes it from triggering paths in most Android applications (it would also be unable to inject the UI key event in our example path). Furthermore, any lapse in the static dependency tracking due to scalability issues results in unresolved dependencies and the inability to execute the target path.

The execution abstraction could be reduced to decrease the amount of dependency tracking required, enabling greater coverage of target code in execution (i.e. completeness). On the other end of the spectrum, Harvester [98] and DirectDroid [119] operate on path slices leading to target code locations (i.e. they abstract at the instruction level). To guide execution, they use static instrumentation to force specific branch outcomes to achieve the control flow required. Rather than resolving dependencies, this essentially bypasses them as the conditions in branch statements are no longer enforced. However, this can easily lead to inconsistent data values such as the forcing of the null check in Line 28, which leads to an inconsistency if the checked object is actually null, the branch is forced anyway, and the object is then dereferenced later in the line. The forcing of multiple branches can also lead to a combination of infeasible branch outcomes as program logic is modified without reconciling the control flow with the data compared within the branches. This can lead to the execution of many unrealistic or unsound paths. As such, forced execution is well-suited for obtaining coverage of target locations but less so for the analysis of security-related behaviors at these locations, which would require tracking how applications access and manipulate system resources and data. It can provide an upper bound for the coverage of target locations that can be achieved by Car or other targeted tools; however, due to its soundness issues, it is not a reliable means for determining if an application will perform a malicious action or not.

Other work, such as FuzzDroid [99] or CrashScope [82], abstracts at the application level. FuzzDroid achieves semi-targeted execution by manipulating the inputs received at application entry-points and variables retrieved from framework and system APIs. However, it does not track or resolve dependencies on state within the application, such as the constraints on the `detectionMode` and `keyword` fields, and would have to execute the dependent paths in order by chance to reach the target code location. This would also be the case for any other untargeted fuzzing tool. Similarly, CrashScope manipulates user

text and system-level resources, such as network or sensor states, to explore different application components that depend on these states (as determined through static analysis). However, this exploration is conducted through random fuzzing and does not handle dependencies on internal application data, which is necessary for full targeting. The semi-targeted execution is geared toward full coverage of the application, which is ideal for CrashScope's goal of triggering crashes, but is less effective when one wishes to trigger only selective security-related locations, which would require fine-grained path guiding.

In Car, we propose a fully targeted approach that achieves a balance between forced branching, which can lead to many unsound paths, and full dependency tracking across the Android system, which is unscalable. Our approach draws on an analogy between targeted execution and developer unit testing. In unit testing, while a test suite aims for full coverage of the tested code, an individual test case is meant to exercise a single code path reliably so that when a test fails, the error location can be pinpointed. Since the path may depend on other parts of the application, to avoid executing those other parts and polluting the test case with irrelevant code, a set of mock classes is used to resolve the dependencies. The developer uses their knowledge of the code base to identify these dependencies and determine how they should be resolved; this resolution is achieved by controlling the values returned from accesses to the mock classes/objects.

With Car, we endeavor to replicate the inherent knowledge of the developer by inferring a path's dependencies from the control and data flow of the application's code. We target specific code paths by abstracting execution at the level of code objects and guiding execution by controlling the inputs, fields, and method return values accessed by a path. The controlled object accesses form the context for a path and are similar to the controlling of accesses to mock classes used in unit testing. There are several advantages to an object-level abstraction for automated dependency resolution. First, unlike bigger, lower-level abstractions such as the Android framework or Android application components, objects are agnostic to the Android system and can be automatically used to handle the injection of different types of inputs and dependencies without custom engineering or domain specific knowledge (e.g. to inject a specific type of framework event or to determine the system APIs to constrain). Furthermore, the smaller abstraction reduces the amount of dependency tracking required to determine how the values that are injected and manipulated (i.e. the objects injected and visible to a target path) affect the dependencies they resolve (object accesses along the path). Dependencies on state within the application and from the framework, such as heap variables or API calls, are implicitly handled as accesses to this state are performed through field accesses and method invocations. Second, unlike even smaller abstractions such as instructions or path slices, injecting and manipulating objects enables the full execution of methods in the target call path rather than specific instructions in a path slice, resulting in more sound execution. Also, the tracking of dependencies on object accesses ensures that the data values accessed within a path are consistent, which would not be the case for forced branching. In our example from Figure 4.1a, Car would trigger the target path by injecting the input object `v` such that the constrained field access to `v.detectionMode` is resolved. It will also ensure that the static field access to `UserInputText::keyword` is resolved by controlling the field value visible to the target path.

In Table 4.1, we summarize the dependency handling techniques used by past tools that execute a restricted set of paths in an Android application. Some of these techniques aim to resolve the dependencies in a path (e.g. framework injection or event chains) while others bypass them altogether (e.g. forced branching). In contrast to these techniques, Car uses the injection and manipulation of code objects to construct a hybrid context for each target path in order to resolve its dependencies. Our

unified approach allows Car to handle general event and dependency types in Android applications while resolving the constraints and dependencies imposed by the target paths, maintaining soundness in the paths' execution.

Table 4.1: Comparison of dependency handling techniques used in past tools and in Car

Tools	**Dependency types**				
	Heap	**Persistent storage**	**Framework & system APIs**	**IPC & callbacks**	**UI callbacks**
IntelliDroid [123]	Statically extracted event chain	Run-time data (shared prefs. only)	Framework injection (SMS and location); Run-time data (specific APIs)	Event chain	Event chain & framework injection (limited subset of click events)
Harvester [98]	Forced branching in static slice	Prepend slice with all static writes	Forced branches; Dummy values	Direct event injection	Direct event injection
FuzzDroid [99]	None (relies on random fuzzing)	Static value provider	Static value provider	Direct event injection	Direct event injection
DirectDroid [119]	Forced branches & null recovery	Static value provider	Static value provider	Direct event injection	Direct event injection
Car [125]	Hybrid context	Hybrid context	Hybrid context	Direct event injection	Direct event injection

4.3 Design

Using the object-level abstraction for targeted execution, Car constructs a context for a target path to resolve its dependencies on inputs, fields, and methods. However, using static dependency tracking to construct this context a priori can fail due to the trade-off between precision, completeness, and scalability. When this happens, the unresolved dependencies limit dynamic target coverage or lead to unsound paths. Car addresses this in a hybrid approach. We start from a set of statically extracted constraints that are necessarily incomplete due to the static analysis trade-offs that must be made (Section 4.3.1). We account for this by separating the constraint extraction and dependency resolution of a path into three levels of progressive approximation (realized, modeled, and unconstrained). The constraints are used to infer an initial approximate context for the path. This context is constructed at run-time using a series of constrained subclasses that can inject constrained inputs, fields, and method return values automatically (Section 4.3.2). We then refine this context dynamically by monitoring for unresolved dependencies and resolving them by re-using objects from the application's execution (Section 4.3.3). We show how the interaction between static inference and dynamic refinement of the context achieves more effective dependency resolution and execution of target paths.

4.3.1 Static constraint analysis

The initial static analysis to extract target paths uses the existing targeted execution framework in IntelliDroid. It first constructs a conservative context-insensitive call-graph to find code paths to a set of configurable target behaviors. For each target path, a context- and flow-sensitive analysis extracts its constraints on inputs, fields, and methods that define its execution. Constraints on inputs can be resolved when injecting the path's entry event with values derived from the solved constraints. Constraints that cannot be resolved through entry-point inputs (e.g. heap or framework API values) form dependencies on other paths in the application.

The separation of the initial path extraction from the constraint analysis balances the need for coverage when detecting target behaviors and the need for precision when extracting path constraints. The precision required to determine the exact program values required for the path (akin to symbolic execution) is expensive and impractical to perform over the entire application. The trade-off between the precision and cost of this analysis impacts its completeness: to fully analyze all constraints, it requires performing precise symbolic constraint analysis over methods directly in the target's call path and all side/auxiliary methods they invoke; essentially, this is comprised of the sub-graph of the call-graph starting at the path's entry-point, which can be prohibitively large. In our earlier example from Figure 4.1, this would require constraint analysis over the methods in the target call path as well as the auxiliary methods `detectMode2()`, `hasKeyword()`, `getText()`, `recordInput()`, `write()`, and any other methods they may invoke.

A purely static analysis can sacrifice the precision of this analysis for scalability such that constraints and dependencies can be tracked across this sub-graph and across the application (albeit imprecisely). However, in a hybrid system where the static results are then used to generate values for execution, this can result in under-constrained program state that does not actually resolve a path's dependencies dynamically, resulting in the incomplete execution of the path. For instance, the imprecise results of an any-path analysis without context sensitivity may indicate that a constrained variable may have one of several values (due to the union of data flows from different paths) while only one of these values is actually valid for the execution of a particular target path. Previous hybrid tools for Android accounted for this by using a different trade-off where precision is maintained but the completeness of the analysis on the sub-graph is heuristically scaled back: IntelliDroid [123] only performs constraint analysis on the methods directly in the target's call path and one level of auxiliary methods they invoke, and Harvester [98] uses a cut-off value to limit the depth of analysis into callers and callees when computing the path slice to force execute. Constraints imposed by code in deeper auxiliary/callee methods are therefore ignored and unresolved. This incompleteness is propagated when constraints are used to perform dependency tracking, ultimately resulting in unresolved dependencies and the incomplete execution of target paths.

The key intuition behind Car's design is that a combination of both static and dynamic techniques can be used to mitigate this limitation in dependency tracking. We directly address the trade-off between precision, completeness, and scalability by separating the dependency resolution of a target path into three levels of approximation, each of which are handled separately. This separation is illustrated in Figure 4.2 for the code example in Figure 4.1.

The approximation is determined by the scope of the static constraint analysis (i.e. the depth of auxiliary methods analyzed). Methods that are within the scope form the *realized* path and are executed in full with constrained inputs. Methods at the boundary of the scope (they are invoked by realized

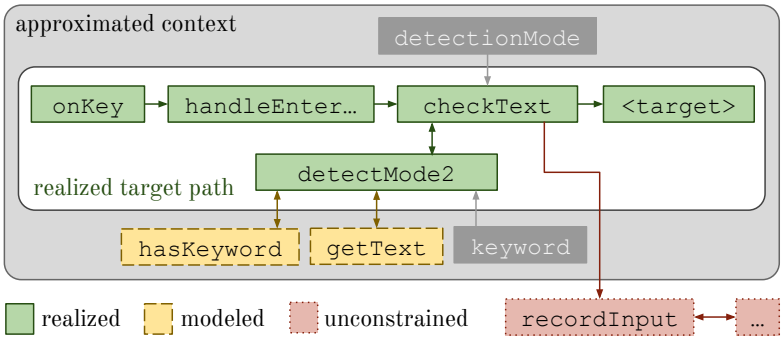


Figure 4.2: Separation of the target path flow based on the approximation level and scope of the constraint analysis

methods but are themselves excluded from constraint analysis) are *modeled* if there are any constraints on their return values. These constraints will be captured by the analysis performed on their realized caller methods. We treat the constrained return values as part of the program state accessed by the path and encode them in the approximated context.

All other methods that might be invoked by the target path are *unconstrained*. We execute them to ensure that any side effects are performed and can be analyzed later, such as updates to persistent storage or the scheduling of future tasks. These side effects are essentially implicit dependencies that our object-level constraint analysis does not track but that might affect the semantics of the path. Modeled methods can also be executed in full, with only their return values constrained; however, we do not do this, as it can result in inconsistencies between any program state they modify (which is outside the scope of constraint analysis) and the constrained return values. While they may also have side effects, our approximation assumes that their primary function is the return value on which there is an explicit dependency by the realized path.

In Figure 4.2, which is based on a constraint analysis that extends to one level of auxiliary methods, the methods in the target's call path and the callee `v.detectMode2()` would be realized, while `v.hasKeyword()` and `v.getText()` are modeled since they must return specific values for the path's control flow. The method `v.recordInput()` is unconstrained, as the realized path imposes no constraints on its return value.

Any program state that influences the normal control flow of the realized path is handled by the values extracted from the constraint analysis. Therefore, the execution of unconstrained methods does not directly affect the realized path, except if an error occurs while the method is running. These errors, which will trigger undesired exceptional control flow and can lead to non-termination of the realized path, arise due to unresolved dependencies on objects; this is a consequence of limiting the scope of the constraint analysis (we do not extract constraints for unconstrained methods). Car instead handles these dependencies as they occur during execution through dynamic context refinement, which tries to recover from them. We describe the refinement process in Section 4.3.3.

The scope of the static constraint analysis and degree of path realization are configurable. We aim to increase the scope of the constraint analysis as much as possible, as it ensures that a greater portion of the target path can be realized and analyzed, with more of its dependencies resolved precisely; however, we must balance this with the scalability and cost (i.e. the time and memory requirements) of the overall targeted analysis. Increasing the constraint analysis scope and realizing more of the target path will

result in more sound execution, but requires more resources. This can reduce coverage if the constraint analysis cannot complete within the allotted time or memory (leading to unresolved dependencies) or if the larger approximated context must satisfy constraints that cannot be resolved statically (e.g. a constrained encrypted value inside an auxiliary method). Reducing the scope will result in more unconstrained methods that may require dynamic context refinement to resolve dependencies, which is based on recovery heuristics and can lead to the execution of unsound paths. In Car, we perform static constraint analysis with one level of auxiliary methods which, from our coverage and false positive results, achieves a good balance of soundness and completeness through the size of the approximated context (i.e. number of solved constraints on program state) and the amount of dynamic recovery required.

4.3.2 Generating an approximate context

While static constraint analysis can determine the values required to resolve a path's dependencies, concrete execution of the path requires that these values be injected into application as it is running. The dependencies can include constrained inputs and global/system state that the path accesses. To inject the constrained values for concrete execution of the target path, Car uses the context to set up the dependent state required.

Given an extracted path, Car triggers the path dynamically by directly invoking its entry-point method. The path's dependencies are resolved through a generated context that is automatically inferred from its constraints. The construction of the context is similar to the generation of input values from constraints in symbolic execution systems; however, in Car, we want to generate the context for input and non-input variables, and enforce it as the target path is executed concretely. To do this, Car uses a static phase that generates code to set up the context (which we call the path-driving framework) and a dynamic phase that invokes the generated code to produce the context at run-time. The majority of the work in constructing the initial approximated context is therefore in the static generation of the code in the path-driving framework. The generation and invocation of this code is fully automated and is performed for each target path.

4.3.2.1 Inferring and constructing the context

For each variable in the extracted path constraints, we infer the context by encoding a value from the solved constraints that can satisfy it—we refer to these as the enforced context value for a given constrained variable. There can be many values that satisfy the constraints and therefore many feasible contexts, though we only generate one for each path. For Figure 4.1, a context that can satisfy the target path's constraints might be:

$$
\begin{aligned}
v = \texttt{UserInputText}\ \{\ & \texttt{detectionMode} = 2, \\
& \texttt{hasKeyword}() = \mathit{true}, \\
& \texttt{getText}() = \text{“}\texttt{a}\text{”}\ \} \\
\mathit{code} = \texttt{KeyEvent.KEYCODE\_ENTER}& \\
\mathit{event} = \texttt{KeyEvent}\ \{\}& \\
\mathit{UserInputText} :: \mathit{keyword} = \text{“}\texttt{a}\text{”}&
\end{aligned}
$$

To enforce context values when injecting the target path, Car considers whether they are input or

```
1 class ConstrainedUserInputText_Path0 extends
2     UserInputText {
3   public ConstrainedUserInputText_Path0(Context c) {
4     super(c);
5     this.detectionMode = 2;
6   }
7   @Override public bool hasKeyword() {
8     return true;
9   }
10  @Override public String getText() {
11    return "a";
12  }
13 }
```

(a) Constrained subclass enforced for the input object for `v`

```
14 public static void PathDriver0 {
15   EnterKeyListener receiver = new EnterKeyListener();
16   View arg1 = new ConstrainedUserInputText_Path0(null);
17   int arg2 = KeyEvent.KEYCODE_ENTER;
18   KeyEvent arg3 = new KeyEvent(0, 0);
19
20   /* Constrain global heap state */
21   UserInputText.keyword = "a";
22
23   /* Inject the path by invoking its entry-point */
24   receiver.onKey(arg1, arg2, arg3);
25 }
```

(b) Path driver method constructing the path's context at run-time

Figure 4.3: Path-driving framework automatically generated by Car for the target path in Figure 4.1

non-input variables and whether they are primitives or objects (we treat strings as primitive-like). The primary challenge is constraining accesses to objects. For example, to enforce the context for Figure 4.1, we must inject the target path with an object for the variable `v` such that when the path accesses the field `detectionMode` at Line 18 and invokes the methods `hasKeyword()` and `getText()` at Line 27, it receives the constrained context values. To accomplish this, Car generates a *constrained subclass* for each constrained object, which is a modified version of the variable's class type similar to the mock classes used in unit testing [80]. Each constrained subclass is only used within the context for one path for a particular constrained object variable. Constraints placed on members of the object are handled by controlling method return and field values. Method return values (i.e. for modeled methods) are handled by overriding the method in the constrained subclass such that they return their context values. Field members are set to their context values in the subclass's constructor when the object is initialized. An example of the constrained subclass generated for the input object for `v` in Figure 4.1 is shown in Figure 4.3a.

The generation of constrained subclasses requires a base class to extend. One might assume that this would be the constrained variable's declared type in the application bytecode, but it often cannot

be directly used for two reasons: (1) the declared type is abstract and cannot be instantiated to create a context object (especially true for constraints on inputs of entry-point methods), and (2) the target path itself may assume a specific type (i.e. it imposes type constraints, like Line 8 in Figure 4.1). To resolve these conditions, when performing static constraint analysis, Car also extracts type constraints for variables used in class- or type-related operations, such as `cast`, `instanceof`, and any object accesses. When constructing a constrained object, it searches the application's class hierarchy tree for a concrete class that fulfills all of the type constraints on the variable. If there are multiple such classes, Car randomly chooses the most specific subclass (i.e. a leaf of the class hierarchy tree), preferring an application class over one declared by the Android framework or Java runtime library. This heuristic is based on the notion that the application has extended a class for a purpose and unconstrained methods may depend on the extra functionality.

We provide details on the initialization of constrained subclasses in Section 4.4.1.1, which is based on the assumption that they should behave in the same manner as their base classes, with the exception of constrained members. The construction of constrained subclasses can be recursive if constraints exist for chained object references. For instance, a path may require that when method `x.foo()` is invoked, it returns an object `y` where the field `y.a` contains a specific value. Car would first construct a constrained subclass for `x` that overrides method `foo()`. This method must return a constrained object for `y`, so it constructs another constrained subclass where field `a` is set to the required context value.

Our current implementation returns only one context value for each constrained variable. However, we can easily return a sequence of values for cases when the path makes multiple accesses to the same object/state (e.g. invocations to the same method) and expects a different value for each access.

4.3.2.2 Driving the target path

In addition to initializing constrained variables with context values (either a solved primitive value or an instantiated constrained subclass object), they must be injected with the target path. Input constraints are enforced by passing their context values as arguments to the invocation to the path's entry-point method. Non-input constrained variables include accesses to static fields and return values for static method invocations. For static fields, we explicitly set the field to its context value prior to injecting the target path. For static methods, we instrument the method to return the context value when the target path is executing and to invoke the method's original functionality otherwise.

To manage the constrained variables and context values, a central path driver method is generated for each path to construct its approximated context at run-time. When invoked, it instantiates the constrained subclasses, sets the input and non-input constrained variables, and injects the path's entry-point method with initialized inputs. For unconstrained primitive inputs, they are set to zero or an empty string. If they are unconstrained objects, they are initialized with an instantiated object of the declared input parameter type (or a concrete subtype if it is abstract), with fields set to a default zero or null value. Figure 4.3b shows the path driver method for the target path in Figure 4.1. During dynamic analysis, this method will be invoked in the application's process to inject the target path, which in turn will trigger the target sensitive location.

4.3.3 Dynamic context refinement

The initial approximated context set up by the path-driving framework is limited by the scope of the static constraint analysis. Unconstrained methods that are invoked by the target path, but that were not analyzed, may depend on program state and objects outside of the generated context. Such an error can occur for the target path in Figure 4.1 for the method `v.recordInput()`, which is an unconstrained auxiliary method. However, its dereference of `detectedInput` imposes a dependency since the field may not yet have been initialized. When the target path is triggered by the path driver method in Figure 4.3b, execution of the path can end in a runtime exception or crash inside `recordInput()` without reaching the target sensitive location. Car handles these unresolved dependencies by refining the path's incomplete context through dynamic dependency recovery. In essence, the dependencies which could not be resolved through static means (i.e. tracked by constraint analysis and resolved through the generated context) are now handled by inferring them from the resulting dynamic dependency error. The refinement process is implemented within the Android runtime executing the application (ART) and is composed of two parts: monitoring for unresolved dependency errors and recovering from the error to return to the target path.

4.3.3.1 Unresolved dependency monitoring

Unresolved dependency errors arise in various forms and usually occur when an unconstrained value in the injected inputs for the realized path (e.g. `detectedInput` in `arg1` in Figure 4.3b) is passed to an unconstrained method that does impose a constraint. The most common error is a runtime-generated `NullPointerException`, mainly due to the default null values used for unconstrained variables when injecting paths. Other common exceptions are `InvalidArgumentException` and `IllegalStateException`, which are thrown by the application after a check for well-formed input. For primitive variables, a runtime `ArithmeticException` can be thrown for divide-by-zero errors, which are caused by the default zero values. In general, technically any exception can be thrown by the application in response to unexpected (i.e. unresolved) program state, including custom exception class types. We instrument the Android runtime's (ART) exception handling code to detect when common dependency-related exceptions occur while a path driver method is executing a target path.

4.3.3.2 Error recovery

Recovery from a dependency error requires the identification of the error's root cause, which is the dependent variable that is incorrectly unconstrained during the execution of an unconstrained method. For exceptions generated by the runtime, such as a `NullPointerException`, the root cause can be identified by the runtime processes that generated the exception. For exceptions that are thrown by the application, the cause of the exception is determined by the application itself (e.g. in a conditional branch statement performing an error check) and the runtime environment only propagates the exception object. Therefore, to identify the root cause of the dependency error, information from the application's control and data flow is required. This can be provided to Car's dynamic refinement process through an intraprocedural static analysis of `throw` instructions to extract their root cause variables, which can be inferred from the conditional branches that directly guard the instructions.

In our current implementation of Car, we only perform full recovery for runtime-generated exceptions where the root cause of the dependency error is available directly from ART. This covers the runtime-

generated `NullPointerException`, which comprises over 75% of dependency-related errors we saw in our evaluation. For other types of dependency-related errors, such as the application-thrown `IllegalStateException` and `IllegalArgumentException`, we perform partial recovery by suppressing the exception and continuing execution in the caller of the method where the exception occurred (i.e. oblivious to the failure [101]). As unresolved dependencies only occur in unconstrained methods, the realized path is not affected. We find that many of these other dependency-related exceptions are due to null variables and are thrown by the application to avoid an unexpected runtime exception later when the variable is used; when they are suppressed, they eventually propagate to trigger a null exception as well, at which point Car can perform full recovery and refine the context. Car's implementation can be extended to instead recover fully from these exceptions by using static information extracted from the `throw` instruction that triggered the exception, which would indicate the root cause variable to recover. We leave this extension to future work.

To recover from a `NullPointerException`, Car identifies the error's root cause through instrumentation in the runtime's exception processing routines, which track the register that caused the error. Car overwrites this location with the address of an object that can resolve the dependency, which we call the *recovery object*, and transparently returns execution to the instruction where the error occurred (i.e. the *recovery location*). Since the register now contains a value expected by the application, execution can continue along the target path as if the error never occurred. The recovery object does not affect the control flow of the realized target path, as dependency recovery is needed only for unconstrained methods and any variables influencing the realized path would have been captured by the constraint analysis and already resolved.

To obtain the recovery object, a seemingly obvious approach would be to simply instantiate an object based on the declared type of the root cause variable. However, it is unclear how to initialize the object properly, as constraints are not extracted for unconstrained methods. Initializing the fields of the object to null values will trigger further errors in the execution, as the application will expect objects to be initialized properly. Invoking the default constructor may set some fields to an expected default value but many application classes define a custom parameterized constructor that populate its fields from the arguments (it may even throw a further exception if these arguments are null). Instantiating other objects to populate the fields or constructor arguments can lead to a series of instantiations due to chained object references. In the worst case, one might reinstantiate all of the objects in the application for each recovery.

Instead, Car heuristically obtains the recovery object by re-using an already instantiated object from the application. It maintains a cache of recently allocated objects and constrains the re-used object based on type compatibility with the root cause variable. By re-using objects in this way, we are essentially resolving a dependency as if the injected path had been triggered normally within the application process and preceded by its dependent paths that would have provided the dependent object. If no compatible object was previously allocated for the recovery object, Car then instantiates a new object and initializes it with null values. Similar to the choice of base class for constrained subclasses, we heuristically choose the most specific subclass to re-use or instantiate, using class hierarchy information passed from static analysis.

If the recovery object does not resolve the dependency error, the error may re-occur. This occurs regularly for certain chained object/class reference patterns that are not enforced by the object's declared type or the application's class hierarchy, but must be followed to obtain the correct execution (e.g. a

wrapper class that is a subclass of the wrapped class, but it cannot wrap itself without entering an infinite loop). Car will try to resolve the dependency with another recovery object, which can be a different re-used object or a new object of a different compatible type. For each error location, Car tracks the recovery objects used in order to avoid performing the same recovery action for a recurring error.

To complete the example from Figure 4.1, if `detectedInput` is null when the realized target path is executed, a `NullPointerException` will be thrown in `recordInput()` at Line 24. When the exception is generated, Car will detect that a dependency error has occurred during `PathDriver0()`, determine that the faulting instruction is a method invocation, and that the (null) receiver object is declared as a `List`. It will search for a previously allocated `List` object or instantiate a new concrete subclass of `List`, which is abstract (e.g. `ArrayList`). After generating the recovery `List` object, Car will set the register for `detectedInput` to its address and return the execution to `recordInput()`, where the invocation in Line 24 will succeed and the execution will continue to the file write instruction and the target sensitive action.

Car's detection of dependency errors is conservative and it may detect cases in which the exception is unrelated to unresolved dependencies and would have occurred during normal execution as well. Furthermore, the exception may not have affected the target path's execution—the application's `catch` blocks, which we avoid, can technically return the execution back to the target path (though we find this to be rare). We assume that even though we bypass normal exception handling in these cases, we still see normal application behavior when executing the target path. However, this can affect the soundness of the paths Car executes and is part of the trade-off made to enable practical dependency handling without requiring expensive and precise dependency tracking.

4.4 Implementation

Car's implementation for Android targeted execution consists of: (1) a static context inference component to extract target paths and generate approximated contexts, (2) a dynamic driving controller to inject the paths, and (3) an instrumented version of the Android OS (AOSP) to facilitate the path driving process and perform dynamic context refinement. The static component is written in Java and operates directly on the application's bytecode (APK file). It uses Soot [116] for its base static analysis and the Java bindings for Z3 [36] for constraint solving. The dynamic controller is written in Python and the instrumentation of AOSP is for Android 10.

4.4.1 Static targeting and context inference

The static extraction of target paths and constraints is based our implementation of IntelliDroid. We ported IntelliDroid to use the Soot [116] static analysis framework, which provides direct support for the instrumentation of DEX bytecode via the smali/dexpler [54] library. Previously, IntelliDroid used the WALA analysis framework [120], which does not have a backend for DEX bytecode. While instrumentation could have been achieved by using WALA with Java-to-DEX conversion tools [37, 85], we found that applications (specifically malicious applications) often use very esoteric aspects of the bytecode specification that are not always supported by conversion tools. In conjunction with the port from WALA to Soot, we also switched from a type-based points-to analysis to an allocation-based analysis (Spark [68]) that is used by default in Soot and provides more precise results. We further augment

IntelliDroid's constraint analysis with greater support for different types of constraints, including class type constraints and non-null constraints for instance object accesses. We also modify the extraction of target paths to extract three paths for each target location to account for the possibility of infeasible paths in the conservative static call-graph.

For each extracted path, Car must generate code to construct the approximated path contexts, including the path driver methods and constrained subclasses. We also instrument application code at target locations to log when the target has been reached for our evaluation. We use Soot's bytecode generation to construct these elements and store them with the rest of the static analysis output (we do not repackage the original APK or binary of the application).

4.4.1.1 Initialization of constrained subclasses

When Car generates constrained subclasses for approximated contexts, it needs to specify how they should be initialized when they are instantiated. We assume that a constrained subclass should behave in the same manner as its extended base class for unconstrained members. For each constrained subclass, we generate a constructor method that invokes the base class's constructor, which will presumably set all unconstrained fields to their initial or default values. Car heuristically chooses to invoke the base constructor method with the fewest input parameters. In cases in which the base class constructor requires input arguments, we set them to a default zero or null value if they are unconstrained by the target path. The generated constructor will set any constrained field members after the base constructor invocation to ensure the enforced context values are visible to the target path

We found that some classes are meant to be constructed in a certain way and may rely on a static initializer method to ensure all state is initialized properly. For instance, the `MotionEvent` class in the Android framework is backed by a native class that provides most of its functionality. Rather than using constructors, static initializer methods are provided to ensure that when the Java `MotionEvent` class is instantiated, the native backend object is created as well—if the native object is not constructed, we find that segmentation faults arise later when the Java object is used. We identified several commonly used classes that are irregularly constructed and specifically invoke their initializer methods when we need to create a constrained subclass for them.

4.4.2 Dynamic driving controller

The dynamic controller receives information from the static component, including the extracted target paths and their path driving code, and runs on a machine connected to a physical Android device. For each application, it sequentially injects each target path by sending a control message to a custom system service running within the instrumented OS on the test device. It monitors the output log to determine whether the target location for each path has been reached and to restart the application on a crash, which usually occurs when an incomplete context is inferred for a target path and our heuristics for dynamic recovery fail.

4.4.3 Custom Android OS

Our modifications to AOSP span ~3000 added or modified lines of code and include instrumentation of the Android framework and the Android runtime (ART).

4.4.3.1 Delivery of injected paths

We instrument the framework to add a custom system service to deliver injected events. A socket is used to communicate with the dynamic controller and upon receipt of a path driving message, the service finds the application process and makes an interprocedural (IPC) call to its main thread with the name of the path driver method. Instrumentation in the main thread will then load the method and invoke it via reflection.

To load our custom path driver methods and constrained subclasses, we instrument the class loader within ART. When a path driving class is requested in the tested application's process, the class loader will instead load it from Car's custom DEX file. We also load instrumented application classes in this way so that the code modifications occur only in memory and the application cannot detect any changes to the original APK or DEX file, which often trip code integrity checks.

The constrained subclasses that Car adds to the runtime may reference classes, methods, or fields that are declared package-private for the application and cannot normally be accessed by other code, such as the path driver methods. We bypass access checks within ART for classes and methods related to Car's path driving. We also bypass enforcement of `final` classes that are extended by constrained subclasses.

4.4.3.2 Dynamic dependency recovery

Car's dynamic context refinement and recovery of dependency errors involve instrumenting ART's code interpretation processes. Monitoring requires the modification of exception handling code to detect when an exception is thrown, either by the runtime or the application. To determine whether this occurs during Car's path driving, we search backward through the stack trace to find a path driver caller method.

For recovery, we instrument locations where null exceptions are generated by ART, which depend on the interpretation mode at the time of the error. When ART is in DEX "interpreter" mode, the routines for handling object access instructions contain explicit checks for a null receiver object. We add a hook into these checks to determine whether the null error is occurring during Car's path driving, which we determine by searching backward through the stack trace to find a path driver caller method. If so, the recovery process is triggered and the recovery object is returned to the original object access routine, which can now operate on the non-null receiver.

When ART is in "quick" mode, it is executing precompiled DEX code. Object accesses in this mode are handled in one of two ways: through a trampoline function to handle a virtual field or method reference (when the compiler cannot statically resolve the receiver type precisely) or a direct native memory access at the resolved field/method offset within the receiver object. For the trampoline case, explicit null receiver object checks are also present within the trampoline function and dependency recovery proceeds in the same manner as the interpreter mode. For direct native memory accesses, a null receiver object will result in a segmentation fault when ART tries to access a field or method within. ART has a custom signal handler that will triage the segmentation fault and eventually generate a `NullPointerException` object to be thrown. We instrument the signal handler at the point when it has determined that the signal is the result of a null pointer error. We then identify the DEX instruction corresponding to the native PC where the error occurred and use the declared receiver class type for the generation of the recovery object. We determine the machine register assigned to hold the receiver variable for the DEX instruction and overwrite it with the address of the generated recovery object.

Using the signal context from the original segfault signal, we restore the machine's context to the PC where the error originally occurred, with the original register values (with the exception of the register now holding the recovery object). The execution should return to the faulting instruction, with the memory access now performed on the recovery object's address rather than a null address.

The recovery process is architecture-dependent due to the manipulation of machine registers, including the PC. We currently implement dependency recovery for ARM and we force ART to run in ARM mode, if possible. This was not an issue with our datasets since applications containing native code usually include both ARM and ARM64 binaries for greater compatibility with different Android devices.

The generation of recovery objects requires a cache of previously allocated objects. We instrument the class initialization process to store the addresses of all allocated objects in reverse chronological order, with a limit of 50 objects for each class type. We also instrument the garbage collector so that deallocated object addresses can be removed. Because recent versions of AOSP use a heap compacting garbage collector, objects can move around in memory. To avoid tracking this movement in the cache, we disable heap compaction when we begin targeted analysis for an application.

4.5 Evaluation

We evaluate Car on a dataset of popular applications from the Google Play application marketplace to demonstrate its ability to generate approximated contexts and trigger a wide range of target sensitive behaviors in large, complex applications. We are interested in answering the following research questions:

Q1: Does the use of context approximation and refinement improve dependency resolution for targeted execution?

Q2: Is Car effective at reaching target code locations in Android applications?

Q3: Are the paths that Car executes sound, despite forgoing full dependency tracking?

Q4: Can Car uncover new security-sensitive behaviors?

4.5.1 Experimental setup

We ran Car's static context inference component on machines with Intel Xeon E5-2650 CPUs, with a JVM configuration of 200 GB of memory and 24 threads. The dynamic driving controller ran on an Intel i7-3770 machine with a tunneled USB-A connection to physical Android devices. [2] We tested on Pixel and Pixel 2 devices running our modified version of Android 10 without Google Play Services installed. We used the same device type for all of the testing for each application. When installing each application, we granted all of the permissions that were requested by the application.

4.5.2 Dataset

To demonstrate Car's generalizability on a variety of commonly used applications, we crawled the Google Play marketplace for the binaries and metadata of all free applications in June 2019. To form

[2] Tunneling over SSH and VPN was required due to COVID-19 access restrictions for the building where our analysis infrastructure was located.

our dataset, we sorted Google Play's application categories into 15 related category groups. For each group, we extracted the 25 most popular applications (determined by the total number of downloads for the crawled version), resulting in 375 applications in total. This set includes well-known applications such as Facebook, WhatsApp, Pokémon Go, MyFitnessPal, Spotify, BBC News, Microsoft Office, eBay, Instagram, Uber, Airbnb, and AccuWeather.

Several applications crashed when launched on our test devices due to incompatibility with the devices or unchecked dependencies on functionality within Google Play Services, which is not included with AOSP. Furthermore, some applications triggered errors in Soot within its code representation and generation; based on our analysis, we believe they were due to incomplete support for highly specific instruction sequences in the version of Soot used. We skipped these applications, resulting in an effective dataset of 310 applications.

4.5.3 Effectiveness of contexts for dependency resolution (*Q1*)

To evaluate Car on a variety of sensitive behaviors and paths, we configured it to target the sensitive source and sink methods from FlowDroid [10]. While we are not performing taint analysis, the methods include a variety of different sensitive actions on Android that are likely of interest when performing security analysis. In addition, we also target invocations to reflection APIs, code loading APIs, and native methods, which may be used for obfuscation.

We compare Car's ability to resolve dependencies for targeted execution against IntelliDroid. As Car uses the initial targeting from IntelliDroid, it also extracts the same target paths. We execute the target paths dynamically with Car and measure the effectiveness of using contexts to resolve dependencies. We were unable to execute the paths within IntelliDroid as we implemented its dynamic framework for Android 4.3, which is now incompatible with our devices and the applications in our dataset (Android 4.3 also predates ART, where we implemented Car's dynamic dependency recovery). Instead, we analyze the output from its dependency analysis (i.e. the event chain mechanism) and generously assume that a lack of dependency tracking errors reported for a target path indicates that the path would also have been triggered by IntelliDroid.

Of the targets triggered dynamically by Car (average of 125 per application), IntelliDroid reported incomplete dependency tracking or unsupported event injection for 72.1% of the paths, with most of the paths reporting overlapping errors. These errors would prevent the tools from executing the target paths at run-time, resulting in IntelliDroid reaching less than a third of the target locations reached by Car—a theoretical improvement of 3.6× in target coverage by Car. Dependency tracking failures, which were reported for 70.1% of the paths, would have resulted in unresolved dependencies, as IntelliDroid would not have been able to set up the expected program state required for the execution of the target paths. The unsupported event injections (39.4%) were due to the lack of support for injecting UI events (the tools would not have attempted to inject these paths at all). This limitation stems from its approach toward static targeting, in which we modeled the execution environment at the framework level. The injection of events into the framework requires each event type to be manually supported, which is difficult to scale for all Android event types—we found over 190 different types of event handlers for the executed paths, including 97 types of UI event methods handling the different ways in which UI elements can be manipulated. The code object abstraction used by Car provides greater flexibility and scales for the injection of different event types without manual effort.

Of the paths that could not have been executed by IntelliDroid, we analyzed the techniques used by

Car to resolve their dependencies and reach the target locations. The use of the statically generated constrained subclasses was necessary in 84.7% of these paths in order to set the expected program state. We can further break this down into 56.2% requiring constrained input objects and 38.2% requiring constrained field or method accesses to heap objects. The approximated contexts generated from these constrained subclasses enabled Car to resolve path dependencies without requiring framework injection or tracking dependencies across the application and framework.

Dynamic context refinement in unconstrained methods was necessary in 53.3% of the paths triggered by Car, which was a result of the trade-offs made in the static constraint analysis to enable scalable tracking and resolution of dependencies in the approximated contexts. In the current implementation of Car, this refinement consisted of recovery from runtime-generated null exceptions, with 92.7% of the recovered paths re-using objects from the application to refine the context. Other dependency-related exceptions that were only partially recovered (by suppressing the exceptional control flow) occurred in 18.1% of the targets, with `IllegalStateException`'s forming the majority of the cases (13.5%). Car's ability to reach the targets despite the approximations made in the static constraint analysis shows the effectiveness of hybrid dependency resolution techniques and the benefit of dynamically inferring part of a path's context rather than relying entirely on static dependency tracking.

4.5.4 Triggering target locations (*Q2*)

We evaluate Car's ability to trigger target code locations against several state-of-the-art code exploration tools. We performed this comparison using two datasets: the Google Play dataset and the EvaDroid test suite [19], which contains evasive behaviors that intentionally try to hide malicious payloads.

4.5.4.1 Evaluation on EvaDroid

Due to lack of availability and/or Android version incompatibilities, we are unable to run previous targeted or forced execution Android tools to compare against Car on the Google Play dataset. Instead, we ran Car on the EvaDroid test suite [19], which was previously used for the evaluation [19, 105] of several targeted and forced execution tools. We compare Car's results with these previousy published numbers for Ares [19] (a forced execution tool for Android), DroidBot [71] (a purely dynamic exploration tool), GroddDroid [3] (a hybrid GUI exploration and forced execution tool), and IntelliDroid [123].

EvaDroid contains 22 synthetic applications representative of evasive malware. In these evasive applications, actions or "payloads" are hidden from dynamic analysis through complex activation conditions, such as timing conditions or device fingerprinting. We ran Car on the EvaDroid dataset and in Table 4.2, we compare Car's payload coverage with the results from the previous studies. We also confirmed that Car triggered no false positive paths on the test applications. Three test cases (`constantCalls1/2` and `divById`) could not be run as-is due to their use of device identifiers, as new security settings in Android 10 limits access to privileged apps; we instrumented the offending APIs to return dummy values that mimic emulator values (even though we actually run on real devices) to verify that Car's contexts can defeat device fingerprinting.

Table 4.2 shows that Car can reach a greater number of malicious payloads than the purely dynamic tool, DroidBot; this agrees with the results from our large-scale evaluation below. It also achieves greater payload coverage than GroddDroid, likely because GroddDroid only forces branches that are encountered during its initial dynamic GUI exploration, which would have had limited coverage. Car

Tool		Payloads triggered
Car		73%
Ares	(evaluation from [19])	86%
DroidBot	(evaluation from [105])	17%
GroddDroid	(evaluation from [105])	37%
IntelliDroid	(evaluation from [105])	33%

Table 4.2: Comparison of payload coverage on the EvaDroid [19] test suite of evasive applications

also outperforms IntelliDroid, which is expected from our previous results in Section 4.5.3 showing that Car can resolve a significantly greater number of dependencies in target paths.

Conversely, the forced execution tool, Ares, reaches more payloads than Car. This also matches our expectations, as Ares can force paths to execute by directly manipulating branch outcomes and skipping code that deliberately stalls execution to evade detection by a dynamic analyzer (e.g. removing invocations to `sleep()`. Car relies on its generated contexts to resolve dependencies and supply consistent data to the target path. Imprecision in the statically extracted constraints, as well as missing support for certain dependency types (such as files or IPC mechanisms), can lead to incomplete resolution. In particular, Ares was able to reach the *postDelayed* and *sleep* payloads, which Car missed since it cannot handle time-based dependencies (Ares can merely nullify the calls to `sleep` to bypass these cases). Car also missed *constants1* due to missing edges in the static call-graph (this appears to be a bug in the version of Soot used).

In general, we find that Car's context-based dependency resolution requires greater support for different dependency types and paths, which forced execution is able to bypass more easily. This forcing can lead to the execution of unsound paths, although this was not an issue for the EvaDroid test suite as the applications are small and only contain a handful of paths. Nevertheless, as noted in Section 4.2, forced execution is better suited for achieving target coverage rather than security analysis due to its bypassing of data dependencies. The forcing of branches can generate behaviors not present in the original code and lead to many false positives in malware detection; we provide further discussion of these soundness issues in Section 4.6.2. We primarily use this evaluation to show that Car's context-based dependency resolution enables it to approach the upper bound of target coverage, of which a forced execution tool such as Ares should provide.

4.5.4.2 Large-scale evaluation

To evaluate Car's ability to dynamically reach sensitive targets in popular applications, we perform a large-scale comparison of target coverage against several state-of-the-art dynamic GUI and model-based exploration tools: Monkey [81], DroidBot [71], and Ape [55]. We ran the tools for an average of three hours for each application, using the default configuration for DroidBot and Ape, and a throttle of 100 ms for Monkey similar to previous work (and gives a slight edge in its coverage [92]). To account for the delays caused by Car's dynamic dependency recovery, we conservatively used a longer wait time of 10 seconds between injections for Car (this could have been significantly reduced, as we explain in Section 3.5.3). We also identified the targets that were trivially triggered without any user input during the first 30 seconds when the application launched. We remove these targets in our comparisons, as they

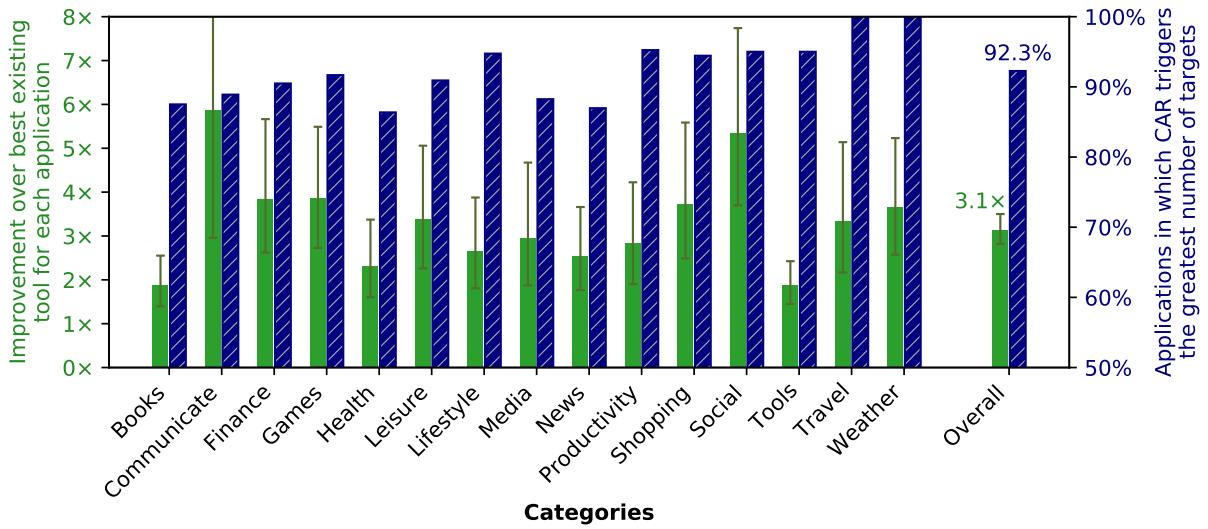


Figure 4.4: Comparison of non-trivial sensitive targets triggered by Car and by existing dynamic tools

The solid green bars (left axis) show the average improvement in the number of non-trivial targets triggered by Car against the best-performing tool (Monkey, DroidBot, or Ape) for each application. The striped blue bars (right axis) show the percentage of applications for which Car triggers the greatest number of targets of all the dynamic tools tested.

can be reached without the use of any tool.

In Figure 4.4, we show Car's improvement in triggering target behavior on our dataset. When comparing Car to each tool individually, on average, Car triggered 4.5× more non-trivial targets in an application than Monkey, 5.2× more than DroidBot, and 7.1× more than Ape. However, we also found that between Monkey, DroidBot, and Ape, some perform significantly better for certain applications than others. We compared Car against the best-performing tool for each application and found that we were able to reach an average of 3.1× more targets. Furthermore, Car triggered the greatest number of targets for 92.3% of the applications.

Car showed the greatest improvement in the "Communication" and "Social" categories ($> 5\times$ more non-trivial targets). These applications often require the user to log in before certain functionality can be accessed; this log-in requirement likely blocked the GUI exploration tools from making much progress (Car bypasses the block since it directly injects paths into the application process). The lowest improvement was seen in "Books and References" and "Tools" with ~1.9× more targets. These applications are fairly simple and they display or perform one specific function. The simple interface makes it easier for the other tools to reach more of the application, resulting in less potential for improvement.

In the applications where Car reached fewer targets than the other tools, it was outperformed by Monkey for 3.5% of the applications, DroidBot for 1.0%, and Ape for 3.2%. We found that this was primarily due to missing implementation/support in Car that disproportionately affected certain applications. We describe the causes for the lower coverage in these applications and for other false negatives in Section 4.5.8.

In addition to the blanket coverage of targets, we also considered which targets were triggered by Car and by the other tools. Figure 4.5 shows an average breakdown of targets triggered in an application by the various tools, including those trivially triggered during the application's launch. A large portion (44.2%) were reached only by Car, showing a significant increase in the coverage of new sensitive

behaviors.

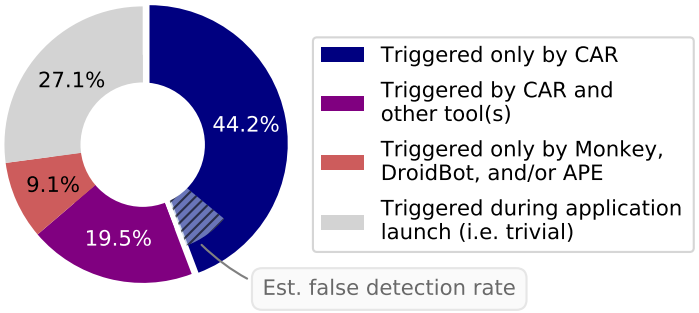


Figure 4.5: Average breakdown of the sensitive targets triggered by the different dynamic tools

4.5.5 False positives (*Q3*)

Of the 44.2% of newly triggered targets in Figure 4.5, we manually inspected a subset to determine whether they are truly reachable or whether they were the result of the execution of unsound paths (i.e. false positives). Our sample set was chosen as follows: (1) for each category, we randomly sampled five applications, with replacement; (2) for each application, we computed the list of targets that were triggered by Car but not by any other dynamic tool; (3) we randomly chose one of those targets and analyzed the decompiled code that would trigger it. In total, we manually inspected 75 targets and their associated paths.

Our primary objective was to determine whether a target newly triggered by Car would be executed during normal execution. From our prior experience, applications often include third-party libraries in their APKs but may not use all of the code within the library. Since Car invokes path entry-point methods directly, it is possible that some of the injected events would not have been registered or triggered during normal execution. For each target in our sample set, we checked whether the invocations, control flows, and data flows were valid across the path methods. We also checked whether the path's entry-point were live (we consider all exported components in the manifest file to be live). For entry-point event handlers that must be registered with the framework before they can be triggered, we recursively checked the path(s) to the registration call-sites. When possible within our testing environment, we tried to trigger the target paths by manually manipulating the application's user interface or injecting system events through the Android debugging (ADB) interface.

From our analysis, we determined that 18.7% of the sampled targets were reached through infeasible paths, where the samples were drawn from the subset of targets that were triggered only by Car. When we translate this rate across all of the targets triggered by Car and assume that overlapping targets that were also reached by Monkey, DroidBot, and/or Ape are true positives, we have a false detection rate of 9.0%. This was heavily skewed by the targets sampled from the "Games" category, where 4 out of 5 inspected targets were determined to be dead code. For the rest of the categories, the average number of false positives found was 0.7 out of 5.

We found the underlying reason to be static imprecision in the points-to analysis when extracting target paths. This manifested in two ways: (1) the entry-point analysis incorrectly identified event handlers as application entry-points due to the conservative aliasing of objects at registration call-sites; and (2) method invocation edges were incorrectly constructed in the call-graph due to conservative

aliasing of the invocations' receiver objects. All the false positives found were located in first- or third-party libraries packaged with the applications as determined by the package name of methods in the paths. The spike of false positives within the "Games" category is likely due to the large number of libraries used, including graphics rendering, analytics, and advertisement libraries. Libraries introduce a great deal of code that must be analyzed but may not necessarily be used by the main application, increasing the effects of conservatism due to imprecision.

4.5.6 Analysis of newly triggered targets

During our manual analysis, we also enumerated the different reasons why the other dynamic tools were unable to trigger the cases where the target was a true positive for Car (i.e. not a false positive from the previous section). For 30%, the target could not be reached by the other tools because the application was blocked by a login screen that the tools could not bypass. For 21%, the target could only have been reached for certain devices or versions or if a specific error, such as a network failure, had occurred. While these conditions could not have been achieved in our test environment, Car's inferred path contexts resolved them and mimicked the environment required.

For the remaining 49%, we found that they could have been reached by normal UI interactions or system event injections in the test environment. However, a few would have required extremely complex interactions between event paths. In one case, the target occurs after the user purchases a travel package through the application, is located at an airport, and has Uber installed on their device. The application will then conveniently show the available Uber vehicles in the area to take them to their hotel. The conditions required for this path include UI flow across multiple screens (purchase process), input that is difficult to generate (purchase information), specific system state (location), and dependence on other installed applications. Car triggered it by directly invoking the location event handler for the target path, thus bypassing the multiple purchase-related requirements, and using a constrained subclass to return a `true` condition when the application queried the framework for the other application.

4.5.7 Analysis of sensitive behaviors (*Q4*)

Of the new behaviors uncovered by Car, we found that they ranged over a variety of different security-sensitive actions that would have been missed by the other tools. In Table 4.3, we break down the security-sensitive actions taken by the applications in our dataset that only Car was able to find. We further show how Car's context-based techniques were essential to reach the new behaviors and measured the percentage that required constrained subclasses or dynamic dependency recovery. Since our original dataset was comprised of benign applications, we also compared Car and Monkey on a set of 91 malicious applications recently labeled as evasion or surveillance "creepware" [103] that were being used for interpersonal attacks, such as harassment or stalking.

We show that we are able to operate effectively on both benign and malicious applications and uncover new sensitive or malicious behaviors that other tools would have missed. This ranged from recurring location accesses that send the location data to the network, to accesses to local email accounts hidden under a misleading calculator UI. We also found cases in which code was dynamically loaded on paths that only Car was able to trigger. This code can be further analyzed and explored to deobfuscate the application. When comparing the two datasets, new sensitive behaviors in malware required 7% less context support, with 39% fewer calls/paths requiring dynamic recovery. This is due to the relative

Sensitive functionality	Google Play			Creepware		
	Apps	**Calls**	**Cxt.**	**Apps**	**Calls**	**Cxt.**
Location	84	237	89%	14	42	86%
Personal data	6	6	83%	4	4	100%
Media	3	3	100%	1	1	100%
Telephony	7	11	100%	4	4	50%
Network	59	99	74%	11	20	85%
Files	220	1199	88%	63	361	10%
Databases	76	187	90%	11	47	94%
Package manager	134	222	93%	12	21	90%
Reflection	169	447	84%	25	113	75%
Code loading	7	7	71%	10	10	60%
Native code	223	1090	88%	29	120	68%

Table 4.3: New sensitive behaviors found by Car that were missed by the other tools

simplicity of the malware and their code paths in comparison to the size and complexity of popular applications in Google Play.

We further analyzed Car's execution of the Creepware [103] dataset (particularly the targets that only Car was able to reach) and determined the malicious or stealthy behaviors that could be detected through Car's targeting. These behaviors were previouly only manually detected or characterized in [103]. We provide an analysis of these cases below to demonstrate Car's ability to uncover malicious or evasive activity in applications, by guiding or targeting execution toward sensitive actions. While a dynamic taint tracking tool would have been ideal for detecting the cases of private data leakage, we were unable to integrate Car with previous taint tracking tools because they were implemented for older versions of Android or their implementation was not fully available [40, 113, 136] We instead relied on manual analysis and instrumentation to determine the flow of data triggered by Car's targeting.

Periodic background location tracking: One malicious application was intended to surreptitiously track the location of an unsuspecting victim. It uses Android's alarm service was used to periodically invoke a location gathering method, which regularly accesses the device's location data and send it to the network. Car was able to trigger the periodic location gathering function and execute this behavior by directly invoking the alarm callback method and using contexts to resolve the dependencies and constraints imposed by the callback path. In contrast, Monkey was unable to access this functionality due to log in and set up requirements that were necessary to enable the tracking. The leakage of location data in the background is performed without user awareness and should be triggered for dynamic security analysis to evaluate whether an application violates the privacy of the device's user.

Location tracking on a location change: There are several methods of accessing location information, including a callback-based approach. One application registers a location callback that is invoked by the framework when the device's location has changed. The application then stores the

new location information into a cloud storage location and sends it to the network asynchronously. Car was able to execute the location leakage path while Monkey was unable to trigger the location change required for the callback to be activated. Similar to the previous case, this access of location information is also performed in the background while the device is in motion and it may leak private data without the user's awareness.

Leakage of private data to the network: A large number of the tested malicious applications (approximately half) are surveillance applications that registers the device for a tracking service. This tracking enables the device's location, phone number, identifiers, photos, videos and/or received messages are sent to a third-party (supposedly for emergency situations, though this functionality can also be used for spying). In some of the applications, Monkey was able to trigger the private data leakage. However, in at least two of the tracking applications, Car was able to trigger private data leakage of device identifiers and location data to the network that Monkey was unable to reach. This was primarily due to a requirement to register the device with the tracking service, which Car is able to bypass and which poses a challenge for dynamic-only tools.

Transmitting microphone recording over Bluetooth: A spying application streams input from the microphone to a Bluetooth headset, allowing someone to eavesdrop on the device's surroundings and on the user's conversations. The functionality is only accessibly if a Bluetooth headset is connected to the device. Other dynamic tools would have difficulty triggering the spying behavior unless the analysis environment specifically includes such a headset. Car used its contexts to resolve the system Bluetooth dependency, enabling it to trigger the access to the audio recording and the flow of data to the Bluetooth media player.

Hidden access to device accounts: One application provided a "cloning" functionality, in which the application's main activity shows a set of public accounts and a secondary hidden activity enables access to a set of private accounts. We manually confirmed that Car was able to access both interfaces and detect access to both sets of accounts. The hidden interface would have been particularly difficult for dynamic tools such as Monkey, as there were complex UI actions required to make the activity visible. While access to accounts is not necessarily malicious, this shows that Car is able to trigger intentionally hidden functionality, which is useful for security analysis of applications.

Misleading UI masking account access: We found several similar evasive applications that hide an account-related activity behind a calculator interface. The hidden accounts are accessible only when a specific password is entered into the calculator. Car was able to explore past the initial calculator interface and trigger the hidden access to the device accounts without having to decipher the correct password.

4.5.8 False negatives

While there is overlap in the targets reached by Car and the other tools, we did find that 9.1% of all triggered targets in Figure 4.5 could be reached by at least one of the other tools but were missed by Car. We consider these to be known false negatives of Car. We manually analyzed 15 of these targets across all the application categories and found the following underlying reasons:

1. Incomplete context handling across Android interprocedural (IPC) invocations.

2. Incomplete handling of constraints, especially for lists and arrays.

3. Reliance on the stack trace for dependency error monitoring, which misses errors in new threads as they have a new stack.

4. Constraints on the results of one-way computations, such as encryption or hashing, which could not be inversed to generate a context value.

5. Any-path extraction of target paths, which may extract infeasible paths to a target due to the conservative call-graph and miss a true path.

Items (1) – (3), which applied to over 75% of the sampled false negatives, are artifacts of the implementation and can be fixed with more engineering. They were the primary reason why Car produced lower coverage in a few applications. (4) is a fundamental limitation of symbolic analysis, although in some cases, it was mitigated when the required result of a one-way computation can be statically extracted and set by a modeled method. (5) can be mitigated by sampling more paths for each target, though the extraction of all paths to a target would ultimately be exponential.

4.5.9 Performance

We measure the performance of the static and dynamic components of Car separately on our Google Play dataset. We ran the static context inference analysis with a timeout of 240 minutes and retrieved partial results in cases in which full analysis had not completed. For most applications, the full time was used, with an average of 35 seconds to process each target code location, including the apportioned call-graph generation time, target path extraction, constraint analysis, and approximated context generation. For the dynamic component, we injected all of the paths extracted by the static analysis (maximum of 2700 for each application). We throttled Car execution with a conservative wait time of 10 seconds for a target to be reached before injecting the next path. On further analysis, this could have been significantly reduced as an average path took 1.4 seconds to reach the target location.

We examined the sensitive targets triggered over the duration of the analysis for each tool and plotted the cumulative number of triggered targets in Figure 4.6. Car steadily made progress as it injected each statically extracted path. For the others, while they were successful at finding targets near the beginning, they made significantly less progress over time. After an hour of analysis, Car already shows a large improvement in the number of triggered targets.

4.5.10 Effect of Google Play Services

Because a number of popular applications tried to access Google Play Services, we also ran our modified version of Android 10 with the "nano" configuration of Open Gapps [89] installed, which provides baseline Google Play Services functionality. We re-ran Monkey on our Google Play dataset with this new execution environment and found that it was able to reach more target code locations, as functionality that was unavailable through the application's user interface might now be accessible to the UI exploration tool. On average, Monkey reached 12.3% more targets for 34.5% of the applications compared to its performance without Google Play Services installed on our test devices. When comparing our original results from Car against Monkey running with Google Play Services installed, Car triggers 4.2× more targets than Monkey, compared to 4.5× previously. While Car's improvement is somewhat lessened,

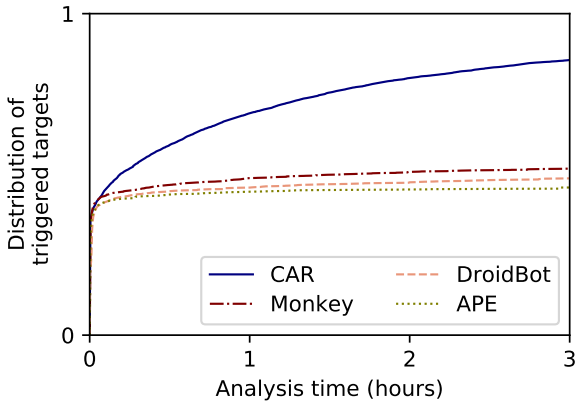


Figure 4.6: Cumulative number of targets triggered over time

this does show that Car is able to trigger functionality that would be missed by the other dynamic tools when a certain component, module, or library is missing from the analysis environment.

4.6 Discussion and Limitations

4.6.1 Completeness

As stated in Section 4.5.8, a fundamental limitation is that some constraints cannot feasibly be solved, such as constraints on hashed or encrypted values. In these cases, a path's context cannot be generated and the target code location would not be reached dynamically due to unresolved dependencies. This can cause the resulting dynamic analysis to miss malicious behaviors, resulting in false negatives. This stems from Car's reliance on constraint analysis to inverse path conditions and resolve dependencies, a limitation that is shared with symbolic execution techniques. It is possible for Car to mitigate this by using modeled methods to directly provide the dependent value in the context rather than attempting to inverse the one-way computation. This would require recognizing such computations (for instance, by adding special cases for common encryption libraries) and automatically modeling their constrained return values regardless of their invocation depth during the static constraint analysis of a path.

4.6.2 Soundness

Another fundamental limitation in Car are the sources of unsoundness in its targeting and context generation. The injection of path entry-point methods directly can lead to the execution of paths that might not occur normally if their entry-points were never registered with the framework. Because the dynamic dependency recovery refines the context heuristically, it may also construct infeasible contexts that would never occur during normal execution (by returning an incorrect recovery object). All of these can affect the soundness of the paths triggered; however, these issues are not unique to Car. Forced execution tools will also execute infeasible paths from unreachable entry-point methods and incorrect data accesses, especially as they enforce branch outcomes without ensuring that the accompanying data dependencies are resolved. Furthermore, without analysis of how data values are used, forced execution can result in infeasible control-flow paths due to the forcing of a normally impossible combination of branch outcomes.

Car's constraint analysis ensures that the extracted path is consistent with respect to the data it accesses; the context will either contain values that satisfy all of the constraints imposed by the path's conditional branches or, if the branches required are incompatible and the path is infeasible, no context would be generated. We believe Car achieves a good balance between its sources of unsoundness and its coverage of target locations, demonstrated by its ability to outperform existing dynamic tools in target coverage and to approach the coverage of the forced execution tool Ares [19], while maintaining a 9.0% false detection rate on popular applications. In addition, our false positive evaluation in Section 4.5.5 showed that the primary source of infeasible paths in Car were due to static imprecision (i.e. over-conservative entry-point detection and points-to analysis), which also affect forced execution.

4.6.3 Obfuscation

Because Car's static analysis uses Soot, it is limited to the Java/DEX parts of the application. Car can extract and trigger paths toward native method invocations, but it cannot handle constraints or dependencies imposed within native code. Applications can also try to obfuscate their code from static analysis, such as through reflection or dynamic code loading. This will affect the initial static targeting and constraint analysis in Car. In the following chapter (Chapter 5), we show how this can be solved by collecting run-time information from the targeted dynamic analysis and integrating it with the static targeting phase to iteratively improve precision and completeness.

4.6.4 Implementation limitations

Other limitations in Car stem from the analysis frameworks used to perform its targeting and dependency resolution. Car's static targeting relies on the Soot static analysis framework [116], and the precision and completeness of the target paths depend on that of the underlying control- and data-flow analyses in Soot. It also relies on the entry-point analysis from FlowDroid [10], which may miss some entry-point methods (resulting in an incomplete call-graph and missing target paths) or may incorrectly identify unreachable entry-points (which can lead to false detection of target behaviors). However, these issues would arise for any other static analysis framework due to the tradeoff between precision, completeness, and scalability in static analysis. Car's combination of static and dynamic dependency resolution makes it more robust to the limitations from this tradeoff and its hybrid context generation can be tuned (by adjusting the scope of the static constraint analysis) such that Car could be implemented on other static analysis frameworks that use a different tradeoff from Soot.

On the dynamic side, the monitoring and resolution of dependencies relies on our instrumentation of ART, which was limited to the ARM 32-bit architecture. This instrumentation also requires more effort if we need to port it to handle newer versions of Android and ART. While this limits Car's ability to handle future applications, the techniques we use to identify recovery locations and objects are not specific to ARM or ART, and can be used in any runtime system. We could potentially implement Car's dynamic component for the Android emulator—while applications can normally fingerprint the emulation environment and hide malicious behaviors from analysis, Car's contexts allows it to resolve dependencies, even those related to device fingerprinting, and reach those hidden target behaviors (Section 4.5.4.1). Instrumentation of the emulator would allow for greater support of different devices and architectures for dynamic dependency recovery, and should make it easier to support newer versions of Android.

While Car was intended for the analysis of Android applications, its techniques are not limited to the

Android environment. While targeted execution could be used for general programs, we believe it is well-suited for the analysis of mobile applications, where a lot of malicious behaviors revolve around abuse of sensitive functionality, which can be targeted. Car could potentially be implemented for the analysis of iOS applications, which are mainly implemented in Objective-C or Swift. They use a dynamic runtime environment that could be instrumented for the dynamic dependency recovery; however, Objective-C and Swift compiles to native code and it may not be possible to perform the same level of symbolic constraint analysis, as native code contains less information than Java or DEX bytecode. This may result in a greater reliance on the dynamic dependency recovery mechanism, as limitations in the static constraint analysis would in turn affect the effectiveness of the initial static context and reduce its ability to resolve dependencies in a target path.

4.7 Related work

Car is most closely related to other targeted dynamic analysis tools. Namely, our previous work in IntelliDroid [123] proposed the targeted execution of Android applications, though it relies on precise static dependency tracking and resolution. Similarly, guided symbolic execution tools such as AppIntent [134], WatSym [91], and [15] implicitly handle dependencies by modeling the program state symbolically. However, the resources required to precisely track and resolve the program state either statically or dynamically can ultimately reduce the coverage of targets due to overhead. Furthermore, guided symbolic execution cannot easily integrate with dynamic analysis tools that require concrete execution of the application. Harvester [98] and DirectDroid [119] propose forced execution, which bypasses a path's constraints on its dependencies. Forced branching can lead to unsound or infeasible paths, resulting in false positives. DirectDroid [119] also performs null recovery, similar to Car.

The constraint analysis used in Car's context inference is closely related to symbolic execution (such as EXE [28] and KLEE [27]) and concolic execution (such as DART [51], CUTE [106], and ACTEve [5]). Rather than tracking and resolving constraints dynamically, which adds significant overhead, Car performs static constraint analysis of specific paths to guide a faster targeted dynamic analysis. Car is also similar to static SAT-based verification, like Saturn [128], though Car's constraint extraction is targeted and not fully complete.

UC-KLEE [96], which performs under-constrained symbolic execution by invoking arbitrary methods directly, is in a way bypassing the dependencies of the path that would normally invoke the method. The preconditions imposed by a method on its inputs [64, 96] are similar to contexts, which are like preconditions on the program state. We can achieve more sound execution by executing paths rather than individual methods. Car is also similar to chopped symbolic execution [115], in which irrelevant function calls are skipped during a symbolically executed path until it encounters a dependent load relying on a function side effect, at which time it enters a recovery state to obtain the value. Chopped functions are similar to Car's modeled methods and any unconstrained methods that may have been aborted, though any dependencies a target path may have on state they modify will be resolved by the context inferred from the path's constraints.

Car is a hybrid tool and is related to other Android tools that perform static-guided dynamic analysis. Brahmastra [20], AppDoctor [59], SmvHunter [109] and SmartDroid [143] guide execution by driving component and UI transitions, which are more coarse-grained than code paths, and they do not handle dependencies between event paths. ContentScope [61] generates inputs for paths in content

providers and also does not handle inter-path dependencies. AppAudit [127] and AppIntent [134] use static analysis to guide approximate or symbolic execution, respectively, for the verification of static information flows. In contrast, Car guides concrete execution and can be integrated with general dynamic analyses.

Car's static analysis is also similar to other static Android tools, such as FlowDroid [10], Epicc [86], Apposcopy [43], Amandroid [122], CHEX [74], and [133]. Car's targeted dynamic analysis is generally more precise. FlowDroid's [10] entry-point extraction for Android applications is used by IntelliDroid's [123] static targeting analysis and is also used by Car.

Car's aims to drive the execution of Android applications, similar to testing frameworks such as DynoDroid [75], EvoDroid [76] and Sapienz [77], and model-based explorers such as Ape [55], Stoat [112], DroidBot [71], TrimDroid [79], and A^{3}E [14]. Fuzzing tools, such as AFL [137], are also commonly used outside of Android. They aim for full coverage, while Car focuses on coverage of target locations. The techniques we use to focus execution to target paths enables greater coverage of locations of interest to an analyzer. Semi-targeted exploration, such as FuzzDroid [99] and Xdroid [97], inject system/framework values, which helps resolve dependencies on framework state. Similarly, generating useful test inputs, such as TextExerciser [58], is also a form of guided fuzzing. These tools do not explicitly handle dependencies between application paths and must rely on randomness to inject paths in an order that satisfies their constraints on program state.

Car for Android targeted execution is meant to aid in the dynamic analysis of target behaviors, such as in CopperDroid [114], TaintDroid [40], TaintART [113], DroidScope [132], DroidBox [66], VetDroid [140], and RiskRanker [53]. Car can potentially be integrated with these dynamic tools to drive execution to behaviors of interest, similar to the integration of IntelliDroid's targeted analysis with TaintDroid; however, we were unable to do so due to Android version incompatibilities or the tool not being fully publicly available.

4.8 Summary

In this chapter, we present Car, a hybrid approach to dependency resolution through context approximation and refinement. We use a combination of static constraint analysis and dynamic error recovery to approximate and refine a context that resolves dependencies on program state and enables the effective execution of isolated target paths. We applied Car to the targeted execution of Android applications and extended our previous work in IntelliDroid to enable more effective targeted security analysis of all types of Android applications. We found that through Car, we were able to reach 3.1× more non-trivial sensitive targets for an application with a false detection rate of 9.0%, demonstrating a significant increase in target code coverage while maintaining reasonable soundness in the executed paths. The sensitive behaviors uncovered by Car, which would have otherwise been missed, are essential for security analysis and vetting of applications.

In the following chapter, we address another limitation of targeted analysis and security analysis in general: obfuscation. The use of static analysis to guide execution to locations of interest can defeat certain forms of dynamic obfuscation in which activity is hidden under specific anti-analysis trigger conditions. However, the reliance on static techniques to extract the target paths makes targeted analysis vulnerable to obfuscation that affects static analysis, such as reflection or code packing. In Chapter 5, we show how we can further extend targeted analysis to form a complete hybrid program analysis loop, in

which information from both the static and dynamic phases feed information to each other to overcome their weaknesses. We use targeted analysis to focus execution to locations where obfuscation is likely to occur and dynamically analyze these locations to deobfuscate them for more complete targeting. The results from either the static or dynamic phases can be used by other security analysis tools to improve their ability to analyze and detect interesting or suspicious application behaviors.

Chapter 5

TIRO: Iterative targeted analysis for deobfuscation

In Chapter 3, we introduced the idea of targeted execution and showed that when used in conjunction with a dynamic security analysis tool, it can improve the effectiveness of the analysis by driving execution of the application to the specific locations of interest. Through the combination of static targeting and constraint analysis with an over-approximation of the the behaviors of interest, we were able to generate inputs that trigger locations deep in the application that the analyzer would have otherwise missed. In Chapter 4, we further refined the execution of these target locations in our development of CAR by simulating a path context that enables the restricted execution of the desired target paths.

A limitation of the program analysis techniques used in the design of IntelliDroid and CAR is code obfuscation, which is a limitation of many static analysis tools. Consider a scenario where a portion of an application's code is obfuscated such that it cannot be analyzed using static techniques. This is common in cases of malware, as the malware developer may want to hide the malicious actions taken by their application from anti-malware analysis. If we were to use targeted execution to analyze such an application, when the static analysis computes its over-approximation of interesting behaviors, it may miss the hidden malicious actions. As such, it would never compute the constraints, context, or input values that will trigger the behavior dynamically, causing the attached analyzer to miss the behavior as well. Alternatively, even if the statically generated over-approximation is complete and includes the malicious actions, obfuscation can also make it difficult to precisely determine the variables used in conditional statements (e.g. through the use of reflected acceses to heap data) or to extract the constraints they impose on the program state, rendering the constraint analysis and context approximation incomplete. The incomplete program state would result in false negatives, as the malicious behavior would not be triggered dynamically.

The underlying problem is the lack of precise information that is only available from the application as it is executing. When the static phase of targeted analysis lacks the information required for precise path targeting, this affects the completeness of the dynamic phase and lessens the effectiveness of targeted execution. The presence of obfuscation does not indicate malicious intent in and of itself, as many legitimate applications employ code obfuscation to protect intellectual property and prevent reverse engineering. However, because of its prevalence among malware, it is crucial that malware analysis tools have the ability to deobfuscate Android applications in order to determine if an application is indeed

malicious.

In this chapter, we investigate methods of mitigating the effects of obfuscation on targeted execution. We previously described targeted execution as a process of over-approximating and refining analysis results until the behaviors of interest in an application can be analyzed, using static information to guide dynamic driving. Now, we look at how information can be passed in the opposite direction—using dynamic information to refine static analysis such that it is robust to obfuscation.

Building on the ideas developed in IntelliDroid, we propose Tiro [124], a hybrid iterative deobfuscation framework. Tiro is an acronym for the automated approach taken to defeat obfuscation—**T**arget-**I**nstrument-**R**un-**O**bserve. Tiro first analyzes the application code to target locations where obfuscation may occur, and applies instrumentation either in the application or runtime to monitor for obfuscation and collect run-time information. Tiro then runs the application with specially generated targeted inputs that will trigger the instrumentation. Finally, Tiro observes the results of running the instrumented application to determine whether obfuscation occurred and if so, produce the deobfuscated code. Tiro performs these steps iteratively until it can no longer detect any new obfuscation. This iterative mechanism enables it to work on a variety of obfuscated applications and techniques.

While Tiro was developed as a deobfuscation tool, it is rooted in an integration with targeted execution and with our previous work in IntelliDroid. In this synergistic combination, IntelliDroid improves Tiro's efficiency by targeting its dynamic analysis toward obfuscation code and Tiro improves its completeness by incorporating deobfuscated information back into the targeting analysis. By using an iterative design that feeds dynamic information back into static analysis for deobfuscation, Tiro can incrementally increase the completeness of this targeting, which further improves its deobfuscation capabilities. Successive iterations allow each to refine the results of the other. The resulting static and dynamic analysis is more complete and can be used to aid other security analysis tools, both static or dynamic.

We used Tiro to explore obfuscation in Android applications and particularly, in Android malware. We found that most employ techniques such as Java reflection, value encryption, dynamically decrypting and loading code, and calling native methods, which have been identified and discussed in the literature [42, 98, 111]. These techniques have a common property in that they exploit facilities provided by the Java programming language, which is the main development language for Android applications, and thus we call these *language-based obfuscation* techniques. In contrast, malware authors may eschew Java and execute entirely in native code, obfuscating with techniques seen in x86 malware [17, 38, 63, 78, 104]. We call this technique *full-native code obfuscation*.

When analyzing malicious obfuscation with Tiro, we identified a third option—obfuscation techniques that subvert ART, the primary Android runtime, and we call these *runtime-based obfuscation* techniques. These techniques subtly alter the way method invocations are resolved and code is executed. Runtime-based obfuscation has advantages over both language-based and full-native code obfuscation. While language-based obfuscation techniques have to occur immediately before the obfuscated code is called, runtime-based obfuscation techniques can occur in one place and alter code execution in a seemingly unrelated part of the application. This significantly raises the difficulty of deobfuscating code, as code execution no longer follows expected conventions and analysis can no longer be performed piecemeal on an application, but must examine the entire application as a whole. Compared to full-native code obfuscation, runtime-based obfuscation allows a malware developer to still use the convenient Java-based API libraries provided by the Android framework. Malware that use native code obfuscation will either

have to use language- or runtime-based obfuscation to hide its Android API use, or risk compatibility loss if it tries to access APIs directly. Our study of obfuscated malware suggests that authors almost universally employ language- and runtime-based methods to hide their use of Android APIs in Java.

In this chapter, we make four main contributions:

1. We identify and describe a family of runtime-based obfuscation techniques in ART, including DEX file hooking, class modification, ArtMethod hooking, method entry-point hooking and instruction hooking/overwriting.

2. We present the design and implementation of Tiro, a framework for Android-based deobfuscation that uses hybrid targeted analysis techniques and that can handle both language-based and runtime-based obfuscation techniques.

3. We evaluate Tiro on a corpus of 34 modern malware samples provided by the Android Malware team at Google. We also run Tiro on 2000 obfuscated malware samples downloaded from VirusTotal to measure the prevalence of various runtime-based obfuscation techniques in the wild and find that 80% use a form of runtime-based obfuscation.

4. We integrate Tiro with the context-based dependency resolution techniques proposed in Car and analyze a variety of popular benign applications from the Google Play marketplace. We compare and contrast the use of language-based and runtime-based obfuscation between malicious and benign applications.

5.1 Background on Android obfuscation

In Chapter 2, we described some common forms of obfuscation in Android applications and how they affect program analysis of applications. We expand on these obfuscation techniques below and categorize them into two categories: language-based obfuscation (reflection, encryption, dynamic loading, and native methods) and full-native code obfuscation. In the following section (Section 5.2), we describe a new form of obfuscation that arises from the tampering of state stored in the runtime environment, which we call runtime-based obfuscation.

Reflection: Java provides the ability to dynamically instantiate and invoke methods using reflection. Because the target of reflected method invocations is only known at run-time, this frustrates static analysis and can make the targets of these calls unresolvable (e.g. by using an encrypted string), thus hiding call edges and data accesses.

Value encryption: Key values and strings in an application can be encrypted so they are not visible to static analysis. When executed, code in the application decrypts the values, allowing the application to use the plain text at run-time. Value encryption is often combined with reflection to hide the names of classes or methods targeted by reflected calls.

Dynamic loading: Code located outside the main application package (APK) can be executed through dynamic code loading. This is often used in packed applications, where the hidden code is stored as an encrypted binary file within the APK package and decrypted when the application is launched. The decrypted code is stored in a temporary file and loaded into the runtime through the use of

the dynamic loading APIs in the `dalvik.system.DexClassLoader` and `dalvik.system.DexFile` classes. Normally, the temporary files holding the decrypted bytecode are deleted after the loading process to further hide or obfuscate it from analysis. In some cases, the invocation to the dynamic loading API may be obfuscated by performing the invocation reflectively or in native code, using multiple layers of obfuscation to increase the difficulty of analysis.

Native methods: Java applications may use the Java Native Interface (JNI) to invoke native methods in the application. When used for obfuscation, malicious behavior and method invocations can be performed in native code. Unlike Java or DEX bytecode, native code contains no symbol information—variables are mapped to registers and many symbols are just addresses. Thus, static analysis of native code yields significantly less useful results and the inclusion of native code in an application can hide malicious activity or sensitive API invocations from an analyzer.

Full-native code obfuscation: Because Android applications can execute code natively, it would also be possible to implement an entire Android application in native code and utilize native code obfuscation techniques. Native code obfuscation has a long history on x86 desktop systems, and can be extremely resistant to analysis [17]. The primary drawback to this approach is that access to Android APIs, which can reveal the user's location and give access to various databases containing the user's contacts, calendar and browsing history, can only be reliably accessed via API stubs in the Java framework library provided by the OS. On one hand, calling APIs from Java code without language- or runtime-based obfuscation would expose the APIs calls to standard Android application analysis [10, 44]. On the other hand, calling these APIs from native code requires the application to correctly guess the Binder message format that the services on the Android system are using. Because the ecosystem of Android is very fragmented [93], this poses a challenge for malware that wishes to avoid executing Java code. As a result, applications that use native code obfuscation still need obfuscation for Java code if they want to be able to make Android API calls reliably.

5.2 Runtime-based obfuscation

Before we describe runtime-based obfuscation, we first describe how code is loaded and executed in the ART runtime. Figure 5.1 illustrates three major steps in loading and invoking code. First, A shows how DEX bytecode must be identified and loaded from disk into the runtime. Second, B is triggered when a class is instantiated by the application and shows how the corresponding bytecode within the DEX file is found and incorporated into runtime state. Finally, C shows how virtual methods are dynamically resolved via a virtual method table (vtable) and execution is directed to the target method code. We describe these steps in more detail below.

5.2.1 DEX file and class loading

In Stage A , DEX files are loaded from disk into memory, a process that involves instantiating Java and native objects to represent the loaded DEX file. The Java `java.lang.DexFile` object is returned to the application if it uses the `DexFile.loadDex()` API; in normal cases, this object is passed to a class loader so that ART can later load classes from the new DEX bytecode.

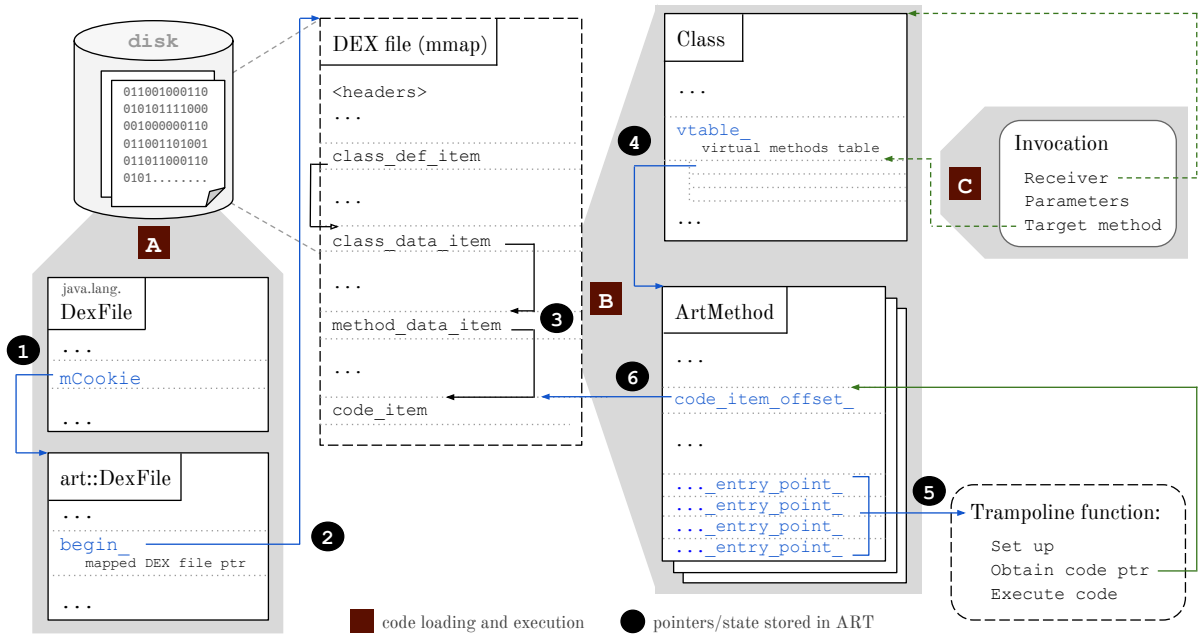


Figure 5.1: ART state for code loading and execution

The class loading process, Stage B , is triggered when a class is first requested (e.g. when it is first instantiated). The class linker within ART searches the loaded DEX files (in the order of loading) until it finds a class definition entry (`class_def_item`) matching the requested class name. The associated class data is parsed from the DEX file, now loaded in memory, and a `Class` object is used to represent this class in ART. In addition, data for class members are also parsed, and `ArtField` or `ArtMethod` objects created to represent them. To handle polymorphism, a vtable is stored for each class and used to resolve virtual method invocations efficiently. The table is initially populated by pointers to `ArtMethod` instances from the superclass (i.e. inherited methods). For overridden methods, their entries in the table are replaced with pointers to the `ArtMethod` instances for the current loaded class.

5.2.2 Code execution

When a non-static virtual invocation is made, marked by Stage C , the target method must be resolved. The resolution begins by determining the receiver object's type, which references a `Class` object. The method specified in the invocation is used to index into the vtable of this class, thereby obtaining the target `ArtMethod` object to invoke (see (4) in Figure 5.1). The actual invocation procedure depends on the method type (e.g. Java or native) and the current runtime environment (e.g. interpreter or compiled mode). A set of entry-points are stored with the `ArtMethod` to handle each case (see (5)); each is essentially a function pointer/trampoline that performs any necessary set-up, obtains and executes the method's DEX or OAT code, and performs clean-up. While Figure 5.1 shows only how the DEX code pointer is retrieved for a method (see (6)), OAT code pointers for compiled code are obtained in an analogous way.

5.2.3 Obfuscation techniques

Runtime-based obfuscation redirects method invocations by subverting runtime state at a number of points during the code loading and execution process outlined above. Because runtime-based obfuscation works by modifying the state of the runtime, it must acquire the addresses of the runtime objects it needs to modify, which is normally done using reflection, and modify them using native code invoked via JNI (since Java memory management would prevent code in Java from modifying ART runtime objects). In total, our analysis with Tiro has identified six different techniques used by malware to obfuscate the targets of method invocations. In Figure 5.1, ① – ③ indicates runtime state that can be modified to hijack the code loading process such that the state is initialized with unexpected data (with respect to the input provided to the runtime from the application). ④ – ⑥ indicates runtime state that can be subverted to alter the code that a method invocation resolves to. We describe these techniques in more detail below:

① ② **DEX file hooking.** When loading a DEX file, the `dalvik.system.DexFile` class is used in Java code to identify the loaded file; however, the bulk of the actual loading is performed by native code in the runtime, using a complementary native `art::DexFile` class. To reconcile the Java class with its native counterpart, the `DexFile::mCookie` Java field stores pointers to the associated native `art::DexFile` instances that represent this DEX file. When classes are loaded later, this Java field is used to access the corresponding native `art::DexFile` instance, which holds a pointer to the memory address where the DEX file has been loaded/mapped. Obfuscation techniques can use reflection to access the private `mCookie` field and redirect it to another `art::DexFile` object, switching an apparently benign DEX file with one that contains malicious code. In most cases, the malicious DEX file is loaded using non-API methods and classes within native code, or is dynamically generated in memory, further hiding its existence.

Similarly, instead of modifying the `mCookie` field, the obfuscation code can also modify the `begin_` field within the `art::DexFile` native class and redirect it to another DEX file. However, this approach can be more brittle since the obfuscation code must make assumptions about the location of the `begin_` field within the object.

③ **Class data overwriting.** Obfuscation code can also directly modify the contents of the memory-mapped DEX file to alter the code to be executed. DEX files follow a predetermined layout that separates class declarations, class data, field data, and method data [35]. Both the class data pointer (`class_data_item`), which determines where information for a class is stored, and method data pointer (`method_data_item`), which determines where information is stored for a method, are prime targets for such modification. Modifying the class data pointer allows the obfuscation code to replace the class definition with a different class while modifying the method definition allows the obfuscation code to change the location of the code implementing a method. This can be done en masse or in a piecemeal fashion, where each class or method is modified immediately before it is first used. We note that there are no bounds checks on the pointers, so while class and method pointers normally point to definitions and code within the DEX file, obfuscation code is free to change them to point to objects (including dynamically created ones) anywhere in the application's address space.

Class declarations (`class_def_item`) are not normally modified by obfuscation code since this top level object is often read and cached into an in-memory data structure for fast lookup. If the obfuscation

code misses the small window where the DEX file is loaded but this data structure has not yet been populated, any modifications to the class declarations will not take effect in the runtime.

④ **ArtMethod hooking.** After the receiving class of an invocation is determined, the target method is found by indexing into the class's vtable. Obfuscation code can obtain a handle to a `Class` object using reflection and determine the offset at which the vtable is stored. By modifying entries in this table, the target `ArtMethod` object for an invocation can be hooked so that a different method is retrieved and executed. The target method that is actually executed must be an `ArtMethod` object, which might have been dynamically generated by the obfuscation code or loaded previously from a DEX file. In the latter case, the use of virtual method hooking is to hide the invocation and have malicious code appear to be dead. The feasibility of this type of modification for obfuscation was established in [34].

⑤ **Method entry-point hooking.** Once the target `ArtMethod` object has been determined for an invocation, the method is executed by invoking one of its entry-points, which are mere function pointers. Similar to `Class` objects, reflection via the JNI can be used to obtain the Java `Method` object and through this, the obfuscation code can determine the location of the corresponding `ArtMethod` object, which is a wrapper/abstraction around the method. By modifying and hooking the values of these entry-points, it can change the code that is executed when the method is invoked.

Although the new entry-point code can be arbitrary native code, there exists a number of method hooking libraries [67, 73, 138] that allow an application developer to specify pairs of hooked and target methods in Java. They use method entry-point hooking so that a generic look-up method is executed when the hooked methods are invoked. This look-up method determines the registered target method for a hooked method invocation and executes it.

⑥ **Instruction hooking and overwriting.** The final stage in the method invocation process is to retrieve the DEX or OAT code pointers for a method and execute the instructions; this is performed by the method's entry-points. These code pointers are stored and retrieved from the `ArtMethod` object. Instruction hooking can be achieved by modifying this pointer such that a different set of instructions is referenced and executed when the method is invoked. Alternatively, instruction overwriting can be achieved by accessing the memory referenced by this pointer and performing in-place modification of the code—this normally requires the original instruction array to be padded with NOPs (or other irrelevant instructions) to ensure sufficient room for the newly modified code. While the invocation target does not change, the obfuscation code can essentially execute a completely different method than what was first loaded into the runtime. The modification of a method's instructions can occur before or after class loading, since the runtime links directly to the instruction array in `ArtMethod` objects. It is even possible to overwrite the instructions multiple times such that a different set of instructions is executed every time the method is invoked.

5.3 Tiro: A hybrid iterative deobfuscator

To address language-based and runtime-based obfuscation techniques, we describe Tiro, a deobfuscator that handles both types of obfuscation. At a high level, Tiro combines static and dynamic techniques in an iterative fashion to detect and handle modern obfuscation techniques in Android applications. The input to Tiro is an APK file that might be distributed or submitted to an application marketplace. The output is a set of deobfuscated information (such as statically unresolvable run-time values, dynamically

loaded code, etc.) that can be passed into existing security analysis tools to increase their coverage, or used by a human analyst to better understand the behaviors of an Android application.

The main design of Tiro is an iterative loop that incrementally deobfuscates applications in four steps, described below and illustrated in Figure 5.2.

Target: We use static analysis to target locations where obfuscation is likely to occur. For language-based obfuscation, these are invocations to the methods used for the obfuscation (e.g. reflection APIs with non-constant target strings). For runtime-based obfuscation, we target native code invocations as these are necessary to modify the state of the ART runtime.

Instrument: We statically instrument the application and the ART runtime to monitor for language-based and runtime-based obfuscation, respectively. This instrumentation reports the dynamic information necessary for deobfuscation.

Run: We execute the obfuscated code dynamically and trigger the application to deobfuscate/unpack and execute the code.

Observe: We observe and collect the deobfuscated information reported by the instrumentation during dynamic analysis. If Tiro discovers that the deobfuscation reveals more obfuscated code, it iterates through the above steps on the new code until it has executed all targeted locations that could contain obfuscation.

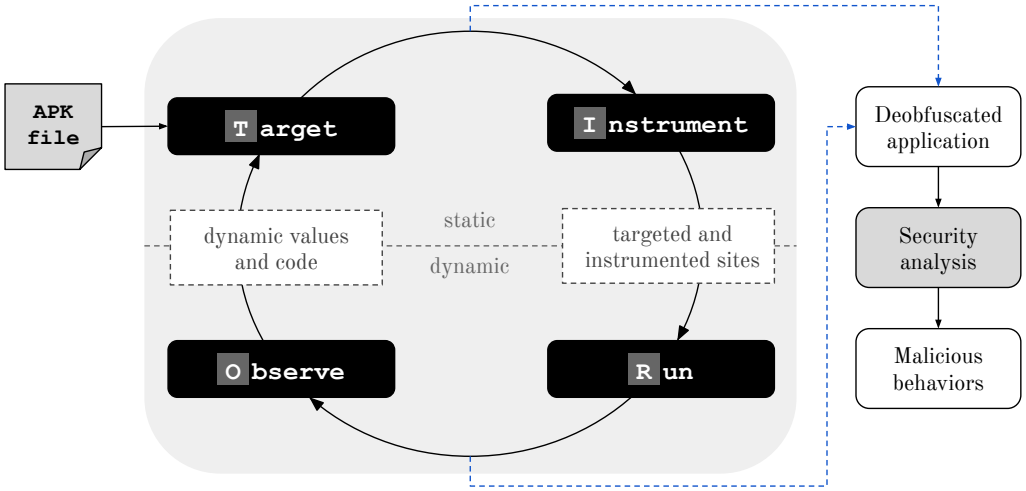


Figure 5.2: Outline of Tiro's design

Tiro's iterative process allows for deobfuscation of multiple layers or forms of obfuscation used by an application, since the deobfuscation of one form may reveal further obfuscation. This is motivated by our findings that obfuscated code often combines several obfuscation techniques and that deobfuscated code often itself contains code that has been obfuscated with a different technique. For instance, an application that dynamically modifies DEX bytecode in memory often uses reflection to obtain classes and invoke methods in the obfuscated code. Without supporting both forms of obfuscation, either the deobfuscated reflection target is useless without the bytecode for the target method, or the extracted obfuscated code appears dead since the only invocation into it is reflective.

5.3.1 Targeting obfuscation

A fundamental part of Tiro's framework is the ability to both detect potential obfuscation (i.e. targeting) and to perform deobfuscation (i.e. observation). Without targeting, Tiro would need to instrument and observe all program paths, which could be infinite in number. Targeting enables Tiro to only instrument and observe the program paths that are involved in deobfuscating or unpacking obfuscated code. For this reason, we build the static analysis portion of Tiro on top of our earlier work in IntelliDroid [123] by specifying locations of obfuscation as targets. While recent Android obfuscators generally automatically unpack application code at startup (and thus require no special inputs), an added benefit of targeting is that we can use IntelliDroid to generate inputs to trigger paths in future obfuscated code that may only unpack sections of code under specific circumstances [108].

For language-based obfuscation, obfuscation locations are visible in static analysis and the targets provided to IntelliDroid are invocations to reflection APIs, dynamic loading APIs, and native methods. For runtime-based obfuscation, while the obfuscated code is executed in the runtime (i.e. in Java/DEX bytecode), the actual obfuscation is done in native code as described in Section 5.2.3. IntelliDroid is currently unable to target locations inside native code. As a result, we instead target all Java entry-points into application-provided native code, such as invocations to native methods and to native code loading APIs (e.g. `System.load()`, which calls the `JNI_OnLoad` function in the loaded native library). While this is an over-approximation, targeting native code will ensure that any runtime-based obfuscation can be detected in the instrumentation phase.

5.3.2 Instrumenting obfuscation locations

Once all of the target obfuscation locations have been identified, Tiro instruments the application and the ART runtime such that any detected obfuscation is reported and deobfuscated values/code are extracted. For language-based obfuscation, Tiro instruments application code since that is where the actual obfuscation occurs. The instrumented code is inserted immediately before the target locations and the instrumentation reports the values of unresolved variables to `logcat`, Android's logging facility. A separate process monitors the log and keeps a record of the dynamic information reported. For example, to deobfuscate a statically unresolvable reflection invocation, the parameters to the invocation are logged (as well as the exact location where invocation occurs, to disambiguate between multiple uses of reflection). To deobfuscate dynamic loading, part of the instrumentation will store the loaded code in a Tiro-specific device location and report this location in the log. Native code transitions are also deobfuscated by instrumenting calls from Java into native code and monitoring for Java methods that are invoked from native code (essentially, the transitions across the JNI interface in both directions). This allows Tiro to create control-flow connections of the type: `Java caller` → `[native code]` → `Java callee`, which helps shed light into what actions are being taken in the native code of an application, even though Tiro does not perform native code analysis.

For runtime-based obfuscation, Tiro instruments the ART runtime. Since the result of this modification is the execution of unexpected code on a method invocation, one approach might be to record the code that was loaded into the runtime for a given method and check whether this code has been modified at the time of invocation. However, this poses a catch-22 situation: to detect the obfuscation, Tiro would have to target the obfuscated method but with runtime-based obfuscation, the obfuscation code could modify any class or method in the program. It would be impractical to target every method

in the program. Instead, we use the fact that runtime-based obfuscation must rely on native code to do the actual state modification. As a result, to detect runtime-based obfuscation, Tiro instruments transitions between native to Java and Java to native code to detect whether runtime state has been modified while the application was executing native code.

The runtime state monitored is specific to the objects used to load and execute code, as described in Section 5.2.3. For example, to detect DEX file hooking, Tiro finds and monitors the `DexFile::mCookie` and `art::DexFile::begin_` fields of all instantiated objects for changes before and after native code execution. If modifications are detected, Tiro reports the call path which triggered the modification, the element(s) that were modified and affected by the modification, and if possible, the code that is actually executed as a result of the runtime-based obfuscation. In some cases, there are legitimate reasons why runtime state may change between initial code loading and code execution (e.g. lazy linking or JIT compilation). We detect these and eliminate these cases from Tiro's detection of runtime-based obfuscation.

Checking all runtime state for modifications can be expensive as there can be many classes and methods to check. To reduce this cost we: (1) only monitor runtime state used in the code loading and execution process, and that are retrievable via the dynamic loading or reflection APIs (i.e. state stored within `DexFile`, `Class`, and `Method` objects); (2) only monitor the objects for methods and classes used by the application, as determined by reachability analysis during Tiro's static phase. This process relies on Tiro's iterative design, since the reachability analysis and subsequent monitoring becomes more complete as the application becomes progressively deobfuscated in later iterations.

5.3.3 Running obfuscated code

Tiro substitutes the original application with its instrumented code and uses IntelliDroid's targeting capabilities to compute and inject the appropriate inputs to run the instrumented obfuscation locations. However, doing this on obfuscated code raises an additional challenge—many instances of obfuscated applications also contain integrity checks that check for tampering of application code and refuse to run if instrumentation is detected. We found that the most robust method for circumventing these checks is to return (i.e. spoof) the original code when classes are accessed by the application and return instrumented code when accessed by the runtime for execution. To avoid conflicts with any runtime state modification that may be performed by obfuscation code, Tiro checks if any state modifications target instrumented code and if so, Tiro aborts execution of the instrumented code and allows the modifications to be performed on the original application code instead. In the next iteration, after extracting the modified code, the previously obfuscated code will be instrumented and executed.

5.3.4 Observing deobfuscated results

Tiro observes how the application either resolves and runs sections of code (to defeat language-based obfuscation), or how the application's obfuscation code modifies the runtime state (for runtime-based obfuscation). The results of this observation and the information provided by Tiro's instrumentation are reported to the user for deobfuscation of the application.

The iterative approach taken by Tiro also relies on these observed results to incrementally deobfuscate layers of obfuscated code. For obfuscation that hides or confuses invocation targets (e.g. reflection, native method invocations, method hooking), Tiro's instrumentation reports the caller method, the

invocation site, and the actual method that is executed. This information is used in the next iteration to generate a synthetic edge in the static call-graph that represents the newly discovered execution flow. Often, this turns apparently dead code into reachable code and Tiro will target this code on the next iteration. For obfuscation that executes dynamically loaded code (e.g. dynamic loading, DEX file hooking, etc.), Tiro's instrumentation extracts the code that is actually executed into an *extraction file*, and a process monitoring Tiro's instrumentation log pulls this file from the device. The extracted code is then included in the static analysis in the following iteration.

5.3.5 Example of iterative deobfuscation

To show how the phases of Tiro can iteratively deobfuscate an application and enable more effective analysis, we present an example of Tiro's process on the *dexprotector* packer, an obfuscation tool for Android applications. The packed application comprises of multiple layers of obfuscation, using a combination of reflection to invoke dynamic loading APIs and to invoke methods in the dynamically loaded code, which then contain further reflection and native code invocation that must be deobfuscated. Figure 5.3 shows how Tiro iteratively applies the T-i-r-o loop to deobfuscate the combination of techniques used by the *dexprotector* packer and to extract a complete application call-graph.

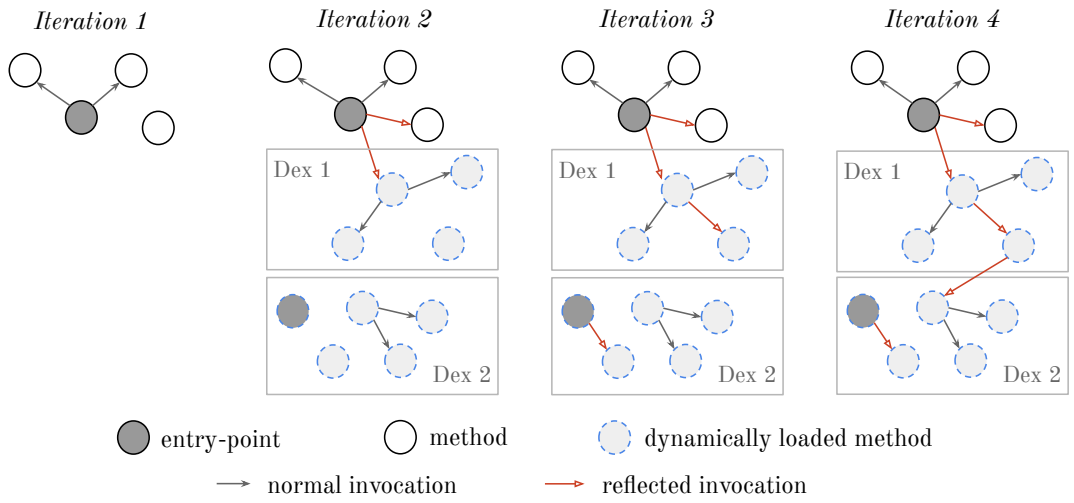


Figure 5.3: Deobfuscated call-graphs produced for an application packed with *dexprotector*

Iteration 1: The scope of the static analysis is limited to code in the application's APK file. Tiro finds locations of reflected method invocations and instruments them to determine the reflection targets. The dynamic phase executes the instrumented code and reports the reflection targets. It also finds two dynamically loaded DEX files.

Iteration 2: The static analysis scope is expanded to include code from these two DEX files. This code includes entry-points into the application that were previously unknown. However, the use of reflection in the dynamically loaded code means that the call-graph may miss certain invocation edges. Tiro's static analysis adds new instrumentation for any obfuscation (namely, reflection) found in the APK code or dynamically loaded code. The dynamic phase will again execute the instrumented code to find the reflection targets.

Iteration 3: Some reflective call edges are resolved in the static call-graph; however, Tiro still sees seemingly-dead code from the second dynamically loaded DEX file. The process is repeated until Tiro encounters no new unresolved obfuscation/reflection.

Iteration 4: The final result is a static call-graph that represents all of the code executed by an application and the method invocation relationships. If used alongside a security analysis tool, malicious actions performed by the application can then be discovered by searching the deobfuscated call-graph.

5.4 Implementation

We implemented the static and dynamic portions of Tiro on top of the targeted analysis from IntelliDroid and added the ART instrumentation that deobfuscates runtime-based obfuscation.

5.4.1 AOSP modifications

The modifications to AOSP are located within the ART runtime code (`art/runtime` and `libcore/libart`). We have implemented these changes on three different versions of AOSP: 4.4 (KitKat), 5.1 (Lollipop), and 6.0 (Marshmallow) due to the portability issues of the DEX file hooking technique, which is performed by most of the malware in our datasets. In order to access the private `DexFile::mCookie` field for DEX file hooking, applications must use reflection or JNI, but the `mCookie` field type has changed from an `int` in 4.4, to a `long` in 5.0, and finally to an `Object` in 6.0. These changes and other conventions that the malware relies upon (such as private method signatures and locations of installed APKs) result in crashes when the applications are not executed on their intended Android version.

5.4.2 Soot modifications

The static component of Tiro is based on the port of IntelliDroid to the Soot [116] static analysis framework. To incorporate deobfuscated values back into the static portion of Tiro, we made several modifications to Soot. Most of these changes were in the call-graph generation code, where we tag locations at which deobfuscated values were obtained and add special edges to the call-graph representing dynamically resolved/deobfuscated invocations. We use a context-insensitive call-graph and the dynamically extracted data used to add the deobfuscated edges does not include context information. However, the dynamic analysis could easily be extended to record contextual data, such as the call stack at the moment of obfuscation, to support the construction of a deobfuscated context-sensitive call-graph. Other deobfuscated values/variables are tagged in the intermediate representation and can be accessed in the post-call-graph-generation phases of Soot.

Some obfuscated applications are armored to prevent parsing by frameworks such as Soot. For example, there were several instances of unparseable, invalid instructions in methods that appear to be dead code. While this code is never executed, a static analysis pass would still attempt to parse these instructions, resulting in errors that halt the analysis. In cases where a class definition or method implementation is malformed (which often occurs for applications performing DEX bytecode modification), we skip these classes/methods and do not produce an instrumented version. If the bytecode is modified at run-time, Tiro will extract them and instrument them in the following iteration.

5.4.3 Instrumenting log statements

To facilitate the communication of information between the static and dynamic portions of Tiro, certain pieces of information are reported through instrumented log statements. This includes the fully-qualified name of the method containing the obfuscation location, as well as the bytecode index of the value being deobfuscated. Currently, the bytecode index is actually an index for the `Jimple` intermediate representation in Soot but since these are fed back into a static analysis that is also using Soot, no conversion to DEX bytecode indices is necessary. For native code deobfuscation, Tiro also reports the Java method that first called into the native code.

5.5 Evaluation

To evaluate Tiro's accuracy, we acquired a labeled dataset of 34 malware samples, each obfuscated by one of 22 different Android obfuscation tools. This dataset was provided by the Android Malware team at Google and were transferred to us in two batches: one in March 2017 and another in October 2017. The samples in the dataset were chosen for their use of advanced obfuscation capabilities and difficulty of analysis, and attention was made to ensure that they represent a wide range of state-of-the-art obfuscators. Each sample was manually confirmed as malware and classified by a security analyst from Google, independent of our own analysis using Tiro. To evaluate Tiro's accuracy, we shared the results of Tiro's analysis with Google and they confirmed or denied our findings on the samples.

In our evaluation, the static portion of Tiro was executed on an Intel i7-3770 (3.40GHz) machine with 32 GB of memory, 24 GB of which were provided to the static analysis JVM. The dynamic portion was executed on a Nexus 5 device running Tiro's instrumented versions of Android 4.4, Android 5.1, and Android 6.0.

We begin by evaluating Tiro's accuracy, as well as detailing the findings made by Tiro on the labeled dataset. Then, to measure the use of obfuscation on malware in the wild, we apply Tiro to 2000 obfuscated malware samples from VirusTotal [117]. Finally, we present an analysis of Tiro's performance.

5.5.1 General findings

Table 5.1 summarizes our findings after running Tiro on the labeled dataset. The table lists the name of the obfuscator, the number of samples from that obfuscator, the obfuscation techniques found by Tiro and the number of iterations Tiro used to fully deobfuscate the sample. We also show the number of sensitive APIs that are statically visible before and after Tiro's deobfuscation. For obfuscation tools where there was more than one sample, the table shows the results for the sample with the most sensitive behaviors detected.

After sharing our results with the Google Android Malware team, we confirmed that Tiro successfully found and deobfuscated the known obfuscated code in the applications, with the exception of the two samples packed with *unicomsdk*, and was able to reach and analyze the original applications (i.e. the bytecode for the underlying application before it was obfuscated or packed). On closer analysis, we found Tiro failed on the *unicomsdk* samples because while Tiro does trigger call paths that invoke dynamic loading, the obfuscation code tries to retrieve bytecode from a network server that is no longer active. Our comparison also showed that Tiro did not have any false positives on the dataset—in no case did

Table 5.1: Tiro deobfuscation results

		Obfuscation								Tiro	Sensitive APIs	
		Language-based			Runtime-based							
Sample	#	Reflection	Dynamic loading	Native code	DEX file hooking	Class data overwriting	ArtMethod hooking	Instruction hooking	Instruction overwriting	Iterations	Before Tiro	After Tiro
aliprotect	2	•	ⓝ	•	•	•				3	0	44
apkprotect	1	•	ⓓ	•						2	8	52
appguard	1	•		•	•					2	0	5
appsolid	1	•	ⓝ	•						2	0	82
baiduprotect	1	•	ⓝ	•	•	•				2	1	2
bangcle	1	•	ⓝ	•						2	1	4
dexguard	3	•								2	0	4
dexprotector	3	•	ⓡ	•						4	0	80
dxshield	2	•	ⓝ	•	•					2	3	25
ijiamipacker	2	•	ⓝ	•	•	•	•	•	•	2	1	93
liapp	1	•	ⓝ	•						2	4	90
naga	1	•	ⓝ	•	•					2	2	2
naga_pha	1	•	ⓝ	•	•	•	•	•	•	2	0	6
nqprotect	1	•	ⓓ	•						2	1	12
qihoopacker	3	•	ⓝ	•	•					2	3	217
secshell	2	•	ⓡ ⓝ	•	•	•				2	200	287
secneo	1	•	ⓝ	•						3	0	12
sqlpacker	2	•	ⓓ	•						2	1	31
tencentpacker	2	•	ⓝ	•	•					3	3	504
unicomsdk	2	•	ⓓ	•						2	226	227
wjshell	1	•	ⓓ	•	•					2	8	13

ⓓ Direct dynamic loading ⓡ Dynamic loading invoked via reflection ⓝ Dynamic loading invoked in native code

Tiro mistake legitimate state modification performed by ART for an attempt to perform runtime-based obfuscation by the application.

We make several general observations about the results. First, all of the malware samples employed basic language-based obfuscation such as reflection and native code usage, while roughly 53% (18/34) of the samples also employed the more advanced runtime-based obfuscation techniques. We note that none of the samples in this set employed method entry-point hooking, perhaps owing to their age as these samples are older than those used in our VirusTotal analysis described in Section 5.5.3. In addition, all used between 2-4 layers of obfuscation, requiring multiple iterations by Tiro. These findings demonstrate the utility of Tiro's iterative design and ability to simultaneously handle multiple types of obfuscation.

Second, many of the obfuscators employed tactics to make analysis difficult. For example, 21 of the 34 samples included code integrity checks that Tiro's code spoofing was able to circumvent. In addition, a common post-loading step in most of the samples was the deletion of the decrypted code file after it had been loaded. This made it marginally more difficult to retrieve the code, since the unpacked DEX file was unavailable after it was loaded; however, since Tiro extracts DEX code from memory during the loading process, this did not impact its deobfuscation capabilities.

Finally, in all cases, the obfuscation was used to hide calls to sensitive APIs in Java (i.e. sensitive behaviors), which were used to perform malicious activity. The number of sensitive APIs shown in Table 5.1 are the number of API calls found by static analysis before and after running Tiro, where the set of sensitive behaviors were obtained from FlowDroid's [10] collection of sources and sinks. On average, Tiro's iterative deobfuscation resulted in over 30 new hidden sensitive API uses detected in each sample. The new sensitive behaviors detected after Tiro's iterative deobfuscation included well-known malware behaviors such as premium SMS abuse and access to sensitive data, including location information and device identifiers.

5.5.2 Sample-specific findings

We now describe in detail some of the interesting behaviors and obfuscation techniques Tiro uncovered:

aliprotect: During Tiro's first iteration, we found that the APK file contained only one class (`StubApplication`) that set up and unpacked the application's code. Static analysis found only one case of reflection to instrument and one direct native method invocation via `System.load()`. During dynamic analysis, we found that the sample used DEX file hooking to load the main application code dynamically. After loading, the obfuscated DEX file was also overwritten prior to class loading to change the bytecode defining the application's main activity. When extracting the modified DEX bytecode, Tiro found that some of the class data pointers referred to locations outside the DEX code buffer (i.e. outside the DEX file). The application stored code in separate memory locations and, via pointer arithmetic, modified the DEX class pointers to refer to those locations. In the second iteration, static analysis showed that most of the methods in the obfuscated (and now extracted) DEX file were empty—when invoked, they would throw a run-time exception. These empty methods and classes appeared to be decoys and were never actually executed by the application. The methods and classes that were executed had undergone DEX bytecode modification, and Tiro successfully extracted the new non-empty implementations.

apkprotect: In the first iteration, Tiro found several classes in the APK file, none of which were the components declared in the manifest. In the dynamic phase, instrumentation of dynamic loading and reflection retrieved the dynamically loaded code and deobfuscated the reflection targets. From the run-

time information gathered, Tiro reported that a number of class objects were requested via reflection, but only one was instantiated via a reflected call to the constructor method.

In the second iteration, Tiro found that only the class that was instantiated was actually present in the dynamically loaded code. Further analysis showed that the application performed a trial-and-error form of class loading, where it looped through class names `app.plg_v#.Plugin` (with `#` a sequentially increasing integer) until it found a class object that could actually be retrieved and instantiated. This form of class loading would have introduced a great deal of imprecision in static analysis since the class name was unknown and obscured by the loop logic; however, with the dynamic information retrieved by Tiro, the static analysis in the subsequent iterations was able to precisely identify the loaded and executed class. During the static phase, Tiro also found two methods within the dynamically loaded code that contained invalid instructions and were unparseable. These methods did not appear to be invoked but attempting to load them without patching Soot resulted in crashes stemming from parsing errors.

baiduprotect / naga / naga_pha: These samples used DEX file hooking to load code dynamically but they would also modify the hooked DEX file multiple times in their execution. Each modification would change the data for one class but also invalidated header values in another; therefore, after the DEX bytecode modification process had begun, no single snapshot of the DEX code memory buffer would result in a valid DEX file. Since Tiro retrieves modified code in a piecemeal fashion as the modification is detected for each class (rather than taking a single snapshot of the buffer), it was able to handle the multiple code modifications and the subsequent mangling of class metadata.

dexprotector: In Section 5.3.5, we described how Tiro deobfuscates the multiple layers of obfuscation used in this sample. It used a combination of reflection, dynamic loading, and native code invocation to hide its actions from analysis. Tiro required multiple iterations to fully deobfuscate the sample and retrieve the full static call-graph.

ijiamipacker: When first installing this APK, the `dex2oat` tool reported a number of verification errors in most of the classes. Tiro's static analysis had similar results but within the parseable classes, it detected instances of reflection, native methods, and dynamic loading. The dynamic phase showed that some of the classes with DEX verification errors were executed without error due to dynamic modification of the classes' bytecode. Furthermore, the methods were modified one at a time as they were loaded by the class loader, which was achieved by hooking a method within the class loader. In the second iteration, Tiro was able to analyze the extracted bytecode for the now-parseable classes and instrumented new cases of reflection.

We also found that this sample suppressed log messages after a certain point in the unpacking process before the main activity was loaded. Since Tiro's feedback system of relaying dynamic information to static analysis depends on instrumented log messages, this initially posed a problem for deobfuscation. Fortunately, this sample did not suppress error logs, so Tiro was modified to write to the error log as well. A more robust approach would be to implement a custom deobfuscation log that only Tiro can access and control.

qihoopacker: In addition to the DEX file hooking obfuscation that this sample employed, we found that it also invoked `art::RegisterNativeMethods()` to redefine the code pointer for the native method `DexFile.getClassNameList()`. This is a form of native method hooking, where the native function

attached to a method is swapped for another. The hooked method `getClassNameList()` does not actually play a part in the class loading process nor was it used by the application; however, it is useful for code analysis as it returns a list of loaded classes and its redefinition made such interactive analysis more difficult.

For completeness, we also found two publicly available method hooking libraries: Legend [67] and YAHFA [73], and used these to create our own application obfuscated with method hooking. For both libraries, Tiro detected the hooked methods, which contained modified method entry-point pointers. These pointers were redirected to custom trampoline/bridge code that resolved the hooked invocation and invoked the target method specified by the developer. Tiro heuristically reported the method objects retrieved by the application that were likely to serve as target methods for this hooking, and in the following iterations, correctly constructed call edges between the hooked and target methods.

5.5.3 Evaluation on VirusTotal dataset

We also use Tiro to measure the types of obfuscation used by malware in the wild. We searched VirusTotal for malware tagged as obfuscated or packed, and downloaded 2000 randomly selected samples that were submitted throughout the month of January 2018. When Tiro was run on this dataset, it exceeded the 3 hour timeout on the static analysis phase for four of the samples and ran out of memory on two others. Of the remaining samples, all proceeded to instrumentation and analysis by Tiro's dynamic phase. Table 5.2 shows the breakdown of the types of obfuscation found by Tiro.

On this dataset, a larger proportion (80%) of these applications used runtime-based obfuscation techniques, compared to 53% on the labeled dataset. In addition, usage of all types of runtime-based obfuscation were observed, including method entry-point hooking. While this dataset is larger, we speculate that these differences and the broader use of runtime-based techniques likely owe more to the fact that the malware in this dataset are more recent than those in the previous labeled dataset.

The most frequent form of runtime-based obfuscation found was DEX file hooking, which is likely due to the ease of implementing the state modification (i.e. the `DexFile::mCookie` field) required for the obfuscation. Likewise, use of instruction hooking was also prominent, since the obfuscation required changing just the DEX code pointer (and possibly the compiled OAT code pointer) in `ArtMethod` objects. Techniques that require overwriting larger regions of memory or more precise determination of a location to modify (e.g. modifying a vtable entry for ArtMethod hooking) were much less common. This may be due to the implementation effort of these techniques, which require greater knowledge of the runtime objects being modified to ensure that any overwriting maintains the expected layout of these objects and preserves the stability of the runtime. However, we do see instances of these techniques in recent malware, and the overall frequency of runtime-based obfuscation techniques in our dataset is likely in response to advances in analyses that can deal with the simpler and more well-known language-based techniques.

5.5.4 Performance

We evaluate the performance of the static and dynamic phases in Tiro separately. The run time of the static component increases as iterations find and deobfuscate more code to analyze. In the first iteration of the static component (where the analysis is only targeting obfuscation locations in the original APK

Table 5.2: Obfuscation in VirusTotal samples from January 2018

Language-based		**Runtime-based**	
Reflection	58.5 %	DEX file hooking	64.0 %
Dynamic loading	79.9 %	Class data overwriting	0.7 %
Direct	52.2 %	ArtMethod hooking	0.5 %
Reflected	0.1 %	Method entry hooking	0.3 %
Native	49.2 %	Instruction hooking	33.7 %
Native code	96.8 %	Instruction overwriting	0.1 %

file), the average static analysis time for the samples in Table 5.1 is 4.3 minutes. However, after the last iteration, the static component takes an average of 12.2 minutes across our dataset.

Tiro's instrumentation also incurs overhead in its dynamic phase. Since the majority of obfuscation occurs in the application launch phase (i.e. when the application unpacks its main activity and other components), we compare the launch time of the application when running in Tiro against the launch time in an unmodified version of AOSP. On average, there is a 3.3× slowdown, with all of the applications launching in under 11 seconds. The majority of this overhead is due to the checking of ART runtime state before and after native code is executed. While this is a noticeable performance impact, we note that Tiro is meant for analysis and not production usage; thus, while the slowdown is large, applications still launch and run in a reasonable amount of time. To further reduce performance overhead, we believe that we can optimize Tiro's monitoring using hardware support. Currently, a full check is performed of all tracked runtime state on every native-to-Java transition. By manipulating memory protections or dirty bits in the hardware page table to identify modified pages, and tracking which objects are stored on those pages, Tiro can reduce the number of objects it must check for modifications.

5.6 Discussion

From our analysis of obfuscation in recent Android malware, we identify and classify a type of runtime-based obfuscation that differs from obfuscation seen in previous work on x86 and Java. The use of a runtime introduces another technique of hiding code that we show is already in use in Android malware.

5.6.1 Obfuscation in benign applications

In addition to our analysis of obfuscation in Android malware, we also ran Tiro on the most popular applications from the Google Play marketplace across a variety of different categories. In order to analyze benign applications from Google Play, which are large and complex, we integrated Tiro with the context-based techniques proposed in Car to enable effective dependency resolution when target obfuscation paths depend on program state from other parts of the application.

Due to the size and complexity of popular applications, we found that our earlier implementation for runtime state monitoring in ART was too performance intensive, with some applications taking over 30 minutes to load when launched. To analyze benign applications effectively, we instead used a less strict form of monitoring, where the memory permissions for pages containing ART runtime state

were set to read-only and a tracing process running `ptrace` on the application recorded the subsequent memory faults when they were modified. A separate monitoring thread periodically checked the modified pages every 50 ms and determined whether runtime state, such as DEX files, classes, methods, and code pointers, were modified since the last check. It is possible for an application to tamper with runtime state, execute the modified code, and revert the changes within this period and thus evade detection; however, for the purposes of analyzing benign applications, this looser monitoring strategy was sufficient.

Using the same dataset of applications from Google Play as in the evaluation for Car (Section 4.5.2), we found the use of language-based obfuscation in most of the applications, with 95.1% using reflection. In addition, 61.5% invoked native methods, where many instances of the native code usage were located in third-party libraries. Dynamic code loading was found in 27.9% of applications as well.

For runtime-based obfuscation, we found 38.1% of the applications made use of method entry-point hooking, which is the technique employed by method hooking libraries such as Legend [67], YAHFA [73], and ZHookLib [138], and by the Xposed framework for modifying Android ROMs [129]. This was the most prevalent form of runtime-based obfuscation found, which is not surprising since the other runtime-based techniques were primarily used by packers and we found no indication that such packers were used by our dataset of benign applications. The methods that were hooked varied across the applications but from their names, seemed to involve access to library or system functionality, such as native cryptographic libraries and data serialization libraries. It is possible that method hooking was used by these libraries to provide device and version compatibility across the multiple Android devices and versions that are still widely used [93]. This can present a consistent interface for applications while changing the operation of the underlying library functionality based on the execution environment, such as invoking different framework or system methods that are only available in certain Android versions. A further 19.8% of the applications used DEX file hooking to load custom DEX files dynamically. Most of these were located in a third-party advertisement library used by multiple applications. The use of DEX file hooking rather than the usual code loading APIs could be due to the use of native code (DEX file hooking forgoes the need for JNI invocations to the framework class loading APIs) or to modify the behavior of certain classes by directly injecting a custom implementation in a DEX file such that it is prioritized over code in existing loaded DEX files in the application's class loader. This would be similar to the injection of instrumented application classes in our modifications to the ART class loading mechanism.

Based on our analysis of benign and malicious applications, the use of certain runtime-based obfuscation techniques is not necessarily indicative of malicious activity but can serve as a prioritization strategy for further analysis of questionable applications. Some run-time based techniques, such as the hooking of DEX files, were used in both datasets but were less common in the benign applications. Others, such as the bytecode modification of DEX files and class data, hooking of ArtMethod pointers, or rewriting of instructions, were detected only in our dataset of malicious applications. Benign applications could potentially use these techniques but there does not appear to be a good use case for them. While the use of obfuscation is not malicious in itself, any detection of runtime-based obfuscation may warrant a closer look at the actions taken by an application to determine why they might be obfuscated or hidden.

5.6.2 Bypassing the runtime

Unlike language-based obfuscation where the application abuses Java language features, runtime-based obfuscation requires modifying runtime data, which must be done using native code. A natural question

is whether runtime-based obfuscation is a stepping stone toward full-native code obfuscation. Static analysis of native code is more imprecise and most existing static malware analyzers for Android are limited to Java bytecode, so a full native code application would make them ineffective. We argue that runtime-based obfuscation is not superseded by full native code but is a complementary technique.

In runtime-based obfuscation, native code is used to modify the runtime state but the execution inevitably returns to Java code after the modifications have been performed. This highlights the main difference between the two forms of obfuscation: in runtime-based obfuscation, the actual malicious behavior can be implemented in Java. Whether this is useful to the malware developer is dependent on the type of malicious activity they wish to execute on a victim's device and how they want to implement it. Many state-of-the-art obfuscators are commercial tools that add wrapper classes to an application to pack them into an obfuscated APK and unpack them when the application is launched. Runtime-based obfuscation allows for complex obfuscation while still allowing the users of these commercial tools to implement their code in Java, which may be preferable due to ease of development. Reusing the existing runtime on Android makes it easier for commercial obfuscation tools to reliably support all forms of Android applications.

In addition, system services are normally accessed through their RPC interface, which would require a transition back into the runtime and would be detected by Tiro's monitoring of native-to-Java transitions. To avoid any Java code (i.e. a true fully native application), the application would have to access system services by calling the low-level Binder interface or Unix `ioctl`s directly. Since the Binder library is not part of the Android NDK, the application is then sensitive to any changes in implementation in the Binder kernel driver or Android service manager. We believe that this is one of the reasons why language- and runtime-based obfuscation is so prominent on Android despite the long history and effectiveness of native code obfuscation on x86. As a result, for the foreseeable future, language- and runtime-based obfuscation techniques will likely still be relevant techniques for obfuscated code on Android.

Another form of obfuscation may be to embed a natively-implemented interpreter within the application that executes a secret bytecode. This is a complementary technique to runtime-based obfuscation and is also a method of bypassing the ART runtime, since the interpreter would be fully implemented in native code. Similar to full-native code obfuscation, access to system services would be limited and invocations to framework methods would still require execution in the ART runtime and would therefore be deobfuscated by Tiro.

5.6.3 Other limitations

Part of Tiro's deobfuscation focuses on retrieving DEX bytecode that the application dynamically loads and executes. This implicitly assumes that any manipulation of the DEX bytecode is reflected in the compiled OAT or ODEX code, and vice versa. Obfuscation code may violate this assumption and perform modifications directly on the OAT or ODEX bytecode, bypassing the current implementation of Tiro. However, in doing this, the obfuscation code forgoes portability across devices, as OAT and ODEX files are device-specific. We did not observe any malware instances that were device-specific in this way. If direct OAT or ODEX modification were to exist, it would be straightforward to enhance Tiro to detect these modifications by monitoring `art::OatFile` objects in the same manner as `art::DexFile` objects.

While we have identified a number of forms of runtime-based obfuscation in Section 5.2.3, there may be others that Tiro currently does not monitor, providing avenues for newer malware to avoid detection and deobfuscation. However, the framework proposed in Tiro is general enough to accommodate the

monitoring of other forms of runtime state as they are identified. A further limitation is that applications can employ x86 obfuscation and hooking techniques to bypass Tiro's monitoring within the ART runtime. While we currently cannot prevent this, due to the shared address space between the application and the runtime environment, future work may explore the separation of application and runtime memory, which would also prevent tampering of runtime state and disable runtime-based obfuscation.

Since Tiro relies on dynamic analysis to report deobfuscated values, full deobfuscation of an application would require executing all of its obfuscation code. Since Tiro was implemented on top of IntelliDroid and extended with Car, we rely on them to execute targeted obfuscation locations. However, because the analysis is limited to Java, while Tiro can target native method invocations, it cannot extract execution paths within native code. Since native code is used extensively by obfuscators, we may miss certain paths. In addition, Tiro may not be able to extract all targeted paths and constraints due to static imprecision and complex path constraints in the code. Tiro can be combined with fuzzers if deobfuscation is required in native code or in execution paths with constraints that cannot be solved.

5.7 Related work

A variety of security and privacy analyzers have been developed for Android, including static [10,44] and dynamic tools [40,113,114,132]. Tiro is a hybrid system similar to [98,99,123,127], which use dynamic information to enhance static analysis. Tools that perform malware classification [9,44] are often based on application semantics and rely on the ability to determine the actions performed by an application. While they are effective against unobfuscated applications, they cannot handle complex code obfuscation and will likely miss malicious actions that the malware performs. While some tools have been designed with obfuscation resilience in mind [48], they often cannot handle the complex obfuscation techniques used by existing Android packers and malware.

The work that most closely resembles Tiro are existing deobfuscation tools for Android. Some focus only on language-based obfuscation. Harvester [98] uses static code slicing to execute paths leading to specific code locations, such as reflection invocations, and can log deobfuscated values. However, code slices do not always produce realistic executions and it does not handle runtime-based obfuscation. StaDynA [142] uses a hybrid iterative approach similar to Tiro to deobfuscate reflection and retrieve dynamically loaded code. However, it relies on instrumentation of reflection and dynamic loading API invocations. Some Android unpackers, such as DexHunter [141] and Android-unpacker [111], handle certain cases of DEX file and DEX bytecode manipulation, but use special packer-specific values to identify the code that must be extracted. They also do not handle any other form of obfuscation, which makes it difficult to analyze the retrieved code if it is further obfuscated in another way. Others, such as PackerGrind [131] and AppSpear [60] have a more general design but their monitoring for bytecode modification is limited to instrumentation of specific methods they expect obfuscation code to use. While these unpackers identify certain cases of DEX bytecode modification, they do not handle other forms of state modification in the code execution process nor do they address the wider issue of runtime-based obfuscation. DroidUnpack [39] uses full system emulation to dynamically extract packed code. While DroidUnpack can extract dynamically loaded code and decrypted DEX files, they do not discuss or indicate if they can handle runtime-based obfuscation the way Tiro can. DeGuard [21] takes a different approach and uses a statistical model to reverse the name obfuscation performed by the ProGuard [56] tool included with the Android SDK. Since Tiro focuses on the actions taken by an application, we do

not deobfuscate class and method names. However, combining the results of Tiro and DeGuard would aid in manual analysis of malware.

Tiro is also similar to deobfuscation tools proposed for general Java applications. TamiFlex [23] deobfuscates reflection by instrumenting the reflection classes loaded by the Java runtime, but does not handle other forms of obfuscation. However, its modification of the class loader in the runtime is similar to the technique used in Tiro to load instrumented application classes. Similarly, Ripple [139] also targets reflection but does so through static resolution, which is less precise. These tools do not address runtime-based obfuscation.

Deobfuscation and unpacking tools also exist for x86 applications. Renovo [63] tracks whether previously written memory regions are being executed and can handle multiple "hidden layers" of packing. Polyunpack [104] checks whether dynamic instruction sequences match those in its static model of the application and returns new unpacked instruction sequences. Ether [38] presents a transparent malware analysis tool that handles emulator-resistant techniques used by packers to prevent reverse engineering. Omniunpack [78] uses an in-memory malware detector to determine if malicious code is being unpacked and retrieves this code from memory. These techniques are more general than those used in Tiro but would require special support to handle the Android runtime and its code loading processes. By focusing on obfuscation for the Android runtime via language-based and runtime-based deobfuscation, we account for the environment in which Android applications are run and produce effective results that can be integrated with existing Android security tools.

5.8 Summary

In this chapter, we described how hybrid program analysis can be fully realized through a feedback loop that passes information between both static and dynamic analysis. This was enabled through targeted execution, where data from static analysis is used to guide execution of the application. Information gather from this execution can then be incorporated back into static analysis. In addition to the security analyses presented in Chapters 3 and 4, we show how this extension of targeted analysis can be used to aid in the deobfuscation of Android applications.

We identify a family of obfuscation techniques used on the Android platform, which we name *runtime-based obfuscation*. These techniques subvert the integrity of the Android runtime to manipulate the code loading and execution processes and execute malicious code surreptitiously. We propose Tiro, a unified deobfuscation framework for Android applications that can deobfuscate runtime-based obfuscation as well as traditional techniques such as reflection or native method invocation. Through an iterative process of static instrumentation and dynamic information gathering that uses **T**arget, **I**nstrument, **R**un and **O**bserve, we show that Tiro is able to deobfuscate malware that have been packed using state-of-the-art Android obfuscators. We also show that runtime-based obfuscation is prevalent among recent Android malware and that effective security analysis will require deobfuscation of these techniques. Using the deobfuscated application information produced by Tiro, it is possible for existing security analysis tools to achieve more complete analysis and detection of Android malware.

Chapter 6

Conclusion

In this thesis, we propose a new method of performing security analysis of mobile applications: targeted execution. We showed how a novel combination of static and dynamic techniques can be used to extract and execute specific behaviors in Android applications such that resources can be used more effectively for the security analysis to be performed.

With our work in IntelliDroid, we present the first prototype for targeted security analysis and show how focusing execution of applications to specific behaviors of interest (namely, APIs to sensitive functionality) can enable greater effectiveness in security analyses such as the detection of malicious activity and of private data leakage. Unlike purely static techniques, the use of dynamic analysis enables greater precision in the resulting analysis. Unlike purely dynamic techniques, the use of static analysis to generate an over-approximation of suspicious behavior focuses the dynamic analysis on interesting paths, thus excluding paths that do not contain interesting behaviors and saving analysis resources. IntelliDroid showed how static and dynamic techniques can together improve security analysis.

In CAR, we address the challenges of restricting execution to specific target code paths in an application. In particular, we identify the issue of dependencies, which create interconnections between different parts of an application, including between target and non-target paths. We use the concept of contexts to represent the constraints a path may have on system state and we resolve these constraints by generating an approximated context for the path's execution and refining it dynamically to account for approximation errors. We showed that the use of hybrid dependency resolution techniques can achieve much greater coverage of target behaviors, improving the analysis and detection of sensitive behaviors for all types of applications.

Finally, we apply the idea of targeted execution to the obfuscation of applications, which is commonly used to hide malicious activity. We show how the hybrid program analysis techniques in IntelliDroid and CAR can be extended to form a fully interactive static-dynamic feedback loop in TIRO that incrementally improves the analysis of Android applications. Through this gradual refinement of the analysis, code and actions that were previously obfuscated or undetected in an application can be uncovered and analyzed. We further show how TIRO can deobfuscate a new form of obfuscation in Android that subverts the runtime executing an application's code such that hidden malicious actions are performed unexpectedly. The use of targeted execution to enable deobfuscation, in addition to the previous use cases of detecting of privacy leaks and sensitive behaviors, demonstrates the flexibility of our approach and its contribution to improving security analysis of Android applications.

6.1 Future work

We believe that hybrid program analysis is the most effective means of performing security analysis of applications. One area of further exploration is the merging of the static and dynamic phases of Tiro such that the iterative process is more fine-grained. For example, when the static component identifies a target path and constructs an execution context, the dynamic analysis could immediately execute it and determine if further refinement or deobufuscation is required. This would improve the performance of the overall analysis, as certain target paths may require more iterations than others to uncover, perhaps due to layers of obfuscation. This fine-grained approach may yield more complete results as analysis resources would be more efficiently used and certain expensive static analyses, such as the construction of the call-graph (which comprises a significant amount of time in the static portion of our hybrid tools), can be refined in place without having to be reconstructed between static-dynamic iterations.

A further area of research for hybrid program analysis is how it can be applied to other forms of security analysis or general program analysis tasks. In our experimentation, we found that a large portion of static analysis time is spent on generating the application's call-graph model and its points-to data-flow model. This process is iterative, as the call-graph uses propagated object aliases to determine the receiver (and receiving class) of method invocations and the points-to analysis relies on the call-graph to propagate object aliases interprocedurally. It may be possible to reduce the construction time by incorporating dynamic information (perhaps achieved through an initial fuzzing phase) into the static call-graph such that locations of imprecision, such as reflective call edges or locations of high aliasing in the control- or data-flow, can be disambiguated more easily by relying on dynamic data rather than iterative propagation of static results. The result hybrid call-graph can be iteratively refined through more targeted dynamic analysis to explore specific paths or regions that were missed in the initial exploration phase, using the techniques proposed in IntelliDroid and Car. This further motivates the idea of a fine-grained hybrid program analysis framework. Further security analysis tools could then be built on the hybrid call-graph, data-flow analysis, or other constructs, similar to how they are currently built on static or dynamic analysis frameworks.

The idea of targeted execution can also be combined with other dynamic code exploration techniques. From our results in Car, we saw that while targeted execution can reach significantly more target code locations than random fuzzing and model-based exploration, these other techniques can also reach locations that were missed by targeted execution. This was primarily due to the limitations in constraint solving and the difficulties of tracking and resolving dependencies through the approximated context. Because Car essentially provides a self-contained environment for a target path's execution, perhaps it can be combined with fuzzing techniques applied on the inputs to this environment (e.g. arguments to the path driver method), resulting in a "fuzzy" targeted analysis that not only explores the target path but also the code surrounding the path that might contain related functionality.

Bibliography

[1] 3GPP. Technical realization of Short Message Service (SMS). TS 23.040, 3rd Generation Partnership Project (3GPP), September 2014.

[2] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. Precise Android API protection mapping derivation and reasoning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS 2018)*, pages 1151–1164. ACM, 2018.

[3] Adrien Abraham, Radoniaina Andriatsimandefitra, Adrien Brunelat, J-F Lalande, and V Viet Triem Tong. GroddDroid: A gorilla for triggering malicious behaviors. In *Proceedings of the 10th International Conference on Malicious and Unwanted Software (MALWARE, 2015)*, pages 119–127. IEEE, 2015.

[4] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. Going native: Using a large-scale analysis of Android apps to create a practical native-code sandboxing policy. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS 2016)*. The Internet Society, 2016.

[5] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2012)*, page 59. ACM, 2012.

[6] Benjamin Andow, Adwait Nadkarni, Blake Bassett, William Enck, and Tao Xie. A study of grayware on Google Play. In *Proceedings of the 2016 IEEE Security and Privacy Workshops (SPW 2016)*, pages 224–233. IEEE, 2016.

[7] Android Studio: Test your app. https://developer.android.com/studio/test, 2020. Accessed: July 2020.

[8] APKParser. http://code.google.com/p/xml-apk-parser/, 2014. Accessed: September 2014.

[9] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. DREBIN: Effective and explainable detection of Android malware in your pocket. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS 2014)*. The Internet Society, 2014.

[10] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick Drew 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 2014)*, pages 259–269. ACM, 2014.

[11] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android permission specification. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS 2012)*, pages 217–228. ACM, 2012.

[12] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic exploit generation. In *Proceedings of the 18th Annual Symposium on Network and Distributed System Security (NDSS 2011)*. The Internet Society, 2011.

[13] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*, pages 1083–1094. ACM, 2014.

[14] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA 2013)*, pages 641–660. ACM, 2013.

[15] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA 2011)*, pages 12–22. ACM, 2011.

[16] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. On demystifying the Android application framework: Re-visiting Android permission specification analysis. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security 2016)*, pages 1101–1118. USENIX Association, 2016.

[17] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC 2016)*, pages 189–200. ACM, 2016.

[18] David Barrera, Jeremy Clark, Daniel McCarney, and Paul C Van Oorschot. Understanding and improving app installation security mechanisms through empirical analysis of Android. In *Proceedings of the Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM@CCS 2012)*, pages 81–92. ACM, 2012.

[19] Luciano Bello and Marco Pistoia. ARES: Triggering payload of evasive Android malware. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft@ICSE 2018)*, pages 2–12. ACM, 2018.

[20] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. Brahmastra: Driving apps to test the security of third-party components. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 2014)*, pages 1021–1036. USENIX Association, 2014.

[21] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of Android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*, pages 343–355. ACM, 2016.

[22] Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An Android application sandbox system for suspicious software detection. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE 2010)*, pages 55–62. IEEE, 2010.

[23] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pages 241–250. ACM, 2011.

[24] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP 2008)*, pages 143–157. IEEE, 2008.

[25] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for Android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM@CCS 2011)*, pages 15–26. ACM, 2011.

[26] Juan Caballero, Pongsin Poosankam, Stephen McCamant, Dawn Song, et al. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010)*, pages 413–425. ACM, 2010.

[27] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, pages 209–224. USENIX Association, 2008.

[28] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2007)*. ACM, 2007.

[29] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, and Long Lu. SAVIOR: Towards bug-driven hybrid testing. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP 2020)*, pages 1580–1596. IEEE, 2020.

[30] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys 2011)*, pages 239–252. ACM, 2011.

[31] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[32] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1998)*, pages 184–196. ACM, 1998.

[33] 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 24th Annual Network and Distributed System Security Symposium (NDSS 2017)*. The Internet Society, 2017.

[34] Valerio Costamagna and Cong Zheng. ARTDroid: A virtual-method hooking framework on Android ART runtime. *Proceedings of the 2016 Workshop on Innovations in Mobile Privacy and Security (IMPS@ESSoS 2016)*, pages 24–32, 2016.

[35] Dalvik executable format. https://source.android.com/devices/tech/dalvik/dex-format, 2020. Accessed: July 2020.

[36] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[37] Dex2Jar. https://github.com/pxb1988/dex2jar, 2020. Accessed: April 2017.

[38] Artem Dinaburg, Paul Royal, Monirul I Sharif, and Wenke Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS 2008)*, pages 51–62. ACM, 2008.

[39] Yue Duan, Mu Zhang, Abishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and Xiaofeng Wang. Things you may not know about Android (Un)Packers: A systematic study based on whole-system emulation. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS 2018)*. The Internet Society, 2018.

[40] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010)*, pages 393–407. USENIX Association, 2010.

[41] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*, pages 235–245. ACM, 2009.

[42] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)*, pages 627–638. ACM, 2011.

[43] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*, pages 576–587. ACM, 2014.

[44] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. TriggerScope: Towards detecting logic bombs in Android applications. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP 2016)*, pages 377–396. IEEE, 2016.

[45] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. SCanDroid: Automated security certification of Android applications. *Manuscript, University of Maryland*, 2009. https://www.cs.umd.edu/~avik/papers/scandroidascaa.pdf.

[46] Ariel Futoransky, Emiliano Kargieman, Carlos Sarraute, and Ariel Waissbein. Foundations and applications for secure triggers. *ACM Transactions on Information and System Security (TISSEC)*, 9(1):94–112, 2006.

[47] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 474–484. IEEE, 2009.

[48] Joshua Garcia, Mahmoud Hammad, Bahman Pedrood, Ali Bagheri-Khaligh, and Sam Malek. Obfuscation-resilient, efficient, and accurate detection and family identification of Android malware. Technical report, Department of Computer Science, George Mason University, 2015.

[49] Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. DyTa: dynamic symbolic execution guided with static verification results. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pages 992–994. ACM, 2011.

[50] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (TRUST 2012)*, volume 7344, pages 291–307. Springer, 2012.

[51] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, pages 213–223. ACM, 2005.

[52] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS 2008)*. The Internet Society, 2008.

[53] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: Scalable and accurate zero-day Android malware detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys 2012)*, pages 281–294. ACM, 2012.

[54] Ben Gruver. smali. https://github.com/JesusFreke/smali, 2020. Accessed: May 2020.

[55] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. Practical GUI testing of Android applications via model abstraction and refinement. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE 2019)*, pages 269–280. IEEE/ACM, 2019.

[56] GuardSqure. ProGuard. https://www.guardsquare.com/en/proguard, 2020. Accessed: July 2020.

[57] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security 2013)*, pages 49–64. USENIX Association, 2013.

[58] Yuyu He, Lei Zhang, Zhemin Yang, Yinzhi Cao, Keke Lian, Shuai Li, Wei Yang, Zhibo Zhang, Min Yang, Yuan Zhang, et al. TextExerciser: Feedback-driven text input exercising for Android applications. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP 2020)*, pages 1071–1087. IEEE, 2020.

[59] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. Efficiently, effectively detecting mobile app bugs with AppDoctor. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys 2014)*, pages 18:1–18:15. ACM, 2014.

[60] Wenjun Hu and Dawu Gu. AppSpear: Bytecode decrypting and DEX reassembling for packed Android malware. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID 2015)*, volume 9404, pages 359–381. Springer, 2015.

[61] Yajin Zhou Xuxian Jiang and Zhou Xuxian. Detecting passive content leaks and pollution in Android applications. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS 2013)*. The Internet Society, 2013.

[62] Roberto Jordaney, Kumar Sharad, Santanu K Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting concept drift in malware classification models. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security 2017)*, pages 625–642. USENIX Association, 2017.

[63] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM 2007)*, pages 46–53. ACM, 2007.

[64] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, pages 553–568. Springer, 2003.

[65] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, Junbum Shin, and SWRD Center. ScanDal: Static analyzer for detecting privacy leaks in Android applications. IEEE, 2012.

[66] Patrik Lantz and Anthony Desnos. DroidBox: An Android application sandbox for dynamic analysis. https://www.honeynet.org/projects/active/droidbox/, 2011. Accessed: June 2020.

[67] Legend. https://github.com/asLody/legend, 2016. Accessed: September 2017.

[68] Ondrej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[69] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE 2015)*, pages 280–291. IEEE, 2015.

[70] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. DroidRA: Taming reflection to support whole-program analysis of Android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*, pages 318–329. ACM, 2016.

[71] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. DroidBot: A lightweight UI-guided test input generator for Android. In *Proceedings of the 39th International Conference on Software Engineering Companion Volume (ICSE-C 2017)*, pages 23–26. IEEE, 2017.

[72] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state based binary fuzzing. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE 2017)*, pages 627–637. ACM, 2017.

[73] Ruikai Liu. Yet another hook framework for ART (YAHFA). https://github.com/rk700/YAHFA, 2017. Accessed: September 2017.

[74] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM SIGSAC Conference on Computer and Communications Security (CCS 2012)*, pages 229–240. ACM, 2012.

[75] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for Android apps. In *Proceedings of the 21st ACM SIGSOFT International Symposium on the Foundations of Software Engineeringg (FSE 2013)*, pages 224–234. ACM, 2013.

[76] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. Evodroid: Segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*, pages 599–609. ACM, 2014.

[77] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*, pages 94–105. ACM, 2016.

[78] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, pages 431–441. IEEE, 2007.

[79] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. Reducing combinatorics in GUI testing of Android applications. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE 2016)*, pages 559–570. IEEE/ACM, 2016.

[80] Mockito. https://site.mockito.org/, 2020. Accessed: June 2020.

[81] UI/Application exerciser monkey. https://developer.android.com/studio/test/monkey, 2020. Accessed: June 2020.

[82] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Automatically discovering, reporting and reproducing Android application crashes. In *Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST 2016)*, pages 33–44. IEEE, 2016.

[83] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (SP 2007)*, pages 231–245. IEEE, 2007.

[84] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430. IEEE, 2007.

[85] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting Android applications to Java bytecode. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2012)*, page 6. ACM, 2012.

[86] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in Android: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security 2013)*, pages 543–558. USENIX Association, 2013.

[87] Philip O'Kane, Sakir Sezer, and Kieran McLaughlin. Obfuscation: The hidden malware. *IEEE Security & Privacy*, 9(5):41–47, 2011.

[88] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. MaMaDroid: Detecting Android malware by building Markov chains of behavioral models (extended version). *ACM Transactions on Privacy and Security (TOPS)*, 22(2):14:1–14:34, 2019.

[89] Open GApps. https://opengapps.org/, 2020. Accessed: June 2020.

[90] Mila Parkour. Contagio mobile. http://contagiominidump.blogspot.ca/, 2018. Accessed: August 2015.

[91] Riyad Parvez, Paul AS Ward, and Vijay Ganesh. Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering (CASCON 2016)*, pages 116–127. IBM/ACM, 2016.

[92] Priyam Patel, Gokul Srinivasan, Sydur Rahaman, and Iulian Neamtiu. On the effectiveness of random testing for Android: or how I learned to stop worrying and love the monkey. In *Proceedings of the 13th International Workshop on Automation of Software Test (AST@ICSE 2018)*, pages 34–37. ACM, 2018.

[93] The Statista Portal. Android operating system share worldwide by OS version from 2013 to 2020. https://www.statista.com/statistics/271774/share-of-android-platforms-on-mobile-devices-with-android-os/, 2020. Accessed: July 2020.

[94] The Statista Portal. Number of available applications in the Google Play Store from December 2009 to June 2020. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store, 2020. Accessed: July 2020.

[95] The Statista Portal. Share of global smartphone shipments by operating system from 2014 to 2023. https://www.statista.com/statistics/272307/market-share-forecast-for-smartphone-operating-systems, 2020. Accessed: July 2020.

[96] David A Ramos and Dawson R Engler. Under-constrained symbolic execution: Correctness checking for real code. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security 2015)*, pages 49–64. USENIX Association, 2015.

[97] Bahman Rashidi and Carol Fung. Xdroid: An Android permission control using hidden Markov chain and online learning. In *Proceedings of the 2016 IEEE Conference on Communications and Network Security (CNS 2016)*, pages 46–54. IEEE, 2016.

[98] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in Android applications that feature anti-analysis techniques. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS 2016)*. The Internet Society, 2016.

[99] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. Making Malory behave maliciously: Targeted fuzzing of Android execution environments. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE 2017)*, pages 300–311. IEEE/ACM, 2017.

[100] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS 2017)*. The Internet Society, 2017.

[101] Martin C Rinard, Cristian Cadar, Daniel Dumitran, Daniel M Roy, Tudor Leu, and William S Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th USENIX Symposium on Operating System Design and Implementation (OSDI 2004)*, pages 303–316. USENIX Association, 2004.

[102] Robotium. https://code.google.com/p/robotium/, 2020. Accessed: July 2020.

[103] Kevin A Roundy, Paula Barmaimon Mendelberg, Nicola Dell, Damon McCoy, Daniel Nissani, Thomas Ristenpart, and Acar Tamersoy. The many kinds of creepware used for interpersonal attacks. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP 2020)*. IEEE, 2020.

[104] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC 2006)*, pages 289–300. IEEE, 2006.

[105] Aleieldin Salem, Michael Hesse, Jona Neumeier, and Alexander Pretschner. Towards empirically assessing behavior stimulation approaches for Android malware. In *Proceedings of the 13th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2019)*, pages 47–52. IARIA XPS Press, 2019.

[106] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005.

[107] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. Andromaly: A behavioral malware detection framework for Android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.

[108] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS 2008)*. The Internet Society, 2008.

[109] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. SMV-Hunter: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS 2014*. The Internet Society, 2014.

[110] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS 2016)*. The Internet Society, 2016.

[111] Tim Strazzere. android-unpacker. https://github.com/strazzere/android-unpacker, 2017. Accessed: September 2017.

[112] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*, pages 245–256. ACM, 2017.

[113] Mingshen Sun, Tao Wei, and John Lui. TaintART: A practical multi-level information-flow tracking system for Android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*, pages 331–342. ACM, 2016.

[114] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. CopperDroid: Automatic reconstruction of Android malware behaviors. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS 2015)*. The Internet Society, 2015.

[115] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*, pages 350–360. ACM, 2018.

[116] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 1999)*, page 13. IBM, 1999.

[117] VirusTotal. VirusTotal. https://www.virustotal.com, 2020. Accessed: June 2020.

[118] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (SP 2010)*, pages 497–512. IEEE, 2010.

[119] Xiaolei Wang, Yuexiang Yang, and Sencun Zhu. Automated hybrid analysis of Android malware through augmenting fuzzing with forced execution. *IEEE Transactions on Mobile Computing*, 18(12):2768–2782, 2018.

[120] Watson libraries for analysis. http://wala.sourceforge.net, 2020. Accessed: April 2017.

[121] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. JN-SAF: Precise and efficient NDK/JNI-aware inter-language static analysis framework for security vetting of Android applications with native code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS 2018)*, pages 1137–1150. ACM, 2018.

[122] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS 2014)*, pages 1329–1341. ACM, 2014.

[123] Michelle Y Wong and David Lie. IntelliDroid: A targeted input generator for the dynamic analysis of Android malware. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS 2016)*. The Internet Society, 2016.

[124] Michelle Y Wong and David Lie. Tackling runtime-based obfuscation in Android with TIRO. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security 2018)*, pages 1247–1262. USENIX Association, 2018.

[125] Michelle Y Wong and David Lie. Driving execution of target paths in Android applications with (a) CAR, 2020.

[126] Michelle Yan Yi Wong. Targeted dynamic analysis for Android malware. Master's thesis, University of Toronto, 2015.

[127] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time Android application auditing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP 2015)*, pages 899–914. IEEE, 2015.

[128] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3):16, 2007.

[129] Xposed framework hub. https://www.xda-developers.com/xposed-framework-hub/, 2020. Accessed: July 2020.

[130] Ke Xu, Yingjiu Li, Robert Deng, Kai Chen, and Jiayun Xu. DroidEvolver: Self-evolving Android malware detection system. In *Proceedings of the 2019 IEEE European Symposium on Security and Privacy (EuroS&P 2019)*, pages 47–62. IEEE, 2019.

[131] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. Adaptive unpacking of Android apps. In *Proceedings of the 39th International Conference on Software Engineering (ICSE 2017)*, pages 358–369. IEEE, 2017.

[132] Lok Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 2012)*, pages 569–584. USENIX Association, 2012.

[133] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in Android applications. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE 2015)*, pages 89–99. IEEE/ACM, 2015.

[134] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and Xiaoyang Sean Wang. AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS 2013)*, pages 1043–1054. ACM, 2013.

[135] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. Droidfuzzer: Fuzzing the Android apps with intent-filter tag. In *Proceedings of the 11th International Conference on Advances in Mobile Computing & Multimedia (MoMM 2013)*, page 68. ACM, 2013.

[136] Wei You, Bin Liang, Wenchang Shi, Peng Wang, and Xiangyu Zhang. TaintMan: An ART-compatible dynamic taint analysis framework on unmodified and non-rooted Android devices. *IEEE Transactions on Dependable and Secure Computing (DSC)*, 17(1):209–222, 2017.

[137] Michal Zalewski. AFL. https://lcamtuf.coredump.cx/afl/, 2020. Accessed: June 2020.

[138] Andy Zhang. ZHookLib. https://github.com/cmzy/ZHookLib, 2017. Accessed: September 2017.

[139] Yifei Zhang, Tian Tan, Yue Li, and Jingling Xue. Ripple: Reflection analysis for Android apps in incomplete information environments. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY 2017)*, pages 281–288. ACM, 2017.

[140] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. Vetting undesirable behaviors in Android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS 2013)*, pages 611–622. ACM, 2013.

[141] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. DexHunter: Toward extracting hidden code from packed Android applications. In *Proceedings of the 20th European Symposium on Research in Computer Security (ESORICS 2015)*, pages 293–311. Springer, 2015.

[142] Yury Zhauniarovich, Maqsood Ahmad, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. StaDynA: Addressing the problem of dynamic code updates in the security analysis of Android applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY 2015)*, pages 37–48. ACM, 2015.

[143] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. SmartDroid: an automatic system for revealing ui-based trigger conditions in Android applications. In *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM 2012)*, pages 93–104. ACM, 2012.

[144] Yajin Zhou and Xuxian Jiang. Dissecting Android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP 2012)*, pages 95–109. IEEE, 2012.

[145] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS 2012)*. The Internet Society, 2012.