TARGETED DYNAMIC ANALYSIS FOR ANDROID MALWARE

by

Michelle Wong

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Targeted Dynamic Analysis for Android Malware

Michelle Wong

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2015

The identification and analysis of Android malware involves either static or dynamic program analysis of the malware binary. While static analysis has good code coverage, it is not as precise due to the lack of run-time information. In contrast, because Android malware is often bundled with applications that have legitimate functionality, dynamic analysis can take a long time to find and analyze the small amount of code implementing the malicious functionality. We propose IntelliDroid, a tool that combines the advantages of both static and dynamic analyses to efficiently analyze suspicious behavior in Android applications. A lightweight static phase identifies possible malicious behavior and gathers information to generate inputs that can dynamically exercise that behavior. IntelliDroid overcomes several key challenges of analyzing Android malware and when evaluated on 30 instances of malicious behavior, IntelliDroid successfully identifies the behavior, extracts path constraints, and executes the malicious code in all but one case.

Acknowledgements

I would like to give my heartfelt thanks to my supervisor, Professor David Lie, for his support and guidance throughout the course of study for my Master's degree. I greatly appreciate the time and care he spends on each student, and I am thankful for the amount of help he provided as I completed my project and wrote my thesis.

I would also like to thank James Huang and Zheng Wei for their help while completing this project. Their ideas and feedback on my research were invaluable.

I am also grateful for the financial support provided by the University of Toronto and the NSERC CGS-M award for my research.

Lastly, I would like to thank my family and friends for supporting me as I complete my degree. I would especially like to thank Yan for being there for me throughout my course of study.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Smartphone malware is and will continue to be a major security threat. One of the most attractive features of a smartphone is the ability to extend its functionality with third-party applications. Unfortunately, such a feature inevitably brings with it the threat of malicious smartphone applications, otherwise known as *smartphone malware*. With a reported 1 billion smartphones shipped in 2013 [18], this large population of potential victims gives malware writers ample motivation to target smartphone devices, as indicated by a recent report by Sophos, which states that the number of new smartphone malware samples detected has doubled from 1000 per day in 2013 to 2000 per day in 2014 [31]. Based on these figures, it is clear that mitigating the threat of smartphone malware is an important problem for the security community. Previous approaches in the literature generally employ program analysis in the form of static analysis [36], dynamic analysis [17, 35], or a combination [39] to detect and analyze Android malware.

Dynamic analysis is inherently incomplete as it can only provide information about code that it executes. As a result, to correctly assess whether an application can exhibit malicious behavior, dynamic analysis needs to execute a fairly significant portion of the paths through the program. While good code coverage and automatic test-case generation can be achieved with techniques like concolic testing [22], these techniques are expensive since they rely on symbolic execution, which can have as much as 70x overhead [13, 23], to extract path constraints.

In contrast, static analysis can efficiently analyze all the code in the application but it is inherently imprecise, meaning that it may miss malicious behavior (false negatives) or falsely detect malicious behavior (false positives). The loss of precision is a cost of an analysis that can scale to an entire program, which often requires the analysis to make trade-offs that reduce precision in terms of context-sensitivity,

1

flow-sensitivity or pointer disambiguation. Such imprecise analyses can make it difficult to accurately disambiguate malicious applications from benign ones as they may have similar behaviors superficially. For example, the *NickyBot* Android malware monitors and intercepts incoming SMS messages from the malware author. Similarly, various legitimate applications such as the *Go SMS Pro*[1], which has over 50 million downloads and 1 million reviews on the Google Play store, permit the user to automate SMS tasks such as automatically responding to and deleting incoming SMS's. Dynamic analyses are generally more precise than static analyses and can provide more information, but dynamic analyses need specific inputs to cause the application to execute the suspicious code.

We propose *IntelliDroid*, which combines static and dynamic analysis to generate the specific inputs needed to trigger suspicious code in applications and produce a dynamic execution trace of that behavior. We observe that there is little benefit to performing expensive dynamic analysis on sections of an application that do not have any possible malicious behavior. Since Android malware is often embedded within a benign application, we should avoid dynamic execution of as much of the application as possible, since the majority of code is benign. Thus, IntelliDroid first uses a simple static analysis that aggressively identifies potentially malicious behavior using a user-specified set of "suspicious" Android API call patterns. For example, such a pattern could specify any code that sends SMS messages to hard-coded addresses, or that uses `abortBroadcast` to intercept and hide notifications for incoming SMS messages. These inputs are then injected into an Android system in a dynamic phase that enables a user to accurately observe the behavior of the application. Thus, instead of performing an expensive dynamic analysis of the entire application, IntelliDroid only performs a statically-guided, "targeted dynamic analysis" on the parts of an application that have been identified as suspicious by a much cheaper static analysis phase.

While the use of cheaper static analysis to inform and guide a more expensive and precise dynamic analysis has been done to analyze Android content providers [39], IntelliDroid is the first instance that we are aware of that applies the technique to Android malware. The analysis performed by [39] is considerably simpler than IntelliDroid as it is restricted to analyzing code within content providers. Dealing with Android malware requires IntelliDroid to handle a broad range of behavior such as inter-event handler execution flow, inter-procedural communication, multiple interactions with the Android OS and dependencies on external network components.

To handle this broad range of behaviors, IntelliDroid overcomes several key challenges. First, Android applications do not have a single entry-point, but are instead composed of a collection of event handlers, which the Android system permits to be called in various loosely-defined orders. However, the path

---

[1] https://play.google.com/store/apps/details?id=com.jb.gosms

to a suspicious API can be dependent on other event handlers, making the path impossible unless a "chain" of event handlers is executed before it. For example, the execution of a suspicious API might be conditionally dependent on a heap variable that is set to a certain state by another event handler, which must be executed first. IntelliDroid iteratively detects such *event-chains* and computes the appropriate inputs to inject, as well as the order to inject them. This problem is unique to event-handler code – regular code that only has a single-entry point cannot, by definition, have reachable paths that depend on others paths that cannot be reached from the single entry-point.

Second, while previous work injects inputs at the application boundary [29, 32, 36, 39], this 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 message that was just received. Simply injecting the SMS notification at the application boundary will result in inconsistent behavior because the application has received the input and expects the message to be in the SMS content provider database, but the Android framework itself has received no such message. IntelliDroid solves this problem by injecting inputs at the lower-level *device-framework interface*, allowing all state in the Android framework to be automatically changed consistently. However, the relationship between how application event handlers will be triggered in response to inputs injected at the device-hardware interface is not specified, so IntelliDroid extracts this relationship using static analysis on the Android source code.

Finally, unlike tools that try to detect vulnerabilities in benign applications, IntelliDroid must handle instances where the behavior of malware is determined by communication with malicious components that are outside of the application. For example, some malware will connect to network servers to receive instructions and modify their behavior accordingly. Constraints that depend on the behavior of these external components cannot be solved statically. As a result IntelliDroid dynamically extracts the necessary dependencies at run-time and dynamically solves the constraints necessary to compute the appropriate inputs in response to the external component behavior.

1.1   Contributions

This thesis makes three main contributions to the field of Android malware analysis:

1. We present the design and implementation of IntelliDroid, which combines static and dynamic analysis to provide a precise and scalable analysis tool.

2. We describe three novel techniques that enable IntelliDroid to accurately generate and inject artificial inputs into potentially malicious applications. Detecting event-chains enable IntelliDroid to exercise event paths that depend on other events. Device-framework interface input injection allows IntelliDroid to precisely generate external events that will consistently trigger specific code in an Android application. Finally, dynamic extraction and solving of constraints at run-time allows IntelliDroid to correctly generate inputs that match constraints derived from network control servers that malware communicates with.

3. We evaluate and demonstrate the ability of IntelliDroid to successfully analyze and generate input sequences for an array of malicious applications. We also show that IntelliDroid is scalable, requiring only 22.7 seconds of analysis time on average to successfully generate inputs to trigger malicious behavior in malware.

1.2   Thesis Structure

We begin by giving relevant background on the Android platform and static analysis in Section 2. Then, we describe the design of IntelliDroid in Section 3, including the steps performed in each stage of the analysis. Details about the implementation are given in Section 4. Evaluation results showing the effectiveness and performance of IntelliDroid are presented in Section 5. Section 6 goes over the limitations of IntelliDroid, while related works are discussed in Section 7. We conclude in Section 8 and discuss possible future work for IntelliDroid.

Chapter 2

Background

2.1   Android System

Android is a popular open-sourced operating system for mobile devices and is currently executed on devices from a number of different manufacturers. It contains a custom Linux kernel and an application-framework implemented in Java that interfaces with third-party applications.

2.1.1   Android Applications and Framework

Android applications are event-driven and communicate with the device via the Android framework API. When a user installs an application, they must first accept the permissions that the application requests. These permissions can include access to sensitive user data or the ability to send data to external parties. Because these permissions are granted at the time of installation, the user knows that the application uses the requested resources but they may not know how these resources are used. Although application descriptions provides an overview of its functionality, it is difficult for users to verify that the application performs the actions claimed and that it does not perform any malicious activity in the background.

Once an application has been installed, it is executed in its own Linux process, providing security and isolation from other applications. When an application is launched by the user, execution begins in the *Android framework*, which loads the application components and manages their lifecycles. This Android framework serves as the middle layer within the Android system, located between the kernel and the application. It processes events from the kernel and requests from the applications, providing each with its own interface (Figure 2.1).

From the application's perspective, the transfer of execution from the framework into application
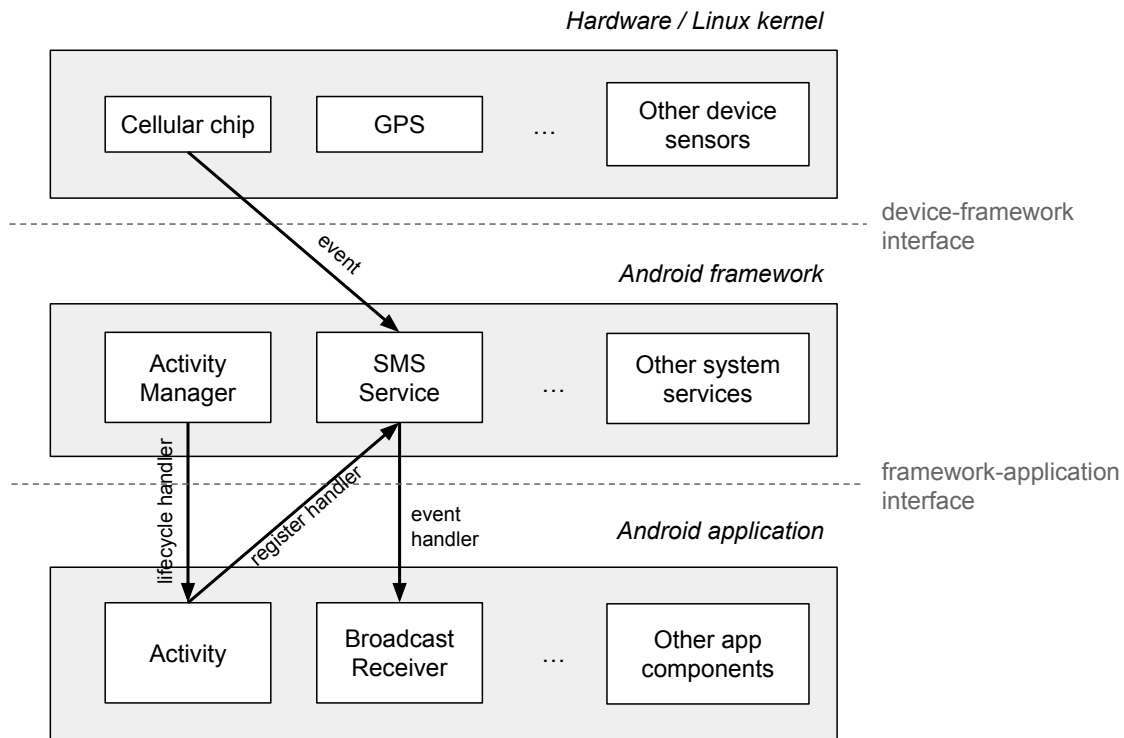
Figure 2.1: Event Propagation and Notification in Android

code occurs at specific *entry-points* that are recognized by the framework. These entry-point methods form the *framework-application interface* that divides the Android framework from code that originates from third-party applications, as shown in Figure 2.1. Execution flows between the framework and the application code frequently via these entry-points and unlike traditional Java applications, there is no `main()` method where application execution must begin. Some of the entry-point methods are automatically created when applications instantiate basic Android components such as Activities, Services, Content Providers, or Broadcast Receivers. For each component, callback methods can be overridden by the application to perform custom operations that execute when application state changes. For instance, the framework will invoke the `Activity.onCreate` method when the activity object is first created, and the `Activity.onResume` method when the user interface for the activity is placed in the foreground. Applications can override these methods to load a particular UI element or start a background process as the application is loaded.

Other application entry-points include event handlers that respond to events on external sensors, such as location change events from the GPS sensor or new SMS events from the cellular chip. Such entry-points are often registered with the framework, which will then invoke them when the associated external event occurs. The registration and callback mechanism can vary for different services. For some services,

such as SMS, events notifications are delivered by the broadcast intent system in Android. Applications must register a `BroadcastReceiver` component either explicitly within the code or by declaring the appropriate intent filters in the application's manifest. These filters specify the precise events for which the receiver should be invoked. Other services may use a special interface; for instance, the location service on Android requires that an application explicitly register a `LocationListener` object containing methods such as `onLocationChange()`; these methods are overloaded with application-specific code.

Overall, an application can consist of multiple components and they belong to one of four types, depending on the framework class that they extend. The different types of Android components are described below.

Activity : An activity corresponds to a particular user-interface (UI) layout of the application. It should contain callback methods for UI actions and load new UI elements as needed. For most applications, the main component that is started when the application is loaded is an activity. An application can contain multiple activities if it requires different layouts. The lifecycle of an activity is managed by methods such as `onCreate`, `onStart`, or `onStop`.

Service : A service allows the application to execute long-running background tasks in a separate process. A service can continue to execute even when the application is closed, allowing the application to monitor and process new events as needed. Services also provide lifecycle methods that applications can override, including `onCreate` and `onStart`. For applications that do not contain a user interface, the main component is a service that runs continuously in the background.

Content Provider : A content provider serves as a gateway for a database managed by the application. It contains methods that allows other components (or applications) to retrieve and/or modify information in the database.

Broadcast Receiver : When certain events occur, a message is broadcasted to all interested applications. A broadcast receiver allows the application to receive these events, which can originate from the Android framework or from other applications. The `onReceive` method within the `BroadcastReceiver` subclass serves as an entry-point into the application. The types of events received depends on how the application has registered the receiver within the framework.

Applications can declare other classes to be loaded by these four main component types. For instance, the `LocationListener` event handler class is often implemented as an inner class within a main application component. Thus, entry-points can be found among all application classes.

Within the Android framework that runs the application, a number system services communicate with the device's hardware components, and receive and respond to application requests. These system services are executed in separate processes and handle the events and data processing required for all applications on the device. When a device sensor receives new data, it notifies the Android framework so that the event data can be processed and disseminated to the applications. The point at which the framework first receives the event is normally located in a system service and the method by which the event is received can vary from service to service. In general, the event notification can be implemented as an inter-process message, a remote procedure call, or a socket message for a monitoring thread. We refer to these locations, where the device sensors notify the framework of a new event, as the *device-framework interface* (shown in Figure 2.1). Upon receipt of an event at this interface, the framework is responsible for notifying the currently-running applications by invoking the callbacks registered for the event. While each system service implements this logic differently, all of the services require that the registered callbacks be stored somewhere within the service. In addition, any parameters affecting how the callback is to be called (e.g. the frequency) is also stored and referenced when the framework determines whether the callback method should be invoked.

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 from the framework at a later time. Some services, such as SMS, store all past event information in a content provider that can be queried and modified by applications with sufficient permissions. Other services, such as location, may store only the last event received. In both cases, the handler invocation in the application and the event information stored in the framework services must be kept consistent for correct execution. That is, when a new event is processed, merely invoking the entry-points at the framework-application interface can cause inconsistency between the application state and the framework state. To resolve this, the framework must also have a record of the event.

2.1.2 Inter-Process Communication

For both the framework and application code, execution flow within a single process is no different than that in traditional desktop applications, given the appropriate entry-points. However, interprocess communication (IPC) is required for different application components to communicate and for a framework service, which runs in its own process, to process an application request. This is especially important for Android applications since the main UI thread must be available and ready to receive UI callbacks in order to avoid a laggy user experience. Android provides an IPC mechanism using the `Handler`
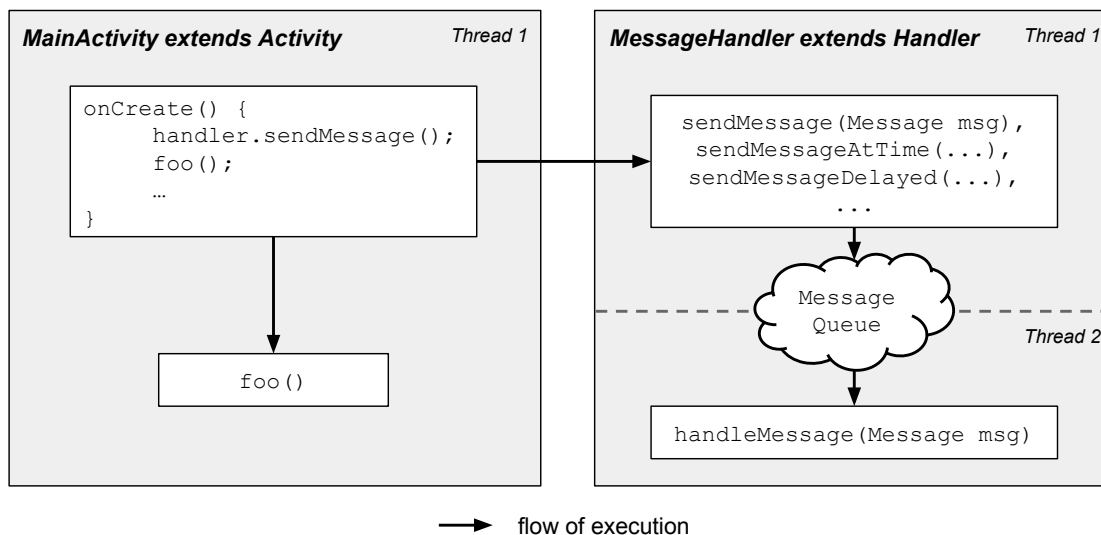
Figure 2.2: Android IPC Mechanism

class, which allows processes to send messages to each other by invoking `Handler.sendMessage` (or its derivatives), as shown in Figure 2.2. Android facilitates the actual message sending and allows these messages to be received within the `Handler.handleMessage` method, which must be overridden in each handler object to process the messages it receives. For classes that are commonly referenced by other processes (e.g. system services), the remote procedure call (RPC) mechanism is used to allow code running in different processes to invoke methods across process boundaries (using IPC as the underlying communication channel).

For application components, the `Intent` framework allows them to trigger one another with specific parameters, without having to implement any message handling code. An intent is an object that contains information about the target component (either the name or a flag that the target has set) and optional data/messages to be sent to the receiver. If the target component has not yet been started, it is loaded by the framework. The target component can be within the same application as the invoking component, or it can belong to another application. For communication between applications, a special flag must be set in the application's manifest to allow a component to be externally invokable. If an intent is explicit, the intent target is identified by its fully-qualified class name and there is no ambiguity regarding the object that receives the intent. An intent can also be implicit, where the intent object specifies a certain action that the target should perform. When the framework processes an implicit intent, it sends the intent to a component that has registered the intent action within its application's manifest. If multiple components have registered for the intent action, the user must decide which component should receive the intent.

In cases where asynchronous execution is needed on a separate thread, Android provides the `AsyncTask` class. Implementors of this class can override methods such as `doInBackground` or `onPostExecute`, which are called by the framework in a predetermined order. Normal Java `Thread` and `Runnable` classes are also supported for asynchronous execution.

2.2 Static Analysis

Static analysis is a commonly used tool in malware detection [20, 21, 26]. For Java applications, static analysis works directly on the bytecode and can perform various analyses such as reconstruct the class hierarchy, find method invocations, and extract control- and data-flow information. In general, static analysis can be divided into intra-procedural analysis within a method, and inter-procedural analysis across method invocations.

2.2.1 Intra-Procedural Analysis

Intra-procedural analysis refers to analysis performed on a method in isolation from other methods in the application. The method body is represented by instructions based on a specific intermediate representation (IR) used by the analysis tool. Instructions can be grouped into basic blocks, which each represent the largest straight-line segment of code that has exactly one entry point and one exit point. The flow of execution between basic blocks is captured by the control flow graph (CFG), which models the relationship between basic blocks in the method. Using the CFG, control- and data-flow analysis can determine how execution flows through the method and how values are passed between variables. In addition, the CFG can also be used to determine the locations (i.e. conditional branch statements) where the execution flow may diverge depending on a variable value. If a certain execution path is desired, the values for these variables must be constrained, and these constraints can be extracted from the branch instructions.

2.2.2 Inter-Procedural Analysis

Inter-procedural analysis refers to analysis performed on the entire application while considering the control- and data-flow relationships between different methods. An integral component is the application's call graph, which models how methods are invoked by each other. Within a call graph, nodes represent methods while edges represent flows of execution. In most cases, these edges represent method invocations; however, a call graph can be customized to contain edges that model non-traditional control flow, such as asynchronous calls or message handling.

Code:

```
onCreate() {
     foo(1,2);
     foo(3,4);
}

bar() {
     foo(3,4);
}

foo(x, y) {
     …
}
```

Context Insensitive

| onCreate() | bar() |

| foo() |

1-Level Caller Context Sensitive

| onCreate() | bar() |

| foo()
Caller: onCreate | foo()
Caller: bar |

Call-String Context Sensitive

| onCreate() | bar() |

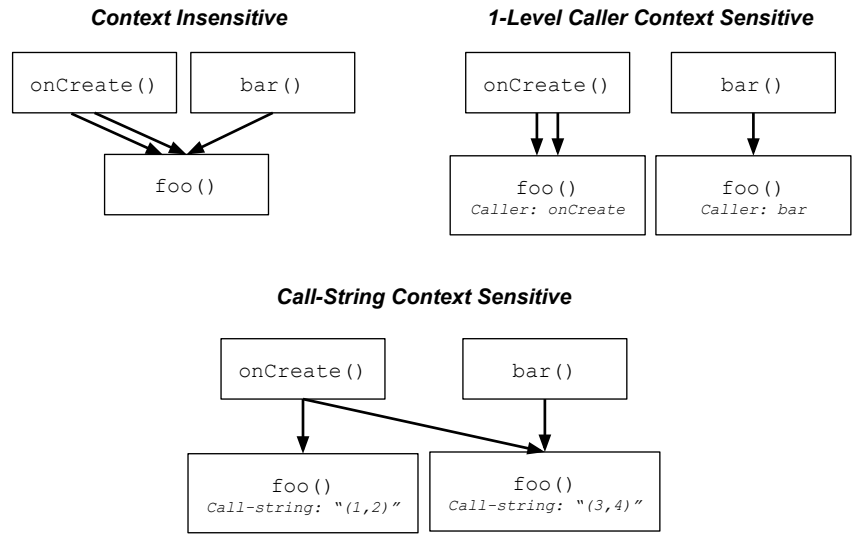| foo()
Call-string: "(1,2)" | foo()
Call-string: "(3,4)" |

Figure 2.3: Effect of Context Sensitivity on Call Graph Generation

When constructing a call graph, the precision of the graph and performance of the analysis can vary greatly depending on the options used. An important option to consider is the context sensitivity of the graph, which determines the representation of methods when they are invoked in multiple locations (i.e., *contexts*). As shown in Figure 2.3, a context insensitive call graph represents each method with only one graph node, which is inexpensive but can cause imprecision when methods are called in different places with different parameters. A context sensitive call graph, on the other hand, may contain several nodes for the same method. These nodes can be differentiated by the invoking method, the receiving class/object for the invoked method, and/or the parameters used in the invocation. Context can be stacked across method invocations; for instance, a call graph with 1 level of caller context sensitivity indicates that a method is cloned into different nodes for each distinct caller, while 2 levels of context sensitivity indicates that method clones are also affected by the caller's caller method. Context sensitivity can also vary across the graph depending on the methods themselves; for instance, a 0-1 context sensitivity indicates that some method invocations are processed with no context while others are processed with one level of context. Increasing context sensitivity results in a greater number of method clones among the graph nodes and increases precision when analyzing the call graph; however, it can become exponentially expensive to generate and process the graph. In most cases, static analysis must strike a balance between call graph precision and the resources (time and memory) required.

While the call graph provides control-flow information about the application, inter-procedural data flow analysis relies on pointer analysis, which models the heap and the objects that pointers are referenc-

Code:

```
onCreate() {
    1 Object x = new CustomObject();
    2 Object y = new CustomObject();
}
```
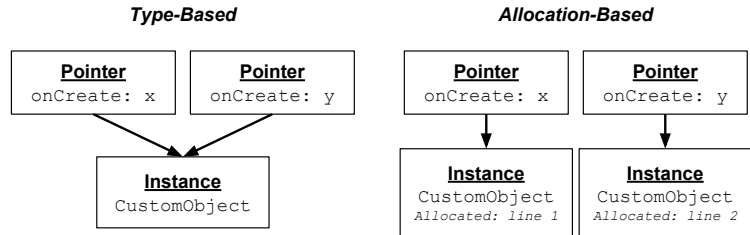

Figure 2.4: Effect of the Heap Model on the Pointer Analysis Graph

ing (Figure 2.4). An example of a simple heap model might assume that each class is only instantiated once, so the references to the same syntactic name point to the same object. However, this can cause significant pointer aliasing for commonly used class types. More precise heap models may differentiate between the allocation sites that instantiate the objects, but this can be expensive. We note that for many Android-specific classes, such as Activities or Services, applications often only instantiate one object for each class so a simple heap model can work in these cases. As with call graph generation, a trade-off between precision and efficiency is required.

2.3   Symbolic Execution

Symbolic execution is a form of dynamic analysis that aims to identify the inputs to an application that result in a particular segment to execute. For many cases, such as in concolic testing, it is used to improve code coverage when testing or analyzing an application. In traditional dynamic analysis, greater precision than static analysis can be acheived due to the run-time information available; however, dynamic analysis can be incomplete since only the code actually executed is analyzed. Code meant for special cases or to be executed under certain conditions may not be triggered during the testing/analysis runs. Concolic testing and symbolic execution can be used to avoid this issue by determining the inputs required for all parts of the application and injecting these inputs to explore the code fully.

To execute an application symbolically, a custom run-time platform is needed to capture the symbolic information of each variable. As data is propagated between variables, the custom platform records information regarding how the variable value was derived. Therefore, rather than storing a concrete value for each varaible, an expression is created that can reconstruct the variable value from the input parameters. When a conditional branch statement is encountered, the symbolic expression of the predicate variables and the conditional expression extracted from the branch instruction can be transformed into a constraint that governs the outcome of the branch. By manipulating the program inputs such that

```
main(int a, int b) {
        int c = a * b;  ⟶   c: a * b
        int d = b + 50; ⟶   d: b + 50

        if (c < d) {
                ...  ⟶————————⟶   (a * b < b + 50)
        } else {
                ...  ⟶————————⟶   !(a * b < b + 50)
        }
}
```

Figure 2.5: Example of Extracted Symbolic Information and Constraints

the constraints are satisfied as necessary, each branch can be explored fully.

The constraints extracted for an execution path model the control flow taken for that path. They are generated for places where the control flow could have diverged due to the value of a variable and represent the required values allowing that specific execution path to take place. Thus, these constraints are derived from predicates in conditional branch statements (where control flow can diverage) and augmented with the propagated symbolic information for the predicate variables (to express the constraint with respect to the input parameters injected). For instance, Figure 2.5 shows a conditional branch instruction whose predicate depends on the results of a series of operations. If the `if` branch was taken and ran during a concolic testing execution, the symbolic information and constraints extracted for this path would represent the control flow dependencies as $(a * b < b + 50)$, where a and b are the input parameters injected. If the branch outcome was toggled in a subsequent run such that the `else` branch was taken, the constraint representing the flow would be the inverse expression $!(a * b < b + 50)$. By solving a and b for either expression, the outcome of the branch (and thus, the execution path) can be controlled via input injection. For more complex functions and operations, the model used to obtain the extracted constraints may vary from system to system.

Recording symbolic information dynamically is a costly process and this form of analysis imposes a large performance overhead. Gathering the symbolic information is timely and storing this information takes up signficant amounts of memory. In addition, exhaustively exploring all branches in the application and solving each constraint can take a long time. Previous work in symbolic execution have tried to mitigate these drawbacks by reducing the amount of exploration required [6, 11] and by simplifying the constraints prior to solving them [10, 11, 22, 30]. Nevertheless, symbolic execution is still a costly process and is much more expensive than regular execution of the application. Furthermore, inefficiency can also be an issue since concolic testing tools use symbolic execution to achieve complete code coverage of the application. When only certain parts of the code is of interest (such as code that performs malicious

activity), such tools would not target those specific paths. Thus, concolic testing and symbolic execution can be inefficient for targeted or selective execution of the application.

Chapter 3

Design of IntelliDroid

To use IntelliDroid, the user specifies a set of suspicious behaviors and an application for IntelliDroid to analyze. IntelliDroid then statically checks if the application can exhibit these behaviors and if so, computes and injects the appropriate inputs to execute the suspicious behaviors. Suspicious behaviors are specified in the form of an Android API method and a set of constraints on the arguments that can be used to call the API. For example, a specification of suspicious behavior might be a call to the `sendTextMessage` API with a destination that is based on a hard-coded constant. Because the user has to provide the specification, it is assumed that they have some idea of the types of malicious activity they wish to detect. On Android, suspicious behavior normally involves a framework method invocation, since the framework abstracts the device resources for the application; thus, the specification of suspicious behavior is a relatively simple process. Given such a specification, IntelliDroid will perform the following tasks on the application to be analyzed:

1. Identify instances of API invocations that match the specification of suspicious behavior. For each instance, identify the *event handlers* where execution is passed to application code and find *call paths* from the handlers that lead to the suspicious behavior.

2. For each suspicious behavior, extract path constraints that control whether the suspicious behavior is invoked.

3. In cases where the constraints depend on execution paths in other event handlers, extract the necessary constraints and the order of the *event-chain* that is required to invoke the suspicious behavior.

4. Using an off-the-shelf constraint solver, solve the path constraints to determine the necessary the
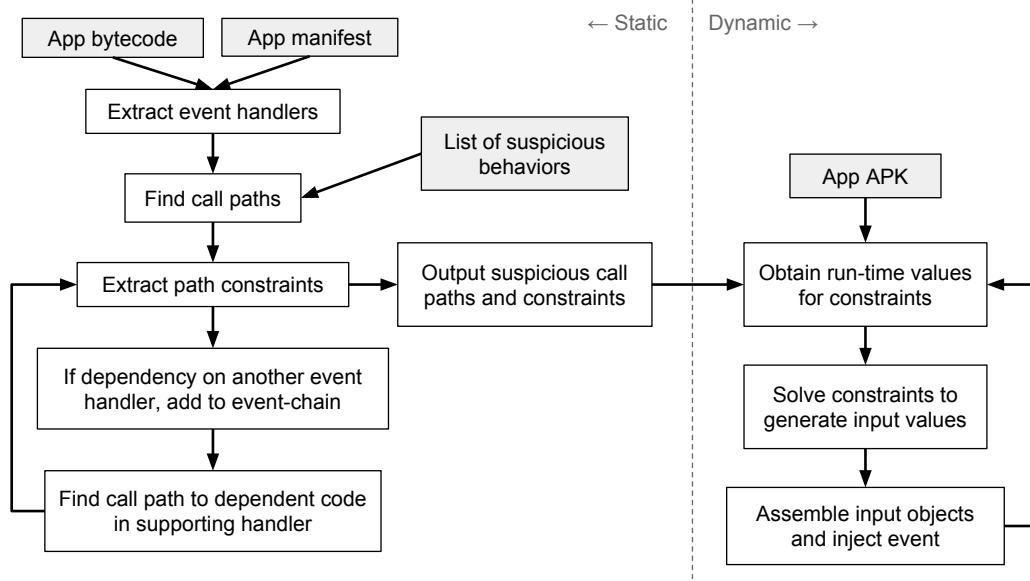
Figure 3.1: IntelliDroid System Diagram

input values that will lead to invocation of the suspicious behavior. Some inputs may depend on external values, such as responses to network requests, so IntelliDroid dynamically extracts the concrete values for these external dependencies during execution so that inputs matching these external dependencies can be generated.

5. Apply the computed input values in the appropriate order to the *device-framework interface* to consistently execute the appropriate paths in the event handlers that are required to cause the suspicious behaviors to execute. IntelliDroid contains a modified Android OS that can extract the required information at run-time and inject inputs at the device-framework boundary.

The flow between these tasks in the IntelliDroid system is shown in Figure 3.1, which also differentiates between the static and dynamic components. Our current prototype uses the WALA [34] framework for static analysis and Z3 [14] as a constraint solver.

By performing a targeted execution of the suspicious behavior, IntelliDroid enables the user to perform dynamic analysis (such as taint analysis), which can help the user determine if the application is malicious, as well as perform reverse engineering on malware to better understand its design and intent. Since IntelliDroid is built on top of a standard Android OS, we believe it is general enough to support any type of dynamic analysis. As a result, we focus on how IntelliDroid generates the necessary inputs and executions for such dynamic analyses, but leave the actual integration of such analyses with IntelliDroid's instrumented Android OS for future work.

3.1   Identifying Paths to Suspicious Behaviors

For a given application and specification of suspicious behavior, IntelliDroid first performs static analysis to identify the locations of suspicious behaviors and the paths leading to those behaviors. 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 life cycle, processing sensor inputs, or responding to UI events. IntelliDroid identifies these event handler entry-points as methods that meet the following requirements: i) the class that declares the method implements a framework interface or extends a framework class, and the method itself must override a parent method in this interface or class; ii) the method is not called within the application code; and iii) the declaring class is instantiated at least once in the application code. We note that this is essentially the same technique used by CHEX to identify Android event handlers [26]. For condition (ii), it is possible that a malicious application may try to avoid detection of certain event handlers by faking event invocations — i.e., invoking the event handler method so that it is called within the application code and therefore, not classified as an entry-point; however, we did not encounter this issue in any malicious applications during the evaluation. In addition, because the event handler would then be reachable from another call path within the application, it would still be part of the application's call graph and would be analyzed in the subsequent steps. Although this entry-point heuristic is used by IntelliDroid, event handler methods can also be manually specified by analyzing the Android framework specifications (via manual effort), and the inclusion of the heuristic does not detrimentally affect IntelliDroid's design.

A call graph is then generated using the event handlers as starting points for the code traversal. However, Android mechanisms such as Intents and asynchronous calls can cause execution to flow between methods in an application even when there is no explicit function call. For instance, Android allows execution to be transferred between different components of an application using the `Intent` interface, so the points at which Android intents are sent and received must be identified and the execution flow between then should be represented by additional edges in the call graph. Details regarding other Android-specific call edges can be found in Section 4.1.

A traversal for each event handler is then performed to identify the event handlers that may lead to a suspicious API invocation as defined by the user-specified suspicious behaviors. For each suspicious behavior that is found, a *suspicious call path* is extracted, which contains the sequence of method invocations from the event handler entry-point to the invocation of the suspicious API.

To illustrate the design of IntelliDroid, we use the code provided in Listing 3.1 as an example through-

Listing 3.1: Code Example.

```java
class SmsReceiver extends BroadcastReceiver {
  String sNum;

  public void onReceive(Context c, Intent i) {
    if (i.getAction() == "SMS_RECEIVED") {
      handleSms(i);
    } else if (i.getAction() == "BOOT_COMPLETED") {
      this.sNum = "99765";
    }
  }

  private void handleSms(Intent i) {
    Bundle b = i.getExtras();
    Object[] pdus = (Object[])b.get("pdus");

    for (int x == 0; x < pdus.length; x++) {
      SmsMessage msg = SmsMessage.createFromPdu((byte[])pdus[x]);
      String addr = msg.getOriginatingAddress();
      String body = msg.getMessageBody();
      // Constraint depends on local function
      if (needsReply(addr, body)) {
        SmsManager sm = SmsManager.getDefault();
        sm.sendTextMessage(addr, null, "YES", null, null);
      }
      // Constraint depends on heap variable
      if (addr.equals(this.sNum)) {
        abortBroadcast();
      }
    }
  }

  private boolean needsReply(String addr, String body) {
    if ((addr.startsWith("10658") && body.contains("RESPOND")) ||
        (addr.startsWith("10086") && body.contains("REPLY"))) {
      return true;
    }
    return false;
  }
}
```

out 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. 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 23 and 27 as suspicious behaviors. IntelliDroid then identifies the suspicious call path from the event handler to each of the suspicious behaviors, which are the paths through the method invocations on lines 4→6→23 and 4→6→27.

3.2 Extracting Call Path Constraints

While the presence of a suspicious call path can indicate that the application exhibits suspicious behavior, the existence of such a path alone does not mean that the path can actually be executed at run-time. To determine if the path is feasible, IntelliDroid generates the appropriate inputs that will cause the path to be executed. The first step in determining the appropriate inputs is to extract path constraints that must be satisfied for the suspicious behavior to occur.

For each method in the suspicious call path, the invocation of the next method in the path may be control-dependent on conditional branches in 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. The control flow analysis determines whether a conditional branch affects execution of the next method's invocation, and if so, it extracts constraints based on the variables used in the predicate of the branch statement. IntelliDroid also extracts symbolic data-flow information about variables to identify other variables they may depend on, or that may depend on it. Similar to traditional data-flow analyses, loop support is implemented by performing the analysis iteratively until the constraint and data-flow output converges.

The forward control-flow and data-flow analysis is obtained by propagating variable information along the method's CFG, both locally within each basic block and across different basic blocks. This is similar to program slicing, where data- and control-flow dependencies for a particular instruction are found, and information is propagated along this flow (in this case, symbolic information and control-flow constraints). The rules used to translate various Android instruction classes into data-flow and control-flow constraints are given in Table 3.1. The instructions listed in the table are given in static single assignment (SSA) form, as this is the intermediate representation used in WALA [34]. If multiple CFG paths are found to lead to the next method invocation, the analysis combines the extracted constraints of

each path with a logical OR ($\vee$), indicating that as long as one path is satisfied, the suspicious behavior is executable. After all of the methods in the suspicious call path are analyzed, the extracted constraints for each method are combined using full context-sensitivity, which is acheivable since the call site of each method along the path is known.

As an example, consider the execution path in Listing 3.1 ending in the `abortBroadcast` invocation at Line 27. The suspicious call path generated by IntelliDroid includes an invocation of `handleSms` in the `onReceive` method. This invocation is dependent on the intent action string; therefore, the constraint (`i.getAction() = "SMS_RECEIVED"`) would be extracted. IntelliDroid's context sensitive interprocedural analysis would indicate that when `handleSms` is called, the parameter with which it is invoked originated from the event handler's input parameter (`Intent i`). Control flow and data flow analysis through `handleSms` shows that the execution of the suspicious method call is dependent on the length of the PDU array and the originating address of the SMS message received. The conditional statement in line 21 is on the execution path, but since the suspicious method invocation can be reached regardless of the branch outcome, it has 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 suspicious call path, their return values affect the execution of the path and the constraints they impose must be extracted. An example of such an *auxilliary method* is `needsReply` in Listing 3.1. To handle such cases, IntelliDroid extracts constraints on the return values and the paths leading to the return sites within the auxilliary method. In `needsReply`, the two return sites at lines 35 and 37 result in the values `true` and `false` being returned, respectively. For each return site, the paths from the method entry to the site are analyzed and their constraints extracted. For instance, one possible path to line 35 would contain the constraints (`body = "RESPOND"` $\wedge$ `addr = "10658"`), while the other path would contain (`body = "REPLY"` $\wedge$ `addr = "10086"`). The constraints for each path are combined with a logical OR to create one cummulative path constraint for each return site. Finally, constraints on the return value are AND-ed to the path constraints for each return site and the constraints for the auxiliary function can then be AND-ed with the path constraints of the main suspicious call path. In `needsReply`, the constraint for the paths to line 35 would be combined with (`needsReply<return> = true`), using a logical AND ($\wedge$). Likewise, the constraints for the paths to line 37 are AND-ed with (`needsReply<return> = false`). The return-specific constraints are combined and inserted into the constraints for the caller method. When the caller method uses the return value, the constraint solver will automatically choose the set of constraints for the return path that produces the desired value. For Listing 3.1, the constraint at line 21 for the path to `sendTextMessage` states that

Table 3.1: Rules for Data-Flow and Control-Flow Constraint Extraction

Instruction	Context	Variables	Constraints
branch	taken	op = relational operator $p1, p2$ = parameters	$p1\ op\ p2$
	not taken	op = relational operator $p1, p2$ = parameters	$\neg\ (p1\ op\ p2)$
invoke	String.equals(); String.matches()	d = defined variable r = receiver p = parameter	$[(r = p \wedge retval = true) \vee (r \neq p \wedge retval = false)] \wedge (d = retval)$
	Object.set()	d = defined variable r = receiver p = parameter	$d = p$
	Message.obtain()	d = defined variable m = message f = field (parameter)	$d = value[m.field]$
	$d = r.method(p1, ...)$	d = defined variable r = receiver pX = parameters	$constraints[r.method] \wedge d = "r.method[return]"$
get	-	d = defined variable r = receiver $field$ = field	$d = value[r.field]$
put	-	r = receiver $field$ = field v = value	$pointer[r.field] = v$
binary op	-	op = operator d = defined variable $p1, p2$ = parameters	$d = p1\ op\ p2$
phi	-	d = defined variable $p1, p2$ = parameters	$(d = p1) \vee (d = p2)$
cast	-	d = defined variable p = parameter	$d = p$
convert	-	d = defined variable p = parameter	$d = p$
arrayload	-	d = defined variable a = array i = index	$d = a[i]$
arraystore	-	a = array i = index v = value	$a[i] = v$

needsReply<return> must be true. To satisfy both this constraint and the constraints inlined from needsReply, the solver would automatically generate output that satisfies a path to the true return site at line 35, thereby "choosing" a path through needsReply that would return true.

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 26 is an invocation to a pure function and IntelliDroid will convert this to the constraint [(addr = this.sNum ∧ equals<return> = true) ∨ (addr != this.sNum ∧ equals<return> = false)]. This simplifies to (addr = this.sNum) since equals<return> must be true to reach the abortbroadcast suspicious behavior. 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 17, 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 createFromPdu, thus allowing IntelliDroid to generate an appropriate input. This is conceptually equivalent to "stitching", which is used to solve constraints that contain similarly complex functions, such as SHA1 and MD5, in BitFuzz [9]. 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 3.4.

3.3   Extracting Event-Chains

At this point, the extracted constraints consist of a boolean expression of concrete values and variables. Ideally, all of the constraint variables should be dependent on the event handler's input parameters. In such a case, solving the constraints for these variables and injecting the solved values will execute the desired suspicious path. However, there may be cases where the path constraints of suspicious path depend on heap variables that cannot be set to the correct values using only the arguments to the entry-point method of the suspicious path. In this case, IntelliDroid must find a definition for these variables that can be executed in a way that sets the heap variable to the required value.

An example of this is shown in Listing 3.1 for the SmsReceiver.sNum heap variable, in the call path to abortBroadcast. This variable is used in a constraint imposed by the conditional branch in line 26, but is defined in another invocation of onReceive where the intent action string is BOOT_COMPLETED. To complete the constraints and execute the suspicious path, two actions must be completed: (i) IntelliDroid must find any additional constraints on the heap variable and add them to the current path constraints; and (ii) IntelliDroid must ensure that the value that is extracted for the suspicious constraints is actually

stored in the heap variable prior to executing the suspicious path. To do this, when the constraints for a suspicious call path depends on a heap variable, IntelliDroid searches for statements where the heap variable is defined and records the event handlers containing the 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 suspicious path. Later, when solving the constraints, a concrete value will be assigned to the heap variable and used to solve the suspicious path constraint. For `sNum` in Listing 3.1, the supporting call path would begin at `onReceive` and the supporting constraints would include (`i.getAction() = BOOT_COMPLETED`). The main suspicious 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 suspicious path form an *event-chain* that results in the activation of the suspicious behavior. In the run-time system of IntelliDroid, this event-chain will result in multiple input injections. For Listing 3.1, the event-chain for the suspicious method call to `abortBroadcast` would include a injected boot event followed by an injected SMS input. In cases of multiple heap variable dependencies, the process is performed iteratively, as shown by the left backedge in Figure 3.1. This forms a event-chain ordered by the data-flow dependency between the variables.

Event-chains are also needed when event handlers are registered 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 certain 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 for the event handler 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 suspicious call path.

3.4   Run-Time Constraints

For the simple case where all constraint variables are input-dependent or can be concretized, the 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 the non-input-dependent constraint variables cannot be determined statically. This may occur for heap variables where the alias analysis is imprecise or for values

obtained from Android API methods that cannot be modeled statically. IntelliDroid's hybrid static-dynamic design allows for these values to be obtained 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 variables that remain unresolved after the static phase can be resolved prior to being passed to the constraint solver.

Run-time constraint variables can be divided into two groups based on the data source. *Monitored variables* depend on an external input that may be malicious, so rather than force these variables to some value, IntelliDroid will monitor these variables and use the real values derived from the external source. *Controlled variables* are derived from the Android framework and OS, which is not malicious; therefore, these variables can be controlled to take a value that, together with the other constrained variables, satisfies the suspicious path constraints and enables the suspicious behavior to be executed. IntelliDroid does not modify variables from a possibly malicious source because this may unrealistically alter the behavior of the malware, but controlling the device state is unlikely to change the intrinsic behavior of the malware. For instance, although setting the device time changes the device state, it does not change the malicious behavior and it merely ensures that the malicious activity is actually triggered. However, changing or controlling the network input received by the application can change the malicious behavior itself, so these values are instead monitored.

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 of this presented in [38] 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 suspicious 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 may be 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 the target suspicious activity is triggered only when the system time is set to 1:00, the device time will be set to this value before injecting the main suspicious event. This is essentially another form of event-chain extraction, where supporting events must be injected prior to executing the main suspicious call path.

3.5   Input Injection to Trigger Call Path

Once the constraints are generated and all run-time values obtained, IntelliDroid can trigger the desired suspicious call path by obtaining the input parameters that fulfill the constraints. As previously discussed, the task of solving the constraints is placed in the dynamic component of IntelliDroid; thus, the input parameters are solved for and generated immediately prior to executing the suspicious call 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. When the user specifies a suspicious call path to be executed, this gateway service transforms the input parameters into input objects that can then be used with the event injection.

Because the constraint solver is a component of the client program on the computer, the input data is serialized when it is received on the device. The `IntelliDroidService` class is responsible for deserializing this data and constructing the input object. The type of object is specific to the type of event desired (for instance, a `Location` object for a location event). Thus, if new events are to be tested and implemented in IntelliDroid some knowledge of the input object structure is necessary. For suspicious call paths that require an event-chain, the events are injected serially and the constraint solver is used prior to each injection to ensure that the desired supporting or suspicious path is executed.

A particularly important component of event injection is the location at which the injection occurs. Perhaps the simplest method would be to inject the input at the *framework-application boundary*, by invoking the event handler method in the application directly. The method parameters can easily be specified such that the necessary application constraints are fulfilled. However, while this technique can trigger the desired application behavior in most cases, it can cause inconsistencies between the framework and the application states. On receiving an event, an application may try to obtain more information about the event by calling into the Android framework. Since the event was triggered at the framework-application boundary, the framework will contain no record of such an event and will return incorrect or error-prone information.

This situation often occurs for Android services that record event information in some persistent

form (e.g. a database). It is especially problematic when applications register event handlers that watch for changes to this persistent data. For instance, new SMS messages are normally detected by an application by registering a `BroadcastReceiver` subclass for the intent action `android.provider.-Telephony.SMS_RECEIVED`. The SMS message content and metadata is passed to the `onReceive()` event handler via the `Intent` parameter. However, some malicious applications (e.g. *CoinPirate*, *GamblerSMS*, and *HippoSMS*) use an alternative, more stealthy technique to monitor the system for SMS messages. These malware applications ensure they are notified when a new SMS message arrives by using the `ContentObserver` class, which is registered by an application to watch a particular database. The `ContentObserver.onChange()` method is an event handler entry-point into the application and provides a notification of when a database change is detected. By registering a `ContentObserver` class for SMS database changes, an application can receive notification of a new SMS message without using the normal SMS and telephony classes/methods.

To ensure that the application receives all new SMS notifications, IntelliDroid could have invoked `ContentObserver.onChange()` directly (at the framework-application boundary). However, because this entry-point is a generic method used for changes in any database, the only event information passed to the method is a `Uri` to the database item triggering the event. To obtain more information, the application will query the database, which poses a problem since the framework has no knowledge of the injected event. This is especially problematic, since malicious activity is often triggered only when the event fulfills certain conditions (hence the constraint extraction). If the framework cannot return the necessary information, the application will never execute the malicious activity while being tested.

This issue could be remedied by intercepting such query calls into the framework; however, this would require IntelliDroid to model the Android device so that it can respond consistently to each query request. Rather, IntelliDroid injects input into the framework; that is, events are injected at the point where the framework would normally receive events from the device sensors. The event input will travel through the framework code as if it were a normal event, and will trigger the desired application behavior while keeping framework state consistent.

The main challenge is to identify the appropriate points in the framework to inject inputs in this manner. Such points should be methods that have a one-to-one relationship with the event handler of interest, so that inputs thus injected will result in one invocation of the desired application event handler. 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, the process calls `PhoneBase.sendMessage`. This method and other injection points can be determined via static analysis of the Android framework, using a backward call graph traversal starting from the event handlers of interest. 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 as input injection points as they present a cleaner interface.

The Android framework may impose constraints on the call path between the injection method and the application event handler. The constraints for this *injection path* are extracted using the same analysis that IntelliDroid performs on applications, and then combined with the application constraints such that the full path from the injection method to the suspicious activity is achieved. The framework constraint variables are dependent on the injection method parameters while the application constraint variables are dependent on the event handler input parameters. Since the Android framework is the same for every application, IntelliDroid can extract the injection path and associated constraints for supported event handlers once, and store them in a library for use at run-time. To simplify the combination of constraints, the static analysis of the framework also tracks the relationship between injection method parameters and event handler parameters; in other words, the data flow through the framework call path is tracked. Combining the framework and application constraints is therefore a matter of joining them with a logical AND, and appending extra constraints specifying how the injection method parameters are related to the event handler parameters.

Chapter 4

Implementation Details

4.1 Static Analysis

For versatility, IntelliDroid performs its analysis on compiled Android applications and does not require the source code. Because they are packaged in APK files and stored using the Dex bytecode format, the applications must be unpacked prior to analysis. In addition, IntelliDroid takes advantage of existing static analysis libraries for Java, which requires that the Dex bytecode be converted to Java bytecode. The dex2jar tool [16] is used for this purpose and the APKParser tool [3] is used to extract the application's manifest file from the APK package. The converted files are then passed to IntelliDroid's static component, which uses the WALA static analysis libraries [34]. WALA provides support for basic static analysis, such as call graph generation, data flow analysis, alias analysis, and an intermediate representation based on SSA.

To perform the actual analysis on the code, IntelliDroid uses WALA to create a call graph using 0-1 context sensitivity with a type-based heap model. Entry-points for the call graph are determined as described previously and represent event handlers where the Android framework passes execution to the application. However, the Android platform provides facilities that allow programs to transfer execution between event handlers without an explicit method invocation. When generating the call graph within WALA, these Android-specific edges need to be added so that the call graph gives an accurate representation of how execution can flow between methods in the application. These additional edges are shown in Table 4.1 and are created by implementing a custom method target selector for the call graph generator.

For normal Java method invocations, nodes are created in a context-insensitive manner. How-

28

Table 4.1: Android-Specific Call Edges

Edge Type	Invoked Method	Resolved Method
IPC	Handler.sendMessage...()	Handler.handleMessage()
Intent	Context.sendService()	Service.onCreate() Service.onStart() Service.onStartCommand()
Thread	Thread.run()	Thread.start()
AsyncTask	AsyncTask.execute()	*fakeMethod()* → onPreExecute() → onExecute() → onProgressUpdate() → onPostExecute()

ever, Android has several cases where some degree of context sensitivity is required so as not to create too many false paths in the call graph. For interprocess communication (IPC) edges, the `Handler.handleMessage()` method that receives the method call can be invoked for a variety of messages and dispatches the appropriate method for each type of message it receives. To ensure that the call path for each message type is represented separately in the graph, the `Handler.handleMessage()` method node is cloned for each type of message – in other words, a context sensitivity of 1 is used for these invocations. The resulting call graph nodes are shown in Figure 4.1. We found that these call graph generation options provided the best trade-off between precision and efficiency, and ensures that the static analysis can complete in a timely manner for most applications.

For most cases, type information in the invoke instruction is sufficient to determine the receiver of a method invocation. For instance, if an application creates a `Handler` subclass called `CustomHandler` and invokes `CustomHandler.sendMessage()`, it is obvious that the class receiving the `handleMessage()` invocation should be `CustomHandler`. However, if the type information is not explicit, IntelliDroid resolves the receiver to any inner classes within the invoking class. That is, if an application calls `Handler.sendMessage()` within the `MainActivity` class and there exists an inner class `MainActivity$-CustomHandler` that is a subclass of `Handler`, the resolved receiving class would be `MainActivity$-CustomHandler`. The inner class can be named or anonymous, as long as it is of the correct type and implements the method needed to resolve the invocation. This heuristic was based on the observation that most custom classes handling interprocess communication are declared within the classes that call them. For more complicated inner class usage, this can cause imprecision and IntelliDroid takes a conservative approach by assuming that an unresolved method invocation can be received by multiple

Code:

```
onCreate() {
    handler.sendMessage(MSG_TYPE_1, ...);
    …
    handler.sendMessage(MSG_TYPE_2, ...);
}
```
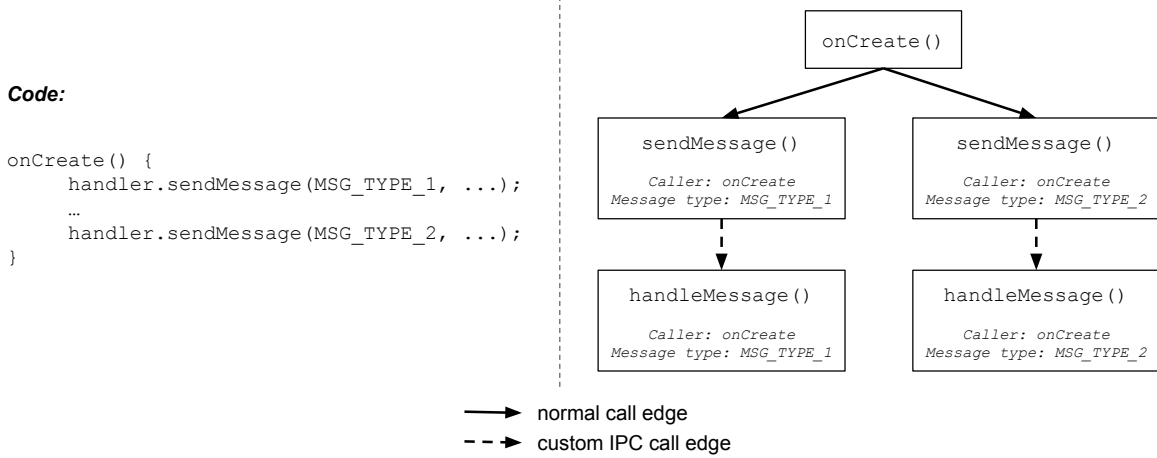
Figure 4.1: Custom Call Edges and Context Sensitivity for Android Handler Class

inner classes. Generally, false call paths can be eliminated when the constraint solving and dynamic run-time components determine that they are infeasible.

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 from 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. Currently, 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 modelled due to the limitations of the constraint solver. For instance, string methods are modelled internally as well as trigonometric operations, since the constraint solver does not support such functionality. In general, the processing of the invocation of a framework method depends on whether it introduces externally-obtained data and whether the constraint solver supports the operations that it performs.

The constraints extracted by IntelliDroid are built internally as binary expression trees. When the constraints are fully generated, they are translated into code for the constraint solver. In addition, to avoid conflicts and syntax errors, variable names for the constraint solver code follow a strict naming scheme and a map is generated to convert between these names and the original variable names. This variable map, along with information about the suspicious call path and possible event-chains, are placed into an app-specific file. When IntelliDroid has completed the static phase, this file will contain all suspicious call paths found in the application, along with information detailing how the dynamic component can trigger them. For a given application, the static component of IntelliDroid needs to

execute only one time, since this file will contain all of the information that the dynamic component requires.

4.2   Dynamic Analysis

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 suspicious call path to execute. It also interfaces with the constraint solver used to generate the path inputs. The Z3 constraint solver [14] is used in IntelliDroid and accessed using Z3's Python API (Z3-py). Communication between this program and the device is done via sockets, using the device port-forwarding feature of the Android Debug Bridge (ADB) [1]. The other endpoint of the socket is located in the gateway Android service, `IntelliDroidService`, and communication is received via a dedicated thread initiated by this service. The `IntelliDroidService` class 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 using the input object.

The custom Android OS used for IntelliDroid is centered around the gateway `IntelliDroidService` class. The service is mostly independent and does not require modifications to existing Android services. When the client program requests that a particular event is to be injected, the input object for the event is assembled. The object generation is customized for each event type (e.g. `Location` for location events or `SmsMessage` for SMS events). While some input objects merely require certain fields to be set, others may be more complicated. For instance, the device receives SMS messages in the PDU bytecode format [1]. While the desired input parameter is an `SmsMessage` object, it must first be derived from the PDU format. When generating the SMS message, the `IntelliDroidService` class encodes the necessary values into the PDU bytes such that the `SmsMessage` generated will contain the correct field values to trigger the suspicious call path. The current prototype of IntelliDroid supports the injection of location, SMS, and boot events. This injection includes the generation of input data that satisfies the constraints imposed by the malicious applications. In addition, IntelliDroid also supports setting various forms of device state, such as the time or date, should the application impose such constraints.

When solving the constraints produced by static component, there may be variables whose values are not dependent on the event handler's input and whose values could not be determined statically.

---

[1] `http://developer.android.com/tools/help/adb.html`

Often, these values are derived from monitored APIs, whose values can affect the malicious behavior. One way IntelliDroid monitors these APIs is by modifying the Android API implementations to record these values and allow the `IntelliDroidService` class to read them and pass them to the constraint solver. However, IntelliDroid also uses other techniques when appropriate. For example, the values used in the constraints may come from the application's shared preferences, and these values are extracted from the application's directory in the device's file system. Finally, IntelliDroid monitors network APIs using a debugger attached to the application. This enables IntelliDroid to intercept the precise values that the application receives, and it also simplifies the monitoring process when the application is using encrypted traffic.

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. Such values can be extracted statically from the application (when the event handler is registered) or controlled by IntelliDroid by setting the device's location state. However, the current prototype of 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 extraction in cases where the event handler registration parameters are not explicit within the application code.

Chapter 5

Evaluation

IntelliDroid was implemented for the Android 4.3 operating system (Ice Cream Sandwich), with the static component and the dynamic client program running on an Intel i7-2600 (Sandy Bridge) CPU at 3.40 GHz with 16GB of memory. We evaluate two aspects of IntelliDroid. First, we evaluate the effectiveness of IntelliDroid at generating inputs to trigger suspicious behavior. Second, we measure the performance of IntelliDroid by measuring the time it takes to extract constraints and inject the targeted inputs.

5.1 Effectiveness

To measure how effective IntelliDroid is at generating targeted inputs that invoke suspicious behavior, we run IntelliDroid on a corpus composed of malware samples obtained from the Android Malware Genome project [38], listed in Table 5.2. We chose the samples from 17 malware families that exhibit the malicious behaviors described below. For each behavior, we describe both the static description of the behavior that is provided to IntelliDroid, as well as the dynamic check we perform to confirm that IntelliDroid is able to successfully cause the suspicious behavior to be executed:

- **Premium SMS**: This behavior occurs when malware sends SMS messages to premium numbers without user knowledge or automatically replies to premium SMS confirmation messages without user input. The static description of this behavior is a call to `SmsManager.sendTextMessage` with a hard-coded string. We confirm that this behavior is executed by instrumenting the `sendTextMessage` method to output the address and the message that is being sent, so that we can confirm it is a premium number.

- **Blocking SMS**: This behavior occurs when malware prevents the user from being aware of any fee confirmation messages by blocking SMS notifications. The static description of this behavior is a call to `BroadcastReceiver.abortBroadcast` from within an `onReceive` event handler. We confirm that this behavior is executed by checking that `BroadcastReceiver.abortBroadcast` is called.

- **Deleting SMS**: This behavior occurs when malware deletes SMS messages as they arrive, usually to hide premium SMS confirmation messages or SMS command messages from the user. The static description of this behavior is a call to `ContentProvider.delete` where the URI is `content://sms`. To confirm that the suspicious behavior is correctly invoked, we check that a successful deletion occurs on the SMS content provider, and that the deleted message is one that was injected by IntelliDroid.

- **Leaking information via SMS**: This behavior occurs when malware sends sensitive information, such as the device's phone number, IMEI, contact information, and messages, in an SMS message. This static description of this behavior is a call to `SmsManager.sendTextMessage` where the content is dependent on data derived from a sensitive source. IntelliDroid uses a simpler heap and inter-procedural analysis than the static taint-tracking in [4] to detect this suspicious behavior since IntelliDroid will remove false positives in the dynamic phase. We confirm the execution of this behavior by confirming that an SMS message is sent and by inspecting the content of the SMS message.

In some cases, the malware constraints depend on values obtained from network requests to a remote control server. As described earlier, IntelliDroid will monitor these network requests to extract the necessary values and solve the constraints necessary to generate inputs that will match these requests. However, for *CoinPirate*, *CruseWin*, and *Pjapps*, the servers were no longer available so the network data values could not be extracted. To test these malware samples, we implemented an HTTP proxy server to act as a fake control server responding to application requests with appropriately formatted replies.

We measure the number of instances where IntelliDroid successfully generates inputs that trigger the malicious behavior. Table 5.1 shows the number of times IntelliDroid was successful in triggering suspicious activity using various injection inputs. Many of the malware samples exhibited multiple suspicious behaviors, in which case they are tested once for each behavior. For each type of injected input and suspicious behavior, we list the number of samples, the number of times IntelliDroid was able to successfully generate inputs that cause the suspicious behavior to be triggered, the number of instances

Table 5.1: Effectiveness of IntelliDroid.

Injected	Behavior	Samples	Successes	Event Chain	Consistency	Run-Time Data
SMS	Premium SMS	8	8	2	1	3
	Block SMS	7	7	1	1	4
	Delete SMS	4	4	1	4	2
	Leak via SMS	8	7	1	1	3
BOOT	Premium SMS	1	1	0	0	0
	Leak via SMS	1	1	0	0	1
LOCATION	Leak via SMS	1	1	1	1	1

the behavior could only have been triggered by an event-chain and the number of times the behavior could only been triggered if the Android framework is consistent (i.e., inputs must be injected at the device-framework interface). Table 5.2 provides detailed information about the suspicious behaviors detected and executed in each malware family. For each malware family, the columns show the input(s) that need to be injected to trigger the corresponding suspicious behavior. Checkmarks (✔) represent instances where IntelliDroid was able to successfully generate the input from extracted constraints and trigger the suspicious behavior. The one ✗ denotes a case where IntelliDroid could not cause the suspicious behavior to execute, which we will explain further below.

5.1.1   Case Studies

In 6 cases of malicious behavior, IntelliDroid needs to generate event-chains to ensure that all constraints are 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 app has been running for certain amount of time. In these cases, simply injecting a single event would not have satisfy the multi-event constraints that these applications impose. The event-chain for these paths includes a separate event to change the device's system time, which is injected prior to the injection of the input that finally triggers the suspicious behavior. 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. In this situation, although the malicious activity was executed in a path originating from the `onLocationChange` event handler, it would not have been executed if the control SMS message was not first received. IntelliDroid's

Table 5.2: Effectiveness by Malware Family

Malware	Injected: SMS				Injected: BOOT		LOC.
	Premium SMS	Block SMS	Delete SMS	Leak via SMS	Premium SMS	Leak via SMS	Leak via SMS
Bgserv		✓		✓			
CoinPirate	✓	✓					
Crusewin	✓		✓				
DogWars					✓		
Endofday	✓		✓	✓			
GamblerSMS				✓			
GGTracker	✓	✓					
GoldDream				✗			
GPSSMSSpy				✓			✓
HippoSMS	✓	✓	✓				
NickyBot			✓	✓			
NickySpy						✓	
Pjapps		✓		✓			
RogueSPPush	✓	✓	✓				
SMSReplicator	✓			✓			
Zitmo				✓			
Zsone	✓	✓					

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, when the application is tested during run-time, IntelliDroid ensures that the SMS event is injected prior to the location event.

In 8 cases of malicious activity, the malware would not have behaved realistically if IntelliDroid had not implement 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 behaviour, 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. 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, 4 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 occured while screening the applications for malware, it would have caused the screening tool to miss potential malicious activity.

In 14 malicious call paths, the extracted constraints for the paths required run-time data. 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 performs the constraint solving in the dynamic component, the statically extracted constraints could be augmented with the run-time data prior to generating the necessary 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. Due to the hybrid system that IntelliDroid employs, this run-time data could be obtained and the malicious activity was reliably triggered.

In 1 case, the suspicious call path and its corresponding constraints could not be fully generated. This occured in the *GoldDream* malware and was the result of data flow through files. In this particular application, an initial set of SMS messages are received and saved into files, which act as persistent storage. In a separate event handler, these files are sent to a third-party through the `sendTextMessage` API method. IntelliDroid currently does not identify dependencies across files, so during execution, it does not generate the initial set of SMS messages. If IntelliDroid recognized file system dependencies the same way it recognized heap dependencies, IntelliDroid would then search for the required writes to the same files and generate the inputs necessary to inject the initial set of SMS messages.

Due to the nature of static analysis, there can be cases where the extracted suspicious call paths or constraints are infeasible. This can be due to control flow within the path methods or imprecision when resolving method calls and heap accesses. In particular, the *NickyBot* malware receives and sends premium SMS messages without user knowledge. The suspicious method calls are performed in the application's `SmsService` class within various asynchronous threads instantiated by this class. Due to aliasing, the static analysis cannot determine the `Thread` or `Runnable` object referenced when a new thread is started. Since IntelliDroid's static component is conservative, there can be paths extracted that are infeasible. However, the dynamic component of IntelliDroid clearly shows that these paths are unrealizeable at run-time and that they do not lead to suspicious activity being triggered. Therefore, although the static analysis was imprecise in this particular situation, the IntelliDroid's hybrid design

ensures that any infeasible paths reported from the static component are filter out by the dynamic phase.

5.2   Performance

We also measure the performance of IntelliDroid at generating and injecting inputs. Input generation and injection 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.

Using the 1260 samples available from the Android Malware Genome project [38], we evaluated the performance of the static component for the supported SMS-based suspicious behaviors. We measure that the time required for the static analysis phase of IntelliDroid to be an average of 22.7 seconds per application, with 91.5% of the applications completing within a 60-minute time limit. This measurement includes both the time to find paths to any of the suspicious behaviors, and the time to extract constraints from any paths found. In most cases, a significant amount of analysis time is taken by WALA's call graph extraction and the search for suspicious behavior, which must be performed on the entire application, and accounts for roughly 22.8% of the static analysis time. The overall static analysis performance is reasonable for IntelliDroid since the static analysis phase is only performed once for each application. Thus, its cost is amortized over dynamic analysis of the application, which often involves the generation and injection of multiple inputs.

Unlike the static analysis component, the dynamic generation of input values must be quick since it is performed for every injected input. In many of the tested applications in Table 5.2, several events must be injected to trigger all of the malicious activity in the code. 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 29 suspicious paths in our dataset to be an average of 45.3 milliseconds.

Because IntelliDroid extracts constraints statically, it is considerably faster than test-generation systems that use symbolic execution [11, 22, 23]. Symbolic execution not only needs to execute the application to extract the constraints, but it may need several such executions before it is able to find the desired path and each execution must be performed on a symbolic execution engine that can be several orders of magnitude slower than the system it is emulating [13]. This is less of a problem when the goal is code-coverage, but the symbolic exploration is especially wasteful when the goal is only to execute one path or a small number of paths in the application. As a result, IntelliDroid uses static extraction of constraints, which allows it to dynamically execute and analyze suspicious behaviors in malware with only tens of seconds of total analysis time.

5.2.1 Performance on Benign Applications

Although IntelliDroid is designed to analyze malicious behavior, we also tested IntelliDroid on corpus of benign applications from exising app markets to determine whether IntelliDroid can be employed for widespread dynamic analysis of Android applications. The static analysis component was executed on a large dataset of applications from Android Observatory [7] and Figure 5.1 shows the commulative distribution of IntelliDroid's performance on the 1273 applications tested. This application set was obtained by filtering the Android Observatory data set to those that contain behavior supported by the current prototype of IntelliDroid, a process that was facilitated by using Android Observatory's metadeta database to extract applications that use the WRITE_SMS or SEND_SMS permissions. As shown in Figure 5.1. the static analysis was able to complete under 45 minutes for 72.7% of applications, and for a majority of these applications, the static analysis completed within 45 seconds.

The average analysis time for the data set of benign applications is 126 seconds ($\sim$ 2 min.), which is significantly greater than the analysis time for the set of malware applications evaluated. This is due to the size and complexity of the applications in the two sets. Malicious applications are generally small and simple, and performs any malicious activity in the background. In particular, the data set obtained from the Android Malware Genome project did not contain many with advertisement libraries, which signficiantly increases the size and complexity of the application. On the other hand, the set of benign applications represent those found in a real marketplace, where advertisements are common. Although an application may only use a small portion of the library, the static analysis component of IntelliDroid must create a complete call graph and extensively search for any suspicious activity. Thus, widespread analysis of applications in Android marketplaces may take some extra time compared to analysis of malware; however, the analysis time is still feasible since IntelliDroid only requires that the static phase be executed once per application.

5.3 Malicious vs. Benign Applications

A randomly selected sample of applications from the Android Observatory dataset was dynamically tested to determine whether the extracted call paths and constraints could be used to trigger suspicious activity. Because these applications are benign, there was very little suspicious activity occurring purely in the background and thus, the extracted call paths often originate from a UI event handler or require input from a UI element. These UI events were manually triggered to determine the feasibility of the extracted paths. For all of the applications tested, the extracted call paths supported by IntelliDroid's
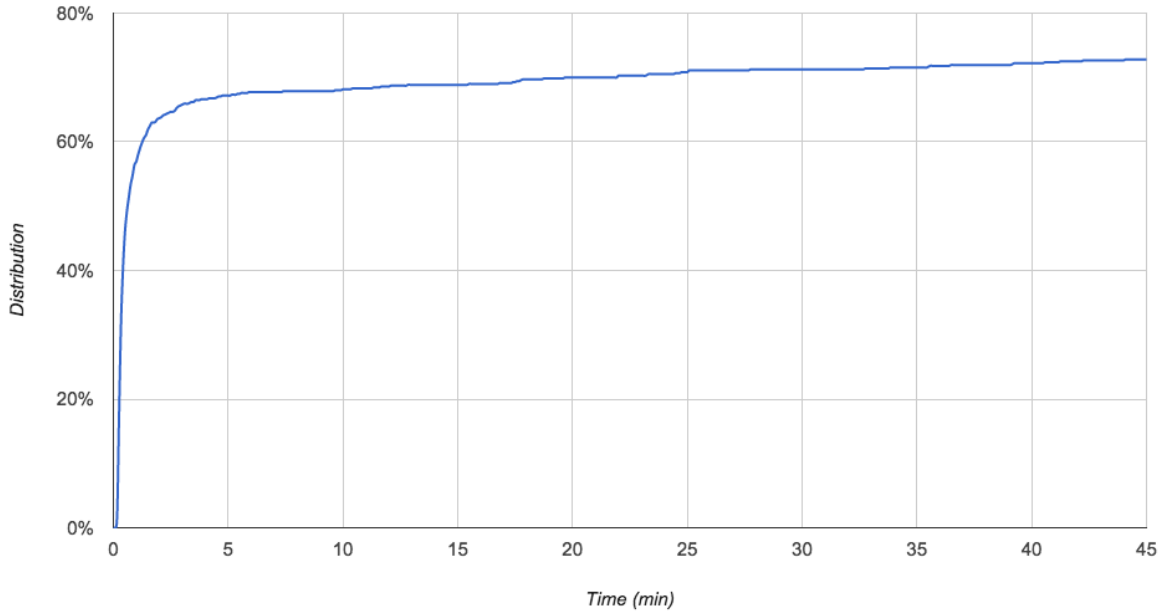
Figure 5.1: Static Analysis Performance for Benign Applications

implementation could be triggered. These applications, and their functionality, are described in Table 5.3.

Our manual dynamic testing suggests that there are certain types of behavior that can distinguish malicious and benign applications. One particular marker for malicious behavior is the lack of user input requested when performing suspicious actions. For the benign applications tested, although the suspicious call paths were verified to be feasible, they require some form of user input or user action before they can be activated. The user input requested might be simple, such as accepting the terms of a privacy disclaimer, or it may require that the user set up the exact phone numbers and data for the sent SMS messages. For malicious applications, this was almost never the case and the suspicious call paths could be triggered without much user action required. Another marker for malicious behavior is the complexity of the constraints extracted for the suspicious call paths. Through manual inspection of the static analysis output, the call paths from malicious applications generally contain more complicated and longer constraints than those found for benign applications. This is not surprising, since malicious activity is often triggered by a specific external message or event, for the purposes of remaining undetected. Although this behavior is not the case for all malicious applications, the complexity of the constraints may serve as a starting point for a new method of classifying malicious and benign behavior.

Table 5.3: Behaviors Found in Benign Applications

Application	Description	Suspicious Paths
Boyfriend Tracker	Forwards all SMS messages to a pre-specified phone number in a background process.	New SMS → Send SMS
CadPage	Provides more detailed location information when firefighters receive a page via SMS.	New SMS → Abort broadcast
Kincent	Messenger app that can also receive/send SMS messages (can replace default SMS app).	New SMS → Abort broadcast
Circle of 6	Sends location information to your friend group to ask for a ride.	Location change → Send SMS
Commodo Mobile Security	Allows user to set a black/white list for incoming SMS messages.	New SMS → Abort broadcast
Where's My Droid Lite	Sends location of phone if a location request SMS message is received.	Location change → Send SMS
	Allows user to set a black/white list of phone numbers that can receive location information.	New SMS → Abort broadcast
SeekDroid Lite	Receives third-party location tracking request via SMS.	New SMS → Abort broadcast
	Sends location tracking confirmation message to requesting party.	New SMS → Send SMS
	Sends location information to requesting party.	Location change → Send SMS
TouchPal	Allows user to create a SMS blacklist.	New SMS → Abort broadcast
	Deletes record of SMS message when sender is on blacklist.	New SMS → Delete SMS

Chapter 6

Limitations

There are several limitations that the design and implementation of IntelliDroid cannot address. Some limitations are inherent parts in the analysis techniques employed, while others are artifacts of the implementation of the current prototype of IntelliDroid.

6.1   Static Analysis

Static analysis is always imprecise due to the lack of run-time information and IntelliDroid takes a conservative approach when this issue arises during analysis. For instance, loops are a common limitation with static analysis since the end condition may not be known and IntelliDroid propagates all data values that could be assigned by any number of loop iterations. While IntelliDroid's design contains mechanisms to overcome some of these issues (such as delaying constraint solving until run-time and relying on the dynamic phase to filter out any infeasible paths), there are cases where the imprecision of the static component carries through past the dynamic phase. For the case of reflection, reflected method calls are not handled by the current implementation of the static component, so it is possible for malicious applications to hide call paths and invocations of suspicious methods by using reflection. If the static component does not detect a suspicious call path, the dynamic component will not trigger the suspicious behavior, resulting in a false negative. Existing work in static analysis also suffer from this issue, although some have tried to address it using ad-hoc methods [19]; nevertheless, a complete analysis currently does not exist. In addition, since WALA is a Java-based static analysis tool, native code within applications is not supported; this can lead to similar situations where malicious applications can hide malicious call paths within the native component.

Malicious applications may also obfuscate suspicious behavior in other ways. As analysis tools become

more sophisticated, so has malware. These applications may try to hide its activities by encrypting strings such that it is difficult for a static analysis tool to determine the data being sent or received. If an application uses reflection to hide method calls, it may also encrypt the method name used in the reflected call such that the analysis tool cannot determine the target method of the invocation. While this information can be obtained at run-time when the invocation is resolved, there is currently no mechanism to funnel this run-time method information back to the static analysis. Therefore, IntelliDroid cannot handle such obfuscation and would miss any suspicious call paths that employ these techniques.

Another limitation of IntelliDroid stems from the nature of the Android platform. The analysis of Android applications is problematic when using a traditional Java analysis tool such as WALA. While considerable effort was made to customize the analysis such that Android-specific mechanisms are supported, we do not claim to handle all Android idiosyncrasies. Rather, the current prototype of IntelliDroid supports common Android mechanisms such as intents, IPC, RPC, AsyncTask's, and threads. It is possible that future versions of Android may contain more mechanisms where the flow of execution cannot be followed via simple method invocations. In such cases, the static component of IntelliDroid would have to be patched to support these additions. Since the current implementation is modular, additional support for Android can be easily added with sufficient time.

Since IntelliDroid is currently a prototype, only events regarding location, SMS, time, or boot are supported. Despite this limitation, IntelliDroid was able to trigger malicious behaviour in a variety of malicious and benign applications. For malicious applications that exploit other services or resources, IntelliDroid must be modified to support these services. This would include changes to the `IntelliDroidService` gateway class to add the ability to trigger the newly supported service. This would also include changes to the component that statically analyzes the framework to find the injection method(s) for the new service and extract any injection path constraints.

6.2   Dynamic Analysis

IntelliDroid is not meant to be an independent dynamic analysis system; rather, it is tool that can aid in malware analysis and improve the effectiveness of existing (and future) systems. In the current prototype, IntelliDroid does not perform a specific analysis such as taint tracking, but it can be easily integrated with existing dynamic analysis tools.

Because IntelliDroid is in the prototype stage, only a limited number of services and resources are supported: location, SMS, boot, and time. Therefore, if a malicious application performs other suspicious activity, IntelliDroid must be patched to extract the suspicious call paths and constraints. In addition,

because UI events are not supported, some call paths extracted during the static component may not execute without input from a human. There are existing automatic UI testing frameworks such as UIAutomator [33] that can extract the UI layout shown on screen and allow the user to programmatically trigger UI events. Such a framework can easily be integrated into the dynamic component of IntelliDroid; however, this is left as future work due to time constraints and since malware generally perform most of the malicious activity in the background.

Adding support for new services and resources is relatively straightforward, given the modular implementation of IntelliDroid. To add suspicious behaviors, the static component must be configured with a new list of suspicious methods to find. To test the effectiveness of IntelliDroid for the new behaviors, the custom Android OS must instrument these suspicious methods (although this is not necessary if IntelliDroid is attached to an existing dynamic analysis tool). The injection of new events is slightly more involved, since the injected input object must be manually specified for each system service (i.e., constructing a `Location` object or a `SmsMessage` object for Location and SMS events, respectively). The method by which the new input object must be constructed and injected should be added to the `IntelliDroidService` gateway class in the custom Android OS. The static analysis for the framework must also be re-run to determine the injection location for this new event type.

6.3   Constraint Extraction and Solving

The ability to generate program inputs to trigger the extracted call paths relies heavily on the abilities of the constraint solver. IntelliDroid currently does not minimize any extracted constraints and relies on the solver to remove any redundancies or infeasibilities that exist within the constraint. Section 5.2 shows that the lack of minimization does not pose a problem for the performance of the constraint solver; however, there may exist malicious applications that try to overload the constraint solver with long call paths containing complicated conditional structures. Although we did not encounter this issue among the Android applications tested, this remains a possible issue with IntelliDroid.

The current constraint extraction process cannot handle complicated code that perform precise byte-wise calculations such as encryption or hashing. Such functions must be modelled and handled as special cases within IntelliDroid. This is an issue with existing symbolic execution systems as well, and IntelliDroid's approach is very similar to the "stitching" mechanism used within BitFuzz [9]. Modelling must also be done for functions that the constraint solver cannot handle, such as trignometric operations (used to calculate GPS coordinates for the location service). In the current prototype, these problematic operations are handled manually by noting when/where they occur and having the dynamic client program

process these calculations outside of the constraint solver.

Because IntelliDroid is currently using the Python API for the Z3 constraint solver [14], the string library is not available. Therefore, string functions such as `equals()`, `contains()`, or `startsWith()` must be modelled and string variable types are handled by the dynamic component as a special case. Due to the heuristics used when modelling such functions, there can be cases where complex string manipulation may not be represented precisely by the extracted constraints.

Despite these limitations, IntelliDroid was designed such that it can be used with any off-the-shelf constraint solver. Although Z3-py [14] is currently being used in the prototype, this can be replaced by any constraint solver provided that IntelliDroid is given a method of translating the constraint expression trees (i.e., the internal data structure used to store constraints) into code for the new solver. Therefore, if there is a future constraint solver that can handle these issues, it can be incorporated into IntelliDroid with minimal effort.

Chapter 7

Related Work

7.1 Symbolic Execution

Symbolic execution allows dynamic analysis of applications to achieve high code coverage by exploring all outcomes whenever execution flow can diverge due to a conditional branch. Dart [22] and Cute [30] show implementations of such symbolic execution systems and they employ various heuristics to optimize constraint generation and constraint solving time. EXE [11] uses symbolic execution to dynamically find dangerous or vulnerable code and optimizes the exploration by toggling constraints to search only for interesting or unexplored paths. Similar to KLEE [10], it also simplifies the generated constraints so that the constraint solver can handle them more efficiently. Tools have also been proposed for concolic testing on the event-driven Android platform. These include DynoDroid [27], which uses an iterative process to capture the application state to trigger new events, and the ACTEve algorithm [2], which handles path explosion by avoiding duplicate paths between event handlers. While IntelliDroid shares constraint extraction, solving and symbolic data analysis with these other techniques, the main difference is that IntelliDroid does not aim for code coverage but rather a targeted execution of a specific section of code. The high cost of detailed symbolic execution is not as big an issue for tools aiming for code coverage because while each execution is expensive, it still increases the amount of code covered. On the other hand, for IntelliDroid, executing a path that does not lead to the suspicious behavior is of no value. As a result, IntelliDroid gathers constraints only for the paths of interest using a more lightweight, though possibly more imprecise, static analysis. In practice, these sources of imprecision do not result in incorrect or incomplete constraint extraction and our evaluation shows that IntelliDroid is able to successfully generate inputs to execute a variety of suspicious behaviors.

BitFuzz [9] addresses cases where constraint solving can fail due to overly complex constraints. It focuses on "encoding functions", such as cryptographic operations, which exhibit a high-degree of interdependence between variables. BitFuzz proposes "stitching", which uses the inverse of such functions to generate appropriate inputs, relieving the constraint solver of having to invert the function to find the necessary input. IntelliDroid handles complex Android API methods or methods that contain semantics that are unsupported by the Z3 solver (e.g. trigonometric functions) in a similar manner. While BitFuzz searches for the inverse functions automatically within the application's code base, the inverse functions used by IntelliDroid are manually implemented.

AEG [6] uses "preconditioned symbolic execution" to automatically find exploits in applications. AEG limits the values that the input data can be assigned such that only exploitable cases in the code are explored. This reduces the complexity of the constraints, and reduces time required to find exploits and test the application. The preconditions must be specified manually and are comparable to how the suspicious behaviors detected by IntelliDroid are specified. However, rather than limiting the input space, IntelliDroid limits the number of paths dynamically explored based on the target suspicious behaviors. This is possible due to the static analysis performed prior to run-time. APEG [8] also generates exploits in a similar manner. However, rather than searching for the exploit, it assumes a patch is available and uses semantic differences between the patched and unpatched versions of the application to identify paths to execute.

7.2 Android Malware Analysis

Static analysis of Android applications has been widely used to detect malicious behavior or vulnerabilities. In many cases [19, 20, 21], IntelliDroid use similar static analysis techniques to trace the control and data flow through applications, and to handle Android-specific difficulties in call graph generation. In particular, the call graph entry-points used in IntelliDroid are discovered in the same manner as CHEX [26].

Because IntelliDroid uses its static analysis results to complement a dynamic component that performs the actual testing or scanning, IntelliDroid can provide more precise and accurate results than other tools that rely solely on static analysis. For example, AndroidLeaks [21] and ScanDal [25] analyzes applications to find sensitive information leakage. CHEX [26] uses static analysis to find component hijacking vulnerabilities in Android applications. SCandroid [20] analyzes applications to provide automated security certification and check whether an applications behavior matches its specifications. ComDroid [12] analyzes code for indications that an application component can be hijacked using mali-

cious intents. FlowDroid [4] uses advanced static analysis techniques to implement static taint tracking. For all of the vulnerabilities and information leakage detected, the dynamic component of IntelliDroid can be used to detect and verify the malicious behavior with greater accuracy while maintaining fast performance though targeted execution. Although IntelliDroid does not implement a specific analysis such as taint tracking, these tools can be attached to the dynamic component without difficulty. In such a case, IntelliDroid can provide similar code coverage using its static component while performing more precise taint tracking in the dynamic phase.

For malware analysis tools relying on dynamic execution, IntelliDroid can complement them to provide more complete and efficient testing due to its targeted execution. For instance, VetDroid [37] identifies sensitive behaviors by using a custom Android OS to track an application's permission usage. If sensitive behavior is detected, taint tracking is used to determine if the sensitive data is leaked. RiskRanker [24] uses several heuristics to determine whether an application is risky. IntelliDroid can be used to statically identify paths to these suspicious or risky behaviors, and efficiently confirm the malicious activity during run-time.

IntelliDroid is also beneficial for manual analysis of malware. The Android Malware Genome Project [38] provides an in-depth study of Android malware resulting from painstaking manual analysis. The evaluation of IntelliDroid shows that a targeted dynamic analysis tool can greatly aid the malware characterization process by executing any suspicious behavior dynamically and providing the exact conditions under which the malicious activity manifests.

ContentScope [39] shows a similar hybrid static/dynamic design to IntelliDroid with the purpose of finding vulnerabilities in application components. Although they also perform constraint analysis to determine vulnerable paths and execute these paths dynamically, their analysis focuses solely on content providers. The call paths in the content providers are much shorter and less complicated than those found in other components used in Android malware. In addition, the event handlers supported by IntelliDroid are used by malicious applications in a greater variety of ways. Therefore, IntelliDroid addresses complications with constraint generation and event injection that would not have been encountered in content providers.

Symbolic execution for Android malware detection has been explored by AppIntent [36], which uses static analysis to identify relevant sections of code to execute. However, the dynamic analysis component is still traditional symbolic execution where constraints are toggled to execute all relevant code. Conversely, IntelliDroid determines the exact call paths to execute in the static analysis component and knows exactly how the input variables must be constrained prior to execution. This greatly reduces the time needed to test the application and execute any suspicious behaviors during run-time.

7.3   Android Static Analysis

The use of static analysis for malware detection has mostly been focused on analysis of the applications. Performing static analysis of the Android framework has its own challenges, as it contains a substantial amount of code. PScout [5] performs static analysis of the Android framework to create a permission map for Android API methods. IntelliDroid uses similar techniques to perform its analysis on the framework, but it focuses on specific execution paths rather than exploring the entire framework. SuSi [28] takes a different approach to analyzing Android code by using machine learning to classify data sources and sinks. The heuristics use for the classification are based on various coding patterns and data-flow patterns through the framework. Although machine learning techniques have not been considered for IntelliDroid's goal of targeted execution, the source and sink methods that SuSi found can be used to expand the list of suspicious behaviors that IntelliDroid detects.

7.4   Android Dynamic Analysis

Although IntelliDroid does not provide a specific dynamic analysis tool, it has been constructed such that any dynamic tool can be used with it. Existing tools such as TaintDroid [17] and DroidScope [35] provide the infrastructure to trace execution during run-time; however, it can be difficult to ensure that all of the code in the application is executed and that the analysis is complete. The targeted execution of IntelliDroid enhances the effectiveness of these dynamic tools by ensuring that any paths to suspicious behavior are executed and subsequently traced by the dynamic tool. Similarly, sandbox tools such as DroidBox [15] can track when suspicious API calls are made as they occur. IntelliDroid can inject events such that these API invocations are actually executed and determine the conditions under which the suspicious activity was triggered.

Chapter 8

Conclusion

Normally, the analysis and extraction of appropriate inputs to cause suspicious behavior in an application is a time consuming manual process. In this paper, we present IntelliDroid, a targeted dynamic execution tool that harnesses the advantages of both static and dynamic analysis to mechanically generate the necessary inputs to trigger suspicious behavior in Android applications. The inherent imprecision of static analysis is mitigated by the dynamic component of IntelliDroid, which executes the suspicious behavior and can be used to confirm or extract more information about the malicious activity. The dynamic execution is enhanced by data collected during the static phase, which extracts the conditions under which the the suspicious activity is triggered. While the constraint extraction and input data construction is similar that used in concolic testing and symbolic execution, IntelliDroid avoids the exorbitant run-time cost of symbolic execution by identifying specific paths leading to the suspicious behavior of interest. Our results shows the IntelliDroid can successfully generate the inputs necessary to trigger suspicious behavior specified by the user in roughly 22.7 seconds on average. IntelliDroid is built to be a versatile analysis tool that can be attached to any existing dynamic tool to improve test coverage and performance.

8.1   Future Work

IntelliDroid is currently in prototype form and can be extended further. The techniques used and the different components implemented for IntelliDroid can be re-used in other tools. For instance, the evaluation presented in Section 5.2.1 shows that the static analysis component of IntelliDroid can be used for widespread analysis of Android applications. Although the techniques used to extract call paths are not particularly novel, future work can re-use this functionality to search for specific actions performed in

the analyzed application. The extracted call paths and constraints can also ease manual analysis, which is especially important for cases where malicious applications obfuscate code and contain circuitous call paths to hide their activity. The infrastructure that was created to support Android-specific control flow and call edges can also be re-used, since there is no Android support within WALA to this date.

As discussed in Section 5.3, the output of IntelliDroid can be expanded and analyzed to differentiate malicious and benign activity. For the purposes of targeted dynamic analysis, the constraints are currently passed to a constraint solver in the dynamic testing phase and serve no other purpose than to ensure that the suspicious call paths are executed. However, via further analysis, they can also provide a more in-depth look into the behavior of an application and perhaps there are certain characteristics in the constraints that distinguish malicious activity from benign activity. These distinguishing features within the constraints can be used as a malware classification criteria to improve current Android malware detection techniques.

As mentioned previously, IntelliDroid does not provide any specific dynamic analysis tools; instead, it can improve the performance of existing dynamic analysis techniques. One particular use-case for IntelliDroid may be to integrate the call path injection mechanism into a taint tracking system such as TaintDroid [17]. Although TaintDroid can accurately determine the propagation of sensitive data, this is only possible if the sensitive data is obtained and all suspicious actions using this data are executed. IntelliDroid can improve this process by ensuring that these actions (i.e., the execution of the sources and the sinks) are executed reliably, regardless of any conditions that an application may impose. By extracting paths to the sources and sinks and executing them, a complete taint tracking analysis can be performed with all possible taints propagated. IntelliDroid can similarly be integrated into existing sandboxing systems such as DroidBox [15], which report any suspicious actions (such as network access or SMS message sending). In order to obtain a complete report from the sandbox, the suspicious actions must be triggered and IntelliDroid can be used to ensure that they are indeed executed. Overall, IntelliDroid can be used in a variety of ways to enhance current analysis techniques for Android applications and Android malware.

Bibliography

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

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

[3] Apkparser. `http://code.google.com/p/xml-apk-parser/`. Accessed: September 2014.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 29. ACM, 2014.

[5] K. W. Y. Au, Y. F. Zhou, Z. Huang, and David Lie. PScout: Analyzing the Android permission specification. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, October 2012.

[6] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic exploit generation. In *Proceedings of the 18th Symposium on Network and Distributed System Security (NDSS)*, pages 59–66, 2011.

[7] 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 Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 81–92, New York, NY, USA, 2012. ACM.

[8] 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*, pages 143–157, 2008.

[9] 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*, pages 413–425. ACM, 2010.

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

[11] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[12] 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*, pages 239–252. ACM, 2011.

[13] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–278, 2011.

[14] 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.

[15] Anthony Desnos and Patrik Lantz. DroidBox: An android application sandbox for dynamic analysis, 2014. `https://code.google.com/p/droidbox/`, Last Accessed Oct, 2014.

[16] Dex2jar. `https://code.google.com/p/dex2jar/`. Accessed: September 2014.

[17] 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 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–6, October 2010.

[18] Ericsson Mobility. Interim update: Ericsson mobility report, February 2014. `http://www.ericsson.com/res/docs/2014/ericsson-mobility-report-february-2014-interim.pdf`.

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

[20] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/˜ avik/projects/scandroidascaa*, 2009.

[21] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. *AndroidLeaks: automatically detecting potential privacy leaks in Android applications on a large scale.* Springer, 2012.

[22] 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 '05, pages 213–223, New York, NY, USA, 2005. ACM.

[23] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *Proceedings of the 15th Symposium on Network and Distributed System Security (NDSS)*, pages 151–166, 2008.

[24] 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*, pages 281–294. ACM, 2012.

[25] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, Junbum Shin, and SWRD Center. Scandal: Static analyzer for detecting privacy leaks in android applications. *MoST*, 2012.

[26] 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 conference on Computer and communications security*, pages 229–240. ACM, 2012.

[27] Aravind MacHiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[28] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *2014 Network and Distributed System Security Symposium (NDSS)*, 2014.

[29] Robotium, 2014. `https://code.google.com/p/robotium/`.

[30] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In Michel Wermelinger and Harald Gall, editors, *ESEC/SIGSOFT FSE*, pages 263–272. ACM, 2005.

[31] Vanja Svajcer. Sophos mobile security threat report, 2014. `http://www.sophos.com/en-us/medialibrary/PDFs/other/sophos-mobile-security-threat-report.pdf`.

[32] Testing fundamentals, 2014. `http://developer.android.com/tools/testing/testing_android.html`, Last Accessed Oct, 2014.

[33] Uiautomator, 2014. `http://developer.android.com/tools/help/uiautomator/index.html`, Last Accessed Oct, 2014.

[34] Watson libraries for analysis. `http://wala.sourceforge.net`. Accessed: September 2014.

[35] Lok-Kwong Yan and Heng Yin. DroidScope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21th USENIX Security Symposium*, pages 569–584, 2012.

[36] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. AppIntent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, pages 1043–1054, 2013.

[37] 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 & communications security*, pages 611–622. ACM, 2013.

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

[39] Yajin Zhou and Xuxian Jiang. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Symposium on Network and Distributed System Security (NDSS)*, 2013.