Abstract

Correlating Sensitive Behaviours with User Interaction on Android

Mariana D’Angelo
Master of Applied Science
Graduate Department of Electrical & Computer Engineering
University of Toronto
2017

In recent years, ransomware and private information leakage have become more prevalent on mobile devices. Distinguishing such applications from benign ones that make use of similar functionality is challenging because they make use of the exact same APIs. In order to disambiguate these uses, a security analyst may wish to consider whether the user was aware of and consented to the sensitive behaviour. This thesis seeks to aid an analyst making this determination by correlating sensitive behaviours with user interaction. In order to assess user awareness, an analyst should consider all paths to a sensitive behaviour. A path to a sensitive behaviour should be preceded by some UI interaction or a read from persistent storage written to as a consequence of UI interaction. A technique capable of correlating sensitive behaviours with preceding UI interaction has been implemented and evaluated on a suite of synthetic applications.
Para mi abuela Mecha quien siempre me dio el animo y la fuerza para seguir adelante.

Te quiero mucho.

Acknowledgements

My supervisor David Lie for giving me the opportunity and helping me bring this thesis to fruition. You have offered thoughtful advice since I started my Masters and I’m a better scientist because of it. The financial support offered by the University of Toronto and the Bell Graduate Scholarship for my research. My brilliant lab mate Michelle Wong who was always willing to help me solve problems and listen to my ramblings, this project wouldn’t exist without your amazing work developing IntelliDroid. A big thanks to all my other lab mates as well: Peter, Sukwon, Ben, Diego, Wei, James, Afshar, and Shirley.

My family for suffering through constant emails and phone calls of me asking for advice about my research and revisions to this thesis. My sister Maria has been my rock through all of the hurdles of grad school, always willing to hear me out and offer good advice with the benefit of experience. My father Ricardo for going through every section of this thesis with a fine toothed comb and taking the time to understand every sentence. My mother Elizabeth for being the best role model a young woman could ask for, with you in my life I’ve never doubted how far I can go. My brother Ricky for offering his support and love even with the distance between us.

My second family: Gorana, Mark, Goran, Branka, Milan, Baka, and Ujak. You’ve helped me be happy (not to mention well fed!) over the course of my Masters. Volim vas sve!

My “adoptive” sisters Patricia, Elise, and Sarah. Patricia, you’ve given me some of the most useful advice imaginable throughout graduate school (and even before!). Elise, you’ve been yourself: supportive, kind, and always willing to go to a rock concert. Sarah, you’ve been there to help me unwind by watching Christmas movies in February while making (an attempt at) buttercream icing in the middle of the night!

My friends Jess, Vuk, Tim, Mahtab, Sylvia, Viola, and Ashley for supporting in me and believing in me even when I wasn’t sure that I’d make it. You guys have made my Masters fun as well as educational, can’t wait to see what you guys have in store for the next four years of my Ph.D.
# Contents

1 Introduction ................................................. 1
   1.1 Malicious API Usage .................................. 1
   1.2 Sensitive Resources .................................. 3
   1.3 Correlating Sensitive Behaviours with User Interaction . . 4
   1.4 Contributions ........................................ 6
   1.5 Thesis Structure ...................................... 7

2 Background .................................................. 8
   2.1 Android Operating System ............................... 8
      2.1.1 Permissions ...................................... 8
      2.1.2 Components ..................................... 9
      2.1.3 User Interface .................................. 10
      2.1.4 Persistent Storage ................................. 11
   2.2 Program Analysis ...................................... 12
      2.2.1 Types ............................................ 12
      2.2.2 IntelliDroid ..................................... 13
      2.2.3 Constraint Extraction ............................. 13
      2.2.4 Event Chain Generation ............................ 14

3 Design ....................................................... 15
   3.1 Correlating Sensitive Behaviours with User Interaction . . 15
      3.1.1 No UI Flow ...................................... 16
      3.1.2 Explicit UI Flow ................................ 16
      3.1.3 Implicit UI Flow ................................ 18
   3.2 Path Identification ................................... 19
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3</td>
<td>Constraint Extraction</td>
<td>20</td>
</tr>
<tr>
<td>3.4</td>
<td>Granularity of Analysis</td>
<td>21</td>
</tr>
<tr>
<td>3.5</td>
<td>Comprehensive Dependency Stitching</td>
<td>22</td>
</tr>
<tr>
<td>3.6</td>
<td>Persistent Storage</td>
<td>25</td>
</tr>
<tr>
<td>3.7</td>
<td>Recursive Dependency Resolution</td>
<td>28</td>
</tr>
<tr>
<td>3.8</td>
<td>Pruning Infeasible Event Chains</td>
<td>28</td>
</tr>
<tr>
<td>4</td>
<td>Implementation</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>Targeted Path Analysis</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Write Operation Identification</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>Comprehensive Dependency Stitching</td>
<td>31</td>
</tr>
<tr>
<td>4.4</td>
<td>Recursive Dependency Resolution</td>
<td>32</td>
</tr>
<tr>
<td>4.5</td>
<td>Pruning Infeasible Event Chains</td>
<td>33</td>
</tr>
<tr>
<td>4.6</td>
<td>Reduced Precision Call Graph</td>
<td>34</td>
</tr>
<tr>
<td>5</td>
<td>Evaluation</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>Synthetic Cases</td>
<td>36</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Single Flows</td>
<td>36</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Multiple Flows</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Experiments on PlayDrone</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>Limitations</td>
<td>41</td>
</tr>
<tr>
<td>6.1</td>
<td>User Interaction and User Consent</td>
<td>41</td>
</tr>
<tr>
<td>6.2</td>
<td>Limitations of Static Analysis</td>
<td>41</td>
</tr>
<tr>
<td>6.3</td>
<td>Obfuscation &amp; Google Play Services</td>
<td>43</td>
</tr>
<tr>
<td>7</td>
<td>Related Work</td>
<td>44</td>
</tr>
<tr>
<td>7.1</td>
<td>IntelliDroid</td>
<td>44</td>
</tr>
<tr>
<td>7.2</td>
<td>User Intention</td>
<td>44</td>
</tr>
<tr>
<td>7.3</td>
<td>Privacy Analysis on Android</td>
<td>46</td>
</tr>
<tr>
<td>7.4</td>
<td>Symbolic Execution</td>
<td>46</td>
</tr>
<tr>
<td>7.5</td>
<td>Scalable Symbolic Execution</td>
<td>47</td>
</tr>
<tr>
<td>7.6</td>
<td>Bug Finding</td>
<td>48</td>
</tr>
</tbody>
</table>
8 Future Work

8.1 Google Play Services Deobfuscation ........................................ 49
8.2 Integration with IntelliDroid’s Dynamic Component .................. 49
8.3 Semantic UI Analysis ......................................................... 50

9 Conclusion ............................................................................. 51

Bibliography ........................................................................... 52
Chapter 1

Introduction

Smartphones have proliferated since being integrated with capacitive multi-touch technology in 2007. In Q4 2016, 432 million smartphones were shipped to end users representing a 7% increase over Q4 2015 [1]. Google’s Android platform dominates the smartphone market with 81.7% of the market, compared to the 17.9% attributed to Apple’s iOS [1]. Unfortunately, having a large user base also makes Android a prime target for malware authors; in 2015, PulseSecure reported that 97% of all mobile malware to date was targeted towards Android [2].

1.1 Malicious API Usage

Traditionally, the most common form of malware on Android involved sending premium SMS messages, which incurred charges to the device owner and sent a portion of the funds to the malware author; this was a lucrative way of monetizing malware. In response to premium SMS malware and the Android malware statistics, Google has consistently improved the security of Android by introducing protections such as prompting the user when an SMS is being sent to a premium number; the user is now able to prevent a premium SMS message from being sent in the background. As such, premium SMS malware has become less lucrative and malware authors have sought alternative monetization strategies. McAfee predicts that 2017’s greatest threat on mobile devices will be ransomware [3], which locks a user out of their device. Unsolicited app installation has also become increasingly popular; a malicious party gets a user to install a seemingly benign application on their device and proceeds to install other applications, without the user’s consent, which generates revenue.
Chapter 1. Introduction

(a) Reveton Ransomware [4]

(b) Facebook Chat Heads [5]

Figure 1.1: Examples of UI Overlay API usage

The challenge in identifying these newer forms of malware is that the application programming interfaces (APIs) used in such malware are not inherently bad and can also be used for benign purposes. For example, the Reveton ransomware shown in Figure 1.1a makes use of APIs to create UI overlays in order to lock a user out of their device; the user is unable to interact with anything other than the

(a) Enabled

(b) Disabled

Figure 1.2: Facebook Chat Heads Settings [6]
malicious app and must pay a fee to remove the overlay. Facebook Messenger uses the same APIs for its Chat Heads feature, shown in Figure 1.1b, which allows users to continue conversations outside of the Messenger app by overlaying pictures of contacts on top of other apps.

Malicious API usage can occur in a variety of scenarios when the user is not aware of the action: sending premium SMS messages, posting to social media accounts, sending emails, etc. In order to distinguish these two uses of the same API, one may consider whether the user was aware of (and consented) to the use. As shown in Figure 1.2, Facebook provides a mechanism for enabling or disabling the Chat Heads feature at will. Due to the nature of ransomware forcing the user to pay to unlock their device, it provides no such mechanism.

1.2 Sensitive Resources

A “sensitive resource” on a mobile device may refer to either device hardware (e.g., camera or microphone) or private information on the device (e.g., location or contacts). A privacy leak occurs when private information is accessed and sent off device without the consent of the user.

In 2015, the HTTP traffic of 55 popular apps from the Google Play Store was analyzed and it was found that 73% of Android applications leak personally identifiable information, such as email addresses or phone numbers, to third parties [7]. Privacy leaks can also occur as a result of routing mobile traffic through cloud providers. A 2016 study found that of the 20 million Android transactions processed through the Zscaler cloud service, 0.3% (60,000) resulted in some form of privacy leakage; 39% of these transactions were location leaks and 3% were leaking personally identifiable information (e.g., phone number, email address) [8].

Accessing other kinds of sensitive resources, such as the device camera or microphone, without the user’s consent is also cause for concern. In 2015, remote access trojans were found on the Google Play store [9] by researchers. Remote access trojan’s can compromise the device such that audio and video captured from device microphones and cameras are uploaded to a server under the control of the attacker.

It is important to note that unauthorized sensitive resource accesses are not solely the result of ill-intent on the part of the programmer; they can also occur due to negligence. A programmer may make errors during development that lead to private information being leaked or they may simply not be aware of the issues (and laws) surrounding the access to private information. Similarly, a programmer may inadvertently record audio or video due to bugs, though this is a more severe breach of privacy and is much less likely to occur accidentally.

Ascertaining whether or not accesses to sensitive resources are authorized is challenging. Similar
to the use of UI overlays in ransomware, one can correlate such sensitive resource accesses with any UI interactions. An analyst may then consider whether the user was aware of (and consented) to the sensitive resource by a particular application.

1.3 Correlating Sensitive Behaviours with User Interaction

A “sensitive behaviour” may refer to the use of an API that may be employed for malicious or benign purposes (e.g., UI overlays), or to the access of private information (e.g., device camera or location). To assess user awareness of (and perhaps infer consent to) such sensitive behaviours, a security analyst may find it useful to correlate sensitive behaviours with any preceding user interactions. For example, Figure 1.3 shows an application that asks the user for permission prior to accessing the device location.

![Figure 1.3: An application asking for permission to use the device location](image)

A “path” refers to a series of method invocations that result in a sensitive behaviour being triggered. In order to assess the awareness of a user, a security analyst should consider all paths to the sensitive behaviour of interest because some paths may involve user interactions, while others may not. Once all paths have been identified, any user interface (UI) interactions that “guard” the behaviour can be extracted. This thesis identifies three types of UI interactions that may precede a sensitive behaviour and refers to them as “flows”. A “background component” is defined as component with which the user cannot interact, a “foreground component” is one with which a user may or may not interact.
Algorithm 1 ExplicitUI Pseudocode

1: procedure FOREGROUNDCOMPONENT()
2:    dialog_selection ← getDIALOGRESPONSE()
3:    if dialog_selection = true then
4:        performSENSITIVEBEHAVIOUR() ▷ Occurs only as a result of UI interaction
5:    end if
6: end procedure

In Explicit UI flows, a user is asked for consent immediately prior to a sensitive behaviour; they are prompted every time it occurs. Algorithm 1 depicts how a UI interaction can be correlated with a sensitive behaviour (e.g., accessing device location): the behaviour occurs only as a direct consequence of user interaction and must occur in a foreground component. An analyst would likely consider a path that is an Explicit UI flow as being benign.

Algorithm 2 ImplicitUI Pseudocode

1: procedure FOREGROUNDCOMPONENT()
2:    dialog_selection ← getDIALOGRESPONSE()
3:    if dialog_selection = true then
4:        setPERSISTENTSTORAGEVALUE(true) ▷ Save result of UI interaction
5:    end if
6: end procedure
7: procedure FOREGROUNDORBAC GROUNDCOMPONENT() ▷ Executed any time after foreground
8:    saved_value ← readPERSISTENTSTORAGEVALUE()
9:    if saved_value = true then
10:       performSENSITIVEBEHAVIOUR() ▷ Occurs depending on prior UI interaction
11:    end if
12: end procedure

In Implicit UI flows, a user is asked for consent once and the response is saved to persistent storage; sometime later, the setting is read and the sensitive behaviour occurs. Implicit UI flows transmit consent via persistent storage and imply consent for the action to be taken any number of times. Algorithm 2 illustrates how user consent may be propagated using persistent storage: the user interacts with a foreground component and their preference is saved, sometime later a foreground or background component can access this stored preference and perform the sensitive behaviour depending on the value. An analyst would likely consider a path that is an Implicit UI flow as being benign.

Algorithm 3 NoUI Pseudocode

1: procedure FOREGROUNDORBAC GROUNDCOMPONENT()
2:    performSENSITIVEBEHAVIOUR() ▷ No preceding user involvement
3: end procedure
Finally, in No UI flows the sensitive behaviour occurs directly without any user involvement or consent. Algorithm 3 illustrates this type of flow: the action occurs without any user interaction in any component (i.e., does not matter if it occurred in the foreground or background). An analyst would likely consider a path that is a No UI flow as being malicious. Note that the API used to generate UI overlays can pertain to different flows: Facebook requires that the Chat Heads feature be explicitly enabled by the user (Implicit UI since the setting is saved) whereas ransomware does not (No UI).

This thesis hypothesizes that it is possible to detect whether there is a UI interaction occurring in conjunction with a sensitive behaviour using a static analysis tool. A security analyst could use information about the different kinds of flows that lead to a sensitive behaviour as a means of inferring whether the user was aware of an intended for the behaviour to occur. In order to discover Implicit UI flows, paths with reads from persistent storage must be associated with paths that write to persistent storage; if the write was guarded by some UI interaction then it is an Implicit UI flow, otherwise it would constitute a No UI flow. This thesis presents techniques to distinguish different kinds of flows by “stitching” together the reads from and writes to persistent storage and identifying any UI interactions that occur along the resulting paths.

1.4 Contributions

This thesis makes three contributions:

1. A classification system for correlating sensitive behaviours with user interaction based on flows, which introduces the notion of propagating user consent via persistent storage.

2. An approach for generating all possible paths to sensitive behaviours (provided by a security analyst) through a technique known as “Comprehensive Dependency Stitching”.

3. An implementation called HydraDroid\(^1\), which pairs Comprehensive Dependency Stitching with tools that assess boolean satisfiability (i.e., SAT solvers) for improved precision.

4. An evaluation on synthetic apps demonstrating the ability to correctly identify different types of flows and discussions on how to interpret mixed-flow apps. A short discussion of experiences applying the tool to a series of real Android applications is also provided.

\(^1\)The “Hydra of Lerna” is a monster in Greek mythology, which was said to have regenerative abilities: for every head cut off the hydra, two heads would take its place.
1.5 Thesis Structure

Chapter 2 provides relevant background on the Android platform and the program analysis tool upon which this thesis is based. Chapter 3 explains the problem of correlating sensitive behaviours with user interface interactions and describes the design of a technique to solve this problem. Implementation details are discussed in Chapter 4, the evaluation is described in Chapter 5, and limitations of this work are discussed in Chapter 6. Chapter 7 compares HydraDroid to existing works. Finally, Chapter 8 describes plans for future work before the thesis concludes in Chapter 9.
Chapter 2

Background

2.1 Android Operating System

Android is a popular operating system (OS) for mobile devices released by Google, Inc in 2008. The Android OS consists of a customized Linux kernel and an application framework, which defines APIs used by third party applications. Most of the application development is done using a combination of Java and C/C++ (native code) and is distributed on markets such as the Google Play store.

2.1.1 Permissions

Every Android application must define an XML “manifest” document. The manifest allows an application to register “components” (described below) with the framework so that their “callback” entry-points may be triggered. Additionally, the manifest is used to declare permissions. A permission is used to protect access to sensitive system resources; these include hardware components such as the camera and/or microphone, as well as software APIs to access sensitive data such as location, SMS messages, etc. An application may declare an existing permission in their manifest to request it. An application may also declare a new permission to restrict access to defined components.

Before Android 6.0, all permissions requested by an application were granted upon installation; users were presented with a list of permissions at install time as shown in Figure 2.1a, which did not provide sufficient context or information for the user to make an informed decision. Since the advent of runtime permission requests in Android 6.0, permissions to access a sensitive resource are now requested at runtime the first time an application tries to access the resource in question (as shown in Figure 2.1b).

Permissions are not a good indicator of user consent because they are only requested once: at install
time prior to Android 6.0, or upon first use since Android 6.0. An application may use a permission for benign purposes, but thereafter use of the permission may result in privacy leakage or be used for malicious purposes. For example, consider an application that requests permission to use the device camera and microphone so the user may record videos and apply special effects to them; a user wanting such features would accept such a permission request. After gaining access to the permission, however, the app could covertly record video and audio without the user’s knowledge constituting a privacy breach. Prompting the user to provide an app with a permission once cannot ensure that user agrees with (or consents to) all ways the app may use that permission.

2.1.2 Components

Unlike traditional programs, Android applications do not have a main method. Android is instead event driven; it performs “callbacks” to trigger overridden framework methods defined in a third-party application. These “callbacks” typically trigger four types of components: activities, services, content providers, and broadcast receivers. Of these four types, activities are the only components run in the foreground and are the only components with which a user directly interacts via the user interface (UI). All components implement a series of “callbacks”, often referred to as the “lifecycle”, which act as entry-points to the application.

Activities define the user interface layout for an application and handle user interaction such as taps,
swipes, etc. An application will typically contain one activity for every screen; for example, one activity may be used to select items from a restaurant menu while another activity handles payment. The core activity lifecycle methods are: onCreate, onStart, onResume, onPause, onStop, and onDestroy. When overridden, these methods are called by the framework as part of the state transitions of the application. For example, when a user navigates to the home screen from an app, the onPause lifecycle method is called on the activity that was dismissed.

A service can be used to implement a long running task performed by the application such as downloading a file. A service continues execution even when the application is closed, allowing it to monitor for new events/commands and process them. Commands sent to a service take the form of intents. An intent is a mechanism for inter process communication in Android, which defines a recipient (or recipients), an action to be performed, and optionally data to perform it with. An application may optionally define multiple services to handle different background functionalities. The core service lifecycle methods are: onCreate, onStartCommand, and onStop.

A content provider helps an application manage data and also share that data with other applications. The primary benefit over simply using persistent storage is the ability to securely share data (e.g., using permissions), while abstracting away the mechanism used to store the data. A content resolver implements methods to query, insert, update, and delete data.

A broadcast receiver is similar to the subscriber of a publish-subscribe pattern. Android apps can send or receive broadcast messages when an event of interest occurs; these messages are wrapped in intents. A broadcast receiver is associated with an “intent filter” in the manifest file, which enables it to register for specific events (i.e., an action in an intent). A permission may optionally be attached to a broadcasted message to restrict the applications which may receive it. Multiple broadcast receivers may coexist within an application; for example, one broadcast receiver may register for the BOOT_COMPLETED event to launch as soon as the device boots up, whereas another may register for the SMS_RECEIVED event to process incoming SMS messages. The broadcast receiver callback is onReceive.

2.1.3 User Interface

As described previously, an activity defines the user interface (UI) layout for a screen in the application. A layout may consist of multiple UI elements known as “views”. These layouts are implemented as XML files using elements to describe different components (e.g., buttons, text boxes, lists, etc.) and attributes (e.g., size, position, etc.). A UI element (e.g., a button) triggers a “callback” (e.g., onClick).

\[\text{Note that although some of these method names are common to activities as well, their implementations are specific to services.}\]
when the user interacts with the element. The mapping between elements and callbacks are performed by implementing “Event Listeners” (e.g., `View.OnClickListener`) and registering a listener implementation with a UI element. This registration can be performed either programmatically in the application code, or statically in the layout XML file using the `onClick` attribute.

There are several ways to interact with the user interface in Android. When a user taps the screen, a listener registered to detect “clicks” or “touches” on the element in question may be triggered. Different listeners are called when a user taps an element (e.g., `onClick`), taps and holds an element (e.g., `onLongPress`), double taps an element (e.g., `onDoubleTap`), or precisely detecting when a user presses and releases an element (e.g., `onTouch`). When a user slides their finger along the screen, a listener registered to detect “scrolls” or “flings” may be triggered. These gestures are typically used when interacting with lists or for games. Again, specific listeners are used to implement moving an item to another position on the screen (e.g., `onFling`) or navigating through a list to reveal other elements (e.g., `onScroll`).

“Dialogs” are a type of UI element that are highly relevant for inferring user intention. A dialog is a small window that prompts the user to make a decision (e.g., allow or deny) or input information (e.g., username and password). A dialog is registered programmatically with a listener to handle the resulting clicks depending on the input of the user. The most common dialog, `AlertDialog`, may have up to three buttons: positive, neutral, and negative; each of these buttons has their own listener method. More complex dialogs may use single/multiple choice lists where all elements share the same listener; the framework determines which item was pressed and passes an identifier to the method so that it may disambiguate the press.

2.1.4 Persistent Storage

Android provides several facilities to persistently store data as files, shared preferences, or SQLite databases. Files can be saved to the device as images or data (e.g., as comma separated values (CSV)).

Shared preferences are key-value stores used to save primitive data (e.g., strings, integers). Shared preference values can be read and written by the application. For example, a well-behaved application would prompt the user with a dialog to access their location, then store the user’s decision in a shared preference for future reference in order to avoid repeatedly prompting the user. Shared preference objects are read using “get” methods and modified using an “editor” class; editing enables the use of “put” methods to set the appropriate key-values before “committing” the result.

`SQLite` is a lightweight server-less relational database management system (DBMS) which uses a
single cross-platform file to persist the database contents to disk. The database can be queried and updated by the application - as well as other applications if a content provider layer is added. An application will typically use a database to store information that is more complex (to some degree) than a shared preference. For example, an application that schedules SMS messages to be sent in the future is ill-suited for implementation via a shared preferences due to the structured data that would need to be stored (e.g., recipient phone number, body text, date/time to send etc) as well as the plurality of items (i.e., having multiple SMS messages scheduled to be sent at a given time). SQLite databases objects allow queries, updates, insertions, and deletions.

2.2 Program Analysis

Program analysis describes the processes by which one may analyze the behaviour of programs with regards to properties such as reliability, safety, and correctness. Program analysis can be used by security practitioners to identify vulnerabilities or for quality assurance for the purposes of bug finding. Program analysis is often classified as being static, dynamic, or a hybrid of both.

2.2.1 Types

Static program analysis can be used to analyze and predict program behaviour without actually executing the application; this is either performed directly on the application source code (e.g., Java) or by interpreting some form of the object code (e.g., Java bytecode). As the application is not actually being executed, static analysis is highly scalable. However, many forms of static analysis are computationally undecidable and may result in a high false positive rate; static analyses may produce infeasible paths as deciding which paths in an application can or cannot be executed can be reduced to the “halting problem”, which is known to be NP-complete.

In contrast, dynamic program analysis actually executes programs on real or virtual processors in order to capture information about their behaviour. For dynamic analysis to be effective, high coverage of the program must be achieved in order to trigger as much code as possible; this can be done by using sufficient test inputs. Coverage can also be achieved through dynamic symbolic execution, which makes use of symbolic inputs when analyzing a program. These symbolic inputs can be constrained in order to force exploration (and thus execution) of all paths in a program. Dynamic analysis has very high fidelity and a lower false positive rate. As the application is actually being executed and all path must be explored, dynamic analysis will use more computational resources than static analysis. Dynamic symbolic execution suffers from the “path explosion problem” as the number of paths that must be
explored is exponentially related to the number of branches in the program.

Hybrid analysis seeks to combine the two approaches in order to reap the benefits of each, while mitigating their disadvantages. A common paradigm applied when combining approaches is to extract some information using a static component and then feed that information into a dynamic component. The static component will restrict the space that must be explored by the dynamic component and the dynamic component will prune any paths that it cannot trigger.

2.2.2 IntelliDroid

The implementation of the techniques described in this thesis make use of some features of a program analysis tool known as “IntelliDroid”. IntelliDroid [10] is a hybrid analysis tool that enables targeted analysis of suspicious code in an application. The dynamic component uses information from the static component to trigger behaviours of interest to improve precision by verifying whether the behaviour can actually be executed. IntelliDroid was designed to trigger behaviours in malicious apps and focuses on finding a single feasible path to a behaviour.

The static component of IntelliDroid first generates a list of entry-points used by the application. Recall that an Android application does not have a main method, but rather a series of “callback” methods which serve as entry-points. IntelliDroid uses an entry-point discovery algorithm first described in the CHEX paper [11], which essentially identifies Android framework methods that are overridden by the application.

A call graph of the application is generated using the identified entry-points. The call graph must then be patched to model Android-specific idiosyncrasies. For example, as Android allows control flow between different application components using intents, the call graph must be patched to model such flows.

A traversal is performed from each identified entry-point to API calls of interest (e.g., sendTextMessage), which are specified by the user of the tool. The resulting “call path” contains the sequence of method invocations required to trigger the suspicious behaviour.

2.2.3 Constraint Extraction

As static analysis is prone to reporting infeasible paths, IntelliDroid performs further analysis to prune these out where possible. For example, a method invocation in the call path may be guarded by a conditional branch wherein it is only executed under some condition. In order to make this determination, IntelliDroid employs a technique known as static symbolic execution.
In symbolic execution, a program is explored with symbolic inputs which are initially unconstrained by design. IntelliDroid uses these symbolic values and interprets Java bytecode instructions in the application as ones that can manipulate symbolic values. Whenever IntelliDroid encounters a branch on a symbolic value, it adds the path condition to a set of constraints known as the “path constraint”. Listing 2.1 illustrates. Prior to line 1 there would be no constraints upon the value of \( x \) meaning that it could take on any value. Once IntelliDroid encounters the control flow statement on line 1 however, a constraint \( x = 10 \) would be generated meaning that \( x \) can only take on that value to trigger this path.

```java
1 if (x == 10) {
2     // sensitive resource access
3 }
```

Listing 2.1: Symbolic Execution Example

For each method in a given call path, constraints are extracted from the method body using forward control- and data-flow analyses on its control flow graph (CFG). The control flow analysis determines whether control flow is affected by a conditional branch and, if so, IntelliDroid extracts constraints from the predicate of the branch. The data flow analysis is used to identify dependences between variables.

IntelliDroid next generates an “event” which contains information about the call path and any constraints on it, formatted as a satisfiability modulo theory (SMT) problem. The SMT problem is to determine whether logical formulas are satisfiable, unsatisfiable, or neither (i.e., unknown). There are many existing SMT solving tools available including Z3 developed by Microsoft [12].

### 2.2.4 Event Chain Generation

Once an event has been generated, IntelliDroid performs further analysis to resolve “dependences” of the event. For example, a heap variable may appear in the event constraints, but cannot be assigned the correct value in the event itself. In such a case, IntelliDroid generates a “supporting event” to any places where such a variable is defined. IntelliDroid also generates supporting events based on event handler registrations. For example, a listener cannot be triggered before it is registered, hence the registration point must be identified to ensure that the dependency is satisfied. Supporting events are generated in the same fashion as events (pathfinding, extracting constraints, etc.).

In order to ensure that the dependency is satisfied prior to the targeted event, the support event must be executed first. IntelliDroid defines as “event chain” as an ordered sequence of events to model this requirement. Thus, a supporting event is pre-pended to the event chain of the targeted event.
Chapter 3

Design

3.1 Correlating Sensitive Behaviours with User Interaction

Access to a “sensitive resource”, such as the device camera, may lead to privacy leakage when information from such resources is accessed without user consent. Accessing a sensitive resource such as the camera is not incriminating in and of itself, but if an application records video without user consent this is considered a breach of privacy. Sensitive resources can also take the form of data stored on the device, such as contact information or SMS messages; such accesses should only performed with consent from the user.

Some application programming interfaces (APIs) can be “morally ambiguous”; they can be used for both benign and malicious purposes. Recall the example from Chapter 1, which demonstrated how the API used for generating UI overlays can be used for malicious purposes by ransomware and for benign purposes such as Facebook’s Chat Heads.

A “sensitive behaviour” may refer to the use of a morally ambiguous API (e.g., UI overlays), or to the access of a sensitive resource (e.g., device camera or location). To assess user awareness of (and perhaps infer consent to) such sensitive behaviours, a security analyst may find it useful to correlate sensitive behaviours with any preceding user interactions.

In order to determine whether a sensitive behaviour is benign, a security analyst may wish to consider whether the user was aware of and consented to it occurring. Such a determination is complex due to the difficulty of inferring consent. In order to aid an analyst seeking to decide whether a user is aware of and has consented to a sensitive behaviour, this thesis proposes an approach for inferring consent by analyzing user involvement for paths through an application that lead to a sensitive behaviour.
A single path through the application to a sensitive behaviour is called a “flow”. A path begins at a method in the application that is called by the Android framework and ends at the sensitive behaviour. A flow can be classified based on the relationship between the behaviour and any UI interactions preceding it on the path. Three types of flows are considered in this thesis: (1) No UI (Section 3.1.1), (2) Explicit UI (Section 3.1.2), or (3) Implicit UI (Section 3.1.3).

3.1.1 No UI Flow

In a NoUI flow, the sensitive behaviour is not preceded by any UI interactions on the path. An analyst would likely consider a NoUI flow as a path in which the user is not aware of the behaviour; they may conclude that the flow is not benign.

```java
public class NoUIActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        LocationManager lm = getSystemService(LOCATION_SERVICE);
        Location location = lm.getLastKnownLocation(GPS_PROVIDER);
        // do something with location
    }
}
```

Listing 3.1: No UI Usage

An example of a NoUI flow is shown in Listing 3.1. The listing shows the code for an activity with an implementation of the lifecycle callback that occurs when the activity is created. On line 4, the system service for device location is accessed; accessing this resource requires the ACCESS_FINE_LOCATION and/or ACCESS_COARSE_LOCATION permissions. On line 5, the sensitive resource is accessed to determine the location of the user; if the GPS of the device is currently on then the current location will be reported, else the last known location will be reported. Note that despite the fact that this flow occurs within an activity, the user is not informed of the access and does not consent to the access. This example can also be applied to services or broadcast receivers without loss of generality.

3.1.2 Explicit UI Flow

In an ExplicitUI flow, the path to the sensitive behaviour contains a UI interaction that precedes the behaviour. An analyst would likely consider an ExplicitUI flow as a path in which the user is consulted prior to the behaviour; they may conclude that it is benign depending on the properties of the UI interaction.

An example of an ExplicitUI flow is shown in Listing 3.2. Similar to the previous example, this
Listing contains code that executes upon the creation of an activity. On lines 4 and 5, a dialog is being created with a message asking the user if the application may access their device location. On lines 6 and 7, different implementations of click listeners are instantiated for the positive (i.e., user consents) and negative (i.e., user does not consent) buttons. On line 8, the dialog is shown to the user. If the user does not consent, the click listener on lines 19-24 is triggered; the body of the method is empty to illustrate the fact that location is not being accessed. In reality, any code may exist within the method so long as it does not access the device location. If the user does consent, the click listener on lines 11-18 is triggered and the device location is accessed. Note that this application lets the user make a decision about the use of their private data (and respects it). As UI elements are presented to the user, an instance of explicit UI usage can only exist in an activity and not in background components (e.g., services).

```java
public class ExplicitUIActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        AlertDialog.Builder db = new AlertDialog.Builder(this);
        db.setMessage("Can we track your location?");
        db.setPositiveButton("Yes", new PositiveDialogListener());
        db.setNegativeButton("No", new NegativeDialogListener());
        db.show();
    }

    public class PositiveDialogListener extends DialogClickListener {
        @Override
        public void onClick(DialogInterface dialog, int which) {
            LocationManager lm = getSystemService(LOCATION_SERVICE);
            Location location = lm.getLastKnownLocation(GPS_PROVIDER);
            // do something with location
        }
    }

    public class NegativeDialogListener extends DialogClickListener {
        @Override
        public void onClick(DialogInterface dialog, int which) {
            // do nothing
        }
    }
}
```

Listing 3.2: Explicit UI Usage

Notice that the application may not respect the user's wishes and access their information in either case or when the user says no. The ExplicitUI flow must be further investigated by the security analyst before concluding that the user did authorize the sensitive behaviour. These cases could also be addressed by performing a semantic analysis of the dialog text similar to AsDroid [13]; this is beyond the scope of this thesis, but possibilities for future work are discussed in Chapter 8.
3.1.3 Implicit UI Flow

In an ImplicitUI flow, the path to the sensitive behaviour is control-flow dependent on a read from persistent storage; a path to the write to persistent storage must be preceded by some UI interaction. An analyst would likely consider an ImplicitUI flow to be one where the user is consulted and sometime later the behaviour occurs depending on the user’s preference; they may conclude that it is a benign flow after further analyzing the properties of the UI interaction.

```java
public class ImplicitUIActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        AlertDialog.Builder db = new AlertDialog.Builder(this);
        db.setMessage("Can we track your location?");
        db.setPositiveButton("Yes", new PositiveDialogListener());
        db.setNegativeButton("No", new NegativeDialogListener());
        db.show();
    }
}

public class PositiveDialogListener extends DialogClickListener {
    @Override
    public void onClick(DialogInterface dialog, int which) {
        SharedPreferences sp = getDefaultSharedPreferences();
        SharedPreferences.Editor editor = sp.edit();
        editor.putBoolean("TRACK_LOCATION", true);
        editor.commit();
    }
}

public class NegativeDialogListener extends DialogClickListener {
    @Override
    public void onClick(DialogInterface dialog, int which) {
        SharedPreferences sp = getDefaultSharedPreferences();
        SharedPreferences.Editor editor = sp.edit();
        editor.putBoolean("TRACK_LOCATION", false);
        editor.commit();
    }
}

public class ImplicitUIService extends Service {
    @Override
    public void onStartCommand(Intent intent, int flags, int startId) {
        SharedPreferences sp = getDefaultSharedPreferences();
        if (sp.getBoolean("TRACK_LOCATION") == true) {
            LocationManager lm = getSystemService(LOCATION_SERVICE);
            Location location = lm.getLastKnownLocation(GPS_PROVIDER);
            // do something with location
        } else {
            // do nothing
        }
    }
}
```

Listing 3.3: Implicit UI Usage
An example of an ImplicitUI flow is shown in Listing 3.3. Similar to the previous example, a dialog is created asking for user consent and shown to the user in lines 4-8. If the user consents to the access, the click listener in lines 11-19 is triggered and the Shared Preference with key "TRACK_LOCATION" is set to true. If a user does not consent, the click listener in lines 20-28 will set the same Shared Preference to false. Note that the application is asking the user for consent and saving their preference to persistent storage. Rather than using a dialog, some applications may implement this notion of consent as menu item that the user may enable or disable at will, but from our perspective these scenarios are equivalent.

The listing also shows, on lines 29-41, the code for a service with an implementation of the lifecycle callback that occurs when the service is started (e.g., using an intent). On lines 34 and 35, the Shared Preference with key "TRACK_LOCATION" is read. If the Shared Preference was set to true, the location of the device is accessed; else, it is not. Notice that the location could only be accessed if the user consented and that this information was propagated from the point of consent to the point of access. The service here could be replaced with an activity or broadcast receiver without loss of generality.

Similar to ExplicitUI flows, the existence an ImplicitUI flow does not guarantee that the user is aware of the behaviour since the user’s wishes may not be respected, but rather indicates a relationship between a sensitive behaviour and user interaction. Future work that could further aid an analyst by employing semantic analysis and comparison like AsDroid [13] is described in Chapter 8.

3.2 Path Identification

Android follows the event-driven programming paradigm, wherein program execution is governed by events such as user actions (e.g., UI interaction, button presses), hardware sensor outputs (e.g., GPS), or messages from other threads/processes (e.g., intents). As Android does not make use of a main method, one must identify the “entry points” of Android applications in order to mitigate the issue of infeasible paths due to dead code. Entry points of an application refer to methods called by the Android OS that result in app code being triggered; entry point methods can also invoke other methods. For example, the entry points for an activity include all of its lifecycle methods; these entry points may also invoke other methods in the application.

This analysis makes use of the entry point discovery algorithm pioneered by CHEX [11] and used by IntelliDroid [10]. In order for a method to be considered an entry point it must satisfy four criterion: (1) the class in which it is defined must implement an interface or subclass a class in the Android framework, (2) the method must override a method in the parent class, (3) the method must not be called directly within the application code, and (4) the declaring class must be instantiated at least once in the app
code. This approach may fail to identify entry points in some cases such as if the developer invokes 
the method manually within the application code, however, that invocation would result in the targeted 
method being included in the call graph and the method would thus be analyzed regardless of whether 
it was an entry point.

In order to generate a complete call graph, the analysis must also model Android-specific edges 
such as intents. For example, a component such as an activity may be started using a method that 
starts components, which takes an intent as a parameter: \texttt{startActivity(Intent)}. The call graph 
must be patched with an additional edge from the call site of the component starting method (e.g., 
\texttt{startActivity}) to the correct event handler identified using the target of the intent. Thus, the analysis 
analyzes the uses of the \texttt{intent} object to find invocations to methods which set the recipient of the intent 
such as \texttt{setClass()}, \texttt{setClassName()}, or \texttt{setComponent()}.

Once a call graph has been generated by identifying entry points and patching Android-specific call 
edges, it may be traversed in order to find paths to sensitive behaviours. A depth first search path finder 
is used to traverse the call graph starting from each entry point to find all paths to the invocations of 
methods associated with sensitive behaviours. Such methods must be specified manually by the security 
analyst in a text file provided as input to the tool.

### 3.3 Constraint Extraction

The existence of a path from an entry point to a sensitive behaviour does not guarantee that the path is 
actually feasible (i.e., can be executed). In order to determine the feasibility of paths produced by the 
path finder, constraints are extracted from the path. These constraints are later used to prune infeasible 
paths (see Section 4.5), but may also be used to generate specific inputs to trigger the behaviour if paired 
with a dynamic analysis.

A “call path” is defined as a sequence of method invocations. A method invocation performed by 
method in the call path may be control-dependent on conditional branches contained within the method 
body; thus, the invocation will only occur under certain conditions. Forward control- and data-flow 
analyses are performed on the control flow graph (CFG) of each method in a given call path both within 
a particular basic block as well as across basic blocks. This analysis is similar to program slicing [14], 
which propagates the control- and data-flow dependences for a particular instruction along the flow.

The control-flow analysis determines whether conditional branches affect the execution of the next 
method in the call path. If a branch does affect whether the next method invocation occurs, constraints 
are extracted from the predicate of the branch statement. Symbolic data-flow information is propagated
to identify dependences between variables. Loop support is implemented using the same approach as
traditional data flow analysis: the loop is analyzed iteratively until convergence between the constraint
and the data-flow output is achieved. If there are multiple paths within a CFG to the subsequent method
in the call path, the constraints are combined as a disjunction. The constraints produced by the analysis
are fully context sensitive since the call site along each path is known.

3.4 Granularity of Analysis

The examples in Sections 3.1.1-3.1.3 illustrate flows where there is only one way to trigger the sensitive
behaviour in question. In real apps, however, there are typically many places where the application
performs sensitive behaviours. Listing 3.4 shows an example where the application asks for consent on
lines 4-8. If the user acquiesced, it creates an instance of the LocationHelper class defined on lines
31-36 and calls the accessLocation method; this method accesses the device location on lines 33-35.
This is an example of an ExplicitUI flow. However, the service on lines 24-30 also invokes the same
method (i.e., accessLocation) and thus also accesses the device location; an example of a NoUI flow.

```java
1 public class ExplicitUIActivity extends Activity {
2     @Override
3         public void onCreate(Bundle savedInstanceState) {
4             AlertDialog.Builder db = new AlertDialog.Builder(this);
5             db.setMessage("Can we track your location?");
6             db.setPositiveButton("Yes", new PositiveDialogListener());
7             db.setNegativeButton("No", new NegativeDialogListener());
8             db.show();
9         }
10 }
11 public class PositiveDialogListener extends DialogClickListener {
12     @Override
13         public void onClick(DialogInterface dialog, int which) {
14             LocationHelper lh = new LocationHelper();
15             lh.accessLocation();
16         }
17 }
18 public class NegativeDialogListener extends DialogClickListener {
19     @Override
20         public void onClick(DialogInterface dialog, int which) {
21             // do nothing
22         }
23 }
24 public class NoUIService extends Service {
25     @Override
26         public void onStartCommand(Intent intent, int flags, int startId) {
27             LocationHelper lh = new LocationHelper();
28             lh.accessLocation();
29         }
```
The example in Listing 3.4 demonstrates how multiple types of flows may coexist within the same application. This example looks at multiple invocations to the same method where the location is accessed, but cases where there are multiple places where the location services are accessed can be considered the same way.

Listing 3.4 also illustrates the issue of granularity. In this context, granularity refers to the number of paths to a sensitive behaviour that must be considered. In IntelliDroid, only one path to a sensitive behaviour is required as the goal is triggering suspicious behaviours in malware. If only one path is reported when identifying different types of flows, it will either identify a NoUI flow or an ExplicitUI flow. The former case is incriminating because the user is not involved whatsoever, resulting in no significant loss of precision; the results of reporting both flows or simply the NoUI flow would likely result in the same conclusion being drawn by the analyst (i.e., that the user was not consulted prior to the sensitive behaviour). However, the latter case suggests that the application respects the intention of the user when in reality it does not. If the only flow to the sensitive is an ExplicitUI flow, then the analyst may conclude that the user is always consulted prior to the sensitive behaviour.

In order to address this problem, HydraDroid returns all paths to the invocation of the interesting behaviour, rather than just one. Notice that IntelliDroid did not do the “wrong” thing: the use-case for IntelliDroid was actually to trigger behaviours in malicious apps so it did not matter how many ways there were to reach the behaviour. In contrast, this thesis seeks to analyze benign applications and correlate sensitive behaviours with user interactions; this could aid an analyst seeking to determine whether the user was aware of and authorized the behaviour.

### 3.5 Comprehensive Dependency Stitching

As described in Section 2.2, a path to a sensitive behaviour can be represented as an event chain. This event chain may consist of multiple “events” in order to satisfy dependences along the path. Recall that the analysis considers both data- and control-flow constraints; the former identifies constraints between
variables and the latter identifies constraints due to conditional branches.

A *dependence* arises when a variable appears in the constraints of an event, but the constraint cannot be satisfied by the existing call path. A different *dependency* exists for each variable in the constraint that needs to be set to a specific value to trigger the sensitive behaviour. Listing 3.5 shows a contrived example where location access (lines 6-8) is preceded by a conditional branch where the predicate is \( \text{val} \neq 0 \). On line 2, \( \text{val} \) is initialized to 0; thus location will not be accessed when the application is first opened\(^1\). An event to access the location API can be generated, but it will have a *dependence* on \( \text{val} \). Each *dependence* encapsulates a variable that must be set to a specific value and so there may be multiple *dependences* associated with a single event.

```java
public class DependentActivity extends Activity {

    public int val = 0;

    public void onStart() {
        if (val != 0) {
            LocationManager lm = getSystemService(LOCATION_SERVICE);
            Location location = lm.getLastKnownLocation(GPS_PROVIDER);
            // do something with location
        }
    }

    public void onPause() {
        val = 5;
    }

    public void onStop() {
        val = 10;
    }

}
```

Listing 3.5: Simple Dependence Example

Portions of the applications where *dependences* can be satisfied are referred to as *operations*; these are places where a variable that affects control flow (i.e., forms a *dependence*) is mutated. The app in Listing 3.5 may be paused if, for example, the user receives a phone call while using an app. In that case, line 13 is triggered. If the user switches to another app, then the app is stopped and line 17 is triggered. When the user returns to the application\(^2\), lines 5-9 are executed and the location is accessed. Two *operations* that can satisfy the aforementioned *dependence* are identified; thus the location can be accessed if the application is either paused or stopped.

IntelliDroid would generate a “supporting event” for one of the writes to \( \text{val} \) and insert it into the event chain for the location access. For this example, satisfying the *dependency* using only one of the

\(^1\)When an application is opened, the framework calls the *onCreate*, *onStart*, and *onResume* methods in that order.

\(^2\)We assume the application was not killed in this span of time.
possible operations is not incorrect as there is no real difference between the result of using each of the two operations to satisfy the dependency.

Consider the same example, but with a different implementation of onPause shown in Listing 3.6. The application asks the user if they can access their location when the application resumes on lines 3-7. If the user consents, then \( val = 5 \); if the user does not consent, \( val = 0 \). In this case, the same dependence arises as before, but there are now two operations that can satisfy it\(^3\). The operation on line 17 of Listing 3.5 will create a NoUI flow. In Listing 3.6, the operation line 12 leads to an ImplicitUI flow.

```java
@Override
public void onPause() {
    AlertDialog.Builder db = new AlertDialog.Builder(this);
    db.setMessage("Can we update your location when you come back?");
    db.setPositiveButton("Yes", new PositiveDialogListener());
    db.setNegativeButton("No", new NegativeDialogListener());
    db.show();
}

public class PositiveDialogListener extends DialogClickListener {
    @Override
    public void onClick(DialogInterface dialog, int which) {
        val = 5;
    }
}

public class NegativeDialogListener extends DialogClickListener {
    @Override
    public void onClick(DialogInterface dialog, int which) {
        val = 0;
    }
}
```

Listing 3.6: Different onPause Implementation

Note that accuracy of the analysis is affected detrimentally when only one of the two operations is considered; the security analyst may conclude that the application asks for user consent before performing sensitive behaviours. In order to solve this problem, a technique known as comprehensive dependency stitching was developed; the technique generates a new event chain for every possible operation that can satisfy a dependency.

There are three phases involved in the technique:

1. **Targeted Path Analysis** generates an event for all sensitive behaviours as in the original Intellidroid; these are referred to as “targeted events”. The analysis also identifies dependences using the constraints on the call path.

\(^3\)There are three operations, however \( val = 0 \) does not satisfy the predicate of the conditional branch (i.e., \( val \neq 0 \)).
2. **Write Operation Identification** generates a “supporting event” for every operation in the application and associates them with the variable they modify; these are stored in a map.

3. **Comprehensive Dependency Stitching** creates a work list from the targeted events generated by the Targeted Path Analysis and checks each for dependences. The variable associated with the dependence is extracted and looked up in the map of variables to supporting events; each dependency is associated with a single variable so multiple dependences may exist for a single event, thus the variable for each dependency is looked up in the map. For every supporting event returned, a new event chain is generated and added to the work list.

### 3.6 Persistent Storage

The techniques presented thus far enable one to find all paths to sensitive resource accesses in an application. However, some interesting types of applications will be misclassified using this approach.

For example, consider an SMS scheduling app that allows a user to input a date, time, message, and recipient for an SMS message to be sent in the future. The application has an activity through which a user can input the information and it also has a service that sends the SMS in the background at the appropriate date and time. Typically, the application will make use of a SQLite database to store the message details and access it when sending the SMS.

Another interesting example involves GPS tracking apps; these apps allow friends/family to track one another’s location in real time. Often, a user can see the location of others without necessarily sharing their own location (e.g., a parent tracking their child). Thus, the application has an activity where the user can decide whether they want to share their location. The application also contains a service that periodically accesses the device location and uploads it to a server so it can be shared with their followers. Typically, the application will use a Shared Preference to store this information.

These two types of applications have something in common: they both store a value related to user input into persistent storage. The user is implicitly providing consent and that consent is “transmitted” through the persistent storage. If an analysis used a traditional classification scheme consisting only of NoUI and ExplicitUI flows, an ImplicitUI flow propagating user interaction through persistent storage would be reduced to a NoUI flow. This may result in an analyst incorrectly concluding that the application does not require user consent prior to the sensitive behaviour despite the fact that programmer consulted the saved result of user interaction.

Note that even with the comprehensive dependency stitching, the existing approach cannot handle flows through persistent storage; IntelliDroid currently supports only inter process communication (IPC)
and heap dependency resolution. Thus, persistent storage dependences and operations are specifically targeted by HydraDroid. Persistent storage dependences are different from those used for existing dependency stitching (i.e., heap, IPC) because the static analysis framework can perform heap analysis to identify reads and writes to the same location, but the same cannot be said about persistent storage (a known limitation of static analysis as discussed in Section 6.2).

```java
SharedPreferences sp = getDefaultSharedPreferences();
SharedPreferences.Editor editor = sp.edit();
editor.putBoolean("TRACK_LOCATION", false);
editor.putBoolean("DONT_TRACK_LOCATION", true);
editor.commit();
if (sp.getBoolean("TRACK_LOCATION") == true) {
    LocationManager lm = getSystemService(LOCATION_SERVICE);
    Location location = lm.getLastKnownLocation(GPS_PROVIDER);
    // do something with location
}
```

Listing 3.7: Shared Preference Keys

In Listing 3.7 a static analysis framework can be used to determine that there are two invocations to Shared Preference editing methods on lines 3 and 4, but it cannot decide whether these two edits are modifying the same or different values. Stitching together events that read and write different values does not resolve a dependency. In Listing 3.7, there is a read dependency on line 6 that must be satisfied in order to trigger the location access. There are two write operations on lines 3 and 4, to keys "TRACK_LOCATION" and "DONT_TRACK_LOCATION" respectively. If any operation to a Shared Preference is naively considered a candidate to satisfy the dependency then the path will be reported as being feasible; this is because an editor reference that sets a key to true is found on line 4 and key that is set to true is required on line 6. Definition-and-use (i.e., def-use) analysis is employed to extract the key being modified by the editor or being read from so stitching can more precisely connect reads and writes.

Due to time limitations, databases have not yet been implemented into HydraDroid, but the approach is similar to that applied to Shared Preferences. This is because databases also make use of method invocations to perform queries, insertions, etc. However, analysis on the variable being read or modified by a database operation can be performed with varying levels of precision, where precision refers to the probability of the variable modified by the operation being the same as the variable in the dependency. In a simple approach, a database can be treated as a “blob” and any reads and writes to the same database file would be considered to be to the same variable. Alternatively, the queries used to read and modify the database can be analyzed and more precise information can be extracted about the exact row being accessed (and thus the variable being used).
public class ConsentActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        Button consentButton = (Button) findViewById(R.id.consentButton);
        consentButton.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                showConsentDialog();
            }
        });
    }

    public void showConsentDialog() {
        AlertDialog.Builder db = new AlertDialog.Builder(this);
        db.setMessage("Can we track your location?");
        db.setPositiveButton("Yes", new PositiveDialogListener());
        db.setNegativeButton("No", new NegativeDialogListener());
        db.show();
    }

    public class PositiveDialogListener extends DialogClickListener {
        @Override
        public void onClick(DialogInterface dialog, int which) {
            SharedPreferences sp = getDefaultSharedPreferences();
            SharedPreferences.Editor editor = sp.edit();
            editor.putBoolean("TRACK_LOCATION", true);
            editor.commit();
        }
    }

    public class NegativeDialogListener extends DialogClickListener {
        @Override
        public void onClick(DialogInterface dialog, int which) {
            SharedPreferences sp = getDefaultSharedPreferences();
            SharedPreferences.Editor editor = sp.edit();
            editor.putBoolean("TRACK_LOCATION", false);
            editor.commit();
        }
    }

    public class ImplicitUIService extends Service {
        @Override
        public void onStartCommand(Intent intent, int flags, int startId) {
            SharedPreferences sp = getDefaultSharedPreferences();
            if (sp.getBoolean("TRACK_LOCATION") == true) {
                LocationManager lm = getSystemService(LOCATION_SERVICE);
                Location location = lm.getLastKnownLocation(GPS_PROVIDER);
                // do something with location
            } else {
                // do nothing
            }
        }
    }
}

Listing 3.8: Recursive Dependency Resolution
3.7 Recursive Dependency Resolution

Notice that in HydraDroid’s final phase (Comprehensive Dependency Stitching) a new event chain is generated for every operation that may satisfy a dependence. Thus, dependences can be resolved recursively; that is, HydraDroid can satisfy dependences that result from satisfying a dependence.

The importance of recursively resolving dependences is demonstrated in Listing 3.8. This example contains one activity (ConsentActivity) and one service (ImplicitUIService). The device location is accessed on lines 43-44, but is only executed if the Shared Preference with key "TRACK_LOCATION" is set to true (line 42). Thus, an event that simply starts ImplicitUIService would not be capable of triggering the behaviour because of an unsatisfied dependence. There are two operations that can satisfy the dependence: lines 23-26 set the Shared Preference to true, and lines 32-35 set it to false. The latter operation contradicts the constraint forming the dependence (i.e., $\text{SharedPref\{TRACK\_LOCATION\}} == \text{true}$) and can be discarded. However, there is a second dependence that arises from generating an event to set the Shared Preference to true; the dialog must be shown. The method showConsentDialog on lines 12-18 displays the dialog that ends up mutating the Shared Preference; this method is invoked on line 8 when a button in ConsentActivity is pressed.

If one were to limit dependency resolution to only the targeted event, they may only consider an event chain with two events: (1) dialog press, and (2) service start. However, this event chain would not be feasible because the dialog must first be displayed. Hence, HydraDroid checks each event in an event chain for dependences. As a result, HydraDroid finds a dependence for the dialog to be displayed and generates a new event chain consisting of three events: (1) button click, (2) dialog press, and (3) service start. There are no remaining unresolved dependences for any events so HydraDroid can report the (feasible) event chain.

3.8 Pruning Infeasible Event Chains

Through a combination of comprehensive dependency stitching, recursive dependency resolution, and support for persistent storage HydraDroid is now capable of identifying all paths to sensitive resource accesses including those through Shared Preferences. However, dependency stitching is very naive and cannot necessarily identify contradictions between write operations and read dependences. For example, HydraDroid would report two flows for Listing 3.3: one where the Shared Preference is set to true (line 16) and another where the Shared Preference is set to false (line 25). By inspecting the code it is clear that the dependency states that the Shared Preference must be true (line 33) and thus only one
In order to avoid such egregious false positives, a constraint on the entire event chain is generated that is the conjunction of the constraints on each event within it. Thus, for Listing 3.3 this is the event chain constraint for the operation on line 25:

\[(\text{SharedPref}\{\text{TRACK\_LOCATION}\} == \text{false} \land \text{SharedPref}\{\text{TRACK\_LOCATION}\} == \text{true})\]

and this is the event chain constraint for the operation on line 16:

\[(\text{SharedPref}\{\text{TRACK\_LOCATION}\} == \text{true} \land \text{SharedPref}\{\text{TRACK\_LOCATION}\} == \text{true})\]

The conjoined constraint can now be passed into an off the shelf SAT solver such as Z3 to determine whether a path is feasible or not. If there is a contradiction between the constraints on different events, the event chain can be eliminated as it is not feasible. For the constraint on line 25, Z3 reports that the formula is unsatisfiable suggesting the path is infeasible. For the constraint on line 16, Z3 reports that the formula is satisfiable. Note that this does not guarantee that the path itself is feasible, simply that the constraints on the path are feasible.
Chapter 4

Implementation

The approach described in Chapter 3 was implemented as a tool called HydraDroid\(^1\). IntelliDroid was extensively re-factored by implementing a layer of abstractions to represent concepts such as read *dependencies* and write *operations*. These changes were performed using the same static analysis framework as IntelliDroid, WALA\(^2\). Changes were implemented to accommodate Android *fragments*\(^3\), which are the most common way to create dialogs in Android.

HydraDroid implements a three phase approach illustrated in Figure 4.1.

![Figure 4.1: New Phases of IntelliDroid](image)

\(^1\)The “Hydra of Lerna” is a Greek mythological monster, which was said to have regenerative abilities: for every head cut off the hydra, two heads would take its place. The nature of comprehensive dependency stitching means that for every *dependency* that is satisfied (i.e., head cut off), a new event chain is generated consisting of every *operation* that can satisfy it, which can in turn have new *dependencies* (i.e., regrowing multiple heads).

\(^2\)TJ Watson Libraries for Analysis [15]

\(^3\)A *fragment* is part of the user interface for the *activity* to which it is attached
4.1 Targeted Path Analysis

The Targeted Path Analysis phase identifies invocations to interesting APIs in the call graph, finds paths through the call graph to those APIs, and generates constraints by analyzing the control flow graph (CFG) of each method. The rudimentary dependency resolution for heap and IPC dependences in IntelliDroid was re-factored and moved into the subsequent phases. A “variable” is an object that can be read from (i.e., generating a dependency) or written to (i.e., forming an operation). Whenever an event is created the dependences on that event are extracted and identified by the variable they read and the constraints on its value.

4.2 Write Operation Identification

The Write Operation Identification phase targets two kinds of write operations: (1) put instructions (i.e., heap writes), and (2) invoke instructions (e.g., Shared Preference edits). A visitor is used to generate call paths to put instructions and to relevant invoke instructions; the latter is limited to interesting invocations using persistent storage such as Shared Preferences because not all invocations are write operations. Currently, HydraDroid does not support databases, but any database writes would be identified during this phase by targeting the relevant API methods.

Once an operation has been identified, information is extracted about the variable associated with it and the value that is written to it where possible. For example, a Shared Preference editor invocation will be associated with its key (e.g., "TRACK_LOCATION") and the write expression (e.g., SharedPref{TRACK_LOCATION = true}). Both pieces of information are required to improve the precision of the dependency stitching itself.

HydraDroid stores all write operations in a map of variables to a list of supporting events, which are referred to as the “Variable-Operation-Map”. This map enables fast lookup when trying to find the candidates for resolving a dependency.

4.3 Comprehensive Dependency Stitching

In order to ensure that the dependency stitching is as comprehensive as possible, all candidate operations must be stitched together with a given dependence. Each targeted event identified by the first phase (Section 4.1) is converted into an event chain with only one event. The resulting event chains are added to a work list so that event chains can be added and removed at will. IntelliDroid keeps all paths in memory until the analysis is complete. However, HydraDroid’s write operation identification and the generation
of new event chains when resolving dependences can lead to significant memory pressure. As a result, HydraDroid writes an event chain to disk as soon as all dependences on the event chain are resolved; it can then be removed from the work list and memory consumption is reduced. The importance of being able to add items to the work list is highlighted in the discussion of recursive dependency resolution (Section 4.4).

The work list is iterated over such that all events within a given event chain are processed. For each dependency associated with the event, the variable is extracted and looked up in the Variable-Operation-Map generated by the Write Operation Identification phase (Section 4.2). The resulting events are considered “candidates” and a new constraint is generated that is the conjunction of the write expression and the targeted event constraints. Constraint minimization is performed on the resulting constraint, which can reduce some contradictory constraints to false and remove them from consideration; Section 4.5 describes limitations to this approach and presents a solution.

IPC dependences are handled differently than read dependences due to their nature. For example, an IPC dependency would exist when a user must click a UI element to trigger some behaviour; in order for the element to become visible the relevant activity must first be opened. Events generated to support IPC dependences can be cached due to their nature, hence, each IPC dependency is resolved only once by generating a supporting event and the result is cached in a map where the key is an “IPC Variable”. An IPC Variable is identified by a node in the call graph of the application, meaning that any time an event using that node must be triggered, the same supporting event may be used. This can save much repeated work that would otherwise be inherent to comprehensive stitching.

HydraDroid generates a new event chain for every remaining candidate (including IPC) supporting event; the resulting event chain consists of the original event pre-pended with the supporting event. HydraDroid does this recursively so that this event chain is then stitched into the original event chain from the work list, which may consist of more than one event. To summarize, new event chains are generated every time a dependency is satisfied and these event chains always inherit all events in the event chain that gave rise to the dependence.

### 4.4 Recursive Dependency Resolution

Recursive dependency resolution is necessary when the supporting event, which satisfies a dependency in a targeted event, has its own dependences that must be satisfied. Recall that an event can be loosely defined as a call path and the constraints associated with it. When an event is created, read dependences are extracted from the constraints by identifying variables that are not set within the call path itself.
HydraDroid stores these dependences in a set, which encapsulates both heap and Shared Preference dependences.

When generating event chains, HydraDroid iterates over the dependences of each event and generates a new event chain every time a dependency is satisfied. Currently, dependences are considered to be “resolved” if attempts have been made to satisfy them. There are cases, however, where there are no operations that can be stitched with a given dependence. In such a case, HydraDroid conservatively reports the event chain to avoid omitting relevant paths.

In order to ensure that all dependences for an event are processed and avoid repeatedly resolving the same dependences, HydraDroid uses a second set to keep track of the ones that have not been resolved. As dependences are processed, they are removed from this second set. As new event chains are generated (i.e., once a dependency is satisfied), they are added to the event chain work list to be analyzed for further dependences. Once an event has been processed, it will remain unchanged and no new dependences will need to be satisfied. As such, HydraDroid uses the set of unresolved dependences for an event to determine what work needs to be done when stitching.

4.5 Pruning Infeasible Event Chains

As noted in Section 4.3, the constraint minimization performed by IntelliDroid is fairly simple and cannot find all contradictions. This is unsurprising as boolean satisfiability was one of the first problems proved to be NP-complete. In order to reduce the number of infeasible paths produced by HydraDroid, the Z3 SMT solver developed by Microsoft [12] is used to prune event chains.

HydraDroid generates event chains as JSON files where each JSON file represents an event chain. If an event chain has constraints, a labelled folder for those constraints is also generated and populated with Z3 compatible Python programs for the constraints on each event. In order to test the feasibility of the entire event chain, the conjunction of the constraints on each event must be fed into an SMT solver. To that end, when HydraDroid is writing the event chains to disk, it also generates this conjunction as a separate constraint file over the entire event chain.

After HydraDroid has completed, the constraints for each event chain are fed into Z3. Note that Z3 can report one of three conclusions: satisfiable, unsatisfiable, or unknown. Unknown cases are conservatively assumed to be satisfiable and are not pruned. For unsatisfiable cases, the event chains are pruned by deleting the relevant files.

Note that this satisfiability testing could also have performed during HydraDroid’s analysis, however, there is one main challenge with this approach. Z3 does not have publicly documented Java bindings.
and the methods they use require constraints to be generated from scratch (i.e., they are incompatible with those produced by IntelliDroid). As such, one is left to execute the Z3 binary or a Python program from within Java, but this has tremendous overhead and retrieving the standard output is difficult.

### 4.6 Reduced Precision Call Graph

IntelliDroid was designed for the purposes of triggering behaviour in malicious applications. There is typically a notable difference in the call graph size and complexity for a benign application and a malicious application\(^4\). As such, IntelliDroid is unable to generate call graphs for many interesting benign applications from the Google Play Store; there appears to be a notable difference, between benign and malicious apps, in the volume of calls to the Java Software Development Kit (SDK) and the complexity of the implementations of those calls. A more complex SDK method will likely invoke many other methods leading to more time spent creating the call graph, especially if the invoked methods also invoke other methods.

In order to address this problem, HydraDroid can use a reduced precision call graph that does not consider method invocations within the Java SDK. This enables a call graph to be produced within a reasonable amount of time, however, the resulting call graph has issues with null object instantiation. That is, if an SDK method returns an object that it instantiated, the reduced precision call graph will not have identified any methods calls from within the SDK method body (e.g., `init`) and will assume that the resulting object is actually null. If the object is considered to be null, WALA ignores any method invocations on that object. In order to overcome this problem, WALA was customized to take a parameter when generating the call graph. If this flag is set, whenever WALA’s call graph generation encounters a method that appears to be invoked on an uninstantiated (i.e., null) object, it will determine the class the method belongs to and instantiate an object of that class. A class may not be instantiated in certain cases, such as if it is an interface or an abstract class. In such cases, the class hierarchy is used to determine all classes that either implement the interface or subclass the abstract class. The customized WALA will then conservatively instantiate objects for all such implementors/subclasses, thus introducing imprecision to the call graph. Only one of these methods will be the actual target of the call, however, without knowing which class the receiver object corresponds to, the call graph must conservatively insert edges to any potential receivers.

In practice, the result of HydraDroid using a reduced precision call graph is typically a superset of the result for a regular call graph. Due to the conservative instantiation of any class that can potentially

\(^4\)With the exception of repackaged applications.
be the recipient of a method invocation, HydraDroid will report a number of infeasible paths along with
the feasible one. As a result, it is undesirable to always use the reduced precision call graph. Hence,
HydraDroid leverages both call graphs. After extensive experiments, it was noted that if the call graph
was not generated within 30 minutes of the analysis, it likely would not finish. Therefore HydraDroid
has a thread that monitors the progress of call graph creation and switches to the reduced precision call
graph if the call graph is not completed within 30 minutes from when the analysis began.
Chapter 5

Evaluation

5.1 Synthetic Cases

HydraDroid was evaluated on hand developed synthetic cases due the obfuscation of Google Play Services in real applications. This limitation is discussed further in Section 6.3 and future work to address the limitation is described in Section 8.1.

5.1.1 Single Flows

Three Android applications were developed to ensure that HydraDroid can identify single flows in an application without introducing infeasible paths (i.e., false positives).

Single NoUI Flow

This app contains a single activity, which accesses the device location once the lifecycle method onCreate is triggered (i.e., when the app is opened). The user is not consulted at all about this private information access.

HydraDroid correctly generates a single event chain consisting of an “activity event” (i.e., an event which is triggered by opening an activity. Z3 reports that the constraints on the event are satisfiable hence the path is reported. It is important to note that despite being within an Android component associated with UI (i.e., an activity), the access does not involve the user at all. This demonstrates the importance of differentiating interactions with UI elements from interactions with activity components.
Single ExplicitUI Flow

This app contains a single activity with a button labelled “Access Location”. If the user clicks the button this is interpreted as consent and the device location is accessed. Otherwise, nothing occurs.

HydraDroid correctly generates a single event chain consisting of one UI click event and Z3 reports that the constraints on the event are satisfiable. Note that although one may not be able to infer consent due to a button press, HydraDroid reports a wealth of information as illustrated in Listing 5.1. An analyst can choose to attribute different “uiTypes” (extracted from the callback listener) to different levels of user consent.

```json
{
  "startMethod": "explicituisingle.MainActivity.openLocation(Landroid/view/View;)V",
  "targetMethod": "android.location.LocationManager.getLastKnownLocation
  (Ljava/lang/String;)Landroid/location/Location;",
  "eventChain": [
    {
      "path": [
        "explicituisingle.MainActivity.openLocation(Landroid/view/View;)V",
        "invokevirtual \" Application, Landroid/location/LocationManager,
          getLastKnownLocation(Ljava/lang/String;)Landroid/location/Location;
        \"@127"
      ],
      "start": "explicituisingle.MainActivity.openLocation(Landroid/view/View;)V",
      "target": "invokevirtual \" Application, Landroid/location/LocationManager,
        getLastKnownLocation(Ljava/lang/String;)Landroid/location/Location; \"@127",
      "type": "ui",
      "dependences": [
        {
          "type": "ipc",
          "variable": "Application,
            Lca/utoronto/marianadangelo/explicituisingle/MainActivity,
            openLocation(Landroid/view/View;)V"
        }
      ],
      "activities": [ "explicituisingle.MainActivity" ],
      "listener": "android.view.View$1",
      "listenerMethod": "openLocation",
      "uiType": "onClick"
    }
    "packageName": "explicituisingle",
    "mainActivity": "explicituisingle.MainActivity"

Listing 5.1: Information Reported by IntelliDroid
Single ImplicitUI Flow

This app contains two activities: ConsentActivity and UseActivity. When the app is opened, ConsentActivity is created and a dialog is shown to the user asking for permission to access their location. If the user consents, a Shared Preference with key "TRACK_LOCATION" is set to true. If the user does not consent (i.e., selects “no”), the same Shared Preference is set to false. ConsentActivity also has a button labelled as “Access Location”, which sends an intent to start UseActivity. Upon creation, UseActivity retrieves the Shared Preference value for key "TRACK_LOCATION"; if its value is true the device location is accessed, otherwise nothing happens.

HydraDroid initially identifies two event chains. Both event chains contain two events and the second event is opening the activity that reads the Shared Preference. This event has a dependence on the variable SharedPreference{TRACK_LOCATION} and there are two operations that modify this variable. HydraDroid generates two event chains by stitching both candidates. One of these event chains will contain contradictory constraints, expecting the Shared Preference to simultaneously be true and false, but HydraDroid cannot make this determination and generates both possibilities.

When the constraints on the entire event chain are fed into Z3, however, those contradictory constraints are identified as being unsatisfiable and that event chain is pruned. The end result is a single event chain consisting of an event to trigger the operation setting the Shared Preference to true, followed by a second event that reads this value and accesses the device location.

5.1.2 Multiple Flows

Two Android applications were developed to test whether HydraDroid can correctly identify different types of flows within the same application. Although these applications consist of only two flows, we believe that this result generalizes to arbitrary numbers of flows.

NoUI and ImplicitUI

This app consists of one activity (ConsentActivity), two services (ImplicitUIService and NoUIService), and one broadcast receiver (BootReceiver). As with the single ImplicitUI flow, ConsentActivity presents a dialog to the user asking for permission to access their location and stores the result in a Shared Preference. As hinted by the name, BootReceiver registers for boot events for the device. When the device boots, BootReceiver will start both services in the application. ImplicitUIService reads the Shared Preference and only accesses the device location if it is set to true. In contrast, NoUIService accesses the device location directly without consulting the Shared Preference (or, by proxy, the user).
HydraDroid generates three event chains for this application. One event chain consists of a single “boot event” in order to start the NoUIService. The other two event chains consist of two events each where the second event is a “boot event” to start ImplicitUIService. Again, this event contains a dependency on the Shared Preference and there are two candidate operations that may satisfy it. As in the single ImplicitUI example, Z3 determines that the event chain with contradictory constraints is unsatisfiable. The result is two event chains reported by the tool: (1) a boot event, (2) a UI dialog click followed by a boot event.

This scenario illustrates some of the value of this technique; an analyst can discriminate the access to which the user consented from the access the user did not consent to. Although this application would inherently be problematic even without the distinction (i.e., any background usage is bad), the next example illustrates a case where the difference is important.

**ExplicitUI and ImplicitUI**

This app consists of one activity (ExplicitConsentActivity), one broadcast receiver (BootReceiver) and one service (ImplicitUIService). When created, ExplicitConsentActivity displays a dialog to the user asking for permission to access their location. If the user agrees, the device location is immediately accessed and a Shared Preference value is set to true. Otherwise, the Shared Preference value is set to false. When the device boots, BootReceiver will start the ImplicitUIService; the service consults the Shared Preference to determine whether the user consented and accesses the device location accordingly.

HydraDroid generates three event chains for this application. One event chain consists of a “UI event” to accept the dialog and trigger the location access. The other two event chains each consist of one of the “UI events” for both options of the dialog (and thus both Shared Preference writes) and a second “boot event” to start the service that accesses the location based on the Shared Preference value. As in the previous example, Z3 determines that one of these event chains is unsatisfiable and prunes it. The result is two event chains reported by the tool: (1) a UI dialog click, and (2) a UI dialog click followed by a boot event.

This scenario illustrates the importance of considering flows through persistent storage. An analysis that only classifies event chains as ExplicitUI or NoUI may consider this application to be malicious as it would find one of each kind of flow; NoUI flows are quite incriminating when considering user intention and consent. In contrast, by introducing the concept of ImplicitUI flows, this application can be considered to have asked for consent prior to accessing the device location.
5.2 Experiments on PlayDrone

This technique could not be evaluated on interesting Android applications. This is a consequence of the obfuscation of Google Play Services discussed in Sections 6.3 and 8.1. The issue is that accesses to the Google Play Service library location APIs are obfuscated in the app and HydraDroid cannot identify them. HydraDroid can only target the framework location APIs which are rarely used by applications as they have not been updated nearly as often as the Google Play Services APIs.

However, HydraDroid was run on apps from the PlayDrone project [16], a Google Play store snapshot from 2014. 200 apps that requested location permissions in their manifests were collected and analyzed. By dynamically switching the precision of the call graph all apps in the collection were able to be analyzed.

Unfortunately, very few of these apps made use of the framework APIs corroborating the claim that Google Play Services APIs are much more common than framework APIs on the Google Play Store, which is unsurprising. Only 21 of the applications (10.5%) had calls to the framework location APIs. By manually inspecting the paths of 18 of these applications, it was determined that the vast majority of the uses were caused by advertising libraries. This intuitively makes sense as advertisers seek to support as many devices as possible including those that do not utilize Google Play Services. In addition, all paths involving advertising libraries did not include any UI events and hence are classified as NoUI flows. There was only one application, known as StarDroid, which made use of an ImplicitUI flow. However, that application also contained several NoUI flows due to its permissive nature: the app accesses location by default unless the user toggles a menu setting to disable that feature. The notion of apps that “assume” consent in such a manner appears to warrant further study.

These apps could not be manually verified to trigger the behaviours and hence statistics are not reported for accuracy. These apps could likely not be triggered due to deprecated ad library SDKs. It seems that advertising libraries maintain a strict standard for serving ads and will not serve ads to applications using deprecated SDK versions. As a result, any execution through an advertising library immediately aborts due to a lack of connection to the advertising network server. Despite this inability to manually verify, we believe the results of the study are interesting and motivate a few avenues for future work: (1) deobfuscating Google Play Services, and (2) creating a web crawler to download recent applications from the Google Play store to avoid the deprecated SDK issue (potentially by updating PlayDrone).
Chapter 6

Limitations

6.1 User Interaction and User Consent

There is a weak relationship between user interaction and user consent. In this work, it is assumed that user consent can be inferred if the user interacted with the app in some way prior to a sensitive resource access. However, there are a few scenarios in which this may not be true. Consider a dishonest developer who creates a dialog to ask the user if they can access their location, but accesses their location regardless of whether they say yes or no. Another adversary may present a dialog to the user displaying an innocuous message about continuing their game, but then record audio with the device microphone and upload it to the server. In both of these cases, the analysis presented in this thesis would not be able to differentiate these malicious cases from a benign case. Future work will seek to integrate HydraDroid with natural language processing to understand the semantics of the UI elements shown to the user; this is further discussed in Section 8.3.

6.2 Limitations of Static Analysis

Static analysis abstracts a program in order to predict program behaviour, but is inherently imprecise due to lack of runtime information. A classic static analysis problem, known as the halting problem, seeks to determine whether a specific program will terminate with a given input. The halting problem is considered undecidable as their is no algorithm that can solve it for all programs and inputs. Static analysis typically suffers from a high false positive rate because it can report infeasible paths through a program; i.e., at runtime the path may be impossible to trigger, but statically it may seem possible. Deciding whether all paths through a program are actually feasible can be reduced to the halting problem,
and is hence also undecidable. This is a known limitation of static analysis and is considered beyond the scope of this work.

Static analysis also has several other limitations including reasoning about unresolvable information. Loops are a common limitation when termination conditions are not known; this was addressed by IntelliDroid, which “propagates data values that could be assigned by any number of loop iterations” [10]. Another issue with static analysis arises from the use of “reflection”. Reflection allows a program to inspect and modify its behaviour at runtime. For example, Android applications can instantiate classes and invoke methods at runtime as long as the names are known. This is problematic for static analysis as it cannot readily reason about such calls; this limitation was also present in IntelliDroid [10] and is considered beyond the scope of this work.

Android allows developers to mix Java code with C/C++ (herein referred to as “native code”) using the Java Native Interface (JNI). Compiled Java bytecode often contains considerable metadata and high-level features that allow it to be decompiled to produce Java source code; this decompiled source code is often nearly identical to the original source code. In contrast, native code binaries do not contain metadata or high level features and are difficult to analyze statically due to the relative dearth of information. As HydraDroid and IntelliDroid are based upon the WALA static analysis framework for Java, native code cannot be analyzed. Native code may use JNI to hide calls which access sensitive resources, which WALA cannot analyze. JNI may also be problematic when identifying dead code in Android applications due to the complexity involved with entry-points. Recall that Android applications do not have a main method, instead relying on lifecycle callbacks as entry-points to program behaviour. For example, consider a program that uses JNI to transfer execution from Java to native code and then back to Java; seemingly weaving in and out of native code. If the second Java method is not invoked by other points in the program then it may be, mistakenly, considered to be dead code. This is also a limitation of several existing Android static analysis tools such as FlowDroid [17].

There are several Android specific idiosyncrasies such as inter process communication (IPC) that HydraDroid cannot handle fully. For example, HydraDroid and IntelliDroid model the control flow of intents by patching the call graph, but they cannot handle the more complex binder mechanism used by Android to implement any IPC. Additionally, HydraDroid relies on UI related callbacks to be identified a priori in order to correctly classify events as having user involvement. There are no guarantees that the classes and methods identified are exhaustive and as such some UI interactions may be missed.
6.3 Obfuscation & Google Play Services

Google Play Services are a closed-source proprietary background service and API package for Android devices. It has been heralded as a solution to the fragmentation problem of the Android platform, which stems from two main issues: (1) the stark difference between Google’s own Android version for Nexus/Pixel devices and the Android Open Source Project (AOSP), which third party vendors can customize, and (2) the disinterest vendors have in updating Android OS versions for their devices, choosing instead to focus on developing Android OS versions for their latest devices. Google Play Services includes functionality such as single sign-on, location, multimedia, and payments.

Due to the rich amount of information offered by compiled Java bytecode, it is often easy to decompile Android applications back into Java source code. In order to protect their intellectual property, many developers seek to “obfuscate” their code in order to make it more difficult to reverse-engineer. Tools such as ProGuard rename classes, methods, and variables to uninformative identifiers such as a. Google includes ProGuard in their SDK and recommends that it be used for all apps in the Android programming tutorials they make available; as a result, ProGuard is widely used. Fortunately, SDK and framework APIs cannot be renamed allowing static analysis to identify such calls. Unfortunately, however, Google Play Services are bundled into Android applications as JAR libraries, which means that they can be obfuscated in such a fashion.

A method for deobfuscating apps needs to be found before the techniques described in this thesis can be applied to a large number of apps. Due to the extensive use of Google Play Services and obfuscation tools such as ProGuard, HydraDroid’s static analysis is currently unable to identify any calls to obfuscated Google Play Services location APIs and can only target framework location APIs. As a result, even applications that are known to use location (e.g., Uber) may find no accesses to location resources.
Chapter 7

Related Work

7.1 IntelliDroid

This thesis presents an enhanced version of IntelliDroid [10], which was originally conceived as a way to target and trigger malicious behaviours in Android applications, with a focus on malware. HydraDroid differentiates itself in several ways for two main reasons: (1) it targets UI usage, and (2) it can also be used on benign applications. First, this thesis introduces the notion of comprehensive dependency stitching. IntelliDroid produced only one possible path to each targeted behaviour. In contrast, HydraDroid produces all possible paths to a targeted behaviour by considering all the ways dependences can be satisfied. Second, this thesis satisfies dependences through Shared Preferences (a type of persistent storage) improving the fidelity of the path produced and making it more likely to be manually triggerable. Third, this thesis uses constraint solving on event chains to prune infeasible paths prior to any subsequent analysis. Finally, HydraDroid introduces the notion of dynamically switching to a reduced-precision call graph to ensure that even benign applications with large and complex call graphs may be analyzed.

7.2 User Intention

Elish et. al. present a classification method based on features they refer to as “TriggerMetrics” for each API call [18]. Static definition-and-use (i.e., def-use) analysis of sensitive operations within the app is used to identify any user inputs/actions prior to sensitive API calls. The TriggerMetric for an API is then the number of call sites guarded by some user interaction divided by all call sites for the API. They then apply a simple threshold approach to the resulting TriggerMetrics for the classification.
Chapter 7. Related Work

Wolfe et. al. improve the precision of the above work by replacing TriggerMetric thresholds with machine learning classification algorithms [19]. HydraDroid improves upon both approaches in several ways. First, precision is improved by introducing the concept of implicit user interaction via persistent storage (such as Shared Preferences). Second, comprehensive dependency stitching increases the number and type of dependences are satisfied meaning that more information is provided for dynamic analysis. Finally, the result of the static analysis is fed into a constraint solver, thus making the feasibility of the reported paths more likely.

SmartDroid combines static and dynamic analysis to reveal trigger conditions based on user interaction [20]. Static analysis is used to generate an activity call graph; an augmented function call graph with modelling for intents in the Android OS. The dynamic analysis seeks to relate UI “callbacks” to listener registrations and specific UI elements. Due to technical complexity when dynamically triggering UI listeners, SmartDroid handles only a limited set of seven UI interactions. HydraDroid improves upon SmartDroid by targeting all UI interactions, but may be less precise due to the nature of static analysis (discussed in Section 6.2). This tradeoff is mitigated by the use of a SAT solver to prune infeasible paths using constraints extracted by IntelliDroid. As described in Chapter 8, integrating HydraDroid with IntelliDroid’s dynamic component is considered an avenue for future work.

AppIntent uses “event-space constraint guided symbolic execution” to discriminate user-intended privacy leaks from unintended ones [21]. They use static analysis to generate an “event-space constraint graph”, to satisfy inter-process communication dependences. They then mitigate the main problem with dynamic analysis (i.e., path explosion) by guiding the dynamic symbolic execution engine based on paths to interesting behaviours in the “event-space constraint graph”. The symbolic execution engine generates inputs, which are fed into a runtime component based on the Android InstrumentationTestRunner to trigger the desired behaviours. The scope of behaviours that can be triggered dynamically is limited by the functionality implemented by the InstrumentationTestRunner. Thus, HydraDroid can cover a more diverse variety of behaviours due to the lack of a dynamic component. Additionally, if integrated with IntelliDroid’s dynamic component, HydraDroid would be able to trigger behaviours beyond those of AppIntent because it uses a custom version of the Android OS to trigger behaviours.

AsDroid uses static analysis to identify stealthy mismatches between information presented to the user and actions performed by the code [13]. They statically extract two pieces of information related to any interesting API call: (1) the UI listeners in which the behaviour occurs (reality), and (2) text from the UI element (user expectation); AsDroid then performs a semantic comparison between the two to determine mismatches between user expectation and reality. This thesis is considered complementary to such an approach because of its ability to handle implicit UI paths through persistent storage, which are
not handled by AsDroid. HydraDroid simply seeks to associate sensitive API calls with UI interactions, but cannot guarantee that the UI presented to the user is consistent with the behaviour. By applying semantic analysis to the UI, the precision of HydraDroid could be greatly improved.

### 7.3 Privacy Analysis on Android

FlowDroid models privacy leakage within the inter-procedural distributive subset problem (IFDS) framework for highly precise static taint tracking [17]. TaintDroid is a modified version of the Android OS which provides system-wide dynamic taint tracking and analysis [22]. Stevens et. al. performed a study on the privacy implications of ads in Android applications [23]. Their study did not include a notion of user consent in the analysis, instead focusing on the difference between app-specific code and ad libraries bundled with it.

RiskRanker implements a risk based classification which differentiates apps that: (1) exploit platform vulnerabilities (e.g., rooting), (2) cause financial loss or privacy leakage, and (3) leak only device-specific or generic information [24]. LeakMiner uses a context-insensitive static analysis to find privacy leaks in Android applications [25]. AndroidLeaks uses taint-aware slicing to identify potential leaks of private information involving phone information, location data, wifi data, and audio recorded by the microphone [26].

None of these tools differentiates flows involving UI interactions, hence they cannot be used to infer user consent. HydraDroid can tackle a similar issue of privacy, but includes user interactions as a way of inferring which private information the user elects to share.

### 7.4 Symbolic Execution

As HydraDroid gathers constraints along the paths to sensitive behaviours in order to determine feasibility, it may be considered a form of static symbolic execution. Existing implementations of symbolic execution tools such as DART [27] and CUTE [28] make use of heuristics to optimize constraint generation and reduce the time required to solve constraints. EXE [29] and KLEE [30] use dynamic symbolic execution to generate test cases for bug-finding; both tools simplify the resulting constraints so constraint solving can be performed efficiently.

AEG [31] and APEG [32] make use of “preconditioned symbolic execution” to automatically generate exploits for application vulnerabilities. AEG limits input values to cases where a vulnerability may be exploited, which reduces the constraint complexity. Preconditions for AEG must be specified manually,
similar to how HydraDroid requires that an analyst specify the methods associated with sensitive behaviours. APEG takes a similar approach to AEG, but assumes that a patch is available for the exploit and drives testing towards the patched code.

HydraDroid also extracts constraints, feeds them into a constraint solver, and performs symbolic data-flow analysis, however, HydraDroid is meant to find all paths to a sensitive behaviour rather than achieve high code coverage.

### 7.5 Scalable Symbolic Execution

Under-constrained symbolic execution [33] enables quality checks on arbitrary functions without initializing any data structures or doing environmental modelling. Under-constrained execution allows variables to be explicitly marked as being under constrained, meaning that they can violate preconditions such as pointers not being null. If an error involving an under-constrained variable does not occur for all values of that variable, it does not flag the error. The ability to reduce the scope of symbolic execution to specific functions decreases the size of the constraints, yielding scalability benefits.

The constraints extracted by HydraDroid are typically not so large that they cannot be solved within a reasonable amount of time by Z3. The relatively small size of the constraints may stem from two factors: (1) Android applications may be simpler than desktop applications in terms of complexity, and (2) the event driven nature of Android could mean that paths to sensitive behaviours can essentially take “shortcuts” in contrast to traditional applications which use a `main` method. These shortcuts could be considered similar to under-constrained execution at a high level, since they reduce the amount of code the symbolic execution must analyze.

EXE [29], KLEE [30], and Sage [34] simplify the constraints produced by symbolic execution in order to address the scalability issue of constraint solving. Driller [35] uses selective symbolic execution to alternate between fuzzing and symbolic execution in order to achieve high code coverage while scaling well. Symbolic Program Decomposition (SPD) [36] improves the scalability of multi-path symbolic execution by “exploiting control and data dependences to avoid analyzing unnecessary combinations of subpaths”. SPD can also further scale at the cost of precision by arbitrarily abstracting away symbolic subterms, thus reducing the size of the resulting constraints. HydraDroid makes use of the Z3 SAT solver to prune infeasible paths. HydraDroid inherits techniques to minimize the size of constraints by eliminating redundancies and detecting simple contradictions from IntelliDroid [10].
7.6 Bug Finding

Program slicing [14] seeks to abstract programs during debugging by reducing that program to a minimal form which still produces that behaviour, called a “slice”. ChiSlice [37] uses “dices” for identifying bugs in a program, where a dice is defined as the set difference between two program slices. A dice that isolates the fault is computed by subtracting the program slice that does not trigger the bug from the program slice that does. ChiSlice is able to reduce the set of statements a programmer must analyze when identifying the cause of bugs in a program. Many other tools exist which make use of program slicing for bug finding purposes [38, 39].

Jackson et. Al. [40] gather constraints governing changes of state within a procedure for the purposes of bug finding; the conjunction of the procedure specification and the negation of the constraints yields a formula whose satisfying assignments are executions that violate the specification. ESC [41] is program checker designed to be used during compilation, which finds common programming errors such as null dereferences, array out of bounds errors, etc. ESC also allows the programmer to record design decisions as annotations in the program and generates warnings if the program violates these specifications.

HydraDroid makes use of program slicing much like bug finding, however, in bug finding the relationship between UI interaction and the bug is unimportant as the goal is to triggering the bug in question. As HydraDroid is not intended to be used for testing, it does not require program specifications to define “correct behaviour”; HydraDroid does not make judgements about whether a flow violates user intention, but instead correlates invocations of certain methods with UI interactions for a security analyst.
Chapter 8

Future Work

8.1 Google Play Services Deobfuscation

Deobfuscating Google Play Services is beyond the scope of this work, however, work has already begun on solving this problem. We have discovered that ProGuard can obfuscate class/method/variable names, but it does not modify package names. Additionally, ProGuard only obfuscates classes that are used by the application meaning that unused classes are not obfuscated. We are experimenting with extracting features from an unobfuscated version of the Google Play Services JAR such as constant strings, method signatures, number of methods in a class, etc. We can also extract such features from obfuscated Android applications that make use of the Google Play Services library and we posit that we can identify specific methods/classes using these features. Our preliminary results have shown that the features listed above are often too generic across multiple methods, but if we introduce a call graph analysis to the approach it is expected to disambiguate such cases.

8.2 Integration with IntelliDroid’s Dynamic Component

Although this thesis work seeks to mitigate the imprecision of static analysis as much as possible, there are no guarantees that the paths reported are feasible without actually triggering them at runtime. Work has already begun on implementing UI triggering in IntelliDroid. Although static analysis can typically associate a callback with a UI element identifier, one must still interact with that UI element in order to trigger the callback. Static analysis can typically not determine the coordinates or size of a UI element as these attributes are typically calculated at runtime based on some constraints specified by the developer. As such, work has been done to search Android View hierarchies at runtime in order to
identify the actual View object to which a callback corresponds. Once the View is identified, techniques
similar to those used by Android’s UI Testing framework Espresso can be applied to inject taps, key
presses, etc.

8.3 Semantic UI Analysis

As described in Section 7.2, AsDroid uses semantic analysis to compare text presented to users on UI
elements with the code executed as a consequence [13]. A known limitation of this work is that we
cannot guarantee that the UI element is actually informing the user that a sensitive resource will be
accessed. It is left to the security analyst to determine whether the user interaction actually constitutes
user awareness. In order to further aid analysts performing this analysis, future work may combine
HydraDroid with a natural language processing (NLP) tool. Semantic analysis of UI elements presented
to the user could be compared to the code executed as a result to ensure that the wishes of the user
are respected. An analyst armed with knowledge of the semantics of the UI elements and the path
information may be more accurate when assessing user awareness.
Chapter 9

Conclusion

The problem of determining whether sensitive behaviours constitute malicious behaviour or bugs on
the part of the programmer is difficult. This thesis takes steps toward addressing this problem by
correlating sensitive behaviours with user interface interactions preceding the behaviour. Traditionally
only two classifications have been considered: (1) the user is directly involved (i.e., ExplicitUI), and (2)
the user is not involved at all (i.e., NoUI). This thesis introduces a third classification: the ability to
propagate user interactions (and perhaps user consent) through persistent storage (i.e., ImplicitUI). Such
flows through persistent storage must be reconstructed by stitching: a path that reads a variable from
persistent storage prior to a sensitive behaviour must be connected with any path that writes to the same
variable. An analyst may use the relationships between different instances of the sensitive behaviour
and UI interactions to assess user awareness by considering all flows to that sensitive behaviour. The
techniques described in this work have been implemented in a tool known as HydraDroid and evaluated
on a series of synthetic apps to prove its efficacy. Additionally, a short case study on experiences with
real Android applications has revealed avenues for future work.
Bibliography


