CHARACTERIZATION, DETECTION AND EXPLOITATION OF DATA INJECTION VULNERABILITIES IN ANDROID

BEHNAZ HASSANSHAHI

B.Eng., AUT (Iran), 2010

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

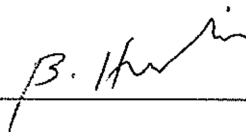
NATIONAL UNIVERSITY OF SINGAPORE

2016

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.



Behnaz Hassanshahi

Wednesday 8th June, 2016

Acknowledgment

First, I would like to express my appreciation to Dr. Roland Yap for supervising me during my PhD journey. He has always been supportive and encouraging. I specially thank him for helping me to develop a critical thinking mindset for solving problems.

I have to thank my husband for motivating me all these years and being with me through the good times and bad times. I would also like to thank my family, specially my parents for always believing in me and helping me to chase my dreams.

To all my friends who have been my second family in Singapore, thank you for being on my side. I cannot list all your names here, but you are always on my mind.

I would like to thank the members of my thesis committee for reading this thesis and giving comments. I would also like to thank the co-authors of my research papers, Dr. Zhenkai Liang, Dr. Prateek Saxena and Yaoqi Jia. It has been a pleasure working with you.

Contents

1 Introduction **1**

1.1 A New Analysis Framework for Android 2

1.2 Web-to-Application Channel in Android 2

1.3 Database Attacks in Android Apps 4

1.4 Contributions and Thesis Organization 4

2 Preliminaries **7**

2.1 Android Applications 7

2.1.1 Android Components 8

2.1.2 Android Manifest File 8

2.1.3 Message Passing in Android 9

2.2 Android Architecture 10

2.3 Android Security 12

3 A Framework for Detection and Exploitation of Vulnerabilities in Android **15**

3.1 Motivating Example 17

3.2 Approach and Design 21

3.3 Source-Sink Pair Identification 24

3.4 Control Flow Graph Construction & Reachability Analysis 25

3.4.1 Control Flow Graph Construction 26

3.4.2 Reachability Analysis 27

3.5 Symbolic Execution and Static Flow Refinement 28

3.5.1 Mitigating the Path Explosion Problem 30

3.5.2 Further Optimizations 34
3.5.3 Interaction with the Environment 36
3.6 Attack Validation and Concrete Value Propagation 38
3.7 Evaluation 40
3.8 Related Work: Analysis of Java and Android Programs 44
3.8.1 Static Information Flow Analysis 44
3.8.2 Symbolic Execution 46
3.8.3 Dynamic Analysis 48
3.9 Summary 48

4 Web-to-Application Injection Attacks on Android 51
4.1 Web-to-Application Injection Attacks on Android 52
4.1.1 Intent Hyperlinks and URI Intents 52
4.1.2 Web-to-App Injection Attacks 59
4.1.3 Categories of W2AI Vulnerabilities 59
4.1.4 A Vulnerable App Example 63
4.2 Web-to-App Channel from a Software Engineering Perspective . . . 64
4.2.1 Browsable Activities as the Main Entry Points 65
4.2.2 Intent Hyperlinks Loaded in the WebViews 66
4.2.3 W2AI Vulnerabilities due to Code Reuse 67
4.2.4 Design Problems of Web-to-App Channel 69
4.3 Detection and Exploitation of W2AI Vulnerabilities 72
4.3.1 Source-Sink Pair Identification for W2AI Attacks 72
4.3.2 Symbolic Execution 72
4.3.3 W2AI Attack Validation 73
4.4 Experimental Results for W2AI Vulnerability Detection 74
4.4.1 Prevalence of W2AI Vulnerabilities in Apps 75
4.4.2 Effectiveness of W2AIScanner in Detecting W2AI Vulnerabilities 76
4.4.3 Reporting Vulnerabilities to Vendors 77
4.4.4 Case Studies 78
4.5 Related Work: Attacks on Android Apps 80

4.5.1 Over-Privileged Malware . . . 80
4.5.2 Privilege Escalation . . . 81
4.5.3 HTML5 and WebView Injection Attacks . . . 81
4.5.4 Web-to-App Injection Attacks . . . 82
4.6 Summary . . . 83

5 Detecting and Characterizing Database Attacks in Android Apps 85
5.1 Overview . . . 86
5.1.1 Public Database Attacks . . . 87
5.1.2 Private Database Attacks . . . 88
5.2 Motivating Examples . . . 89
5.2.1 Vulnerable Public Database Example . . . 92
5.2.2 Vulnerable Private Database Example . . . 93
5.3 Threat Model . . . 94
5.4 Detection and Exploitation of Database Vulnerabilities . . . 94
5.4.1 Source-Sink Pair Identification for Database Attacks . . . 95
5.4.2 Constructing the Control Flow Graph and Reachability Analysis . . . 96
5.4.3 Symbolic Execution for Detecting Database Attacks . . . 97
5.4.4 Database Attack Validation . . . 105
5.5 Experimental Results for Database Vulnerability Detection . . . 109
5.5.1 Database Vulnerability Detection Results . . . 109
5.5.2 Case Studies . . . 111
5.5.3 Comparing DBDroidScanner and ContentScope . . . 112
5.6 Related Work . . . 116
5.7 Summary . . . 118

6 Conclusion and Future Work 121

Bibliography 132

Appendix A Translating Jimple IR to SMT-LIB Format 135

Summary

Android is a popular mobile platform for which a huge number of apps (applications) have been developed during the past few years. However, the complexity in Android programming increases the possibility for developers to introduce vulnerabilities. In this thesis, we present a novel analysis framework to detect and confirm data injection vulnerabilities in benign Android apps. We study two important classes of such vulnerabilities and use our analysis framework to show that many existing apps are vulnerable. As we are able to find many such vulnerabilities, we believe that a significant number of Android apps can be exploited by such attacks.

First, we develop an automated vulnerability detection system for Android apps which not only finds data injection vulnerabilities but also confirms them with a proof-of-concept zero-day exploit. Our tool employs a novel combination of static dataflow analysis and symbolic execution with dynamic testing. We also use several optimizations to tame the path explosion problem in symbolic execution. We show through experiments that this design significantly enhances the detection accuracy compared with an existing state-of-the-art analysis.

Next, we present a detailed study of a new class of application vulnerabilities in Android that allows a malicious web attacker to exploit app vulnerabilities. It can be a significant threat as no malicious apps are needed on the device and the remote attacker has full control on the web-to-app communication channel. Analyzing real apps from the official Google Play store – we found many confirmed vulnerabilities which suggest that these attacks are easy to mount and developers do not adequately protect apps against them.

Finally, we conduct a systematic study of the attacks targeting databases in benign Android apps. We present a comprehensive classification of database attacks.

These attacks can be triggered either from content providers or intents received throughout the app. In order to detect and exploit zero-day database vulnerabilities, we utilize our analysis framework and extend it with models for symbolically executing operations on the URI-based objects that are involved in database management. We evaluate our analysis framework by analyzing real-world Android applications and generating the corresponding proof-of-concept exploits. We find both public and private database vulnerabilities in real-world apps. We also show new ways to exploit the previously reported and fixed vulnerabilities.

List of Tables

4.1 Security control matrix for W2AI attack in Android development frameworks. 69

4.2 Overall statistics of vulnerable apps in each W2AI attack category. 75

4.3 Representative vulnerable apps for each W2AI vulnerability category. 78

4.4 Sinks and policies/settings for representative apps. 78

5.1 This table shows SMT formulas for modeling methods of URI-based classes. 101

5.2 Overall statistics of apps vulnerable to the database attacks. 110

5.3 Public and private database vulnerabilities in representative apps of [ZJ13]. 113

A.1 Translating representative Java methods to SMT-LIB(v2) syntax. . 136

List of Figures

3.1 The code snippet is chosen from the WorkNet (kr.go.keis.worknet version 3.1.0) app which is vulnerable to data injection attacks. This app may obtain parameters from the malicious intents. There are three classes separated by dashed lines: `MainActivity`, `MyWebView` and `MyRunnable`. `MainActivity` is the browsable activity, `MyRunnable` is an inner class of `MainActivity` and methods are shown in boxes. 18

3.2 CFG for the motivating example in figure 3.1. The gray box contains the lifecycle methods of the `MainActivity`. This graph contains a call cycle which is painted in red. The `IrrMethod()` method represents irrelevant methods which do not affect the data dependency analysis but contribute to long paths. The dashed arrows are not original edges in the CFG. They summarize the methods and immediately connect the callsite to the successor statement. The `p` statements represent conditional statements (predicates). 20

3.3 Analyzer Architecture 21

3.4 We have randomly chosen 200 applications vulnerable to data injection attacks. (a) Shows the number of conditional statements with data dependency on input on paths that reach data injection vulnerabilities. (b) Compares the number of source-sink pairs that analysis has to iterate over with (FlowDroid Pairs) and without (Total Pairs) FlowDroid. 22

3.5 The dashed boxes contain the CFG of a method. (a) Immediate post-dominator (ImPodm) inside the method is marked as a merge point. (b) If method does not contain an immediate post-dominator but both branches of the If statement are reachable to the sink statement, we create a dummy immediate post-dominator. 32

3.6 Ratio of number of paths generated by our analysis framework and vanilla FlowDroid. 41

3.7 Number of missing edges in the initial CFG which were found and added by our analyzer. All of these apps have at least one potential vulnerable sink. Apps are sorted based on the ratio in Figure 3.6. . 42

3.8 FD_sinks are number of FlowDroid false positive sinks and new_sinks are number of new vulnerable sinks found by our analyzer. Apps are sorted based on the ratio in Figure 3.6. 42

3.9 Total execution time for static analysis in seconds. Apps are sorted based on the ratio in Figure 3.6. 43

4.1 This figure shows how an intent hyperlink starts a benign app which has been granted sensitive privileges by the user. 55

4.2 W2AI attacks on Android apps. 1) A user clicks a malicious link that redirects to the attacker's site in her mobile browser. 2) The site loads the malicious intent hyperlink in an iframe or a new tab. 3) The browser parses the hyperlink, generates the URI intent and launches the corresponding activity in the vulnerable app. 4) Therefore, the payloads derived from the URI intent running in the app can access the user's private information or perform privileged operations on behalf of the app. 58

4.3 (a) The original activity used for financial purposes. (b) Injected URL loaded by the App. This application uses the same UI framework for different activities. 62

4.4 (a) The original or phishing page in WorkNet. (b) The malicious page running in the `WorknetActivity` steals the user's private data (e.g., device information, contacts, local files and geolocation), sending it to the attacker's server. 63
4.5 This chart shows the percentage of lines of code exposed to web by browsable activities. 66
4.6 Two-way communication between the Android app and web. 71
4.7 Attacks in Android smartphones can be classified to four categories: (a) over-privileged malware; (b) privilege escalation; (c) injection attacks in HTML5 and WebView attacks; and (d) web-to-app injection attacks; 80

5.1 Database attack scenarios: CP API stands for Content Provider API. 87
5.2 SFT for `UriMatcher.match(Uri)`: ϕ is the path constraint; the fields of the registered and argument URI are denoted by $f_{1,i}$ and $f_{2,i}$ respectively; c_1 checks the constraints for the authority and c_2, c_3 and c_4 check the constraints for the path segments of the URIs and d is the default integer registered in `UriMatcher` object. 102
5.3 SFT for `Uri.compareTo(Uri)`. The label for each transition is a constraint (c_i) for a particular field of the base and argument URIs: c_1 for scheme, c_2 for userinfo, c_3 for host, c_4 for authority, c_5 for port, c_6 for path, c_7 for query pairs, c_8 for fragment. 104

Chapter 1

Introduction

Android is a popular mobile platform for which a huge number of apps (applications) have been developed during the past few years. It offers many useful functionalities, thereby highly adopted by developers in practice. However, not all these apps are developed with security in mind. Complexity of features in Android has raised many confusions for software developers which has resulted in many vulnerabilities. Attackers may exploit these vulnerabilities to launch devastating attacks.

In this thesis, we focus on data injection vulnerabilities in Android apps through which the attacker can control the input and behavior of the app. These vulnerabilities may be exploited by attackers to launch a wide variety of attacks such as cross-site scripting (XSS), SQL-injection, cross-site request forgery (CSRF), privilege escalation, sensitive data leakage, etc. Such vulnerabilities may exist in SDKs (e.g., PhoneGap [PHO]) which can affect thousands of apps incorporating them. However, understanding the vulnerabilities in a particular SDK does not give the big picture of the problem. Moreover, there are already millions of apps ready for download [GPL] which may suffer from different types of vulnerabilities. Hence, we aim for a systematic approach which helps us to detect, confirm and characterize data injection vulnerabilities in real-world apps.

In order to automate the process of identification and exploitation of data injection vulnerabilities in Android apps, we introduce a new analysis framework. This framework helps us to detect different classes of vulnerabilities and successfully generate working exploits for them. Our analyzer integrates static analysis with dynamic testing and is designed in a way to be suitable for the Android ecosystem.

Another goal in this thesis is to study how some of the popular features and functionalities of Android affects the security of benign apps. Our focus is on two important classes of attacks which abuse the web-to-app channel and databases

implemented in apps. We study and characterize these attacks in detail. Using our analysis framework, we find many apps which are vulnerable to these two attack families and generate proof-of-concepts to exploit them.

1.1 A New Analysis Framework for Android

At the high level, detecting many classes of vulnerabilities can be considered a source to sink reachability analysis for Android apps. In general, pure static information flow analysis techniques would give potential reachability and existing techniques for Android apps [ARF+14, LLW+12, GZWJ12] are no different.

Many analyzers have been developed for Android applications. FlowDroid [ARF+14] and CHEX [LLW+12] are some representative static analyzers tailored for Android applications. FlowDroid is an open source static taint analysis framework built upon Soot [LBHD11] which is often used in academic studies. CHEX is another static dataflow analysis framework designed to find component hijacking vulnerabilities in Android. It builds data-dependence graph (without control dependence) to report potential vulnerable flows in the program. These analysis frameworks do not handle conditional statements in the program and are heavily based on abstractions which result in high false positive rates and reporting infeasible flows. Moreover, they are not adequate for confirming the vulnerabilities.

Our goal is not only to analyze for potential vulnerabilities but also to confirm them. For this more ambitious goal, we want to be able to generate working exploits which actually drive the execution to reach and also affect a security-critical program point, in other words, automatically generating some form of zero-day attacks. This makes the task of understanding and confirming a vulnerability significantly easier for security analysts or the app developers.

To estimate the prevalence of vulnerabilities, we have developed an automated vulnerability detection system for Android apps which not only finds vulnerabilities but also confirms them. Analyzing real apps from the official Google Play store, we found many confirmed vulnerabilities. Our tool employs a novel combination of static dataflow analysis and symbolic execution with dynamic testing. We show through experiments that this design significantly enhances the detection accuracy compared with an existing state-of-the-art analysis.

1.2 Web-to-Application Channel in Android

Presently, both mobile applications and traditional browsers are playing essential roles in users' devices. To provide a seamless experience for users, Android and iOS

support *scheme* mechanism to handle web-to-application inter-operations [ANU, IOS]. This mechanism makes it possible for a web page to invoke an installed app. When a user clicks a web hyperlink of a certain format, Android parses the hyperlink and invokes a specific application. With this technique, the mobile app can be launched by a custom URI on any website in the user's browser. For example, by clicking the phone number in a website, the Phone app will be launched to call the number. This customized service is convenient to users, but it also exposes vulnerable apps to web attackers.

Without the proper sanitization or check on the URI or extra parameters derived from the intent hyperlink, the vulnerabilities in apps may become accessible to the remote attackers. Thus the malicious intent hyperlinks can misuse the vulnerable apps to conduct illegitimate manipulations on the victim's device, e.g., stealing the victim's contacts. In this thesis, we report the first systematic study on *Web-to-App Injection (W2AI)* attacks, which exploit the scheme mechanism to hijack the vulnerable mobile apps.

W2AI forms a new class of application vulnerabilities that *do not* require the user to have installed a malicious app, but merely to have visited a malicious website or advertisement in a mobile browser. Such attacks permit remote attackers to exploit natively installed Android applications, without the risk of publishing malware on application market or enticing users to install malicious applications that request suspicious permissions at install-time. This interface can be a significant threat as the remote attacker has full control on the web-to-app communication channel.

Some previous works discover attacks through scheme mechanisms [WXWC13]. Rui Wang et al. reveal confused deputy attacks on Android and iOS applications which abuse channels provided by mobile OS. One of these channels is the scheme mechanism through which attacker can invoke apps on the phone by crafting intent hyperlinks and publishing on web. They present a CSRF attack on the Dropbox SDK in iPhone [WXWC13] launched through an intent hyperlink. However, our attacks differ because in our attack model, the user clicks on an intent hyperlink in the default browser so it does not need to be started from the benign app and can leverage safer channels like default browsers.

We present the first detailed study of this new class of application vulnerabilities on Android. Moreover, we investigate which vulnerabilities can be exploited once the attacker can manage to start an application via intent hyperlinks. We study the prevalence of these attacks in Android apps and generate exploits for a large number of them.

1.3 Database Attacks in Android Apps

Android apps often include internal databases for data management. We conduct a systematic study of the attacks targeting these databases and detect them in real-world apps. Content provider components are designed to provide shared content among applications. When an application requests to access another application's data using the `ContentResolver` object available in its context, the system delivers the request based on the URI provided in the request. The URI also contains additional data which might be processed by the destination application for different purposes like locating tables in a database. Even though the main functionality of content providers is to insert or query data from other applications, they might be abused by the attackers if not protected carefully.

Developers sometimes implement internal databases without utilizing content providers. Android apps may have private databases typically in the form of `SQLite` databases which cannot be accessed using content providers. These databases can be implemented in any component of an app. Even though such internal databases are usually incorporated into the apps to be used privately, adversaries might be able to launch pollution, data-leakage or file access attacks as described later due to the inherent vulnerabilities in the benign applications.

In order to detect and exploit zero-day database vulnerabilities, we utilize our analysis framework and extend it with mechanisms for symbolically executing operations on the URI-based objects that are involved in database management. We use our analysis framework to analyze real-world Android applications to find and generate working exploits for them.

ContentScope [ZJ13] is proposed for detecting pollution and leakage attacks on content providers in Android applications. However, it only deals with the vulnerabilities in public databases triggered through standard content provider APIs. We present a more comprehensive study of database vulnerabilities in Android apps and analyze them for both "public" and "private" database attacks. Moreover, our analyzer can detect public and private vulnerabilities and generate proof-of-concept for them.

1.4 Contributions and Thesis Organization

This thesis makes several contributions in improving the security of Android applications. We introduce a scalable approach for automatically detecting vulnerabilities in Android apps focusing on web injection attacks and databases. Our approach is novel as we not only detect but create proof of concept (zero-day)

exploits to demonstrate the vulnerabilities. Our current techniques can be extended and customized to cover more classes of vulnerabilities in Android. With our detection system, we have found zero-day vulnerabilities in many real-world Android apps.

To the best of our knowledge, we are the first to conduct a systematic study on web-to-app injection attacks , which abuse the web-to-app bridge to hijack vulnerable Android apps without any installation of malware [HJY+15]. We demonstrate 8 different categories of W2AI attacks. These attacks target critical vulnerabilities that can be exploited silently without user interaction.

We perform a comprehensive classification of database vulnerabilities in Android apps. In order to detect and exploit these vulnerabilities, we construct symbolic models for URI-dependent Android libraries. Using our analysis framework, we have analyzed Android apps and found many of them vulnerable to the database attacks and generated exploits for them. We also compare our results with ContentScope [ZJ13]. We find both public and private database vulnerabilities. We also show new ways to exploit the previously reported and fixed vulnerabilities.

This thesis is organized as follows: Chapter 2 briefly describes the architecture of Android, its security mechanism and applications developed for it. Chapter 3 introduces a new framework for detecting and confirming vulnerabilities in Android. We also evaluate our framework and compare it with a state-of-the-art dataflow analyzer. Chapter 4 studies and classifies Web-to-App Injection attacks that target Android applications. We also present a customized version of our analysis system tailored for W2AI vulnerabilities. We have analyzed real-world Android applications and detected and confirmed W2AI vulnerabilities which is reported in this chapter. Chapter 5 performs a comprehensive study of database vulnerabilities and show how to detect and exploit them in benign Android apps. We conclude in Chapter 6.

Chapter 2

Preliminaries

In this chapter, we introduce the main elements of an Android application which are relevant in the attacks presented in this thesis. Understanding these elements are also essential for analysis purposes. We also briefly present the Android architecture and its security mechanism. Throughout this thesis, we use app and application interchangably to refer to the Android programs.

2.1 Android Applications

Even though Android apps are mainly written in Java-like languages, they are very different from standard Java programs. Some of these differences are:

1. Android applications consist of four types of components which can be points through which system enters the app (unlike Java programs which have a `main` method that can be the entry point). Each component has a different lifecycle which identifies how the component is created, destroyed, etc.

2. Android is heavily based on callbacks and asynchronous calls handled by the framework. For instance, in addition to the Java threads, Android provides higher-level implementations for threads such as `AsyncTask` and `Handler` as part of the system's framework which have particular semantics that are not available statically.

3. Android provides RPC/IPC among apps which mainly works based on serialization similar to Java but through different mechanisms. In particular, a new message passing mechanism is possible in Android in which messages (called intents) can be used to invoke a specific component in the same or another app.

4. The access control mechanism designed for Android apps is very different from Java programs. Android apps do not have a language-level security mechanism while the Java security manager enforces security in Java programs at the language level.

2.1.1 Android Components

Android components are the building blocks of Android applications. There are four types of components: activity, service, content provider and broadcast receiver. The data injection vulnerabilities studied in this thesis can be triggered by invoking one of these components depending on the attack category.
Activity. Activity is an Android component which is in charge of the user interface of the app. Usually, activities are the first components launched once an app starts. A different app can invoke any of the activities of the app (if it allows) and transmit data by sending Intent messages.
Service. A service component performs long-running operations (such as download a file) in the background without a user interface. A service can be bound by other components to establish a client-service interface through the Binder.
Content Provider. Content provider is another Android component used as an interface to manage access to a structured set of data. The developer can implement this component to share data with other apps on the device. In order to access data in a content provider of another app, developer has to use the `ContentResolver` object available in the application's context. `ContentResolver` allows applications to locate and interact with content providers in the phone. Once the content provider receives and handles a request, the results are returned by the `ContentResolver` back to the requesting app.
Broadcast Receiver. Broadcast receiver is a component that can be registered to receive broadcast messages and notifications sent by other apps or system. The broadcast messages can be the system events, such as when the system boots up or any custom intent.

2.1.2 Android Manifest File

Every Android application needs an `AndroidManifest.xml` file (manifest file). The manifest file provides information about the application including its components and the requested permissions. It is also possible to specify which `Intent` messages a component can handle through intent filters as explained shortly. Listing 2.1 shows part of an example manifest file which provides specification for an activity component.

2.1.3 Message Passing in Android

A component in an Android app can request another component (possibly in another app) to perform an action. Intent is the primary way to facilitate this communication to share data with other apps and access system services [INA]. Unfortunately, it can also be crafted by attackers to exploit data injection vulnerabilities in the apps installed on the device. An intent is an abstraction of an operation to be performed which is implemented by the Android Binder IPC. An intent object carries information as shown below which is used by the Android framework to determine which component to start executing, plus information that the recipient component uses to properly perform the action:

- **data** is a URI that the intent is targeting. It can consist of elements such as scheme, host, path, etc.

- **action** is a string that specifies the generic action to perform (such as view some information).

- **category** defines a string containing additional information about the kind of intent that will be handled by this component.

- **component name** can be set to send the intent to a particular component in an application.

- **extra parameters.** Extras is a bundle of any additional information carried through intents. It is the parameter returned by `get[type]Extra()` API where `type` can be `String`, `Int`, etc. Extras is set through the `Bundle` data structure. A `Bundle` is a mapping from values to `Parcelable` types whose instances can be written and restored from a `Parcel`. A `Parcel` is a container for data or object references that can be sent through the Android IPC.

- **flags.** It is possible to use flags in Intent messages. By setting special flags, we can control how the intent is handled. `FLAG_GRANT_READ_URI_PERMISSION` is an example flag that can be set to grant `readPermission` to the recipient activity for a set of data.

In the manifest file, it is possible to specify which `Intent` messages a component can handle through intent filters using `<intent-filter>` tags. The `<action>`, `<category>`, and `<data>` tags are the sub-elements that describe most of the contents of the intent filters.

The <data> tag declares the type of data accepted, using one or more attributes of data URI (scheme, host, port, path, mime type, etc.). Listing 2.1 depicts a sample intent filter for an activity named MainActivity picked from the manifest file of an application. An Intent message is accepted by a component if its fields match the intent filter attributes.

```
1 <activity android:name=".MainActivity">
2     <intent-filter>
3         <action android:name="android.intent.action.VIEW"/>
4         <category android:name="android.intent.category.DEFAULT"/>
5         <category android:name="android.intent.category.BROWSABLE"/>
6         <data android:scheme="myuri" android:host="foobar" android:path="
              /"/>
7         <data android:mimeType="text/plain">
8     </intent-filter>
9 </activity>
```

Listing 2.1: This Listing is a part of a manifest. It shows an intent filter of an activity named MainActivity.

2.2 Android Architecture

The Android architecture consists of five layers: applications, application framework, libraries, runtime and kernel. The benign apps analyzed in this thesis reside in the applications layer which can be exploited to access system resources and libraries in lower layers. In order to exploit the vulnerabilities, attacker can leverage the services in the application framework layer to access the information and interfaces of the benign app.

Applications

Applications is the top layer in the Android architecture where all applications installed on the phone (e.g., browsers, games, etc.) can be found.

Application Framework

Application framework is the layer below the applications layer where Android apps are plugged in and contains system services that perform system-level jobs. Next, we describe some of the key system services that perform important operations:

- **Activity Manager** is the system service which controls the application lifecycle. It interacts with the overall activities running in the system. This

service manages the inter-component communication in Android. For example, it is responsible for delivering intents to the recipient activities.

- **Package Manager** is responsible for APK installation and maintaining information about installed applications. It stores the permissions granted to each application in certain files and later, apps can query this service to determine whether an app has the required permissions.

- **Content Providers** provide the interface to manage and share data among running processes.

Libraries

This layer provides libraries such as `SQLite` database, `WebKit` browser engine and `Libc` to applications at higher levels. It also includes libraries specific to Android which are available to the applications as Java classes. Next, we describe the most relevant Android libraries to the attacks studied in this thesis.

- **android.webkit.** It provides Java classes to perform web-browsing including WebView. WebView is an in-app browser that provides the basic functionalities of normal browsers (e.g., page rendering and JavaScript execution). However, the UI (User Interface) is not similar to normal browsers. In fact, the UI in WebView is similar to any other activity and the URL bar is not visible to the user. For instance, user cannot distinguish `http` from `https` URLs. Therefore, if a malicious web page loads in a WebView, phishing attacks are feasible which can result in credential theft. WebView also enables access to various interfaces (e.g., HTML5 APIs and JavaScript-to-native bridge) that can be exploited by malicious URLs. WebKit, the rendering engine in WebView, is a framework library to display web pages. In addition to phishing attacks, if the malicious URL loads in the WebView, attacker can exploit vulnerabilities (e.g., [CVWb, CVWa]) in the WebView or WebKit (other than the vulnerabilities in third-party apps) to bypass the same origin policy or invoke arbitrary Java code. WebViews are used in a group of vulnerable applications studied in Chapter 4, which are exposed to attackers allowing arbitrary JavaScript execution.

- **android.database.** This library contains `SQLite` classes including `SQLiteQueryBuilder` and `SQLiteDatabase` which can be used to access the data shared by content providers. These libraries can also be implemented in

applications as private databases. If developers do not provide enough protection for these databases, attackers can launch different classes of attacks which are studied in Chapter 5.

Android Runtime

Android runtime contains the Dalvik Virtual Machines where Android applications run. This layer incorporates standard Java libraries to be utilized in Android applications.

Android Kernel

The bottom layer in the Android operating system is the Linux kernel. The user-based access control in the Linux kernel plays an important role in the Android's security mechanism.

2.3 Android Security

In order to design reasonable attack models and report realistic vulnerabilities, it is necessary to understand the protection mechanisms provided in Android. Android has two levels of security: system level and application level. The system level protection is provided by the user-based access control in Linux. Each app gets a unique UID and GID at the installation time based on which it determines which system resources can be accessed.

At the application level, in order to invoke sensitive APIs, apps have to request for certain permissions in the manifest file. By installing an application, user grants the requested permissions.

Android classifies permissions as follows:

System Permissions. These permissions are owned by the system and allow access to the system resources. `android.permission.RECEIVE_SMS` and `android.permission.INTERNET` are examples of system permissions that once granted, allow the application to receive SMS and access the Internet respectively.

Self-declared Permissions. Applications can declare their own permissions with specific protection levels in the manifest file to protect their components (e.g., activity).

Content Provider Permissions. In addition to the self-declared permissions, a content provider can protect its data with `readPermission`, `writePermission` and `path-permission`. Developers can choose to require the requesting apps to

obtain these permissions at install time. Alternatively they can delegate permissions to some applications at runtime. For this purpose, they need to specify `grantUriPermission` in the manifest file for the whole provider or a particular set of paths. The per-URI permission delegation happens by setting certain attributes in an Intent message. When the client app receives this intent, it can access the specific set of data.

The Android permissions can have four protection levels:

- **Normal permissions** which are automatically granted to applications. Permissions by default are normal.
- **Dangerous permissions** are more sensitive and will be presented to users for approval.
- **Signature permissions** require the app to have same signature as the app that has declared the permission.
- **Signature or System permissions** are only designed for system applications and third-party apps cannot request them.

Chapter 3

A Framework for Detection and Exploitation of Vulnerabilities in Android

In this chapter, we design and introduce a new analysis framework for analyzing Android applications (APKs) to detect and exploit data injection vulnerabilities on a large scale. Existing static analysis techniques alone are insufficient for conducting such analysis as the complexity and size of applications limits the precision of static analysis.

Our goal is not only to analyze for potential vulnerabilities but also to confirm them. For this more ambitious goal, we want to be able to generate concrete exploits which actually reach and also affect a sink, in other words, automatically generate some form of zero-day attacks. This makes the task of understanding and confirming a vulnerability significantly easier for security analysts or the app developers. Also, we opt for an analysis framework which is independent of Android versions and can analyze APKs built for any SDK. At the high level, finding data injection vulnerabilities of an Android app can be reduced to a source to sink reachability analysis. We assume that the source and sink method signatures are given and design our analysis system based on the source-sink pair invocations found in the program.

In general, static analysis techniques would give potential reachability and existing techniques for Android apps [ARF+14, LLW+12, GZWJ12] are no different. These analysis frameworks heavily use abstractions and often do not deal with control dependencies. As a result, there can be many potential source-sink flows detected with possibly many false positives (i.e., potential vulnerability is signaled as a flow, even though it can never occur during execution). Moreover,

these analysis systems are not adequate for exploit generation.

The random input generators in Android [MTN13, SR14] mainly focus on UI events and are more suitable for stress testing. These tools might also generate redundant inputs and they are less likely to be useful for semantically-rich vulnerabilities studied in this thesis.

Symbolic execution is an alternative analysis technique used to achieve higher precision and generate inputs that cause each part of the program to execute. This analysis technique faces the complementary problem of path explosion [CS13]. Another well-known challenge in symbolic execution is dealing with external code such as libraries and frameworks [CDE08]. Even though existing dynamic analysis systems [CDE08, GLM08] use heuristics to tame such problems while analyzing possibly large programs, they might not be directly appropriate for our purposes. The reason is the heuristics used by these works mainly aim for improving path coverage. However, we need a targeted symbolic execution which optimizes the exploration process to reach a specific sink method. Moreover, these systems may not be suitable for the Android app ecosystem. That's due to the huge number of applications in App-stores which can still grow on a daily basis and each of those applications might be replaced by new versions very frequently. Hence, our analysis design should be suitable not only for large programs but also for a large number of them. For this purpose, we offer a framework which achieves a good balance between precision and efficiency.

In this work, we employ static analysis integrated with dynamic testing to overcome the challenges of these individual techniques. Our analysis system is able to automatically analyze APK files (we don't need the source-code) to produce working exploits (zero-days) for various forms of data injection vulnerabilities. It consists of several stages which refine the results generated in the previous stage and produce more precise results. Due to this flexible design, the security analyst can optionally stop the analysis at each stage and use the results. For instance, our analyzer performs a fast static symbolic execution at the third stage to find vulnerabilities and potential exploits and reports a subset of the apps as potentially vulnerable. The results from this phase can directly be used by the security analysts to confirm the reports manually. Alternatively, it conducts an automatic testing to confirm the exploits or improve the precision of the generated inputs and potentially generating new exploits which are more accurate.

Our analysis framework shows a significant enhancement of the accuracy and precision over the results generated by purely static state-of-the-art dataflow analysis. It constructs a witness exploit (e.g., Intent), to be subsequently used by the security analysts or app-store managers to construct specific attack payloads for

determining the severity of discovered vulnerabilities. We employ several optimizations to tame the classical problems in symbolic execution such as path explosion and precision.

In summary, the contributions of this chapter are:

- A new analysis framework which is capable of generating zero-day exploits for data injection vulnerabilities in Android apps.
- A staged analysis which combines static dataflow analysis and symbolic execution with dynamic testing to analyze potentially large Android apps.
- Several optimizations that tame the path explosion and dealing with external code in symbolic execution of Android apps.
- Comparison with a state-of-the-art static analysis and showing improvement in precision and accuracy.

3.1 Motivating Example

Our aim is to design a system which both *detects* and also *confirms* data injection vulnerabilities by generating working exploits. We look for *sinks* defined as sensitive/critical Android and Java APIs used to inject data which make the application vulnerable. API calls which fetch data that are under the control of the attacker are called *sources*.

Apart from the classical challenges in symbolic execution, the complexity of the Android environment and apps raises additional practical challenges. Figure. 3.1 shows a simplification of the code of the WorkNet (kr.go.keis.worknet version 3.1.0) app which is vulnerable to data injection vulnerabilities. In this example, the source method is `getIntent()` and the security-critical sink methods are `WebView.loadUrl()` which loads URLs in the browser implemented in the app and `Context.startActivity()` which invokes other apps on the device.

The activity component that is triggered by the incoming malicious intent is `MainActivity`. Unlike Java programs, Android apps do not have a `main` method. When an intent invokes an activity, what happens is that the Android runtime invokes the `onCreate()` which is part of the activity component lifecycle or `onNewIntent()` callback method. Next, the `getIntent()` and `onNewIntent()` methods obtain the Intent messages. Once an Intent is sent to an activity, any invocation of the `getIntent()` method throughout the activity yields the same Intent until `setIntent(Intent)` is called. Thus, the intent objects at Lines L9 and L17 will refer to the same intent object.

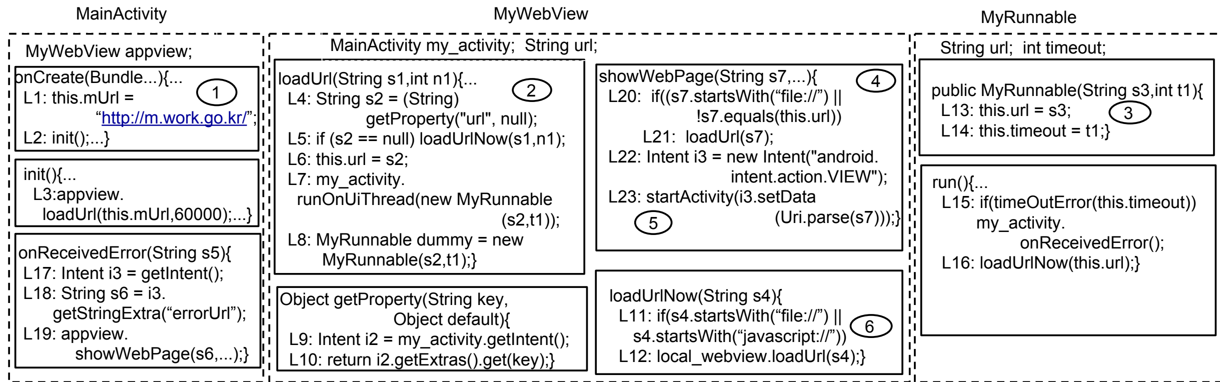


Figure 3.1: The code snippet is chosen from the WorkNet (kr.go.keis.worknet version 3.1.0) app which is vulnerable to data injection attacks. This app may obtain parameters from the malicious intents. There are three classes separated by dashed lines: `MainActivity`, `MyWebView` and `MyRunnable`. `MainActivity` is the browsable activity, `MyRunnable` is an inner class of `MainActivity` and methods are shown in boxes.

We explain three possible execution paths in Figure 3.1 where the `MainActivity` loads malicious parameters from the malicious intent. These three execution paths are explained using the steps shown in Figure 3.1:

1. The `MainActivity` is launched and `onCreate()` is invoked storing the default URL in `this.mUrl` used by `loadUrl()` at L3.

2. However, the application does not load the default URL into the WebView immediately. Instead, `getProperty()` is called which invokes `getIntent()` at L9. This method looks for the “extra parameter” (explained in Section 2.1), having the key `"url"`. If this parameter exists in the intent, `runOnUiThread()` at L7 is called which runs the `MainActivity`’s UI thread.

3. Next, `MyRunnable` class is instantiated storing the malicious URL in field `this.url` and the `run` method is invoked by the runtime. Line L15 in `MyRunnable` forks a thread (not shown) to check whether the network connection times out within `timeout` limit. In case of timeout, it calls the `onReceivedError()` in the `MainActivity`. This method looks for another extra parameter with key `"errorUrl"` at L18.

4. If the string conditions at Line L20 are met, the malicious URL is eventually loaded to the WebView (path 1 with sink 1, `loadUrl`, at Line L12).

5. Otherwise, the string will be incorporated into a new intent and the attacker succeeds to confuse this app to start another app (path 2 with sink 2, `startActivity`, at Line L23).

6. Alternatively the malicious URL obtained at Line L4 will eventually be loaded into the WebView (path 3 with sink 1, `loadUrl` at Line L12).

In this example, there are 2 vulnerable sinks at Lines L12 and L23 with 3 paths to reach them. However, analyzing these vulnerabilities requires dealing with challenges that are not currently dealt with satisfactorily in existing systems. Existing systems such as FlowDroid have limitations in constructing the control flow graph (CFG) from the Dalvik code due to incomplete models for Android-specific asynchronous calls. In the example, we saw that the vulnerable flows occur due to (nested) inner threads and `runOnUiThread` which changes execution to the main UI thread of the activity. Moreover, typically static dataflow analysis frameworks do not deal with conditional statements which may result in reporting infeasible flows.

Our analysis not only aims for accuracy in finding the paths for the source-sink flow but also needs to generate exploits (e.g., instances of intents) to confirm the vulnerability. This means that symbolic reasoning of the string type and operations is needed (Lines L11, L20) in addition to the numeric operations. Exploit generation for semantically-reach vulnerabilities is not currently supported in the existing analysis systems for Android.

The operations on Intent parameters can be dependent on the intent filters in the app manifest. Hence, in addition to the bytecode analysis, intent filters from the app manifest need to be taken into account in the analysis and the corresponding constraints should be used in the symbolic execution as pre-conditions. This example also shows that `getIntent()` method may be called in various parts of the component. All of these invocations should give the same Intent message. In general, analysis needs to determine which intent `getIntent()` refers to. Furthermore, the analysis needs to be object-sensitive to refer to the correct instance of `MyRunnable` class. It should also be field-sensitive, since the malicious URL is stored in the `this.url` field. Hence, symbolic execution should incorporate a symbolic heap model.

We have observed that real Android applications often include application-level (inter-procedural) cycles or *call cycles*, i.e., method calls in the callgraph form a strongly connected component. For example, consider the code in Figure 3.2 which shows the vulnerable paths of the motivating example in Figure 3.1. In this figure, the sink methods are `startActivity()` and `loadUrl()`. As you can see, this application contains a call cycle (the call chain is a loop) which is painted in red. Moreover, the `IrrMethod()` in this CFG embeds a long path that is irrelevant to the analysis which might prevent us from reaching the program points that we look for due to time and space limits.

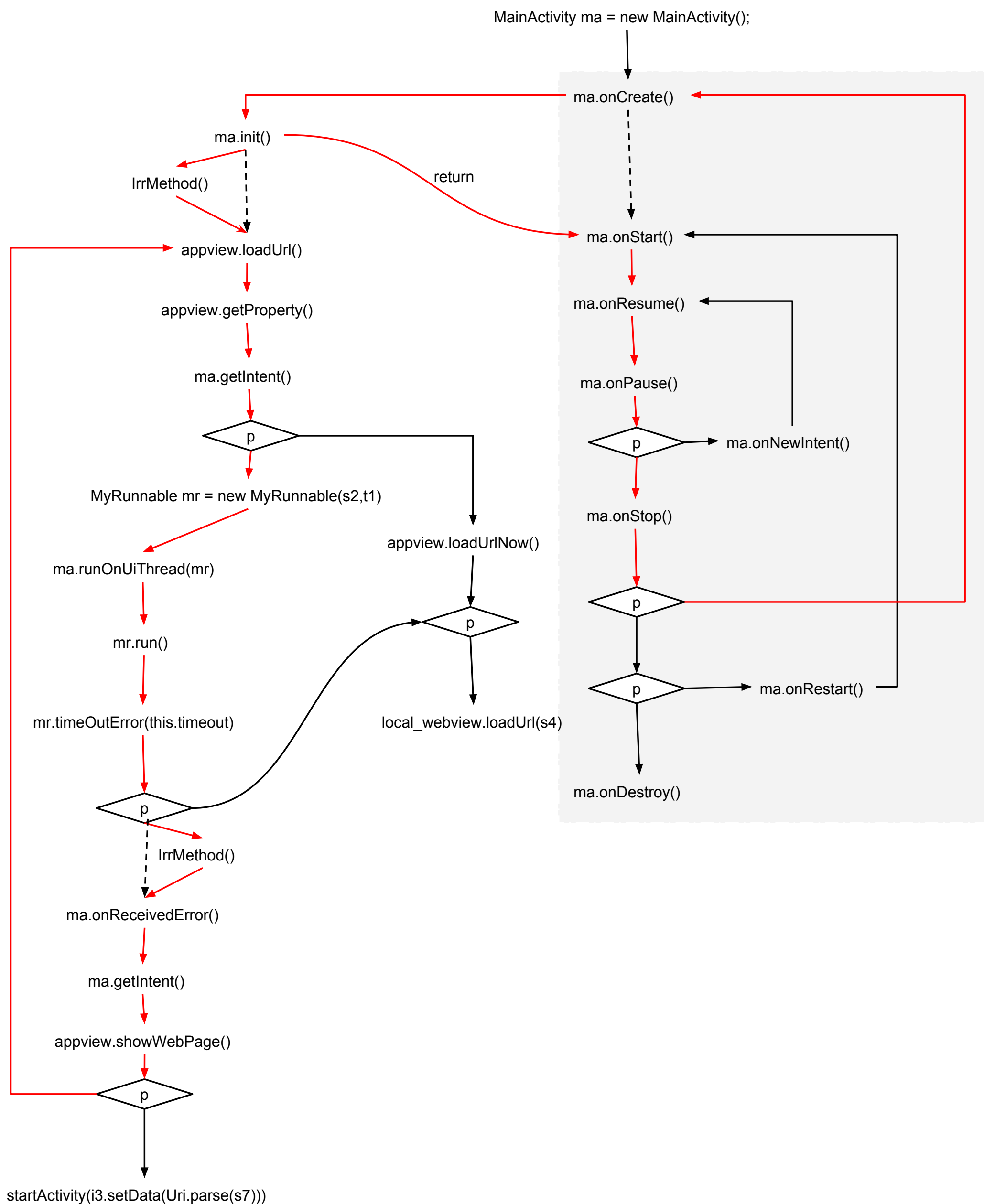


Figure 3.2: CFG for the motivating example in figure 3.1. The gray box contains the lifecycle methods of the `MainActivity`. This graph contains a call cycle which is painted in red. The `IrrMethod()` method represents irrelevant methods which do not affect the data dependency analysis but contribute to long paths. The dashed arrows are not original edges in the CFG. They summarize the methods and immediately connect the callsite to the successor statement. The `p` statements represent conditional statements (predicates).

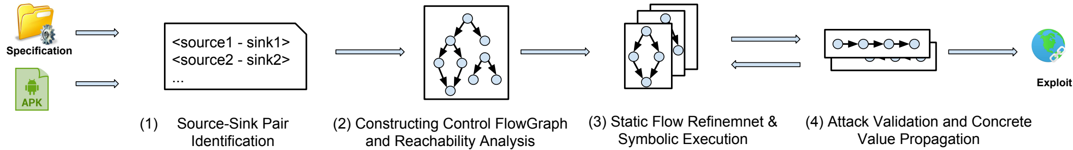


Figure 3.3: Analyzer Architecture

If these challenges are not handled properly, analysis might get stuck in one path and miss the critical sinks which reside on other paths. Our goal in symbolic execution is not to achieve full path coverage but to find certain classes of vulnerabilities and exploiting specific sink methods. Therefore, our analysis has to be equipped with ways to avoid traversing the irrelevant execution paths and reach the particular program points of interest.

3.2 Approach and Design

Our analysis consists of several individual components that are put together to detect and exploit data injection vulnerabilities in Android apps. In order to maintain the balance between precision and efficiency, we have integrated the static dataflow analysis, symbolic execution and dynamic testing. The initial static dataflow analysis can be fast but less precise which is followed by static symbolic execution, a more precise but slower phase. The final dynamic testing of Android apps is the most time consuming phase in practice. Therefore, we try to reduce the number of flows that have to be tested and confirmed by the dynamic testing phase using the other components of our framework.

The core analysis technique used by our system is symbolic execution which enables us to generate an attack exploit. To understand the importance of symbolic execution for generating attack exploits, we have randomly selected 200 apps from our data set which are reported by our analyzer to be vulnerable to data injection attacks. Figure 3.4-(a) shows that execution paths triggered by intents often contain constraints on variables that have data dependency on them. On average, the paths that trigger the vulnerabilities in these apps have 19 conditional statements with data dependency on inputs and 32% of them have more than 20 data dependent conditional statements. Therefore, simply fuzzing with random inputs might result in many false positives. A new efficient approach is required which is scalable and takes the advantage of accurate techniques such as symbolic execution.

Note that the Android framework offers a very large API to applications con-

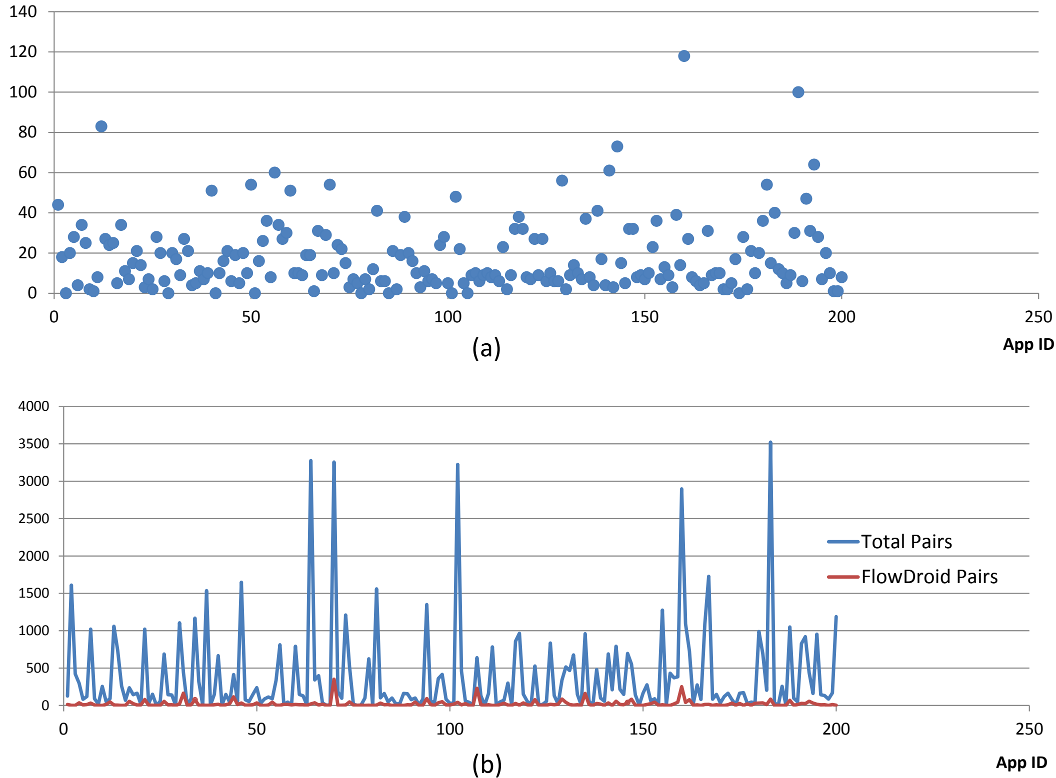


Figure 3.4: We have randomly chosen 200 applications vulnerable to data injection attacks. (a) Shows the number of conditional statements with data dependency on input on paths that reach data injection vulnerabilities. (b) Compares the number of source-sink pairs that analysis has to iterate over with (FlowDroid Pairs) and without (Total Pairs) FlowDroid.

sisting of thousands of methods. Hence, obtaining a complete control flow graph (CFG) of the application is challenging in principle and it also requires the analysis of the framework. Despite the progress in static analysis techniques, often the CFG constructed for real applications is incomplete. Even though these techniques [ARF$^+$14, LLW$^+$12, GZWJ12] are meant to be conservative through over-approximations performed in the analysis, unfortunately, in practice some of the flows are missed as explained in Section 3.1. We try to alleviate these problems by combining static analysis with dynamic testing and modeling some parts of the framework.

One well-known barrier in symbolic execution is path explosion. In order to "tame the path explosion problem", we have developed techniques which are explained in this chapter: (1) a search heuristic which chooses the next symbolic state based on its distance from a sink method; (2) the use of bounded recursion and recognizing cycles; (3) a node visiting strategy to avoid long and expensive execution paths; (4) merging symbolic states using our search heuristic.

Initial Setup of the Framework

Our implementation extends the Soot analysis framework [LBHD11] which provides a three-address intermediate representation (Jimple IR) for analyzing Java and Android applications. This framework is considered to be a start-of-the-art framework for analyzing Android apps [GKP+15]. The term *variable* usually used in Java is called *register* in Jimple and Dalvik VM. We use variables and registers interchangeably in this chapter.

Initial Control Flow Graph. The initial CFG used by our analysis is the inter-procedural control flow graph in Soot [LBHD11] constructed based on the callgraph created by SPARK [LH03]. As explained before, Android apps don't have a single `main` method. Instead, each Android component contains several callback methods (e.g., `onCreate()`) that are invoked by the Android framework in a special order. FlowDroid [ARF+14] models the Android component lifecycle, in the form of a *dummy main* method. We use the same lifecycle model in our analysis. The gray component in Figure 3.2 depicts a dummy main method used by our system. We remark that the initial model created by FlowDroid does not call the `onNewIntent()` as part of the activity lifecycle. Instead, this method is called as a callback if FlowDroid is configured in callback mode.

SPARK builds the callgraph starting from the dummy main method. It conducts a field sensitive points-to analysis to build the callgraph. Given a set of entry points, it starts with a Class Hierarchy Analysis (CHA) [DGC95] to find the reachable methods from which it creates the pointer assignment graph. Then it simplifies the pointer assignment graph and performs points-to propagation. Even though SPARK uses CHA initially, it creates the final graph on-the-fly at solving time by removing all the initial inter-procedural edges and only adding the edges as the points-to propagation continues.

Our analyzer works in phases as shown in Figure 3.3: (1) the first step identifies the pairs of source and sink program points and creates the initial CFG; (2) step 2 takes the source-sink pairs produced by the previous step and performs a sink reachability analysis which is utilized as a pre-computation for the search heuristic in the next step. Also, if our analyzer finds any edges which reflect missing execution paths at this step, it adds them to the initial control flow graph; (3) the third step performs bounded static symbolic execution to generate inputs which will be incorporated into the final exploits; (4) the runtime executor constructs the final exploits (e.g., intents) and runs them to log the execution trace for further exploit validation. Additionally, the feedback from phase 4 to phase 3 in Figure 3.3 enables our vulnerability detection system to incorporate the con-

crete values obtained from the runtime execution to the path constraints which are solved again by the solver to possibly assist the symbolic execution to generate more precise exploits. In what follows, we describe each of these phases.

3.3 Source-Sink Pair Identification

Our analysis framework starts with a specification provided by the security analyst. The specification contains lists of source and sink method signatures as well as attack settings for a particular attack model. In the first step, we generate pairs of source and sink program points for the given specification.

Definition 3.1. *Method Signature. Two of the components of a method declaration comprise the method signature – the method's name and the parameter types.*

"`<android.app.Activity: android.content.Intent getIntent()>`" is a sample source method signature. It fetches the Intent objects and provides data inputs to the app. "`<com.android.webview.chromium.WebViewChromium: void loadUrl(java.lang.String)>`" is an example sink method signature. It loads URLs in the in-app browser.

There are two design choices for selecting these source and sink program points. In the first approach, we can locate all possible program points in the initial CFG, simply by comparing the method signatures in the source code for reachable methods. Since the CFG created by Soot [LBHD11] is constructed in a way that it only includes methods reachable by the entry points, these source and sink program points are a subset of all source and sink program points in the whole application source code.

Alternatively, we can use an existing dataflow analysis system like FlowDroid [ARF$^+$14] to collect source-sink pairs which have data dependency on inputs. Figure 3.4-(b) compares the number of source-sink pairs that the symbolic executor has to iterate over using each of these two approaches. This figure shows the results for the same 200 applications randomly selected for Figure 3.4-(a). The total number of source-sink pairs has been counted by comparing the method signatures in the source code for reachable methods. The FlowDroid source-sink pairs are those reported by FlowDroid which are potentially vulnerable to injection attacks using taint analysis. As you can see, there is a big difference between these two approaches, using FlowDroid we need to perform the analysis for fewer source-sink pairs, thereby decreasing the analysis time.

As discussed in Section 3.8.1, FlowDroid is a static state-of-the-art analyzer for Android built upon Soot [LBHD11]. Even though it is not path-sensitive and the

paths generated by this framework might not be the execution paths, it scales well. The reason is that it is based on the Inter-procedural Finite Distributive Subset (IFDS) algorithm [RHS95] which has worst-case complexity $\mathcal{O}(ED^3)$ where D is the set of dataflow facts and E is the number of control flow edges of the program. IFDS can be applied to problems that have finite dataflow facts and the meet operation is distributive. These two properties allow creating representations which summarize the effects of a procedure. Such summarizations have helped the IFDS framework to handle recursions efficiently.

FlowDroid uses the CFG explained above to find the pairs of source and sink program points using dataflow analysis and also generates a set of witness flows for the detected tainted sinks. The initial dataflow analysis done by FlowDroid has to be more conservative and possibly not missing any potential vulnerability. It has a configuration setting that can be adjusted for the analysis. We have configured FlowDroid with a conservative setting. For instance, it is possible to choose the flow-sensitivity of the backward alias search and conservatively, we choose it to be flow-insensitive.[1]

In the next step, we utilize these source-sink pairs in the reachability analysis and refine the initial CFG constructed by Soot.

3.4 Control Flow Graph Construction & Reachability Analysis

The less precise dataflow analysis in the previous step might have many false positives and the same constructed CFG might miss edges (informally, we call them as gaps). This step is essentially a preparation for the next phase where we perform an accurate on-demand refinement of the analysis and symbolic execution.

This step has two objectives: (i) to refine the CFG by filling in the gaps as much as possible; (ii) and to find potential vulnerable regions in the CFG using reachability analysis. This is used to tame the state explosion problems in the symbolic execution phase.

If the CFG traversal in symbolic execution is only based on limited depth-first search, the long paths on the call cycle either cause the analysis to miss the sinks or results in path explosion. Therefore, if we perform a pre-analysis to mark the irrelevant parts of the execution paths and prevent the symbolic execution from examining them, analysis will scale better. The `IrrMethod()` in Figure 3.2

[1] The authors of FlowDroid also recommend to make the alias search flow-insensitive for large applications [FDG].

represents these irrelevant parts of the paths. Moreover, since the sources and sinks are known to our system and analysis has to be conducted per source-sink pairs, if we can detect the irrelevant pairs using a less expensive analysis and do the more-expensive symbolic execution for the rest, the efficiency will improve.

3.4.1 Control Flow Graph Construction

The current implementation of SPARK partly supports `Thread` and `AsyncTask` but it is not complete and precise enough. Android provides more mechanisms to support threads: `runOnUiThread()`, `Handler`, `Executor.execute(Runnable command)`, `ThreadPoolExecutor.execute(Runnable command)` `FutureTask`. Each of these mechanisms might have several methods to execute a thread. For example, `Handler` is an Android class which provides `post(Runnable)`, `postAtTime(Runnable, long)`, `postAtFrontOfQueue(Runnable)`, and `postDelayed(Runnable,long)` methods to start threads. Such thread mechanisms are not supported in the current version of SPARK.

On the other hand, `AsyncTask`, a helper class for `Thread` and `Handler`, is partly supported in SPARK. This Android class has a special lifecycle that needs to be modeled.

Since all the static analysis phases are dependent on the CFG, it is important to make it as precise and complete as possible. In Figure 3.1, a node for method `run` in `MyRunnable` class has to be added because the CFG misses the edge from L7 to this method. The class object for the `MyRunnable` class is resolved using a backward search similar to the copy constant search explained in Section 3.5.2. We choose to look for such gaps in the CFG and fill them to decrease the number of false negatives.

While analyzing the Android apps in our dataset,[2] we have detected some edges missing due to failure in properly handling cast operations. For these cases, the methods belonging to the class casted by the cast operation are not reachable in the callgraph. Among the other missing edges found in our refinement phase, some are due to the improper handling of inner classes. Our analyzer successfully deals with such cases and adds the missing edges to the CFG.

Given a source method, S_c, and a sink method, S_k, our analysis traverses the CFG with S_c being the starting node. We traverse the graph with an optimized depth-first search for more coverage and less memory space consumption. If a new sink is detected during this phase, it is added to the source-sink pairs to be examined later by the symbolic execution. In Section 3.7, we show that accurately

[2]Our dataset is a collection of Android apps that we analyze to detect vulnerabilities.

handling threads helped us to find interesting vulnerabilities that could not be detected by an existing state-of-the-art analyzer [ARF+14].

Our analysis needs to find the possible targets of the calls which are not reachable in the original CFG due to the missing edges discussed before. In Java and Android applications (we assess the parts written in Java), a group of methods can be overridden by inherited classes (also called virtual methods). Java also provides *interfaces* as types of reference variables. Any instance of a class that implements the interface can be assigned to such reference variables. Therefore, there might be more than one implementation for methods of an interface. In order to find the correct targets of such methods while traversing the parts of the CFG which are added by our analysis, we employ a backward use-def chain analysis to find the allocation site of the object and use its type as target.

The backward use-def chain analysis is 1callsite+1object sensitive. Note that in a context insensitive call graph traversal, results computed for a method is used for all of its callsites. Traditionally context sensitivity has been a standard vehicle to increase precision. There are many flavors to context sensitivity including callsite, object-based and type-based analyses. 1callsite+1object context sensitivity means that the analysis qualifies each method invocation with the receiver object of the method (i.e., 1object) and the callsite of the method where the receiver object is allocated (i.e., 1callsite).

3.4.2 Reachability Analysis

The search heuristic used by our symbolic execution is based on the distance of a program point from a given sink program point. The reachability analysis explained in this section computes this information to be utilized subsequently by the symbolic executor. Symbolic executors may not explore all program paths, and hence they often make heuristic choices to prioritize path exploration. Our work focuses on finding paths that reach certain program points, whereas most prior work has focused on finding paths to increase code coverage [GKS05, CGP+06, CDE08].

While refining the CFG, a reachability analysis is also performed simultaneously for the selected sink program points. We define the program statement B is reachable from the program statement A if there exists a path in the CFG from A to B. Each statement in the CFG has a unique corresponding program point. When analysis reaches a method call, first it checks if the corresponding edge exists in the CFG. If this edge is not present, it attempts to detect and add the edge as explained before. Next, it examines the reachability and distance of the

method to the sink, S_k. The reachability analysis in this phase is 1callsite+1object context sensitive to compute distances more precisely.

The reachability analysis traverses the CFG via a limited depth-first search. If the callsite for a method invocation statement in a certain context is visited again, the previous sink reachability result is reused and the method is not traversed again. This strategy is used to handle recursive calls.

Another problem in the analysis is that S_c can be invoked anywhere in the program. Therefore, the caller of the method where S_c resides might not be known (e.g., Line L9 in Figure 3.1). Our analysis is conservative, thus, it returns to all possible callsites to continue the analysis. Note that a path might have more than one sink. In that case, the analysis continues until it reaches the S_k sink.

3.5 Symbolic Execution and Static Flow Refinement

The initial dataflow analysis in the first step might produce a large number of flows, many of which are false positives. Therefore, a strategy is required to reduce the number of false positive flows. On the other hand, static analysis is generally not sufficient to confirm vulnerabilities. Rather, concrete execution is needed for such confirmation. However, concrete execution requires input, in the form of an attack (e.g., intent). As discussed in Section 3.2, each phase in our analysis framework improves the precision of the results generated by their prior phases.

We employ a bounded symbolic execution [Kin76] commonly used for automated test generation to help in generating the input along with a reaching definition analysis. The final generated exploit is the result of a combination of the symbolic execution and validation phases. Our symbolic executor does not require any initial inputs; there are optimizations to improve the scalability and reduce the number of paths that need to be explored by utilizing the sink reachability analysis conducted in the previous step.

At the high level, our analyzer achieves an initial reduction by removing the infeasible paths using symbolic execution. A path is feasible if there exists a program input for which the path is traversed during program execution, otherwise the path is infeasible [Kor90]. So we immediately remove the infeasible paths.

Symbolic executor runs a program on symbolic input which is initially allowed to be unconstrained. It substitutes program inputs with symbolic values, hence operations manipulate symbolic values rather than concrete values. When program execution reaches a conditional statement which is dependent on a symbolic

value, the system can follow both branches. On each path, it maintains a set of constraints, called path condition, which must hold on the execution of that path. When a path terminates or a sink statement is reached, the path condition is sent to an SMT solver to generate an input which triggers the same path at runtime (if the program is deterministic). The inter-procedural analysis in our system handles three kinds of edges: call edge, return edge and normal edge.

Along with the symbolic execution, we perform a reaching definition analysis [ASU86] to be able to track the variables which have data dependency on the input (source) variables.

Definition 3.2. *Data dependency. We say statement s_2 is* data dependent *on statement s_1 if s_1 writes to the memory that s_2 later reads.*

Reaching definition is a dataflow analysis which computes all the definition statements which may reach a given program point. Given a variable, we use the computed use-def chains to determine if it is dependent on the input variables. Our reaching definition analysis is conservative. For instance, if analysis reaches a library method which is not available statically and the arguments are dependent on inputs, we assume that the output is also dependent on inputs.

Our static symbolic execution executes programs by keeping track of symbolic states and at each program statement, it transitions from a symbolic state to another.

Definition 3.3. $\sum$*. A symbolic state* $\sum$ *is defined as a tuple* $(s, \phi, \delta, \mathcal{H}, \mathcal{S}, \eta)$*.*

Given a symbolic state $\sum$, a transition step gives us a new symbolic state by translating its program statement s to a symbolic expression. If s is a conditional statement and it is dependent on symbolic variables, a constraint is derived and added to the path condition ϕ. Hence, ϕ records which conditional branches have been chosen so far. Each symbolic state maintains a mapping δ between local variables of the currently running method, i.e., variables on the call stack, and symbolic expressions. Once s is translated, δ is updated with the translated symbolic expression if a local variable is modified. Otherwise, if s is a store operation to an instance or static field, symbolic state's heap $\mathcal{H}$ or $\mathcal{S}$ are updated respectively.

Static fields are referenced by the class name where they are declared and $\mathcal{S}$ maps class names to static fields. However, instance fields are defined for a specific class object. Hence, instance variables have to be distinguished based on the class object where they are invoked. We distinguish objects using unique identifiers. $\mathcal{H}$ maps each object to its instance fields. Finally, η is the cache containing the list

of statements executed on the path which are data dependent on inputs which is kept for improving the performance.

Our symbolic analysis applies String, Integer and Boolean theories. We also handle equality constraints for object references using their unique identifiers.

Algorithm 1 gives the simplified pseudocode for the main loop of our symbolic executor and further details are provided in Section 3.5.1. Analysis picks a source-sink pair, S_c-S_k, starts from the source statement S_c and symbolically executes the program until it reaches the sink, S_k. First, an initial state $\sum$ is created and s is set to S_c. In the beginning, ϕ, $\mathcal{H}$, $\mathcal{S}$ and δ are all empty. The initial symbolic state is added to a worklist. At each iteration, the symbolic executor chooses the next symbolic state from the worklist to analyze. It calls `select` to choose a symbolic state from the worklist based on the search heuristic which is explained shortly. If s is a conditional, `fork` is called which derives the constraint C and queries the SMT solver to decide which branch must be taken next. A new symbolic state is forked for a branch if it is satisfiable. If C is dependent on symbolic variables (using the reaching definition analysis results), it is concatenated to ϕ and the SMT solver is queried. If s is not conditional, the symbolic executor runs `translate` to execute s and updates $\mathcal{H}$, $\mathcal{S}$ or δ if the instruction has any side-effect. Finally the new symbolic states are added to the worklist and execution continues until the worklist is empty or the sink statement which we look for is reached.

Algorithm 1 Symbolic Executor's Main Loop

1: S_k = a sink statement
2: **while** worklist $\neq \emptyset$ **do**
3: $\quad\sum$ = select(s, worklist, S_k)
4: $\quad$**if** sink is found **then**
5: $\qquad$exit and analyze next source-sink pair
6: $\quad$**end if**
7: $\quad$**if** $\sum.s$ is conditional **then**
8: $\qquad$fork($\sum$) and add to worklist
9: $\quad$**else**
10: $\qquad$translate($\sum$) and add to worklist
11: $\quad$**end if**
12: **end while**

3.5.1 Mitigating the Path Explosion Problem

As we have discussed before, if a conditional statement has more than one feasible branch, the symbolic executor has to choose which branch to explore first (`select` at Line 3 in Algorithm 1). We need to choose the selection strategy in a way to be able to reach the security critical statements that we are interested in successfully in practice. Moreover, the number of feasible paths may grow exponentially as the symbolic execution forks symbolic states for all feasible paths. In this section,

we introduce strategies that we use to reach the security critical statements and to mitigate the path explosion problem.

Search Heuristic

The search strategies of a symbolic executor can play an important role in alleviating the path explosion problems. For instance, SAGE [GLM08] uses a generational search strategy and KLEE [CDE08] guides the exploration towards the path closest from an undiscovered instruction which yields more path coverage. Unlike these works, our objective is not to increase path coverage but to reach a specific program point.

Algorithm 2 shows how our search heuristic picks a symbolic state. If the current state's program statement is not a control statement (i.e., if , goto, switch or call statement), it picks the last state added to the worklist. Otherwise, `least` or `randomReachable` are called randomly. Each program statement has a distance from a given sink. `least` selects the symbolic state whose program statement has the shortest distance from the S_k. The distances between program statements and sinks in the control flow graph are computed in the sink reachability phase explained in Section 3.4.

Alternatively, `randomReachable` chooses the next branch only if it leads to the S_k. If more than one branch leads to the S_k, one of them is chosen randomly.

Algorithm 2 Search Heuristic used by Symbolic Executor

1: **function** select(s, worklist, S_k)
2:     **if** s is a control statement **then**
3:         $\sum$ = (least(worklist, S_k) or randomReachable(worklist, S_k))
4:         return $\sum$
5:     **else**
6:         $\sum$ = pop(worklist)
7:         return $\sum$
8:     **end if**
9: **end function**

Merging States

We use state merging to decrease the number of paths that needs to be explored by the symbolic executor. Similar to the state merging strategies in [KKBC12], our state merging is based on our search heuristic, thereby not interfering with it. We choose to merge states at a branching node if the branches are reachable to a given sink.

Our analysis employs "Phi-node folding" or "If conversion" [CCF03] which statically merges program paths when branches form a diamond pattern in the control flow graph. Instead of branching for both true and false cases, the whole

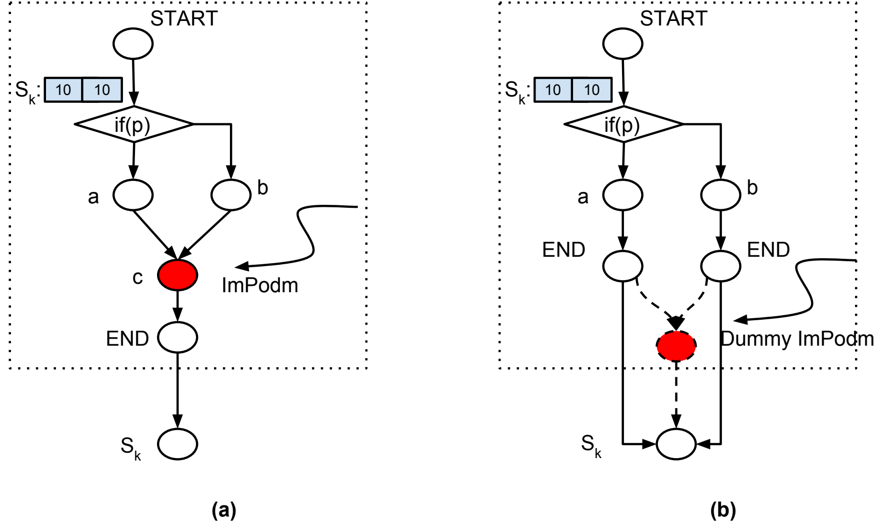


Figure 3.5: The dashed boxes contain the CFG of a method. (a) Immediate post-dominator (ImPodm) inside the method is marked as a merge point. (b) If method does not contain an immediate post-dominator but both branches of the If statement are reachable to the sink statement, we create a dummy immediate post-dominator.

diamond pattern is replaced by a single basic block, thereby, reducing the number of paths.

In addition to the classic state merging through Phi-node folding, we utilize the sink reachability results explained in Section 3.4 to perform a special form of state merging.

When analysis reaches an If statement, the sink reachability result is examined for the true and false branches. If none of the branches are reachable to the S_k, no new job will be added to the worklist and the next path will be traversed. If only one of the branches is reachable, that branch will be taken. Finally, if both branches are reachable to the S_k, we employ the following optimization.

First, we search for the ImPodm inside the method.

Definition 3.4. *ImPodm (immediate post-dominator). Given a control flow graph G, node b is said to postdominate node a if every path from a to the END node (the exit node) of G contains b. If $a \rightarrow b$ is an edge in G, then the ImPodm of a postdominates b.*

Based on the CFG of a method, if analysis finds an ImPodm inside the method, a new pending merge state will be added to the merge stack $\mathcal{M}$. Figure 3.5-(a) shows the CFG of a method (enclosed in the dashed box) which contains an ImPodm. In this figure the condition for the If statement is referred to as p (i.e., predicate) and the sink reachability result for this node is expressed by a

vector with two elements: the left element refers to the left branch and the right element refers to the right branch. In this method, both of the elements have sink reachability distance $< \infty$ which means that both of the left and right branches are reachable to the sink S_k.

Another possible scenario is presented in Figure 3.5-(b). As you can see, even though the sink reachability result for the If statement shows that both of the left and right branches are reachable to S_k, the method does not contain any ImPodm. Note that if there is no ImPodm inside the method, even though both of the branches eventually reach the same node (S_k), path merging is not possible. To avoid these distinct paths forked for each branch, we introduce a dummy ImPodm:

Definition 3.5. *Dummy ImPodm (dummy immediate post-dominator). Given a control flow graph G, node a is a conditional statement whose both branches are reachable to the sink S_k. A new node b is created such that every path from a to all of the END nodes of G pass through b. Furthermore, b is an ImPodm of node a.*

Now we explain how these dummy ImPodms are added and handled. (1) we add a merge state to the merge stack when execution reaches an always sink reachable conditional statement and the merge point can be any exit statement of the method. An exit statement is a program point where execution exits a method (END nodes in Figure 3.5); (2) when execution reaches any exit statement, it does not exit the method. Instead, if merge stack ($\mathcal{M}$) contains a pending merge job, the merge job is processed. If the merge job belongs to a conditional statement which has unexplored path, the unexplored path is added to the worklist; (3) finally, when all feasible paths inside the method are traversed and execution is exiting the method, the states at all of the exit statements which have data dependency on inputs and the constraints for the class fields are merged and there will be only 1 merged state for all exist statements. In order to choose the program statement for this dummy ImPodm, we also create a dummy exit statement.

After merging states, there is only one formula with disjunctions in the path condition. If variables that appear on different paths have different content, a new symbolic variable is added to the symbolic variable pool, σ, and the disjunction of the values is added to the path formula, ϕ. Merging the values of two variables requires a form of type checking, i.e., the types of the two variables should match each other. Otherwise, the solver produces error when it solves the generated constraint. Merging two variables are allowed if they have similar types[3] or one is `NULL`.

[3]These types have to be supported by the solver.

Loops, Recursions and Cycles

Symbolic execution of code containing loop, recursion or call cycles may result in infinite number of paths if the termination condition is not known to the analysis. In practice, one needs to put a limit on the search, e.g., a timeout or a limit on the number of paths, loop iterations or exploration depth.

Our analysis detects loops inside methods [ASU86] and conducts a bounded symbolic execution (i.e., runs loops for m times). We also detect the inter-procedural cycles (call cycles) by finding the **strongly connected components** in the callgraph. Similar to the loops, we employ a bounded symbolic execution for call cycles (i.e., iterating for n times).

Node Visiting Strategies

If a program point is visited in the same context (i.e., the same callsite) for k times and the path condition has not changed, the corresponding symbolic state is not further explored.

3.5.2 Further Optimizations

To enable detection and exploitation for vulnerabilities in Android apps on a large scale, we employ further optimizations which improve efficiency and precision. These optimizations are performed at different stages of the analysis explained earlier in this chapter.

Reusing DataFlow Analysis Results

In section 3.5, we explained that a combination of symbolic execution and reaching definitions analysis is used to accurately generate inputs which are subsequently embedded in zero-day exploits. Since the path-sensitive analysis in this phase is expensive, we do the following optimization to avoid re-computations as much as possible.

Our analysis design differs from the classical reaching definitions analysis by **reusing** the reaching definition results at branching nodes. Symbolic states store the pointers to data dependent statements (η cache in Definition 3.3). This allows us to **reuse** the computed results in branching statements. Otherwise the analysis has to recompute the use-def chains from the beginning of the path to maintain the path-sensitivity when a new branch is taken. This design enables us to efficiently check whether the η part of the symbolic state contains a definition present on use-def chains.

Copy Constant Search

In an execution path, there may be variables whose values are used but not resolved. We employ an (on demand) inter-procedural copy constant search to increase the chance of generating more accurate inputs in symbolic execution.

In general, constant propagation is conducted as follows: given statement s_1: $a = c$ where a is constant and s_2: $t = a$ op b, if statement s_1 reaches s_2 and no other definition of a reaches s_2, then t is replaced by c op b.

However, we query for copy constants in a partial backward search similar to the demand-driven approach in [Due96]. If analysis reaches a variable whose value is dependent on the intent filters in the manifest file, the value is obtained from the manifest file. We have modeled the Intent class to map the methods of this class to the elements of intent filters in the manifest file.

The analysis starts backward from statement s and only relevant information is collected (e.g., if the variable we are interested in is affected). The search terminates as soon as it finds a solution (i.e., a copy constant for the variable).

If a method invocation is reached on the path, the search continues from the exit statement of the method and the problem is replaced by the new problem for the return variable in the exit statement. In case the unresolved variable has dependency on method arguments or class fields, analysis continues from the definition statement outside the method recursively.

The inter-procedural copy constant search approach explained above leads to the following over-approximations: (i) if the variable is a method parameter, we consider all possible callers of the method. Therefore, the result might be a set of possible values; (ii) if the variable is a class field, we do an over-approximation by considering all of the objects that the field variable points to using the points-to analysis in the SPARK [LH03]. Note that if two values are resolved due to conditional statements, the intersection of the two values is reported.

In Section 3.4, we mentioned that type resolution is required to resolve methods and more specifically handle interfaces and inheritance for abstract and other Java classes. Our system resolves class types using a search similar to the copy constant search described in this section. The difference is that the search terminates if a `new` statement is reached (i.e., the class object is instantiated).

Reusing Cached Results for Identical Symbolic States

When the analyzer reaches a sink statement and sends the path conditions to the SMT solver, we cache the results. If the analysis reaches a particular sink statement with the same symbolic state, this optimization avoids sending queries

to the solver and reuses the already cached results.

3.5.3 Interaction with the Environment

Symbolic execution is not always able to handle programs completely. Static symbolic execution usually does not scale to analyze libraries and frameworks. Also, some parts of the program might be too complex, might contain native code which is not supported by our symbolic executor or they may only be available at runtime. Whenever symbolic execution is not possible, models can be used to approximate the behavior of the unavailable code. Another way is to use concrete values to simplify constraints and carry on the analysis with a simplified partial symbolic execution [CDE08]. However, this might result in over-constraining [CKC11] and interesting paths might be lost. We use models for some libraries and use symbolic variables for the rest. Once we get the concrete execution trace by running the generated exploit, we obtain the concrete values and try to improve the path constraints.

Our analysis is a hybrid of pure static symbolic execution and dynamic testing. We have modeled the frequently used library functions that are necessary for generating precise exploits for data injection attacks. `android.content.Intent`, `java.lang.String`, `java.lang.StringBuffer` are examples of such classes and symbolic reasoning for them is crucial. For the rest of the libraries, we use a symbolic variable for the return value of external method. Once the initial inputs are tested by the dynamic executor, the concrete values are obtained from the concrete execution path and we try to replace the concrete values with the symbolic variables used for the external method to generate more accurate inputs if possible.

Modeling Libraries

String Classes. The library classes which implement the semantics of strings (e.g., `java.lang.String` and `java.lang.StringBuffer`) are modeled using SMT formulas. Since the SMT solver used by our system supports the String theory, most of the methods of these libraries can directly be translated to SMT formulas (see Appendix A).

Container Classes. Our analysis uses models for container libraries (e.g., `android.content.ContentValues`, `java.util.List`) to provide more precision. The `android.content.ContentValues` class is used to map a set of keys (column names) to values. This class is usually used in the database APIs. Our analysis generates a unique identifier for each `ContentValues` object and stores `ContentValues` column name and values in the field map for the `ContentValues`

object. If the key parameters are not resolved by the symbolic execution, we use the copy constant search, explained in 3.5.2. If the analysis fails to resolve the key names, an arbitrary string value is generated. The `containsKey()` method in this class captures the constraints related to the key parameters of the `ContentValues`. In this way, we are able to trace the values stored in these objects more precisely. We have modeled other Java container classes such as `java.util.List` in a similar way.

Intents. Our model for the Intent objects is similar to the one used for container classes. The symbolic model for Intents has several fields such as `action`, `category`, `Extras`. During the analysis, we define Intent methods as entry methods if the intent is the source input. The methods of the Intent class are also mapped to the elements of the intent filters in the manifest file. For instance, when analysis reaches the `Intent.getAction()` method, the analyzer parses the manifest file and finds all possible `actions` registered for the intent object and adds equality conditions to the path condition. The `data` fields of Intent class which is a URI is not precisely modeled in this chapter. Therefore, if the fields of the URI cannot be determined using the manifest file, we try to track the values returned by the URI methods (e.g., Uri.getPath()) by generating unknown values explained before and follow their data dependencies on the inputs conservatively.

Bundle. `Bundle` is a class used to set extra parameters of intents. For instance, `Intent.getStringExtra(String key)` returns the extras in the `Bundle` field of an intent whose type is string and is mapped to `key`. This class behaves similar to the other container classes. We need to find keys corresponding to each input parameter to find the values of the arguments of API calls such as `getStringExtra(String name)`. If the analysis fails to resolve the key names, an arbitrary string value is generated.

On-demand support for arrays. Handling arrays in program analysis is usually expensive. In practice, precision is usually sacrificed for the performance and the indices of these data structures are not distinguished in the analysis. We take a conservative approach and for most of the cases avoid tracking the individual elements of arrays. However, if the source variables in the analysis have array types (e.g., some of the parameters of source methods in the public and private database attacks studied in Chapter 5 have array types), we keep track of the individual elements of the arrays.

Threads. We handle different ways provided by Android to use threads and also support binding arguments for them. Usually threads are initialized with arguments which are stored in class fields. Later, these class fields are queried in the body of the `run` methods. Keeping track of these objects is possible using our

field-sensitive analysis. There are also more complicated thread models used in Android apps for which an obvious one-to-one mapping between actual parameters at callsite and formal parameters of the method does not exist. As an example, the argument of the `AsyncTask.onPostExecute(Result)` is the return value of the `AsyncTask.doInBackground(Params...)`. We also handle such cases.

Once we get the abstract description for all the sinks and external methods as constraints (SMT formulas), we solve them and check the feasibility of each path. For feasible paths, a solution to the constraints is a witness which can be used to construct an exploit to drive the execution down this path. These exploits are used at the last step to dynamically execute the program. We employ the CVC4 SMT solver [LRT^{+}14] which supports String, Integer and Boolean constraints to solve the generated formula.

Once the solver has generated values for the symbolic variables, we use them to instantiate an exploit. In order to incorporate the generated inputs to the exploit, analysis should resolve other pieces of information (e.g., the key-value mappings) in the exploit (explained shortly).

3.6 Attack Validation and Concrete Value Propagation

Even though the symbolic execution in the previous step can remove some of false positives, it is not sufficient for confirming attacks. The analyzer aims to automatically generate exploits (e.g., intents) for the data injection vulnerabilities. The generated exploit might need additional data which correlate to the intent filter elements in the manifest file. For example, an exploit in the form of an intent should be configured (e.g., using the action and data element of the intent filter of the target component) to trigger a specific source method in the victim app.

Once we have all the necessary inputs for the source-sink flows and the intent filter specifications for the target component, the system puts all of these elements together to generate working attack exploits. Note that due to the state merging in the previous section, a group of paths generated in the symbolic execution phase might contribute to a single exploit.

There are several possible ways to send data to an application. Some applications communicate with other apps by directly calling their exposed APIs (e.g., public database attacks in Chapter 5), while some others send Intent messages (private database attacks in Chapter 5 and W2AI attacks in Chapter 4). Depending on the attack model, the actual exploit may follow a different structure, e.g.,

malware app or intent hyperlink. In Chapter 4 and 5 we show how the analyzer is customized to generate exploits for the W2AI and database attacks respectively.

Attack Validation. The exploit generated in the static phase are used by the dynamic executor explained below to validate whether they exploit the sink methods. In general, an exploit is considered a true positive exploit if it satisfies the following conditions: (i) causes the execution to reach a sink program point; (ii) the sink program point has a data dependency on the malicious exploit, hence controllable by the attacker; (iii) and the triggered execution path conforms to the attack policy (e.g., the security analyst can specify certain methods which should be present on the execution path).

The generated exploit can guide the execution to reach a critical sink method. For some sink methods, this might be enough for launching an attack. However, some sink methods are only exploited if they are tainted by attacker inputs. We call the latter data injection attacks. The attack policy consists of rules for every class of vulnerability. Depending on the category of the sink method reached on the execution trace, the attack validation component applies different policy checks. The validator component invokes exploits and logs the execution traces.

After testing the generated exploits (e.g., executing the app with an intent), two possible scenarios can happen: (1) the sink method is invoked at runtime and the generated input is accurate enough to cause the desired attack. In this case, the validator component reports the generated exploit as a proof-of-concept exploit; (2) the sink method is invoked but it is not exploitable. In this case, first we use the concrete values obtained from the runtime execution path and assign them to the variables of interest whose values were unknown at the symbolic execution phase. The new path formula is passed to the solver again and our system generates a new exploit. This procedure continues until exploits do not change any more (i.e., analysis reaches a fixed point).

The validation component has to verify whether the generated exploit results lead to true positive attacks. This decision is based on the execution trace, concrete values and the attack policies provided by the security analyst. First we verify whether the sink method is reached on the execution trace.We should also check whether the concrete values of the sink method parameters are directly affected by the generated exploit fields. For this purpose, we compare the values resolved for the sink method parameters in the symbolic execution phase with the values observed after running the exploit. We also check for other methods (if provided in the policy) on the execution path that should exist so that the exploit is not prevented from occurring. After confirming the sink method to be exploitable, the exploit will be reported to be true positive.

Note that generating a working exploit cannot always be fully automatic. As an example, our system may conclude that the exploit should be

```
"intent:///A.html#Intent;scheme:myapp;action=android.intent.action.VIEW
;end"
```

and the vulnerable sink is `deleteFile(‘‘A.html’’)`. This means that the `path` segment of the URI should point to the html file which will be deleted by the `deleteFile` method. In this generated intent hyperlink, `A` denotes an unconstrained string. The security analyst has to replace `A.html` with an existing path on the victim device and the validator verifies if this input taints the sink method.

In order to run the generated inputs and obtain the execution trace, we chose to use a high-level but standard interface, the Java Debug Wire Protocol (JDWP) [JDW] which is supported by the Android runtime (both Dalvik and ART). Therefore, we don't have to upgrade our detection system for new releases of the Android framework. Specially, this factor is important when the security analyst aims to compare the behavior of the application in different framework releases.

An additional complexity is that the execution is running in Dalvik bytecode but we use the Jimple IR (the three-address IR used in Soot) in our analysis. Dexpler (the Dalvik byte code to Jimple converter in Soot) [BKLTM12] keeps a mapping between byte code instruction addresses and Jimple statements. In order to assign the concrete values of variables from execution trace to Jimple registers, for each method, we have to find the relation between variables on the execution stack and the Jimple registers in the method Body. Moreover, Jimple local registers might be reused (because Jimple is not a Static Single Assignment (SSA) representation).

After running the generated exploits, we will use these register mappings to find out accurately which Jimple registers' values should be updated. These values will be further processed to construct more accurate exploits as explained before.

3.7 Evaluation

In this section, we assess the effectiveness of our analysis framework against FlowDroid [ARF+14], a state-of-the-art static dataflow analysis for data injection vulnerabilities. We have the following goals: (i) the potential vulnerabilities found by the analyzer should have only few false positives; and (ii) the analyzer should find vulnerabilities which may be missed due to the imprecision at the initial CFG

construction.

We have analyzed 1,729 apps in Ubuntu 12.04 on an Intel Core i5-4570 CPU PC desktop (3.20GHz) with 16 GB of RAM. Our experiments show that our analysis framework is able to effectively reject false positive flows. In contrast, a purely static dataflow analysis like FlowDroid has a large number of false positive flows reported as potential vulnerabilities. We detect missing edges in the CFG constructed by Soot framework which results in finding more vulnerabilities.

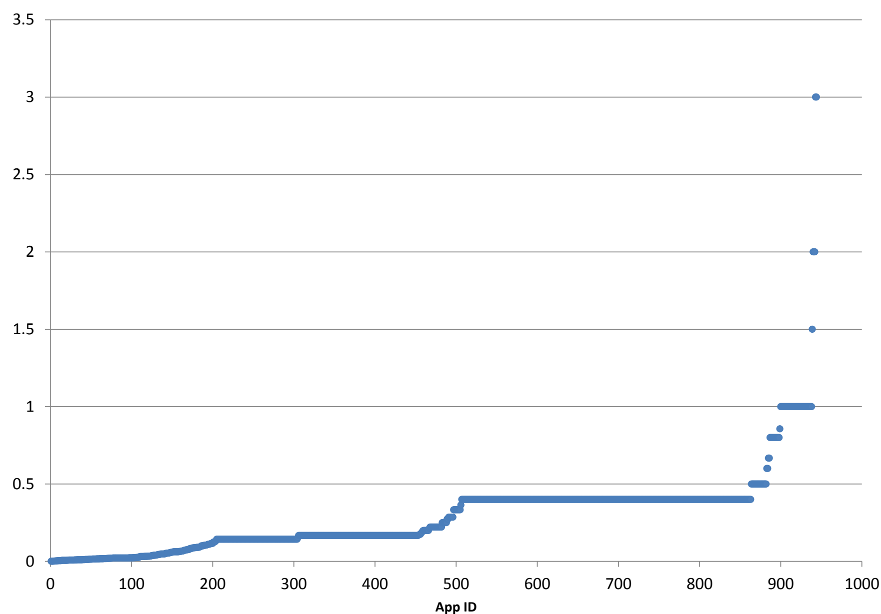


Figure 3.6: Ratio of number of paths generated by our analysis framework and vanilla FlowDroid.

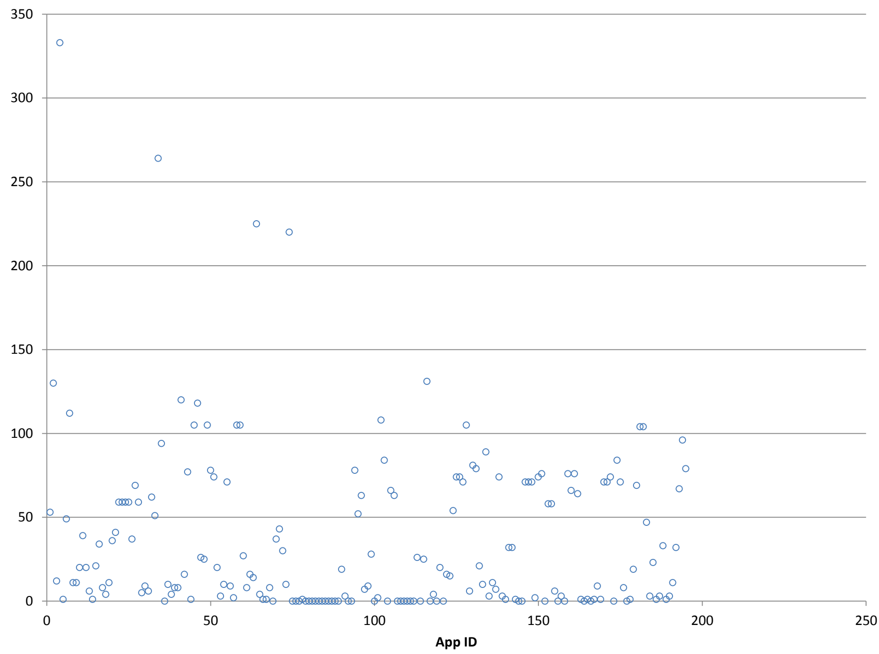


Figure 3.7: Number of missing edges in the initial CFG which were found and added by our analyzer. All of these apps have at least one potential vulnerable sink. Apps are sorted based on the ratio in Figure 3.6.

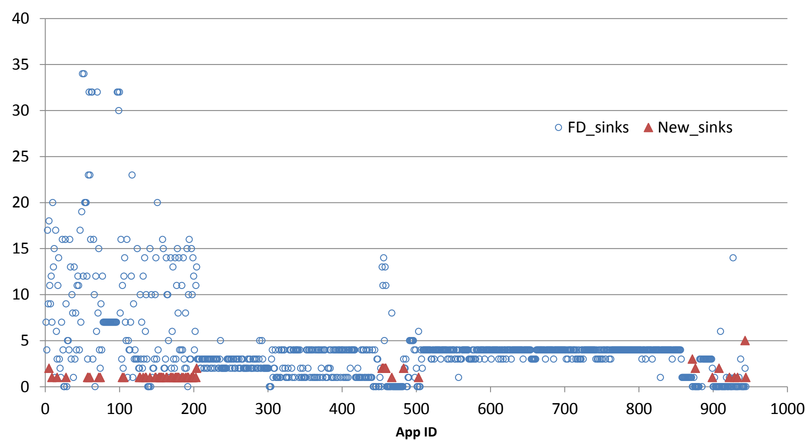


Figure 3.8: FD_sinks are number of FlowDroid false positive sinks and new_sinks are number of new vulnerable sinks found by our analyzer. Apps are sorted based on the ratio in Figure 3.6.

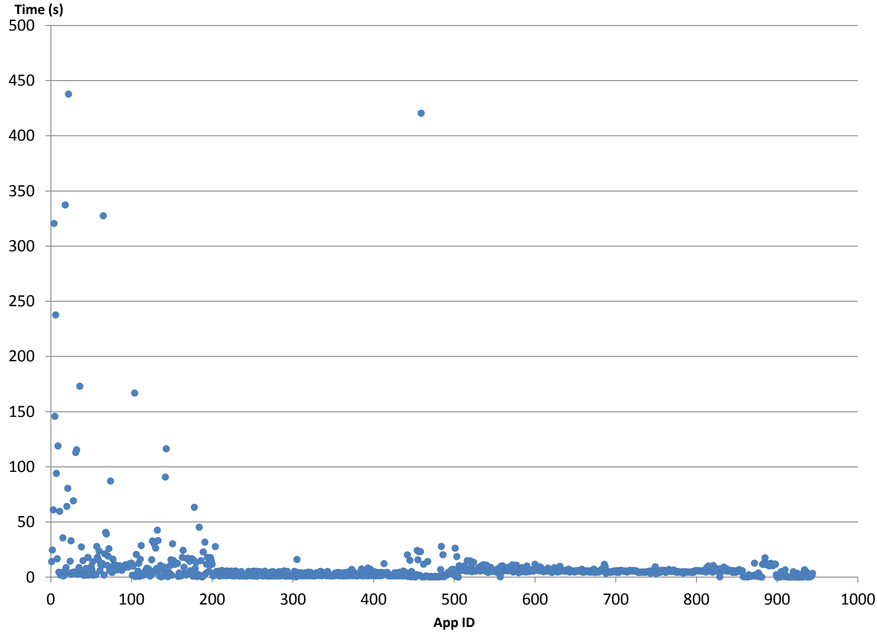


Figure 3.9: Total execution time for static analysis in seconds. Apps are sorted based on the ratio in Figure 3.6.

Figure 3.6 depicts the ratio of number of paths generated by our framework and those reported by vanilla FlowDroid. For most of the apps, there is a considerable reduction in the number of reported flows which means that either most of the false positive flows are rejected or the combination of symbolic execution and dataflow analysis has effectively reduced the number of generated paths. For some of the applications, the path ratio is bigger than 1 which is due to the new vulnerabilities detected by our analysis framework which cannot be found by FlowDroid.

Figure 3.8 shows that our analyzer is able to effectively detect false positive sinks. Our system is able to find sinks which have been missed by vanilla FlowDroid. In Figure 3.8, these sinks are shown as new_sinks. Note that if we don't find any new sinks for one app, we don't put 0 in the chart. In some cases, all of the sinks reported by FlowDroid are false positives while our analysis finds the true positive ones.

In total, we find 82 new true positive sinks in 69 applications after refining the CFG constructed by FlowDroid. The new sinks found in 39 applications are due to thread executions. Figure 3.7 shows the number of missing edges in the CFG constructed by Soot and also used by FlowDroid for each application in our data set. In total, we find 863 missing edges in the initial CFG of apps which are due to thread invocations.

The total execution time for static analysis phase can be found in Figure 3.9. For most of the applications, analysis takes less than 30 seconds. The execution

time for dynamic analysis phase tends to be higher on average. We measure the execution time as per flow (the execution time for running the exploit representing the flow) for 8 applications each representative for each attack category. The average execution time per flow is around 48.3 s. A large portion of the cost for the dynamic phase is due to operations such as networking, graphics rendering, etc.

In Figure 3.6, it can be observed that for the first 200 apps, the number of paths reported by vanilla FlowDroid is much higher than our analyzer (the ratio is less than 0.2). Figure 3.8 also shows that FlowDroid has many false positive sinks for the same apps. This shows that our system can successfully reduce the number of generated paths for these apps by rejecting the false positive sinks.

3.8 Related Work: Analysis of Java and Android Programs

Many of the recent works have been devoted to the analysis of Android apps and symbolic execution. In this section, we review the existing works on static information flow analysis, symbolic execution and dynamic analysis of Android applications.

3.8.1 Static Information Flow Analysis

Static analysis of Android applications for vulnerability detection is employed by many works [GZWJ12, ZJ13, GZJS12, GCEC12, EBFK13, CHY12, YY12, KYYS12, FHM+12, SSG+14, HUHS13]. There are also works that use analysis techniques for malware detection [FADA14, ASH+14, HZT+14]. In what follows, we describe the analysis frameworks that are closely related to our vulnerability detection system.

Woodpecker [GZWJ12] is a static analysis tool that detects capability leaks in pre-loaded Android applications. The authors define capability leaks as situations where an app can gain access to a permission without actually requesting it. The analysis is conducted in two steps: (1) finding possible paths between entry points designated by the manifest file and some use of the capability; (2) performing symbolic path simulation on each path one by one, to reject paths whose constraints are infeasible. Woodpecker's main objective is to report potential vulnerabilities that are reachable from entry points. Therefore, they do not address the challenges which are specific to input generation.

It is not also clear whether they can track instance fields of a particular class object. More specifically, it is not clear whether they distinguish objects originating at different allocation sites but reaching the same program point. Finally, Woodpecker is designed for specific pre-loaded apps which have high privileges. However, our goal is to detect vulnerabilities in any benign third-party application.

CHEX [LLW+12] is a tool designed to find component hijacking vulnerabilities in benign Android applications. In this work, the app execution is approximated as a sequential permutations of "splits". The inter-procedural dependency among heap objects is built using a callgraph constructed by 0-1-CFA analysis. The precision of this analysis affects the practicality of this approach as an imprecise callgraph may result in infeasible split permutations. The analysis in this work is limited to data dependency analysis and does not handle control dependencies. In contrast, we use symbolic execution and check the feasibility of paths by solving constraints collected along the paths. CHEX abstracts objects by their allocation sites whereas our analysis generates unique identifiers for each object during symbolic execution which yields better precision.

FlowDroid [ARF+14] is a state-of-the-art dataflow analyzer tailored for Android applications. It is built upon Soot [LBHD11] and implements the Interprocedural Finite Distributive Subset (IFDS) algorithm [RHS95] to improve the scalability. The analysis in FlowDroid incorporates a modeling of the component lifecycle of apps and achieves precision by performing a field-sensitive and an on-demand alias analysis. While doing the backward aliasing, in order to avoid false positives, a flow-sensitive analysis is employed which uses *activation statements*: After spotting a field definition in the backward analysis, FlowDroid propagates inactive taints in the forward direction. This taint, however, is not active and only becomes active when activation statement is called in the call-tree. However, this solution is expensive and in practice (for real applications) the flow-insensitive approach (also suggested by the authors in [FDG]) scales better.

Similar to FlowDroid, our analysis is also object and field-sensitive. For the cases where backward aliasing is required for field objects, if the aliasing statements are on the execution path, symbolic execution naturally captures the dataflows through the aliases. Otherwise, we use the results from the points-to analysis in the Soot framework. Similar to the previous systems, FlowDroid is designed to report potential vulnerable flows and reachable vulnerable sinks.

3.8.2 Symbolic Execution

Symbolic execution [Kin76] can be used as a general software testing technique to generate inputs that cause each part of a program to execute. It can be employed statically [CK03, CKL04]. Alternatively, it can be combined with concrete execution, [SMA05, GKS05, GLM08, CDE08] also called dynamic symbolic execution to enhance coverage and be able to deal with calls to framework APIs and library functions using the runtime concrete values. There are different lines of research that try to address the challenges inherent in symbolic execution.

Interaction with Environment and Precision. Whenever symbolic execution is not possible, concrete values can be used to simplify constraints and carry on with a simplified partial symbolic execution [GKS05]. Concolic execution [GKS05, GLM08] allows calling the actual code if it is not available statically. Another approach used by symbolic executors is to purely run symbolic execution without executing the program [JMF12]. This latter has to model the underlying platform. Execution Generated Testing Approach implemented by [CGP+06, CDE08] checks before every operation if the values in the operands are all concrete. The system interacts with the environment to run the operation if all of its operands are concrete and leverages the symbolic models of the external libraries otherwise. Even though dynamic symbolic execution helps to generate test inputs for executions that traditional symbolic execution cannot handle, it may miss some execution paths due to simplifications of concretization.

Our analysis is a hybrid of pure static symbolic execution and dynamic testing. We have modeled the frequently used library functions that are necessary for generating precise exploits for data injection attacks. For the rest of the libraries, the path condition is extended with the constraint that the relevant symbolic expression be equal to a concrete value. Initially, these concrete values are unknown. Once the initial inputs are tested by the dynamic executor, certain concrete values are obtained from the concrete execution path. These values help us to generate more accurate inputs on demand.

Scalability and Path Explosion. Scalability is a challenge that symbolic execution faces due to (1) exponential number of paths; (2) expensive constraint solving; (3) and interaction with environment. While the main objective of most prior works is to increase code coverage [GKS05, CGP+06, CDE08], our work is based on directed symbolic execution which focuses on finding paths that reach certain program points.

One way to control the exponential search space is using search heuristics. [CGP+06] selects next states if the statement is visited the fewest number

of times. This may lead to missing critical paths if the program makes extensive use of few utility methods. Our search heuristic is related to the approach used by KLEE [CDE08] which guides the exploration towards the path closest from an undiscovered instruction. Another approach is to use sound program analysis techniques to simplify the path exploration problem. For instance, the RWset technique [CS13, BCE08] discards paths that reach the same program points and while their symbolic constraints have not changed. Another approach for reducing the number of explored paths is merging the path constraints statically and passing them to constraint solvers [CCK11]. We also use merging to tame the path explosion problem.

Satisfiability checking is NP-hard for the constraints used in the formulas. Moreover, invoking queries at every branch is very expensive. Several optimizations have been designed to make the SMT solver problems less expensive: (1) expression rewriting; (2) constraint set simplification; (3) implied value concretization; (4) removing independent constraints; (5) counter-example cache KLEE [CDE08]. Our system also tries to make queries less frequent by caching the results and reusing them if the symbolic states are identical.

Recently, symbolic execution has been used for analyzing Android applications [ANHY12, MMP+12, JMF12]. [ANHY12] is a concolic executor built upon Soot which automatically generates event sequences at runtime to mimic user interactions. SymDroid [JMF12] is another symbolic executor which proposes a new language with fewer instructions (compared to Dalvik). It can be used to discover conditions under which high privileged operations in apps might occur. Unfortunately, SymDroid currently does not support symbolic strings. Compared to other Android symbolic executors that support event sequences, it requires the user to write a driver which reflects the entry points for symbolic execution. Unlike the existing symbolic executors for Android apps, currently we focus only on data inputs that will lead to injection attacks. However, existing event sequence generation techniques can be combined to the existing system in future to support more categories of attacks.

On the other hand Mirzaei et al. [MMP+12] perform symbolic execution and handle both event sequences and data inputs by first making an abstraction for modeling event sequences and then use Symbolic Java Path Finder (JPF) to perform symbolic execution on the sequences. JPF is an explicit-state model checker which is built on top of a customized Java Virtual Machine. Programs are instrumented to enable JPF to perform symbolic execution; concrete types are replaced with corresponding symbolic types and concrete operations are replaced with calls to methods that implement corresponding operations on symbolic expressions.

This work creates stubs and drivers that simulate Android framework, event handling and external libraries. Scaling a precise static analysis to real-world Android apps (in APK form) is challenging. Existing works, do not reflect this challenge when it comes to automatic data input generation for exploitable vulnerabilities.

3.8.3 Dynamic Analysis

Apart from the symbolic execution techniques, event sequences can be generated through capture-replay or model-driven approaches [AFT11, MTN13]. Dynodroid is a tool to generate UI event test inputs for Android applications [MTN13]. It is based on a modified Android framework and involves humans to pass some of the app pages. CopperDroid [TKFC15] is a dynamic malware behavioral analysis system which performs system call-centric analysis through virtual machine introspection to reconstruct the behaviors of the malware. Our work can be complementary to CopperDroid by providing inputs to stimulate the app and trigger certain system calls which can be captured and analyzed by CopperDroid.

There are also some works which create Intents to test the robustness of Android applications [MABR12, YCZJ13, SR14]. Intent Fuzzer [SR14] generates Intents to test Android components which are exposed publicly. This work uses a path-insensitive traversal of the CFG to collect possible parameter keys and string literals to use as parameter values to create well-structured Intents. During our analysis, we have observed that the data injection exploits can heavily rely on the path constraints and such an imprecise analysis may result in a large number of inaccurate Intents (which incorporate the data inputs). Therefore, the methodology used in [SR14] is more suitable for testing the resilience of Android apps to arbitrary inputs rather than exploit generation for zero-day vulnerabilities.

3.9 Summary

We have proposed a practical framework to detect and confirm zero-day vulnerabilities in Android. We have designed an analysis framework for the Android ecosystem. It reduces the false positives and improves scalability and gives good efficiency. We employ heuristics and optimizations to tame the path explosion problem in symbolic execution. Our analysis gains further precision by applying a hybrid of static analysis and dynamic testing. With our analyzer, we also validate the exploits for the vulnerable apps.

We have evaluated our system and compared the results with a state-of-the-art dataflow analysis, FlowDroid [ARF+14]. The results show that our analyzer can

avoid many false positives statically and even find new vulnerabilities missed by FlowDroid. Our analysis is efficient enough to be used at a large scale. In this thesis, we show that the analysis framework explained in this chapter is capable of finding and exploiting important classes of attacks. We find and exploit 286 W2AI vulnerabilities in 1,729 candidate apps (Chapter 4) and 153 database vulnerabilities in 924 candidate apps (Chapter 5).

Chapter 4

Web-to-Application Injection Attacks on Android

In this chapter, we study the security implications of the web-to-app channel which exposes the vulnerabilities in the application layer to web. These vulnerabilities may further allow remote attackers to access the sensitive resources and libraries in the lower layers of the architecture.

The Android platform, much like its smartphone OS counterparts, is designed to protect users from installing malicious apps. Typically, applications are served on official, vetted application markets that are monitored from malware [BOU]. Further, each application is isolated/sandboxed and restricted to a set of permissions that are granted by the user at the time of installation.

There is a growing transparency between the web and applications. Users want a seamless experience when finding and using information on the Internet. In this chapter, we study a new class of application vulnerabilities that *do not* require the user to have installed a malicious app, but merely to have visited a malicious website or advertisement in a mobile browser. Such attacks permit remote attackers to exploit natively installed Android applications, without the risk of publishing malware on application market or enticing users to install malicious application that requests suspicious permissions at install-time.

First, we study the mechanisms of *Intent* messages in Android and show how they differ from intent hyperlinks which convey untrusted input from the web. We introduce the web-to-app bridge, an underexplored channel through which attackers can exploit vulnerabilities in benign applications remotely. This channel is currently adopted largely for several legitimate objectives. It also expands the attack surface targeting Android applications.

To the best of our knowledge, we are the first to conduct a systematic study

on web-to-app injection (W2AI) attacks [HJY+15], which abuse the web-to-app bridge to hijack vulnerable Android apps without any installation of malware. We demonstrate eight different categories of W2AI attacks.

We find that W2AI attacks introduce a broad range of possible exploits in installed Android applications analogous to vulnerabilities commonly known to occur in web applications — such as open redirect, database pollution, file inclusion, credential theft, and so on. Further, these vulnerabilities are not specific to implementations of certain application frameworks (or SDKs), as they can arise in apps written in different SDKs.

Even though, many of the developed mobile applications leverage the web-to-app channel, few of them are implemented securely. We elaborate on common implementation mistakes made by Android developers that might lead to serious security problems in Android devices.

Finally, we assess the prevalence of W2AI attacks using our analysis framework at a large scale and find 134 apps out of 1,729 browsable apps vulnerable and detect and exploit 286 vulnerabilities in total.

In summary, the contributions of this chapter are:

- Studying an characterizing W2AI attacks.
- Investigating the software engineering implications of web-to-app channel.
- Extending the analysis framework introduced in Chapter 3 and detecting and exploiting W2AI vulnerabilities at a large scale.

4.1 Web-to-Application Injection Attacks on Android

4.1.1 Intent Hyperlinks and URI Intents

Presently, both mobile applications (apps) and traditional browsers play essential roles in users' devices. To provide a seamless integration of web and native apps for users, Android and iOS support a *scheme* mechanism to handle web-to-application inter-operations [ANU, IOS]. Therefore, it is possible for a web page to invoke an installed app. In this case, the target app should declare one of its activities as browsable. For example, `Yahoo Mail` is an Android app with more than 100 Million downloads used to send emails. This email app can be invoked via an intent by any other app. Setting one of its activities as browsable, it lets

any web page to open its email composing activity of Yahoo Mail.

```
"intent://foobar/#Intent;scheme=myuri;action=android.intent.action.VIEW;S
    .url=https://google.com;end"
```

Listing 4.1: An Intent Hyperlink Example

Browsable Activities.

Activities which intend to be invoked via the web have to be declared in a specific category called the BROWSABLE category in the app's manifest (Line 5 in Listing 4.2).

```
1 <activity android:name=".MainActivity">
2     <intent-filter>
3         <action android:name="android.intent.action.VIEW"/>
4         <category android:name="android.intent.category.DEFAULT"/>
5         <category android:name="android.intent.category.BROWSABLE"/>
6         <data android:scheme="myuri" android:host="foobar" android:path="
              /"/>
7         <data android:mimeType="text/plain">
8     </intent-filter>
9 </activity>
```

Listing 4.2: The Browsable Activity's Manifest

When a user clicks a web hyperlink in a certain format, Android translates it into an intent. Such *intent hyperlinks* begin with intent:, as shown in listing 4.1. We call intents created from such hyperlinks as *URI intents*. In this thesis, we will say "intent hyperlink" when referring to the link or its string, while "URI intent" refers to workings of the mechanism in Android. Intent hyperlinks carry parameters contained in the hyperlink, the fragment identifier, and information about the target activity specified as a tuple ($scheme, host, path, action, category$), and some additional metadata. This is the web-to-app bridge defined in Android. For example, an intent hyperlink can be used to launch the phone app when a user clicks a hyperlink showing the phone number on a website.[1] Some of the mainstream Android browsers (e.g., Opera) even allow activities not marked as browsable to be launched via URI intents which allows even more attacks.

The data inputs which make up an intent hyperlink are derived from the Intent class methods. An intent hyperlink follows a specific syntax. Here is a simplified intent hyperlink example:

[1]When user clicks on the number 1234, the web page is redirected to an intent hyperlink (e.g., `intent:1234#Intent;scheme=tel;action=android.intent.action.DIAL;category=android.intent.category.BROWSABLE;end`) and the phone app launches.

```
intent://HOST/PATH?query=[string1]#Intent;action=[string2];
scheme=[string3];S.key =[string4];end
```

where data input can be sent through the [*string*] fields to the Android application code. There are several possible ways to send data via an intent hyperlink:

- A data URI which references the data resources consists of the scheme (`[string3]`), host and path that should match the `<data>` element specified by the manifest file (line 6 in listing 4.2). It can also include query parameters (`[string1]`) which are the key-value mappings preceded by the "?";

- Intent extras, the key-value pairs whose type can also be specified in the Intent URI (e.g., the S in S`.key=[`$string_4$`]` refers to the string extra). Note that an intent hyperlink can have more parameters with other types, e.g, int;

- Intent filter parameters such as categories, actions (`[string2]`), etc., explained in section 2.1 that can be sent as string values.

The syntax (as regular expression) for intent hyperlinks is provided below:

```
intent://HOST/PATH[?query1=string(&queryi=string)*]#Intent;
action=string;[scheme=string;][category=string;][type=string;]
([S|B|i|l|f|d].key=string;)*end
```

where the fields inside `[]` are optional and $*$ means that the fields inside `()` can be repeated for zero or more times. The types for extras in intent hyperlinks can be specified by `S`, `B`, `i`, `l`, `f` and `d` which refer to String, Boolean, int, long, float and double types respectively.

Figure 4.1 shows the steps from where a user clicks on an intent hyperlink in the default browser to where a benign app gets launched.

- Step 1: Initially the user installs a benign app which requests for a set of permissions.

- Steps 2 and 3: The Package Manager component in the Android framework checks the AndroidManifest.xml file for the requested permissions and prompts the user.

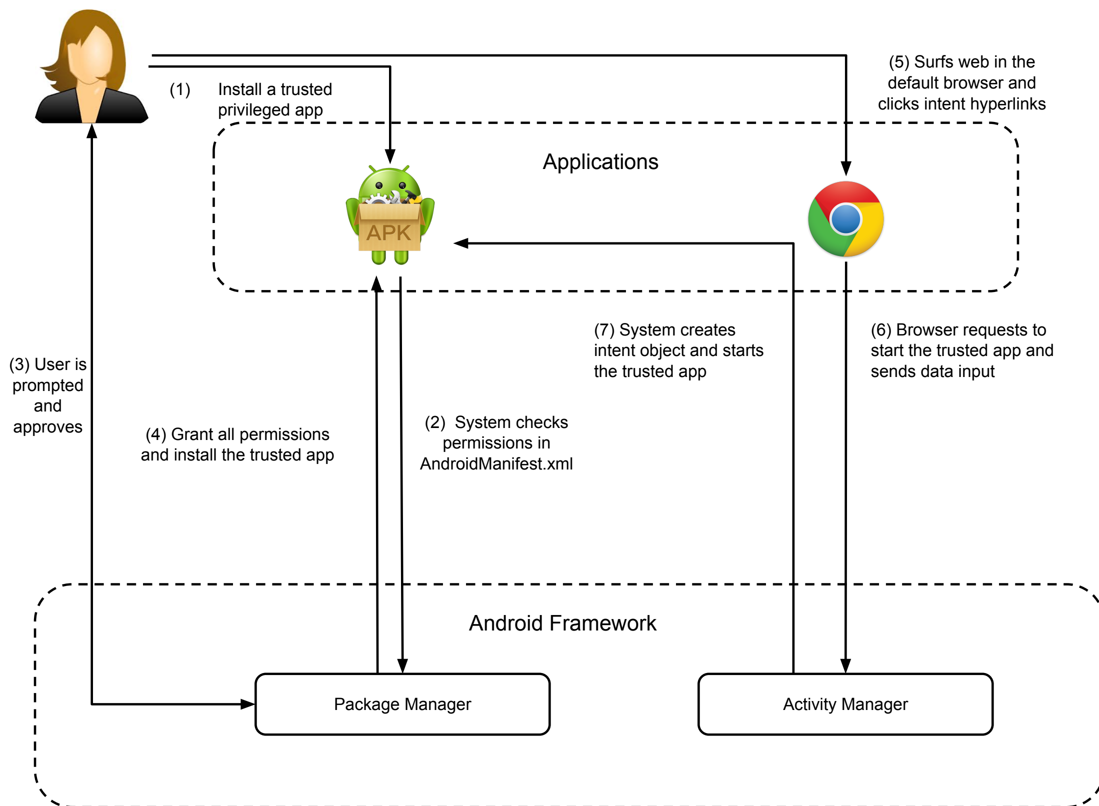


Figure 4.1: This figure shows how an intent hyperlink starts a benign app which has been granted sensitive privileges by the user.

- Step 4: The user grants privileges to the benign app.
- Steps 5 and 6: The user continues by surfing through the default browser. While surfing, she clicks on links formatted specifically to launch Android apps (intent hyperlinks). Therefore, the browser redirects to the Android framework to request to start the target app.
- Step 7: This time, the Activity Manager checks the $(scheme, host, path, action, category)$ tuple in the intent hyperlink at step 7 and launches the component of the app which matches this tuple. Note that no further permission checking is conducted by the Android framework at this stage.

In what follows, we present some use-case scenarios to highlight why intent hyperlinks are largely adopted by existing applications and therefore, a large-scale analysis of the security side-effects of this channel is crucial.

Deep Linking and App Indexing

A pervasive use-case for intent hyperlinks is deep linking and app indexing [AIN]. These features allow Android applications to appear in Google search results,

letting users to land exactly on a specific function within the app. The use of app indexing allows Google systems to improve search results, giving better ranking results, query autocompletions for device users and *Now on Tap*, a Google feature which advertises related apps while visiting a specific app.

Companies such as Google, Yelp and Twitter already offer this deep linking experience in their search results and mobile applications. The documentation for deep linking shows that developer is meant to take the user straight away to the relevant content (also called First-Click-Free) without showing the logging screen or further authentication. This requirement increases the possibility of the attacks, explained later in this chapter, which leverages the same channel as deep linking.

While deep linking is handled locally on the device by the underlying Android framework, app indexing is managed by the Google crawler remotely on the web side.

If a URL starts with `android-app://`, Google crawler considers it as a link for app indexing. When Google crawler sees such a link on a web page, it registers that link for app indexing. If user searches for content that matches this link, then Google shows the corresponding app page as the search result. For this purpose, the developer has to put the link in the HTML code of the webpage as shown below:

```
  <link rel="alternate" href="android-app://package_id[/scheme
/host[/path ]][#Intent;...];">
```

There are several case studies that show a widespread adoption of deep linking [CAS]. Statistics show that 15% of Google searches on Android return deep links to apps through app indexing and the number of clicks on app deep links has seen an increase by 10x, over just one quarter. As an example, `Etsy` is a marketplace where people sell and buy unique goods around the world. This market has seen an 11.6% increase in average daily app traffic from referral links, thanks to app indexing. Additionally, after a series of product improvements, `Etsy` has seen a 254.7% increase in impressions and a 32.5% increase in clicks.

In order to get support for deep linking, apps need to add the browsable category to the intent filter for activities of interest in the manifest file. Additionally, the intent filter should specify the `android.intent.action.VIEW` action.

Although applications benefit by adding support for deep linking, they are also opening up new channels for attackers to inject (malicious) inputs. Therefore, if applications do not apply appropriate validation on the incoming data, potential vulnerabilities will be accessible to remote attackers. To motivate the necessity

of a systematic study for these group of applications, we present an Android app which is vulnerable to W2AI attacks. `Equalizer music player booster` is a music listening Android application with more than 10 million downloads. This Android application aims to support deep linking and sets one of its activities as browsable in the manifest file. However, the browsable activity also exposes its point earning system used for marketing purposes to remote attackers. A remote attacker can request or give points on behalf of the user by sending malicious data through an intent hyperlink.

Authentication and Authorization

Another scenario where Android applications might get benefit from the web-to-app bridge is for authentication and authorization purposes. There are several identity provider SDKs available that can be integrated into the Android applications. OAuth is an authorization protocol used by most of these providers which has been shown to be poorly implemented in existing applications [CPC+14].

Both of the versions of this protocol (OAuth1 and OAuth2) use browser redirection extensively for delivering OAuth tokens. Since browsable activities in Android apps can be started by a web page, they can play the role of browser redirection in web. Service providers can send access tokens and other particulars to the applications using intent hyperlinks.

This phenomenon opens up many of the existing applications to sensitive data provided by unknown parties on the web. While application developers might have used browsable activities only for authorization or authentication purposes, we believe that they should be studied carefully for the side effects of poor implementations or misunderstandings that potentially make these apps vulnerable to different classes of attacks.

`Docs To Go (4.0)` is a document manager application with more than 10 million times being downloaded. This application uses Dropbox for storing data. The Dropbox SDK version that is used in this application implements OAuth2 and allows the remote attacker to provide the redirect URL. Therefore, remote attackers can inject their own `oauth_token`, `oauth_token_secret uid` and `state` query parameters to launch an authentication exfiltration attack.

File Management

Sometimes, Android applications expose their functionalities to other applications to enhance the usability. Streaming audio/video or opening a PDF or image and other MIME types from the SDcard are examples of such features. For this

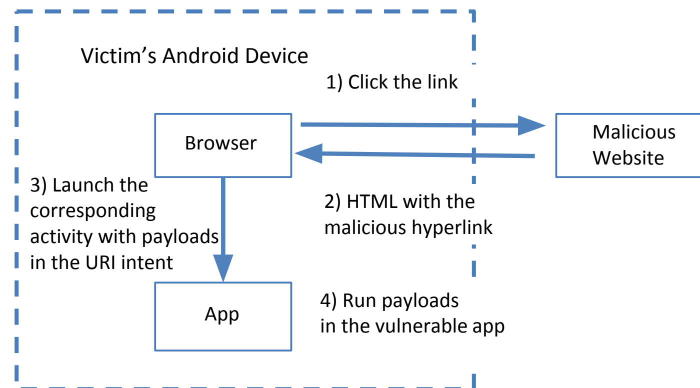


Figure 4.2: W2AI attacks on Android apps. 1) A user clicks a malicious link that redirects to the attacker's site in her mobile browser. 2) The site loads the malicious intent hyperlink in an iframe or a new tab. 3) The browser parses the hyperlink, generates the URI intent and launches the corresponding activity in the vulnerable app. 4) Therefore, the payloads derived from the URI intent running in the app can access the user's private information or perform privileged operations on behalf of the app.

purpose, the intent filter is configured to be accessed by other applications. If the intent filter is configured to be browsable, not only the local applications on the phone, but any remote party from web can also access these services. These applications usually accept the details of the operation to be performed through the `data` segment of a URI or extra parameters of the intents. The same segments can also be configured by the intent hyperlinks. If the input data is not handled correctly by the recipient application, it might lead to exploitation of potentially existing vulnerabilities.

`HD MP4 Video Downloader (1.0)` is a video downloader application which can be started by other applications or remotely through intent hyperlink. The path of the file to be downloaded can be specified by the data URI segment of the intent. However, the data URI is also concatenated with JavaScript code and loaded in the inner WebView of the application. Therefore, a remote attacker can launch a Cross-Site Scripting attack [XSS] by embedding a malicious playload into the intent hyperlink.

4.1.2 Web-to-App Injection Attacks

URI intents expose a new channel of attacks targeted at installed apps. In this chapter, we present the first comprehensive study of web-to-app injection (W2AI) attacks in Android.

Threat Model. In a W2AI attack, we assume that the adversary is a standard web attacker [ABL+10], who controls a malicious website. To expand the coverage of victims, the attacker can disseminate the shortened URL of the malicious site through emails, social networks, advertisements and other channels. Once a user clicks on a link, our attacks do not require any further interaction and vulnerabilities can be exploited silently. We make the following conservative assumptions but a real attack may be even worse by combining W2AI with other Android attacks (see Chapter 5). We assume that the victim, Alice, only installs legitimate apps from Google Play on her Android device, and does not install any malware. We assume that at least one app on her device is benign but buggy, hence a W2AI vulnerability exists. Note that as the app is benign, it has adequate permissions to achieve its functionality (See Figure 4.1). The W2AI attacks studied in this chapter do not request for system or dangerous permissions explicitly. In contrast, once the user grants permissions (with different sensitivity levels) to a third-party application, the remote attacker can leverage the already existing permissions to access system resources.[2]

W2AI Attacks. As Figure 4.2 depicts, when surfing the Internet, the victim Alice clicks a link that redirects to the attacker's site in her Android browser (step 1). The attacker's page can then automatically launch the vulnerable activity via an intent hyperlink. For example, it can load a maliciously-crafted hyperlink in an iframe causing it to generate a URI intent (step 2). The intent launches the vulnerable activity passing it the data from the malicious hyperlink (step 3). Depending on how this malicious data is used by the vulnerable activity, a broad category of vulnerabilities can arise (step 4).

4.1.3 Categories of W2AI Vulnerabilities

Android applications typically use the data derived from URI intents through various API interfaces. These can be divided into two categories — WebView interfaces and native interfaces. If the attacker-controlled data is used in these interfaces without any validation, the attacker can feed payloads to abuse the APIs. We divide the arising vulnerabilities that either abuse WebView or Android native

[2] For example, if a contact manager app has the APIs to read and write contacts, the app must have the `READ_CONTACTS` and `WRITE_CONTACTS` permissions in the manifest.

app interfaces, and explain the damage via these exploits.

Note that all W2AI vulnerabilities arise due to dataflows that start in the native Android code, and not in the application logic written in HTML5 code [CW14, LHD+11, GJS14, JHY+14]. Unlike other vulnerabilities that exploit app-to-app communication interfaces [ZWZJ12, ZJ12, LLZW14, CQM14], W2AI attacks do not need an installed malicious app on the device to launch attacks.

Abusing WebView Interfaces. As we explained before, WebView is an in-app browser that provides the basic functionalities of normal browsers (e.g., page rendering and JavaScript execution) and enables access to various interfaces (e.g., HTML5 APIs and JavaScript-to-native bridge). Certain applications take parameters in the URI intent and treat them as web URLs, thereby loading them into WebView during their execution. If such a behavior exists, the attacker's HTML code runs in the WebView. Additionally, if the vulnerable application enables execution for JavaScript in the WebView, the attacker can run JavaScript in its HTML page, and can access all interfaces exposed to it by WebView. We classify the vulnerabilities arising from unfettered access to the exposed interfaces into 4 sub-categories:

1) Abusing the JavaScript-to-Native Bridge. JavaScript code loaded in the WebView can access native methods on Android.[3] The accessible native methods are specific to each application and tend to be quite large. In our experiments, we have found up to 29 distinct JavaScript-to-native interfaces accessible in a single application. For example, many applications enable access to interfaces that retrieve the device's UUID, version and name, thereby opening up the threat of privacy-violating attacks. Furthermore, several interfaces allow reading, updating and deleting the user's contact list and app-specific local files.

2) Abusing HTML5 APIs. WebView enables access to standard HTML5 APIs, akin to normal web browsers. For example, if the vulnerable app has the proper permissions and WebView settings,[4] the attacker's web page running in the WebView can use JavaScript to call the HTML5 geolocation API directly. For instance, we find that 29 applications allow the attacker to track the user's current geolocation.[5].

3) Local File Inclusion. When the user visits the malicious site, the site can trick the browser to automatically download a HTML file into the user's SD-

[3]JavaScript can access native methods via the `android.webkit.JavascriptInterface`.

[4]`ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION` permissions and `setJavaScriptEnabled` and `setGeolocationEnabled` settings give access to geolocation sensors.

[5]Using the API `navigator.geolocation.getCurrentPosition`

card by setting the HTML file as not viewable.[6] When the site triggers the browser to parse the intent hyperlink that refers to the downloaded HTML file (e.g., `file:///sdcard/Downloads/attack.html`), it launches the vulnerable app to load the HTML file in its WebView. If the vulnerable app has certain settings[7] for the WebView, the malicious JavaScript code in the HTML file can read any files under the app's directory or the readable system files (e.g., `/etc/hosts`) and send them to the attacker.

4) Phishing. The attacker's web page can impersonate or phish the user interface of the original application. Since there is no address bar displayed by WebView that users can use to inspect the current page's URL, users cannot distinguish the phishing page from the normal page, as shown in Figure 4.4-(a). Such attacks via the web-to-app bridge are harder for users to discern than the conventional phishing attacks on the web [FW11].

Even worse, we have detected applications that use a default UI for all the activities including the activity that the WebView is loaded in. Figure 4.3 shows a zero-day phishing attack on `com.sigmaphone.topmedfree (1.0.92)` application which is downloaded more than 1 million times. In this example, `attacker.com` is a page identical to an existing activity in the application and entices user to do sensitive operations (e.g., **Buy stuff** from `attacker.com`). The user has no means to distinguish the original app (on the left) from the injected page from `attacker.com` (on the right).

Abusing Android Native App Interfaces. Android Apps, even those which do not use WebView, can expose native Android interfaces to URI intents without proper validation on input. These open the following four category of exploits in our experiments:

1) Database Access. Android provides native interfaces for apps to execute SQL statements to manage the app's database. Therefore, if the SQL statement parameters are derived from the URI intent, it allows the web attacker to pollute (e.g., add or update the table's fields) the vulnerable app's database.

2) Persistent Storage Pollution. Android native interfaces enable apps to store persistent states, e.g., authentication tokens, in the persistent storage.[8] Many vulnerable apps directly treat the parameters from the URI intent as the content to add or update the persistent storage. Hence, the attacker can craft a proper

[6]Setting the `Content-Type` header with `binary/octet-stream` for a HTML file can make it not viewable.

[7]The settings are `setAllowFileAccess`, `setJavaScriptEnabled` and `setAllowFileAccessFromFileURLs` (or `setAllowUniversalAccessFromFileURLs`).

[8]Persistent storage includes `SharedPreferences`, local files, etc.

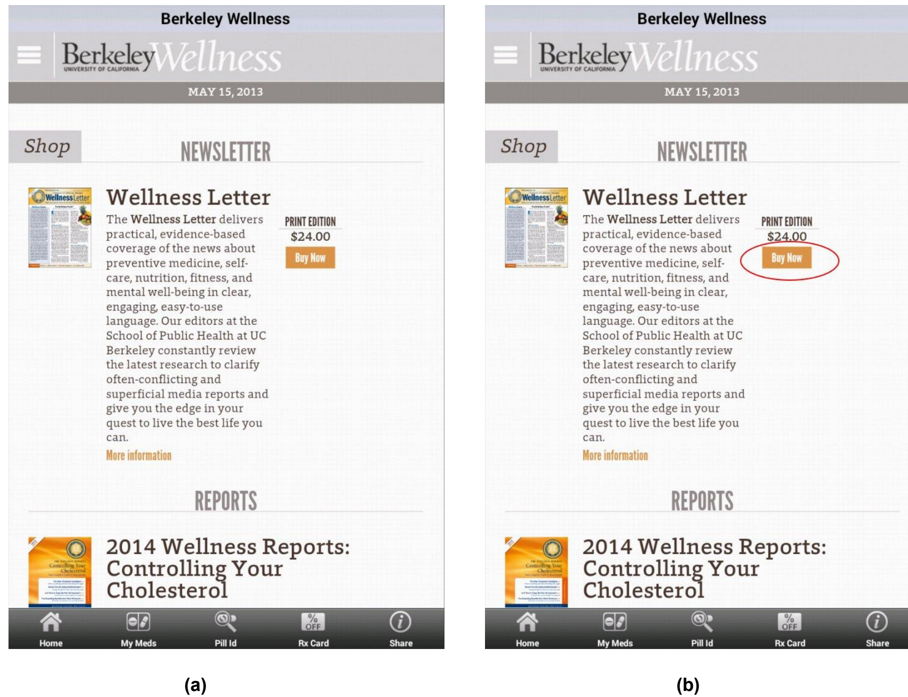


Figure 4.3: (a) The original activity used for financial purposes. (b) Injected URL loaded by the App. This application uses the same UI framework for different activities.

URI intent to pollute the target app's persistent storage.

3) Open Re-delegation. Android native interfaces provide the ability to launch specific activities addressed by name.[9] If the name parameter is derived from URI intent, it allows the malicious web attacker to invoke any in-app private activities directly, which may not be marked browsable. Moreover, attacker might embed an additional intent hyperlink as a parameter to the original intent hyperlink and force the benign app to invoke another app. This leads to a broad range of problems such as permission redelegation [FWM+11]. Permission re-delegation is a confused deputy problem whereby a vulnerable app accesses critical resources under influence from an adversary. Though these attacks are previously known to be possible via the app-to-app [FWM+11], we show that they can be affected under influence of a website through the web-to-app bridge, without requiring installed malware. Furthermore, W2AI attacks allow calling in-app private activities beyond what traditional privilege redelegation attacks provide.

4) App-Specific Logic Flaws. Android enables apps to perform various operations (e.g., popping up messages) via native interfaces. Due to the app-specific logic flaws, the vulnerable app directly uses the data from the URI intent as parameters to these operations. For instance, in our experiments we find that the web attacker

[9] `Class.forName(`x`)` enables invoking a class called x.

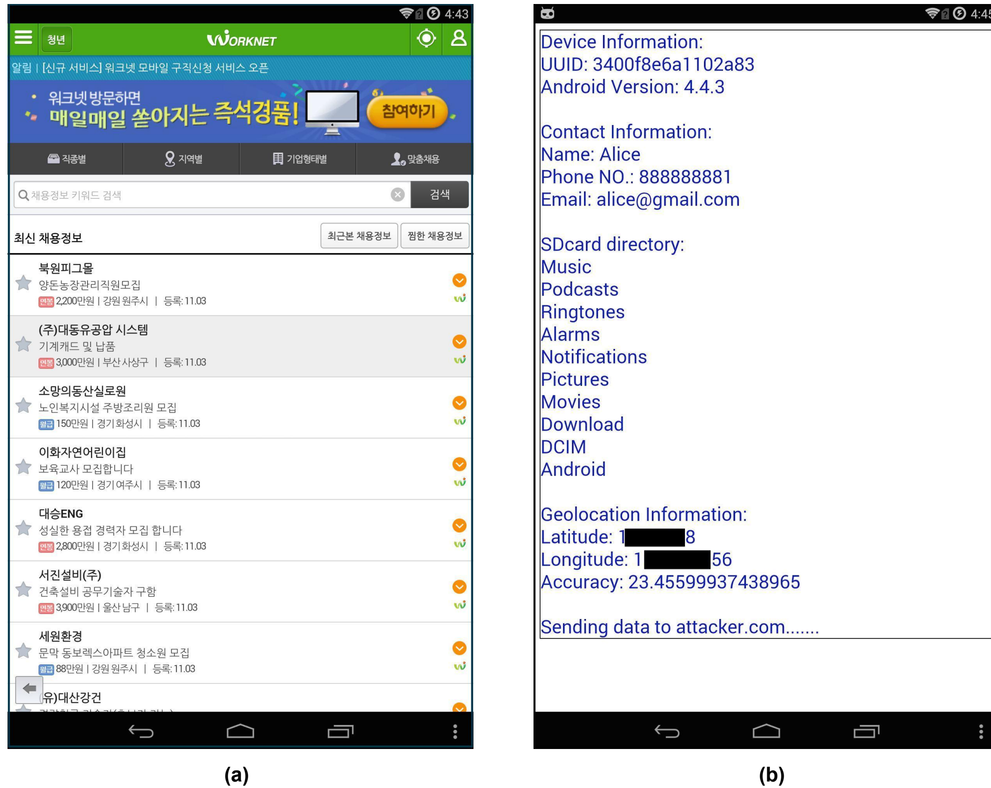


Figure 4.4: (a) The original or phishing page in WorkNet. (b) The malicious page running in the `WorknetActivity` steals the user's private data (e.g., device information, contacts, local files and geolocation), sending it to the attacker's server.

can utilize the flaws to instruct the vulnerable apps to display fabricated PayPal transaction status.

4.1.4 A Vulnerable App Example

Now we use a real app as an example to explain how the W2AI attack works.[10] The example app is WorkNet (`kr.go.keis.worknet`), which provides job information in Korea and has 1 - 5M downloads. It has a browsable activity,[11] which loads arbitrary URLs in URI intents and is vulnerable to the following W2AI attacks: abusing JavaScript-to-native bridges, abusing HTML5 APIs, local file inclusion and phishing. The attack's life cycle is depicted in Figure 4.2 as follows:

1. The attacker hosts a malicious website, which loads the hyperlink below into a new tab using `window.open`.

```
"intent://#Intent;scheme=worknet;action=android.intent.action.VIEW;S
    .url=http://attacker.com;end"
```

[10]We have disclosed the vulnerability to the application vendor and Google.
[11]The activity `kr.go.keis.worknet.WorknetActivity`.

The attacker posts the site's shortened link on social networks, e.g., Facebook.

2. When the user visits the attacker's site (by clicking the link on social networks, ads, and so on) in her Android browser, the site loads the hyperlink in a new tab.

3. The user's browser parses the hyperlink to the URI intent that contains extra parameters and launches the `WorknetActivity` activity with the intent.

4. The activity loads the URL (`http://attacker.com`) derived from the malicious URI intent into the WebView without any validation. Now the attacker's site is loaded with its JavaScript code running in the WebView. The attacker can utilize whatever is available to this activity, e.g., abusing JavaScript-to-native bridges.

   We find that WorkNet has 21 such interfaces, e.g., accessing contacts, local files, device information and so on. Therefore, the attacker's site can mimic the same UI of the original page as Figure 4.4-(a) shows. In the background, the scripts access the user's private data (e.g., device information and contacts) and send them to the attacker's server as Figure 4.4-(b) demonstrates (logged on the server). In addition to abusing the JavaScript-to-native interfaces, the web attacker can also abuse HTML5 APIs to track Alice's geolocation and leak the content of local files via file inclusion in this app.

From this example, we can see that the W2AI attacker can not only mount conventional web attacks (e.g., unvalidated redirection in the example), but also can hijack the vulnerable app to perform privileged operations on sensitive resources (e.g., local files and contacts) without any installation of malware in the user's device.

4.2 Web-to-App Channel from a Software Engineering Perspective

We find that among many of the developed mobile applications which leverage the web-to-app channel, few of them are implemented securely. From our experiments, it appears that either developers do not understand how this channel works or they give little attention to the security implications of a vulnerable implementation. In

this section, we elaborate on common implementation mistakes made by Android developers that might lead to serious security problems in Android devices.

While implementing the code to support intent hyperlinks, Android developers might introduce vulnerabilities in several ways. As explained in Section 4.1.1, programmers need to both add intent filters to the manifest file and handle the Intent messages received in the app via the web-to-app channel. Recently, Google has provided plugins in the development environments for giving useful warnings to programmers, specially for intent filter configurations. For instance, if the `pathPrefix` specified in the intent filter of a browsable activity does not start with `'/'`, the developer gets warning. However, the logic that handles the intents received by the app is not analyzed. Therefore, developers might fail in enforcing crucial validations.

In this section, first, we present some Android-specific programming practices that might increase the W2AI attack surface. Next, looking at the most popular Android development frameworks, we estimate how the software engineering practices lead to devastating security vulnerabilities. Finally, we study the request methods in the HTTP protocol to understand whether app indexing, an important use-case for web-to-app channel, is akin to `GET` methods. For this purpose, we compare the existing web-to-app channel design with HTTP request methods and find ambiguous differences which might lead to vulnerabilities.

4.2.1 Browsable Activities as the Main Entry Points

In Chapter 2, we explain that Android applications are different from Java programs. An Android app consists of four types of components: activities, services, broadcast receivers and content providers. These differences might raise confusions in developers. For instance, an activity can serve as the main entry point into the app if it has an intent filter with the `android.intent.action.MAIN` action in the manifest file. This action indicates that the corresponding activity is the main entry point and does not expect any Intent data.

We have found 453 applications in our data set (26% of the browsable apps) where the main entry point activity is set to be browsable. This means that developers of these apps have possibly exposed the main functionalities of their applications to web. This behavior extends the potential attack surface which developers should particularly try to avoid.

Figure 4.5 shows the percentage of Lines Of Code (LOC) which are reachable from the browsable activities in the apps studied in our data set. In order to count the reachable LOC, the number of statements of the application methods

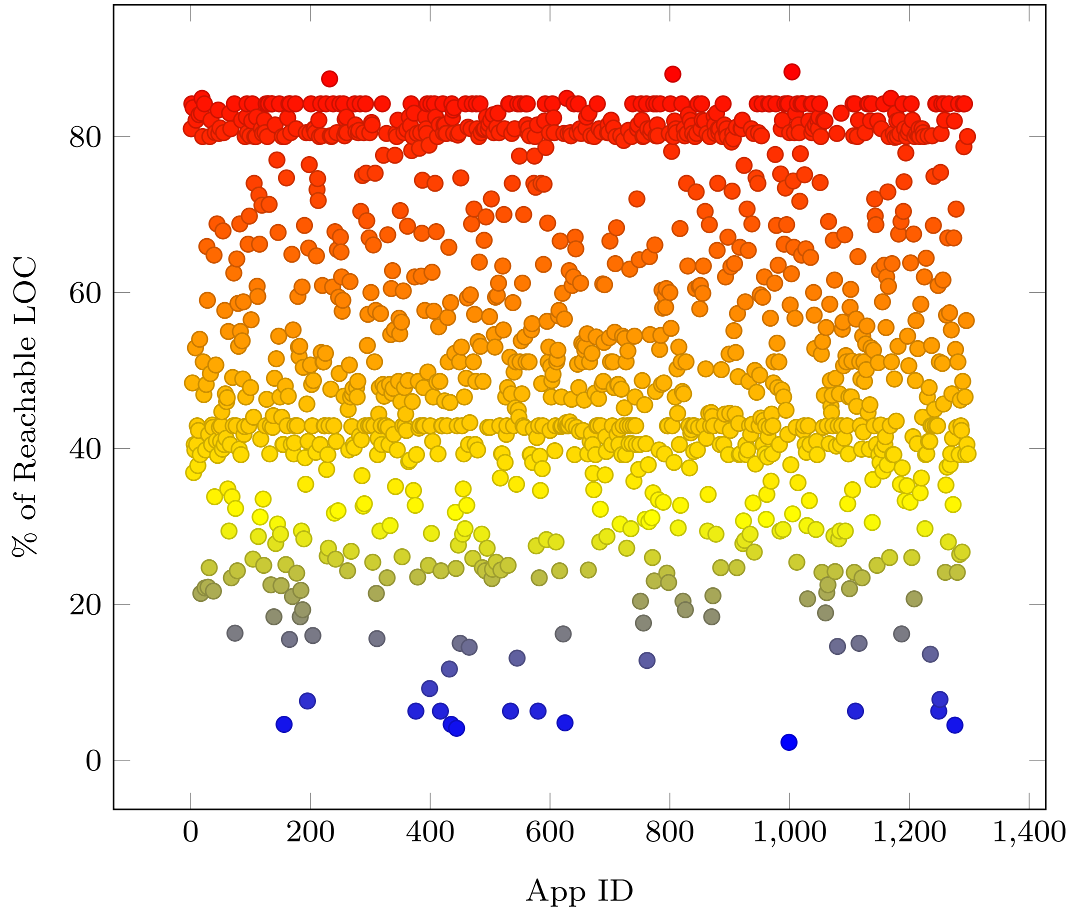


Figure 4.5: This chart shows the percentage of lines of code exposed to web by browsable activities.

which are reachable in the call graph are calculated. Our study shows that about 26% of apps expose more than 80% of the code-base to the web while only 10% of them expose less than 30% of their code-base to web. The high percentage of reachable code-base to the browsable entry points show that developers might not have isolated the implementation for handling intent hyperlinks carefully. Also, we found that none of the apps which expose less than 10% of their code-base to the web are vulnerable to the W2AI attacks. This suggests that as expected, there is a correlation between code exposure and possibility of exploitation.

4.2.2 Intent Hyperlinks Loaded in the WebViews

We observe that Android developers often load untrusted and arbitrary URLs which are obtained from the intent hyperlinks in their customized internal WebViews. Our results in Section 4.4 show that 84 applications in our data set load such URLs in their internal WebViews which cause the browsable activities to be

invoked. This large number of vulnerable apps suggest that the developers might use the web-to-app channel as a source of URLs which can be processed by the app without additional security checks and that they don't understand their security implications.

Whitelisting on URLs in WebView. To abuse WebView interfaces, the web attacker needs the vulnerable app to load the malicious URL from the URI intent in the customized WebView. Setting the proper whilelist for URLs to prevent the unauthorized web pages being loaded in the app's WebView can mitigate W2AI attacks to abuse WebView interfaces. Currently, some hybrid frameworks (e.g., PhoneGap) provide such whitelisting rules [WHI], but the vulnerable apps in our experiments do not deploy the proper settings. Researchers have also proposed origin-based defenses that require developers to set whitelisting policies [WXWC13], however, they have not been widely deployed on Android.

4.2.3 W2AI Vulnerabilities due to Code Reuse

Developers are often keen to either extend existing frameworks or copy code from one program to another. In order to provide the transparency between the web and applications and a seamless experience when finding and using information on the Internet, existing frameworks usually support and utilize intent hyperlinks. Therefore, a developer who embeds an existing code-base to her application might not even notice that it is exposed to the web.

While creating a hybrid Android application, integrating both web and native Android functionalities, many developers prefer to use existing frameworks such as PhoneGap [PHO], SeattleClouds [sea], etc. In addition, providing special services such as sharing data on social networks, cloud storage, etc., requires incorporating SDK libraries into the application. Even though the developers are generally aware of the need for security, the inherent security of the used frameworks largely affects the security guarantees of the application.

Even though some of these frameworks do have built-in protection, they have very little or none at all for intent hyperlinks. The major drawback of building applications on top of the frameworks is the inevitable security flaws in the framework itself. Even if developers pursued best practices, the resulting application would remain vulnerable. It is necessary to factor in additional research on security properties of frameworks before picking them. However, the developers may not be equipped to do so.

Table 4.1 shows the security control parameters that we have used to analyze the protection mechanisms of existing Android frameworks against web-to-app

injection attacks.

The first parameter which is crucial to avoid W2AI attacks is input validation and sanitization. Input validation checks if the input meets a set of criteria (such as a string contains no standalone single quotation marks). Input sanitization modifies the input to ensure that it is valid (such as doubling single quotes). These two techniques are used to protect the program from injection attacks. For instance, the PhoneGap framework[12] extracts an extra parameter `url` from the intent hyperlink, applies an input validation by checking whether it starts with `javascript:` or `file://`. Otherwise, it checks if the url follows a specific pattern. Unfortunately, most of the developers choose ".*" as the pattern which allows any URL to be loaded in the Webview of the local app. This vulnerability leads to attacks such as accessing contacts, local files, device information and so on. Unfortunately, this platform does not apply any sanitization and the extra parameters reach security critical methods without being modified.

PhoneGap does not check the origin of the Intent messages. The data passed through the intent hyperlinks are sent as raw strings. Hence, any third-party app which can hijack the intent on the phone can see the query parameters and data. Finally, this framework does not use the web-to-app channel for authentication or authorization.

Many developers often integrate cloud storage services using available SDKs like Dropbox. Dropbox provides SDKs both for Android and iOS to allow applications to store data on the cloud. Android developers can include the Dropbox SDK in the application as a library to authenticate or authorize the user. The Dropbox SDK leverages the web-to-app channel for this purpose as shown in Figure 4.6. Unfortunately, due to the lack of proper input validation and authorization, a web adversary can exploit W2AI vulnerabilities inherent in this SDK and launch authentication exfiltration attacks. In this case, even if the developer is cautious about the security aspects of the application, she ends up publishing a vulnerable application due to the vulnerability in the Dropbox SDK.

Tencent (QQ) framework is another well-known SDK which is mainly included by Android developers to authenticate or authorize users for sharing data on the social networks. This framework provides several implementations for this service and one of them utilizes the web-to-app channel as shown in Figure 4.6. Moreover, this SDK has W2AI vulnerabilities mainly due to the lack of proper input validation and sanitization which allows adversaries to launch injection attacks such as Cross-Site Scripting [XSS].

[12]PhoneGap [PHO] is a distribution of Apache Cordova [COR]. The actual code-base that implements the web-to-app channel is part of the Apache Cordova.

Table 4.1: Security control matrix for W2AI attack in Android development frameworks.

Framework	Version	Input Validation /santization	Checking Origin	Leakage	Authentication / Authorization
PhoneGap	3.3.0	Partial	No	Yes	Not Supported
Dropbox SDK	1.6.1	Partial	Yes	Yes	Can be bypassed
Tencent (QQ) SDK	2.8	Partial	Yes	Yes	Can be bypassed
SeattleClouds	6.5.1	Partial	No	Yes	Not Supported

SeattleClouds is another framework utilized by many developers to build hybrid apps. This framework also provides financial services such as PayPal transactions. Unfortunately, due to the inherent flaws in the SDK, web adversaries can pop up arbitrary messages on the phone such as "transaction status failed" or they may force the app to open arbitrary files. Similar to the previous frameworks, the W2AI vulnerabilities in this SDK make the application vulnerable while the developer might not even be aware of the problem. These vulnerabilities could be effectively avoided with proper input validation and sanitization.

4.2.4 Design Problems of Web-to-App Channel

Even though web-to-app channel is getting more and more popular, its design does not seem to be mature enough. For instance, intent hyperlinks are sent in plain text as part of the URL both for app indexing (see Section 4.1.1) and sensitive information transmission (e.g., authentication tokens) which may result in confidentiality issues.

Sometimes intent hyperlinks are intended to be used for side-effect-free actions. App indexing is one example where an application is invoked as the result of a search action and no further side-effect is expected to happen. On the other hand, an intent hyperlink might be received and processed if it is sent from a trusted party. If these two scenarios were distinguished systematically (i.e., transparently by the Android platform), Android developers could handle these requests more reliably. Such design problems have been raised already and addressed in the web context where sender and receiver are web client and server. In this section, we briefly describe the key concepts of HTTP protocol used in the World Wide Web which can possibly be re-defined for web-to-app interactions in Android.

The HTTP protocol defines request methods like GET and POST to clearly identify the action to be performed on a specific resource (e.g., a file). The information returned to the user depends on the implementation of the server.

`GET` **Requests.** The `GET` method is designed to request information from a server. Such requests should only retrieve data [FGM+99]. This guideline specifies that this method should be safe which means that it should not have any side-effect on the server. The `GET` request essentially embodies some data that is passed via the URL's query string. The query string specifies the data that needs to be retrieved from the server such as date, account balance, etc. Web crawlers such as search engine indexers normally use the `GET` and `HEAD` methods exclusively, to prevent their automated requests from performing such actions. HTTP `GET` should not be used where sensitive information, such as user names and passwords, have to be submitted along with other data for the request to complete.

Side-effect-free intent hyperlinks (e.g., used for app indexing) resemble `GET` methods supported by the HTTP protocol and hence they can be considered safe. The important difference between intent hyperlinks and `GET` messages is that unlike `GET` requests, intent hyperlinks do not send back any direct response to the web user. Instead, an application is launched on the phone.

`POST` **Requests.** The `POST` method requests that the server accept the data enclosed in the request as a new subordinated web resource identified by the URI. The data `POST`ed might be, for example, a message for a bulletin board, newsgroup, mailing list, comment thread or an item to add to a database. `POST` requests allow altering data on the server side. The action performed by the `POST` method might not result in a resource that can be identified by a URI. Intent hyperlinks which are used for authentication or file management are akin to `POST` requests which should be transmitted more securely and handled more cautiously.

Intent hyperlinks are sent in plain text. Unlike `GET` messages which are explicitly advised not to include sensitive data, there is no such restriction for intent hyperlinks. The sensitive data is advised to be sent via `POST` requests in the HTTP protocol. Since there is no such distinction available in the web-to-app channel design, developers have no choice but to send the sensitive data in the URI itself. Similar to HTTP requests, intent hyperlinks are also prone to attacks based on file and path names. The RFC guideline [FGM+99] clearly warns HTTP servers not to process arbitrary files or leakage of configuration/setting files. However, there is no such guidelines for smartphone developers.

HTTP Headers. HTTP requests can have security-related headers configured by the web server:

`strict-transport-security` is an HTTP header which enforces secure (HTTP

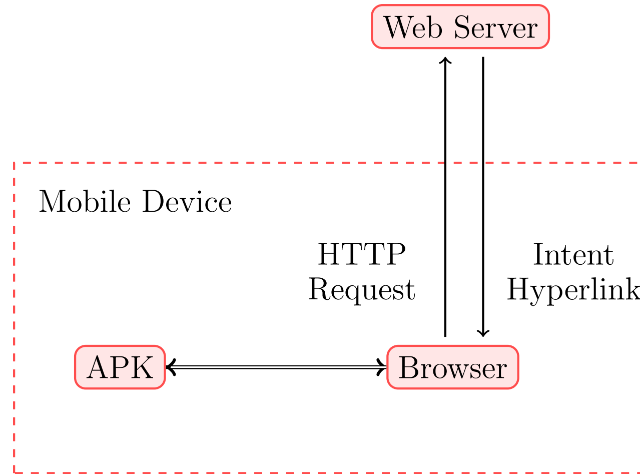


Figure 4.6: Two-way communication between the Android app and web.

over SSL/TLS) connections to the server. Another HTTP header example is `content-security-policy` which can prevent Cross-Site Scripting [XSS] and other cross-site injection attacks.

Unfortunately, these security-related HTTP headers are not available in intent hyperlinks. Therefore, communications from the Android app to web might be problematic as shown in Figure 4.6. Assume an Android application establishes a connection with a web-server using the HTTP protocol while the web server responds using intent hyperlink. Since the security-related HTTP headers are not available in the intent hyperlinks, this transaction might lead to security problems. Unfortunately, there are Android SDKs in the wild such as QQ that support identical scenarios and are incorporated in many real world apps.

In the HTTP requests, it is possible to set the referrer field which identifies the address of the webpage (i.e., the URI) that is linked to the resource being requested. By checking the referrer, the new webpage can see where the request originated from. The Android API level 22 and later versions have a similar additional feature that provides a way for developers to figure out the sender of an intent. `EXTRA_REFERRER` and `EXTRA_REFERRER_NAME` are extra parameters that allow the sender of an intent to set `Uri` object or `Uri` string respectively to help the recipient application know where the intent comes from. Recipients can obtain this extra parameter by either calling `getReferrer()` or directly reading the parameter. However, this *is not* a security feature – developer cannot simply trust the referrer information, since applications can spoof it.

As we have discussed in this section, unlike the web world, there is no `GET/POST` notion defined for the intent hyperlinks. Therefore, the same code-base that handles intent hyperlinks will be used for requests with and without side-effects. In

fact, whether the intent hyperlinks have side-effect or not is highly dependent on the Android application itself. Our results in Section 4.4 show that implementations handling intent hyperlinks in many existing applications indeed have side-effects, while the same code-base is used for app indexing. Unfortunately, they don't distinguish safe and unsafe requests – the same code-base handles any arbitrary intent hyperlinks in the same way. If developers want to provide both functionalities, i.e., app indexing and processing data in the intent which possibly has side-effects, they need to implement the code for distinguishing the requests by themselves since Android does not provide an interface for that.

4.3 Detection and Exploitation of W2AI Vulnerabilities

We aim to examine the prevalence of W2AI attacks at a large scale. For this purpose, we customize and use the vulnerability detection system that we introduced in Chapter 3 and call it W2AIScanner [HJY+15]. Next, we discuss the customization needed for some of the stages of the analysis.

4.3.1 Source-Sink Pair Identification for W2AI Attacks

We have modified the entry point selection implementation to pick the browsable activities. Starting from the browsable activities, FlowDroid finds pairs of source and sink program points using dataflow analysis which will be utilized in the sink reachability analysis.

The `getIntent()` and `onNewIntent()` methods are the source methods that fetch the Intent objects which start the activity and provide data inputs to the app from an intent. `onNewIntent()` is called for activities that set launchMode to `singleTop` or the `FLAG_ACTIVITY_SINGLE_TOP` flag is used in the Intent which invokes the activity. We choose a subset of the sink methods provided by Susi[RAB14] based on the attack categories discussed in Section 4.1.3.

4.3.2 Symbolic Execution

The symbolic variables for analyzing W2AI vulnerabilities are the intent objects received in browsable activities and their fields such as URI data, action, extra parameters, etc. Our analysis starts with a set of pre-conditions created based on the intent-filters specified in the manifest file for browsable activities. For instance, the intent-filter in the manifest file might be defined to only accept intents whose

actions are specific string constants. By adding these constraints to the path formulas, we avoid generating non-realistic exploits.

4.3.3 W2AI Attack Validation

W2AIScanner has to automatically generate intent hyperlinks that exploit vulnerabilities in Android apps. An intent hyperlink can be divided into two parts: (i) the action, category and parts of the data URI which have to be matched with the intent filter for the activity defined in the manifest file; and (ii) the data inputs which are in the form of key-value pairs.

The first part is collected by our manifest parser component which we use to obtain the intent filter specification for the source activity. It creates all possible schemes that will match the intent filter. Path is one part of the data elements in intent filters that will be checked for accepting an intent. The Android developer can specify a special form of regular expressions in the manifest file to match the intent hyperlink path pattern. W2AIScanner uses the same algorithm implemented in the Android framework to match against intent hyperlink path patterns and generates a counter example. Some of these values might also be obtained from the symbolic execution phase in which case we directly use the values generated by our symbolic executor.

The second part which includes the key-value pairs of input data are obtained by our analysis. Usually, the keys in key-value pairs are resolved using copy constant search as explained in Section 3.5.2 and the values are generated as instances of symbolic variables. It is also necessary to resolve the types for extra parameters which is dealt with in the symbolic execution phase (e.g., by evaluating cast operations). When an intent hyperlink is clicked by the user and the target application is invoked, the data inputs which make up an intent hyperlink are derived from the Intent class methods. In this chapter, we discussed that an intent hyperlink follows a specific syntax:

intent:HOST/PATH?[$string_1$]=[$string_2$]#Intent;action=[$string_3$];
scheme=[$string_4$];[$type_1$].[$string_5$] =[$string_6$];end

where data input can be sent through the fields (e.g., [*string*]) and their types can be specified using [*type*].

Once we have the key-value pairs and their types, other necessary inputs for the source-sink flows and the intent filter specifications for the target browsable activity, all of these elements are put together to generate an intent hyperlink.

The validation component confirms whether the generated intent hyperlink

is a true positive exploit. In this chapter, we classified W2AI vulnerabilities and showed that these classes depend on the categories of the sink methods. If the sink method is reached on the execution trace, W2AIScanner applies different policy checks. There are two main classes of vulnerabilities: abusing WebView interfaces and abusing native app interfaces.

The first category is validated by sending a malicious website URL through the intent hyperlink parameters. When a vulnerable application loads the malicious URL, the data retrieved from the device is sent to our test server and we can confirm the attacks according to the category settings. The Policies/Settings column in Table 4.4 shows the methods for the representative sinks in our attack categories that should be invoked on the execution path for an attack to be confirmed. As an example, if the app has flows that reach the `WebView.loadUrl` sink and enables `setAllowFileAccess`, `setJavaScriptEnabled` and `setAllowFileAccessFromFileURLs` settings, the app is vulnerable to local file inclusion attacks.

Attacks on abusing native app interfaces are more complex to validate. If the sink method is reached on the execution trace, we check the concrete values of the sink method parameters. We compare the values resolved for the sink method parameters in the symbolic execution phase with the values observed after running the intent hyperlink at the sink invocation site. Then, we look for the category setting method invocations on the execution path and report the intent hyperlink if the vulnerability is confirmed to be exploitable.

4.4 Experimental Results for W2AI Vulnerability Detection

In this section, we employ W2AIScanner on real-world Android applications and report our results. We assess the prevalence of web-to-app injection attacks at a large scale. We choose the top 100 apps of all categories in Google Play plus the data set used in [JHY+14] which are 12,240 in total. Among these apps, 1,729 apps have at least one browsable activity. Since numerous apps in the data set used in [JHY+14] were out of the shelf (this data set contains 15,510 apps), we could download only 9,877 apps on Google Play in April, 2014.

New Vulnerabilities Found: W2AIScanner detected and validated 286 exploitable vulnerabilities of 134 apps which corresponds to 7.75% of apps with browsable activities being vulnerable. Since Google Play has millions of applications ready to download, we believe that we can expect a higher rate for the whole

Table 4.2: Overall statistics of vulnerable apps in each W2AI attack category.

Category	Sub-Category	# of Sinks	# of Vulnerable Apps	ID
Abusing Web-View Interfaces	Abusing JavaScript-to--Native Bridge	9	52	1
	Abusing HTML5 APIs	10	29	2
	Local File Inclusion	9	63	3
	Phishing	11	84	4
Abusing Native App Interfaces	Database Access	14	10	5
	Persistent Storage Pollution	72	7	6
	Open Re-delegation	39	23	7
	App-Specific Logic Flaws	16	18	8

app store.

Experimental Setup

We have conducted a systematic analysis on 12,240 apps in Ubuntu 12.04 on an Intel Core i5-4570 CPU PC desktop (3.20GHz) with 16 GB of RAM.

To validate the exploits that abuse WebView interfaces, W2AIScanner sets the malicious site's URL (our attack web server) as a parameter in the malicious intent hyperlink and utilizes the `adb` command, as shown below, to automatically launch the corresponding activity via the intent hyperlink in the emulator or Android device.

```
adb shell 'am start "intent://#Intent;scheme=worknet;action
=android.intent.action.VIEW;S.url=http://attacker.com;end"'
```

Thus, the vulnerable app loads the attacker's page into its WebView, the test webpage calls the exploitable JavaScript functions for that category of vulnerability identified by W2AIScanner to retrieve the data and send it back to our test server. By checking the responses in the server, we can automatically confirm whether there are exploitable vulnerabilities in apps from the category of W2AI attacks which abuse WebView interfaces.

To validate the exploits that abuse Android native app interfaces, W2AIScanner utilizes the `adb` command to launch the activity to perform privileged operations (e.g., inserting data into the app's database). By analyzing the concrete values in the log traces and mapping them to the Jimple registers as explained in 4.3, W2AIScanner can automatically determine whether the exploits work (e.g., successfully pollute the app's database) or not.

4.4.1 Prevalence of W2AI Vulnerabilities in Apps

We ran W2AIScanner on 1,729 apps and detected 286 W2AI vulnerabilities in 134 apps. This shows that our system can scale as a vulnerability detection tool

for W2AI attacks. (We, in fact, analyze 12,240 apps, first rejecting those without browsable activities).

Table 4.2 gives a breakdown of the detected vulnerable apps into our 8 categories for W2AI attacks. The column, # of sinks, gives the number of sinks defined by our specification for that category. There can be overlaps among the different categories of sinks. For example, `WebView.loadUrl` can be the sink for the first 4 categories. In total, we have specified 153 distinct sinks for 8 categories in our attack specification. Table 4.4 shows the representative sink method for each attack category.

The column, # of vulnerable apps, gives the number of apps which are vulnerable to a class of vulnerability. An app can be vulnerable to more than one attack category. For instance, the WorkNet example has vulnerabilities from categories with ID 1 to 4. Thus, the sum of that column is greater than the number of vulnerable apps.

We have found at least one application with more than 1 million downloads for each category. One of the popular applications is Wikipedia (1.3.4) which is vulnerable to categories with ID 2 and 4. We have also detected and confirmed 14 Dropbox applications to be vulnerable to open-redelegation attacks where attacker can force them to invoke other apps on the phone. One app-specific logic vulnerability appears in 587 apps so we count it as one unique vulnerability to avoid skewing the results. Once this vulnerability is exploited, attacker can send fake PayPal payment notifications to the phone. In Section 4.4.4, we present case studies that show detailed results on 8 applications which are representative of each of our attack categories.

Validation on Data from Intent Hyperlinks. In our experiments, we have found 46 vulnerabilities in Android apps that pass data from intent hyperlink to the native Android interfaces as parameters without proper validation. We believe that by validating and sanitizing the data from the intent hyperlink before passing it to the native interfaces, the app developer can effectively fix such W2AI vulnerabilities. For instance, Android apps should not process arbitrary URIs referencing files (e.g., configuration/setting files).

4.4.2 Effectiveness of W2AIScanner in Detecting W2AI Vulnerabilities

We successfully generate accurate intent hyperlinks that follow complex patterns and allow us to find zero-day vulnerabilities. For example, `Letv` is an Android app which only processes an intent hyperlink if it has a query parameter with

`from` as the key and `baidu` as value. Another example is `Kobobooks` which requires that action parameter of the intent hyperlink that invokes the app be not equal to `android.intent.action.VIEW`. W2AIScanner can successfully report such paths as infeasible and avoid false alarms by the use of symbolic execution and validation. An alternative approach to symbolic execution is fuzzing but we believe that any fuzzing without some symbolic reasoning is unlikely to give good results.

4.4.3 Reporting Vulnerabilities to Vendors

Using our vulnerability detection system, we have found several critical zero-day W2AI vulnerabilities. In this part, we explain the steps we took to detect and report these vulnerabilities.

Analyzing the applications in our data set, we found a critical W2AI vulnerability in some of the PhoneGap hybrid applications in July 2014.[13] Our system is able to automatically generate proof of concept exploits for these applications. The static analysis in our system reported two categories of vulnerabilities for these apps: Abusing WebView Interfaces and Open Re-delegation. However, only the first category of vulnerabilities were confirmed by our system. The attacks for the second category of vulnerabilities failed due to a null pointer dereference which would cause the application to crash. We have reported these vulnerabilities to all of the application vendors and got positive feedback from some of them.

We also found a vulnerability in Dropbox SDK used in Android applications where attacker can provide a redirection URL through which they are able to launch open re-delegation attacks and also inject OAuth tokens. We detected this vulnerability in July 2014 and reported to some of the vendors.

Our system has detected and confirmed a W2AI vulnerability in McAfee application (an antivirus for Android apps) which enables attackers to launch phishing attacks.[14]

We have reported all of the vulnerabilities explained above as well as the rest of the vulnerabilities to Google. We have also provided the proof of concept intent hyperlinks to exploit these vulnerabilities.[15]

We have also noticed a dangerous use of intent hyperlinks in some of the applications that integrate QQ authentications. These attacks might result in Cross-Site Scripting [XSS] and token exfiltration attacks. The security team in QQ has confirmed the vulnerability and introduced a patch in the subsequent releases of the SDK.

[13] The Cordova vulnerability [Kap] was reported afterwards in August 2014 by IBM security..

[14] Unfortunately the McAfee security team has not responded yet.

[15] Google has encouraged us to release our system to be used in the application vetting process.

Table 4.3: Representative vulnerable apps for each W2AI vulnerability category.

ID	App	Version	# of Downloads
1	WorkNet (kr.go.keis.worknet)	3.1.0	1 - 5 M
2	Wikipedia (org.wikipedia)	1.3.4	10 - 50 M
3	WeCal Calendar (im.ecloud.ecalendar)	3.0.8	1 - 5 M
4	IPharmacy (com.sigmaphone.topmedfree)	1.0.92	1 - 5 M
5	i2X RDP (com.tux.client)	11.0.1899	1 - 5 M
6	MoneyControl (com.divum.MoneyControl)	2.0	1 - 5 M
7	Caller ID (com.callapp.contacts)	1.56	1 - 5 M
8	Sina Weibo (com.sina.weibo)	4.3.0	5 - 10 M

ID: Category ID
App: Representative App (Package Name)
Version: App's Version
Download: # of Downloads (Million)

Table 4.4: Sinks and policies/settings for representative apps.

Category	Sub-category	Representative Sinks	Policies/Settings	ID
Abusing WebView Interfaces	Abusing Java-Script-to-Native Bridge	`WebView.loadUrl`	JavaScript-to-native interfaces, `setJavaScriptEnabled`	1
	Abusing HTML5 APIs	`WebView.loadUrl`	`setGeolocationEnabled`, `setJavaScriptEnabled`	2
	Local File Inclusion	`WebView.loadUrl`	`setAllowFileAccess`, `setJavaScriptEnabled`, `setAllowFileAccessFromFileURLs`	3
	Phishing	`WebView.loadUrl`	`setJavaScriptEnabled`	4
Abusing Native App Interfaces	Database Access	`SQLiteDatabase.insert`	-	5
	Persistent Storage Pollution	`SharedPreferences.Editor.putString`	-	6
	Open Re-delegation	`Class.forName`	-	7
	App-Specific Logic Flaws	`TextView.setText`	-	8

4.4.4 Case Studies

In what follows, we detail the reached sinks which are exploitable and the damage caused by the vulnerabilities for each representative app in Table 4.3.

Abusing JavaScript-to-Native Bridge. WorkNet with 1 - 5 million (M) downloads provides job information in Korea. W2AIScanner detects `WebView.loadUrl` in this app. This app enables settings for JavaScript and JavaScript-to-native interfaces in its configuration file (`config.xml`). After running W2AIScanner on WorkNet, we have detected and exploited the `WebView.loadUrl` sink. This app enables the following settings: `setJavaScriptEnabled`, `setGeolocationEnabled`, `setAllowFileAccess`, `setAllowFileAccessFromFileURLs`. Hence, the web attacker can mount all the attacks in the abusing WebView interfaces category on WorkNet. As we have explained in this chapter, its WebView which loads arbitrary URLs exposes the Java native methods to the JavaScript code. Once the user clicks the malicious link, WorkNet loads the URL from the intent hyperlink's parameters in the WebView. Therefore, the malicious page running in the WebView can invoke 21 JavaScript-to-native interfaces to access private user data (e.g., contacts) and perform privileged operations (e.g., modifying local files).

Abusing HTML5 APIs. Wikipedia with 10 - 50 M downloads is the free encyclopedia containing more than 32 M articles in 280 languages. It contains 2 paths that reach the `WebView.loadUrl` sink and enables JavaScript and `geolocation` settings. The combination of this sink and setting enables the malicious site running in the WebView to access the GPS sensors and send out the user's current location to the attacker to track the user at any time.

Local File Inclusion. WeCal Calendar is a calendar assistant, which synchronizes with the Google calendar, takes notes, sets alarm and so on. W2AIScanner detects that the app has flows that reach the `WebView.loadUrl` sink and enables settings for JavaScript and the file's access. The settings are: `setAllowFileAccess`, `setAllowFileAccessFromFileURLs`. After validation, we find that with loading the local HTML file (whose URL comes from the intent hyperlink) in the WebView, the file can utilize `XMLHttpRequest` to read the local files (e.g., /etc/hosts) and leak the content to the attacker.

Phishing. IPharmacy with 1 - 5 M downloads provides medical products. W2AIScanner detects that the `Webview.loadUrl` sink in this app is reachable and exploitable. Therefore, this app can be exploited to load a phishing page whose URL is derived from the intent hyperlink crafted by the web attacker in the customized WebView.

Database Access. 2X RDP Client is a popular remote desktop app. The exploitable sink reported by W2AIScanner is `SQLiteDatabase.insert`, which adds items to `farms` table. The web attacker can set sensitive attributes, e.g., credentials, in the intent hyperlink to pollute the app's database.

Persistent Storage Pollution. MoneyControl is a popular business and marketing app. W2AIScanner detects paths that inject data to the `SharedPreferences.Editor.putString` sink. Exploiting this vulnerability, the web attacker can make permanent changes to the storage.

Open Re-delegation. Caller ID - Call Blocker is a caller-ID app in Google Play that identifies a billion unknown callers. The reached sink for this app is `Class.forName`. The attacker can set a private activity's name in the intent hyperlink's parameters. When this app launches with the malicious intent hyperlink, it invokes the designated activity.

App-Specific Logic Flaws. Sina Weibo is a microblogging client for Android phones. A W2AI vulnerability in this application allows the attacker to show arbitrary titles to the victim user (e.g., attacker can set the title to `https://www.paypal.com`) which can be used in social engineering attacks. The vulnerable sink in this application is `TextView.setText`. The attacker can launch an injection attack by putting an arbitrary title as query parameter in the mali-

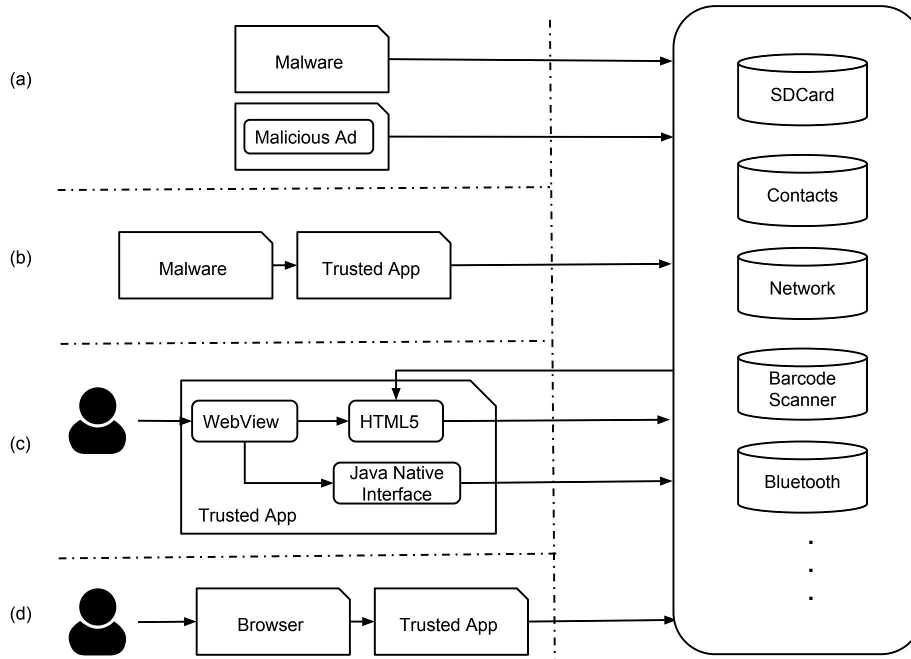


Figure 4.7: Attacks in Android smartphones can be classified to four categories: (a) over-privileged malware; (b) privilege escalation; (c) injection attacks in HTML5 and WebView attacks; and (d) web-to-app injection attacks;

cious intent hyperlink.

4.5 Related Work: Attacks on Android Apps

The popularity of smartphones along with the fact that these devices store a large amount of private user data (e.g., contacts, user data files and geolocation) has drawn much interest in the security research community. The Android operating system restricts access to the user's private data through a permission-based security model. We compare three classes of attacks targeting Android smartphones with the new class of attack (Web-to-App Injection) that is studied in depth in this chapter.

4.5.1 Over-Privileged Malware

The first attack model depicted in Figure 5.1-(a) assumes that the Android user installs a malicious application and grants critical permissions (e.g., sending SMS). Android malware may use several ways to seem legitimate to the Android users [ZJ12]. Felt et al. showed that it is very common to find over-privileged Android apps in the app-stores [FWM+11]. Sometimes, a malicious advertisement

library integrated into an application forces the app to request for high-privilege permissions [GZJS12]. Since the Android permission system does not separate privileges of Android apps from the embedded third-party libraries, the privileges will be granted to the libraries as well. In our attack model, we don't need any kind of malware to be installed on the phone.

4.5.2 Privilege Escalation

Figure 5.1-(b) demonstrates another class, known as Android privilege escalation attacks. While Android provides a well-structured permission system, it does not protect against transitive permission usage. This ultimately results in allowing an adversary to perform privileged operations (e.g., sending premium SMS) which the sandboxed application is not authorized to do [DDSW10, LLZW14, CQM14, ZLZ+14]. This class of attacks target unprotected components in benign applications. Enck at al. [EOMC11] have discussed the existence of unprotected broadcast receivers which receive arbitrary intents from other applications on the phone. ContentScope [ZJ13] is another work for finding pollution and leakage attacks on content providers in Android applications. If a vulnerable Android application exposes a content provider component without any protection, installed malware on the phone can launch pollution and leakage attacks to steal data or manipulate the sensitive data. These works all assume that the malicious apps are present on the victim's Android device.

4.5.3 HTML5 and WebView Injection Attacks

More recently, it has been shown that the WebView and hybrid apps [LHD+11, JHY+14, GJS14, CW14] can lead to new classes of attacks (See Figure 5.1-(c)). Luo et al. [LHD+11] observe that malicious JavaScript code can access the sensitive resources via the native bridge by invoking `addJavascriptInterface` method. Georgiev et al. carry out an analysis on hybrid apps, and demonstrate vulnerabilities that affect the native bridge mechanisms [GJS14].

Jin et al. introduce code injection attacks on HTML5-based mobile apps via internal and external channels [JHY+14]. These attacks require the user to use external resources such as bluetooth or barcode scanner to read malicious input which will potentially exploit vulnerabilities in the HTML5 code. Alternatively, the user has to visit the malicious page *directly* in the WebView of the hybrid apps. Our attack model differs, since, W2AI attacks utilize intent hyperlinks to convey the payload simply by clicking a link in the default browser, which is more probable than users choosing to browse a malicious page through a specific

app. Moreover, hybrid apps, like any other app can be prone to W2AI attacks. Therefore, these apps are only a subset of applications which are systematically studied in our work.

4.5.4 Web-to-App Injection Attacks

Figure 5.1-(d) shows our attack model, Web-to-App Injection attacks. We distinguish the threat models described above from the Web-to-App Injection attacks, by the assumption that the user chooses to install the malware app on her phone. However, the threat model studied in this chapter is much more probable and has less requirements since the user only needs to click a link to exploit a vulnerability. The attack targets are any third-party applications, including the hybrid apps in HTML5 and WebView injection attacks (figure 5.1-(c)). The difference between these two models is the program code where attacks start. In W2AI attacks, the default browser parses a malicious intent hyperlink and invokes the vulnerable app. Therefore, attacks start from Java code in the app and involve program paths which might finally reach HTML5 code and even exploit its vulnerabilities. In contrast, in HTML5 and WebView injection attacks, the malicious URL loaded inside the in-app WebView exploits vulnerabilities in HTML5 code and as a result may reach Java methods which are explicitly exposed via `addJavascriptInterface()`.

Some previous works have discovered attacks through the scheme mechanisms [WXWC13]. Rui Wang et al. reveal confused deputy attacks on Android and iOS applications which abuse channels provided by mobile OS. One of these channels is the scheme mechanism through which attacker can invoke apps on the phone by crafting intent hyperlinks and publishing on web. This work studies the problem where user surfs through the web in customized WebViews of benign applications and launches confused deputy attacks abusing the benign app's "origin". They present a CSRF attack on the Dropbox SDK on the iPhone [WXWC13] launched through an intent hyperlink. However, our attacks differ because our attack model is more general, the user clicks on an intent hyperlink in the default browser, so it does not need to be started from the benign app and can leverage trusted channels like the default browsers. More importantly, we investigate which vulnerabilities can be exploited once the attacker can manage to start an application via an intent hyperlink. We have conducted (the first) systematic and large-scale analysis of existing apps which shows that our analysis system is not only able to detect vulnerabilities with high precision but also automatically generate exploits.

Takeshi Terada [Ter] presents three browser vulnerabilities exploitable via In-

tent scheme URLs [Ter]. This work is based on manual analysis.

4.6 Summary

In this chapter, we have presented an in-depth study of W2AI attacks which can introduce a broad range of possible exploits (i.e., abusing WebView interfaces and native app interfaces) in installed Android apps. It can be a significant threat as no malicious apps are needed on the device and the remote attacker has full control on the web-to-app communication channel.

Web-to-app injection enables web-based adversaries to trigger intents (with arbitrary parameters) the same way as if the adversary had a malicious APK installed on the victim's device, without really installing the malware. Therefore, the web adversary essentially plays the role of a pseudo-malicious application which doesn't need any actual Android permissions (e.g., reading "public" files on compact flash drive) and does nothing except issuing (arbitrary) Intent messages. As no malware is needed for our attacks, the root cause of the problem is that apps do not validate Intent parameters. Even those users who are very careful to never install malicious or "pseudo-malicious" apps are vulnerable.

In this chapter, we have presented some Android-specific programming practices that might increase the W2AI attack surface. Looking at the most popular Android development frameworks, we have shown how reusing third-party software components lead to security vulnerabilities. We have also studied the request methods in the HTTP protocol to understand whether web-to-app channel can borrow similar concepts to allow the developers utilize it more securely.

To discover the prevalence of W2AI vulnerabilities, we have employed our analyzer to automatically detect the apps vulnerable to W2AI attacks and generate exploits for them. With our analyzer, we also validate the exploits for the vulnerable apps. By running a customized version of our analyzer, W2AIScanner, on 1,729 candidate (browsable) apps identified among the apps downloaded from the Google Play, we have found 286 vulnerabilities in 134 apps. Specially our system has automatically detected and confirmed devastating W2AI vulnerabilities in `PhoneGap` and `Dropbox` applications.

We have observed that developers often expose a large percentage of the app code-base to web without really needing to do that. We have also observed that in many apps the main activity is browsable. We recommend the developers to avoid making the main activities browsable and minimize the exposure to web to reduce the attack surface. We also believe that white-listing the URLs processed b the apps and validation and sanitization can help mitigating the W2AI attacks.

It would also be useful to distinguish the URI intents with and without side-effect to avoid vulnerabilities which can be exploited to control the behavior of the app. Finally, we suggest the developers to check the origin of the incoming intents and handle those coming from the web more cautiously.

Chapter 5

Detecting and Characterizing Database Attacks in Android Apps

Android apps often make use of data stored in databases. Furthermore, apps in Android are designed to provide functionality to each other, which means that an app can interact with the database(s) managed by another app. A vulnerability in an app can allow malware to violate the integrity and confidentiality of data stored in its databases.

Apart from ContentScope [ZJ13], there has been little work on detecting database vulnerabilities in Android apps. However, their work only studies vulnerabilities in public databases which are managed through the Android content provider APIs. On the other hand, content management in apps is not limited to content providers. Developers also utilize internal databases, private databases, without implementing content providers. Even though such internal databases may be intended to be used privately, malware may be able to launch pollution, leakage or file access attacks. We indeed show that vulnerabilities in the benign apps can have private database attacks. While studying the private database usage in apps which were previously only studied for public database attacks [ZJ13], we discovered new privilege escalation attacks which are triggered from private database vulnerabilities but end up exploiting protected content providers (public databases) of other apps.

In this chapter, we study and classify different approaches taken by Android developers to implement databases and the security controls used to protect them. This is used to design our analysis system. We also study whether the public database vulnerabilities in [ZJ13] still occur after Android's changes to secure

the default settings of content providers. In order to assess the security of the databases in real-world apps, we extend the analysis framework introduced in Chapter 3 and propose an end-to-end analysis framework, DBDroidScanner, which finds and confirms database vulnerabilities more comprehensively than [ZJ13]. We analyze for both public and private database vulnerabilities.

The Android database implementation relies heavily on URI objects. URIs are used in Android to reference resources such as text files, images or structured data. Paths in the app manifest referring to a particular set of data in content providers are based on URIs. Code in the apps using database methods often employ URI library methods. Inaccurate analysis of such libraries may result in generated exploits which do not work. Existing works [ZJ13, ARHB15, BJM+15] do not discuss much about (symbolic) models for URIs. As far as we are aware, we are the first to present symbolic implementations for the semantics of URI operations and related objects in Android.

Using DBDroidScanner, we analyze 924 apps which are among the top 100 apps of all categories in Google Play. We automatically detect and confirm that 153 database vulnerabilities are exploitable. We found some of the applications which were detected in [ZJ13] but since updated to still have vulnerabilities, e.g., `Maxthon Android Web Browser`. More importantly, there are apps where the content provider reported to be vulnerable to the public database attacks in [ZJ13] is updated with the vulnerability fixed, but it remains exploitable via other components due to private database attacks.

In summary, our contributions in this chapter are:

1. A comprehensive classification of database attacks in Android apps.
2. Accurate models for URI-based libraries which are essential for analysis of apps using databases.
3. A detection and exploit generation framework for zero-day Android database vulnerabilities.

5.1 Overview

Android developers often open up the databases implemented in their apps to "other apps" on the device using content provider components which provide APIs for public database access from other apps. They can also implement databases without exposing them through the public database mechanism by instead using the inter-app communication channel, i.e., intents. We call the former group

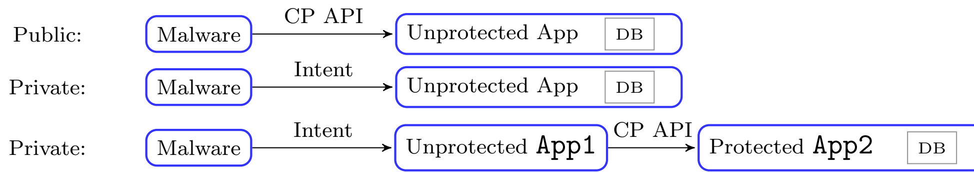


Figure 5.1: Database attack scenarios: CP API stands for Content Provider API.

of databases, *public databases* and the latter, *private databases*. A vulnerable database (public or private) which is not protected properly may compromise the security of the system. In addition, we show that public databases by design have more robust and reliable protection mechanisms. For our purposes, database means what Android provides for data storage, namely the `SQLiteDatabase` library and files (e.g. `ParcelFileDescriptor`).

5.1.1 Public Database Attacks

The first category of attacks targets the public databases which are accessible through content providers. Figure 5.1 shows the public database attack scenario where a malware app uses the parameters of an unprotected content provider API to exploit the database vulnerabilities of a victim app. In Android, components should be exported to be accessible by other apps by specifying the `android:exported` attribute in the manifest file. Content provider is a special case. Line 4 in Listing 5.3 shows the `android:exported` attribute of the content provider tag in the manifest file. By default, this attribute is set to `false` for Android SDK 17 (released Nov 2012) and higher which means that the content provider is not available to other apps. However, content providers in apps built for SDK 16 and lower are exported by default, hence accessible by all apps. In this thesis, we study the public database attacks for the SDK 17 and higher.[1] While the `android:exported="false"` attribute isolates a database from other apps, unfortunately, it is coarse-grained preventing all legitimate apps from using public database functionalities.

Developers can protect content provider components using existing or their own custom permissions.[2] We consider cases where content provider is not fully protected in our attack scenarios. Developers can restrict access to the data in content providers across applications at different granularities. Setting the `android:exported="false"` attribute is the least fine-grained option to protect

[1]ContentScope [ZJ13] analyzes apps for SDK 16 and lower versions.
[2]A custom permission is declared by the developer and has to be added to the manifest file separately. If the protection level of a permission is `normal`, all applications can get it.

the component. Alternatively, they can specify the following permissions: (1) `android:permission` for the whole component which is coarse-grained and prevents apps (malware) lacking this permission from directly accessing the content provider; (2) `readPermission` and `writePermission` which restricts access based on the request, i.e., query or data manipulation. These permissions are more fine-grained and partially protect the content providers; (3) `path-permission` for protecting particular sets of data in databases which is the finest-grained option protecting specific paths of the content providers.[3] More details on choosing the candidate content providers can be found in Section 5.4.1.

5.1.2 Private Database Attacks

The second category of attacks targets the private databases which are accessible through inter-app communication. An unprotected app has an unprotected component (except for content provider) which is exported[4] but not protected by any `dangerous` or more restrictive permissions. Figure 5.1 shows two private database attack scenarios. In the first scenario, the malware sends malicious intents to the victim app's components (e.g., activity) to exploit its database vulnerabilities. In the second scenario, two victim apps are involved: the malware first sends malicious intent to the victim `App1`; next, `App1` invokes the content provider APIs of victim `App2` which results in exploiting the vulnerabilities of the latter. Note that the victim `App2` might have correctly protected its content provider with permissions. It is also possible that the victim `App1` calls its own protected content provider APIs.

Unlike public databases, Android does not provide any explicit protection mechanism for private databases. Hence, developers have to implement their own (possibly buggy) access-control code to secure internal databases. If a component (e.g., a broadcast receiver) allows an app to access the private databases by sending messages (e.g., intents), there is no further access control in Android to protect these private databases. However, many Android apps heavily rely on these private databases to organize various data types such as contacts and app private information.

Intents are the main means of communication in Android and developers often fail in checking its origin properly. They may use intents for communication among

[3]Android allows developers to temporarily override the content provider permissions using the `grantUriPermission`. This case is not considered as the app which owns the content provider should explicitly send an intent or call `grantUriPermission()` method to allow the app to temporarily access its data.

[4]I.e., `exported="true"` is specified. Alternatively the `exported` attribute is not specified explicitly but the component has intent filter.

internal components, but forget that intents can also be created and sent by other apps. Handling an incoming intent which modifies the internal database's data is not different from handling other intents and this might result in the programmer's confusion. As a result, intents may trigger undesired behaviors which result in security violations of private databases. Apps may accidentally expose access paths to private databases by allowing portions of the input string from an intent to directly be passed to the SQL methods (e.g., `insert()`), which allows attackers to manipulate the database. We remark that the "private databases" discussed in [ZJ13] are considered here as public databases since they can be accessed through the content provider APIs.

We adopt the classification in ContentScope [ZJ13] and categorize our public and private database vulnerabilities to leakage and pollution. A leakage vulnerability results in leaking sensitive data while pollution vulnerabilities allow attackers to manipulate the data. Additionally, we use the file access category for attacks which allow the attackers to control obtaining the file descriptor for the database files which may cause both leakage and pollution of data.

The question of whether the developer intends certain functionality allowing other apps (malware) to access its database is a tricky one. For the public database attacks, one view is that only protecting a specific set of paths of the content provider in the manifest file is intentional. However, a valid argument is that the developer may have chosen a wrong path pattern, thereby creating a vulnerability – this seems to be the case for some of real app vulnerabilities found. Private database attacks are even more susceptible to unintended behavior as the protection in Android is limited only to the permissions specified for the entire exported component which is too coarse-grained. Hence, developers might give up on protecting a component since it restricts the app functionality too much without realizing its exposure to attacks. In this work, we are conservative and make the reasonable assumption that a database which is available to all apps including malware but not fully protected is a candidate to be analyzed for public and private database vulnerabilities.

5.2 Motivating Examples

Listing 5.1 and 5.2 show two example components of app `A` (our example benign victim app) which are vulnerable to the public and private database attacks respectively. App `M` in Listing 5.4 is the malware app which exploits the vulnerabilities in app `A`.

```
1  public class PublicDatabase extends ContentProvider{
2    UriMatcher uriMatcher = new UriMatcher(0);
3    PublicDBHelper dbHelper;
4    public boolean onCreate(){
5      dbHelper = new PublicDBHelper(...);
6      uriMatcher.addURI("com.example.app.PublicDatabase", "#", 1);
7      uriMatcher.addURI("com.example.app.PublicDatabase", "contacts/", 2);
8      return true;}
9    public Uri insert(Uri uri, ContentValues values){
10     SQLiteDatabase db = dbHelper.getWritableDatabase();
11     switch(uriMatcher.match(uri)){
12       case 1:{
13         String table = getTableName(uri.getLastPathSegment());
14         if(!table.isEmpty())
15           return db.insert(table, null, values);
16         return null;
17       }
18       case 2:{
19         return db.insert("Contacts", null, values);
20       }
21       default:
22         return null;
23     }
24   }
25   String getTableName(String id){
26     switch(id){
27       case "1":
28         return "ScheduledMessage";
29       default:
30         return "";
31     }
32   }
33   ...
34 }
35 class PublicDBHelper extends SQLiteOpenHelper{
36   public void onCreate(SQLiteDatabase db) {
37     db.execSQL("CREATE TABLE ScheduledMessage (message TEXT, ... )");
38     db.execSQL("CREATE TABLE Contacts (_id INTEGER, ...)");
39   }
40   ...
41 }
```

Listing 5.1: Public database example code in app A.

```java
public class ExampleBroadcastReceiver extends BroadcastReceiver{
  PrivateDBHelper  dbHelper;
  ...
  public void onReceive(Context paramContext, Intent paramIntent){
    ContentValues values = new ContentValues();
    if(paramIntent.hasExtra("task")){
      String task = paramIntent.getStringExtra("task");
      String data = paramIntent.getStringExtra("data");
      values.put(task, data);
      if(values.containsKey("message")){
        int contact =  paramIntent.getIntExtra("contact");
        values.put("contact", String.valueOf(contact));
        handleSendMessage(contentValues);
      }
    }
  }
  void handleSendMessage(ContentValues values){
    SQLiteDatabase db =  dbHelper.getWritableDatabase();
    db.insert("SendMessage", null, values);
  }
}
class PrivateDBHelper extends SQLiteOpenHelper{
  public void onCreate(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE SendMessage (message TEXT, ... )");
  }
  ...
}
```

Listing 5.2: Private database example code in app `A`.

```xml
<permission android:name="com.example.app.Write" android:protectionLevel="
    dangerous" />
<permission android:protectionLevel="normal" android:name="com.example.app.
    permission"/>
...
<provider android:name=".PublicDatabase" android:authorities="com.example.app.
    PublicDatabase" android:exported="true">
  <path-permission android:pathPattern="/contacts/" android:writePermission="com.
      example.app.Write" />
</provider>
<receiver android:name="com.example.app.ExampleBroadcastReceiver" android:
    permission="com.example.app.permission">
  <intent-filter>
    <action android:name="com.example.app.event.Trigger" />
  </intent-filter>
</receiver>
```

Listing 5.3: The content provider and broadcast receiver specification of app `A` (victim app) specified in its manifest file.

```
1  class DatabaseMalware extends Activity{
2    void insertPublic(){
3      Uri targetUri targetUri = Uri.parse("content://com.example.app.PublicDatabase
         /1");
4      ContentValues values = new ContentValues();
5      values.put("message","Sensitive Data");
6      ContentResolver contentResolver = getContentResolver();
7      Uri confirmationUri =  contentResolver.insert(targetUri, values);
8      if(confirmationUri != null){
9        //attack has been launched successfully.
10     }
11   }
12   void insertPrivate(int phoneNo, String message){
13     Intent intent = new Intent("com.example.app.event.Trigger");
14     intent.putExtra("task", "message");
15     intent.putExtra("data",message);
16     intent.putExtra("contact",phoneNo);
17     sendBroadcast(intent);
18   }
19 }
```

Listing 5.4: App `M` which is a malware accessing the public and private databases of the victim app `A`.

5.2.1 Vulnerable Public Database Example

Line 4 in Listing 5.3 shows the content provider tag of app `A`. The `android:exported` attribute in the manifest allows the content provider to be accessed by other apps on the device. The developer has tried to protect this provider using `<path-permission>` at Line 5. This means that any request which targets URIs with the `"/contacts/"` path intending to modify the data managed by this provider will be allowed if the requesting app has the `com.example.app.Write` permission. However, there are some bugs in the code in Listing 5.1. Other paths in this code allow the attacker to pollute the database with sensitive data which will be sent out later (e.g., via SMS).

In the lifecycle of the content providers (see Section 5.4.1), `onCreate()` is the first method called by the Android framework. The two URI patterns are registered in the `android.content.UriMatcher` object: Line 6 maps URIs whose paths only consist of digits to `1` and Line 7 maps URIs whose paths are `"/contacts/"` to `2`. Due to the `<path-permission>` in the manifest file (Line 5 in Listing 5.3), only the second URI pattern is protected by the `writePermission`. However, the first URI pattern is not protected by any permission so malware can use it to pollute the database by invoking the `insert()` API in the content provider.

Now, we explain the public database attack launched by app `M`, the malware in Listing 5.4. The `insert()` method of the content provider in app `A` at Line 9 in Listing 5.1 is called once app `M` calls `ContentResolver.insert()` at Line 7 in Listing 5.4. App `M` also passes appropriate data as arguments of this method to

send sensitive messages to the contact list on the device. The `targetUri` at Line 3 is crafted by the attacker in a way to pass the check at Line 11 in Listing 5.1 to reach the `SQLiteDatabase.insert()` statement at Line 15. In this example, the bug which leads to the public database attacks can be fixed either by enforcing proper permissions in the manifest file or changing the implementation of the content provider. Instead of protecting a particular subset of paths in the manifest file, the developer can protect the whole provider by the `writePermission` which is inflexible. Alternatively, she could remove Line 6 in Listing 5.1 and only allow the path patterns that are already protected in the manifest file to execute the program to reach the `insert()` method.

5.2.2 Vulnerable Private Database Example

Listing 5.2 shows an example broadcast receiver in app `A` which is vulnerable to private database attacks. The manifest file of this app in Listing 5.3 shows that the broadcast receiver component declares the permission at Line 2 with `"normal"` protection level. Therefore, any application can have this permission including app `M`. When app `M` creates and sends a malicious intent to the broadcast receiver, the `onReceive()` method gets invoked in Listing 5.2. This method processes the Intent message and if it contains a particular set of data, parts of its content will be stored in the private database at Line 19.

The private database attack explained above can be prevented by protecting the entry points which make the private internal databases reachable. For example, the broadcast receiver in this example could be protected by a permission with `"dangerous"` protection level. In this way, the malware would not be able to send arbitrary intents to this component because it does not have the required permission. However, as this protection mechanism is all-or-nothing, it may not be flexible enough. Alternatively, the developer can implement and incorporate validators in the code to prevent the malicious payloads reaching the database. Experience shows that writing a correct validator is error prone, e.g., the vulnerable code in Listing 5.2 tries to validate the incoming Intent message before inserting its data to the database, but it still has a vulnerability.

Our example database attacks show that even though the targets in the private and public attacks are similar, e.g., `SQLiteDatabase`, the attack channels are different and attackers have to bypass different security mechanisms. Public databases have more standardized security mechanisms. Private database apps seem to be written more arbitrarily and their security are mostly based on the app's implementation.

These examples also show that detecting and generating exploits for database attacks demands accurate analysis. A malware installed on the device can construct malicious inputs using data structures and objects which are more complex than primitive types. There are also several libraries with particular semantics on the execution path which have to be handled by the analysis to lower down the false positive rates. In addition, since the number of entry points for the private database attacks can be large and the private databases can be implemented anywhere in the code-base, the analysis needs to be scalable and efficient. Next,we discuss our attack model and explain how we detect and confirm database attacks in real-world Android apps.

5.3 Threat Model

The adversary in our attack model is a malware installed on the Android device. We do not make any assumptions about the permissions requested by the malware, i.e., malware does not need to request for any permission with `dangerous` protection level. We assume that at least one app on the device is benign but buggy, hence a database vulnerability exists. The malware can attack either public or private databases of unprotected apps as shown is Figure 5.1. It needs to craft malicious input (which can be string, URI objects, intent or other types of objects) and send it to the relevant component of the vulnerable app.

5.4 Detection and Exploitation of Database Vulnerabilities

We aim to generate inputs which can exploit these vulnerabilities. ContentScope [ZJ13] is an analysis framework which detects and exploits the public database vulnerabilities using reachability analysis and constraint solving. However, it does not deal with the private database vulnerabilities. Our experience shows that private database vulnerabilities are more scattered throughout the code-base of Android programs. Hence, compared to ContentScope, our analysis has to be equipped with the techniques which deal with the scalability challenges and handle framework libraries more precisely. We present DBDroidScanner and show how it extends and customizes different components of the analysis framework introduced in Chapter 3.

5.4.1 Source-Sink Pair Identification for Database Attacks

The first component of our analysis framework is to identify the source-sink pairs – this is customized for the class of vulnerabilities studied by the system namely, database vulnerabilities. In order to find the entry points which lead to the database sink methods, first, we study possible ways of accessing the public and private databases in the Android apps.

Direct Invocation of Content Provider APIs (Public Databases)

Content provider components encapsulate local content and export them through standardized APIs: `query()`, `insert()`, `openFile()`, etc. These are the interfaces which can be invoked by other apps to operate on the internal `SQLite` database and internal files. A malicious app can call these (standardized) APIs to launch pollution, leakage or file access attacks. Once the possible entry methods are determined, we mark the parameters of these methods as source variables.

A content provider is candidate for analysis if it is not fully protected by appropriate attributes and permissions in the manifest file. A content provider is fully protected: (1) by default (equivalent to `exported="false"`) in Android SDK 17 and higher; (2) if `exported="false"` is specified explicitly; (3) if the `android:permission` attribute is specified in the `<provider>` tag in the manifest file.[5]

A content provider is partially protected if developers use the `readPermission` and `writePermission` or `<path-permission>` for a more fine-grained protection. If the content provider is protected by these permissions, some of the APIs do not need to be analyzed for the public database attacks. We remove the permission-protected APIs from the source method list based on the category that they belong to and the permission chosen by the developer: (i) if the content provider is protected by the `readPermission`, clients installed on the device must request for it to be able to query the component. Therefore leakage APIs (e.g., `query()`) cannot be a source method; (ii) Similarly if a component is protected by `writePermission`, the pollution APIs (e.g., `insert()`) cannot be source methods; (iii) For the APIs which are used to obtain the file handlers (e.g., `openFile()`), `readPermission` and `writePermission` are checked based on the requesting access mode. If the access mode is `"w"`, `writePermission` is checked and if it is `"r"`, the `readPermission` is checked. If the content provider is protected by the `readPermission`, we analyze the app for the `"w"` mode and if it is protected by the `writePermission`, we analyze

[5] If a content provider is protected by a `permission` which has a protection level higher than `normal` (e.g., `dangerous`), it is not chosen as a candidate entry point for analysis since the user will be prompted to grant it.

it for the "r" mode. Since the attacker can always gain the file handlers in one of the "r" or "w" access modes, the file access APIs are always source methods.

A content provider which is protected by the `readPermission` and `writePermission` simultaneously will not be chosen as candidate for analysis since such providers are not reachable by the malware. The analysis does not generate an exploit whose reported path is protected by the `<path-permission>`. For instance, Line 5 in Listing 5.3 shows that the developer has protected the "/contacts/" path by `writePermission`. Therefore, our analysis does not generate exploits consisting of URIs which have "/contacts/" path to invoke the `insert()` method of the content provider as they will be false positives.

Intents and Data Access (Private Databases)

Android apps may have private databases typically in the form of `SQLite` databases which are not accessible through content providers. Private databases can be implemented in any component (class) of an app. A component (except for content provider) is a candidate entry point to be analyzed for private database attacks if it is not protected by permissions and it is exported (see Section 5.1). A malware installed on the device may indirectly access private databases by crafting malicious intents without calling any of the methods of the `ContentResolver` to invoke the `ContentProvider` APIs. Instead, it sends an intent that starts a component (e.g., activity), which is part of the provider's app. Hence, the destination component is in charge of retrieving and processing the data. These intents are obtained by APIs such as `getIntent()` in activities or `onStart()` in services, etc., which are our source methods.

We have classified the sink methods to the leakage, pollution and file access categories.[6] The sink methods used in our analysis for the privilege escalation attacks are the `ContentResolver` APIs (e.g., `ContentResolver.insert()`). These sinks are used to detect both public and private database attacks.

5.4.2 Constructing the Control Flow Graph and Reachability Analysis

The control flow graph (CFG) construction for analyzing database attacks is similar to the one in Section 3.4. Android apps do not contain a main method so the CFG is dependent on the lifecycles of the entrypoint components. The entry

[6] E.g., `SQLiteDatabase.query()`, `SQLiteDatabase.insert()` and `ParcelFileDescriptor.open()` for the leakage, pollution and file access categories respectively.

point component for the public database attacks is content provider. The content provider APIs are called by the underlying Android framework as callback methods in a specific order as follows. First, the `ContentProvider` is instantiated and the `onCreate()` method is called. Next, one of the entry methods of the content provider which is overridden by the app is invoked and this process is repeated for the rest of the overridden entry methods. Notice that the `onCreate()` method in content providers which is supposed to be called before any other API is anlayzed before other callbacks. The lifecycle for the entry point components which trigger the private database attacks are the same as the models explained in [ARF+14]. A sink reachability analysis is performed on the CFG to identify whether a sink method is reachable from a program point and what their distance is. This information is later used by the search heuristic of the symbolic execution phase to optimize the path traversal for reaching sink methods, thereby reducing path explosion.

5.4.3 Symbolic Execution for Detecting Database Attacks

As we have discussed in Chapter 3, symbolic execution would have to deal with low-level data structures, e.g., collection classes which are essentially a barrier for most of the existing analyses. Also, some parts of the libraries might not be supported by the analysis, e.g., native code.

The analysis introduced in Section 3 takes a hybrid approach of static symbolic execution and dynamic testing to interact with the Android framework. While the hybrid approach helps for scalability, modeling certain libraries is crucial for certain classes of vulnerabilities. The examples in Listing 5.1 and 5.2 show that the execution paths that lead to the database attacks might include Android and Java library methods which have to be handled more accurately by the symbolic execution and constructing exploits to run these paths is not trivial.

Among these libraries, those which construct a URI [BLFIM98] object and use its semantics to build filters are particularly important for building public and private database exploits. For instance, the entry methods of content providers which have to be invoked by the malware in the public database attacks require a URI parameter to identify the data in a database. URI objects are different from strings and have more complex semantics. The analysis framework in Chapter 3 supports load and store operations for the fields of URI objects but not more complex operations. This is not sufficient for generating the database exploits which rely on these operations, hence, we model semantics of complex URI methods.

Note that ContentScope [ZJ13] which is designed to detect the public database

vulnerabilities also has to handle such libraries to be able to generate working exploits. However, the authors do not discuss how they model them. Our approach for handling URI-based libraries combines the classical symbolic execution which is dependent on SMT solvers with automata-based theories. We use the following approach to construct symbolic models for URI-based libraries: (i) we use SMT formulas if a method of a library can be directly translated to an SMT formula and the formula is tractable enough. Examples of such methods can be found in Table 5.1; (ii) sometimes, directly translating the semantics of library methods to SMT formulas is complex and the resulting formula is large (possibly unbounded). If the method maps an input string to an output string, we model them as Symbolic Finite Transducers (SFT) [HLM+11] to simulate the I/O relationship. In what follows, we study the structure of URIs and present our models using the approaches discussed above.

Background: Structure of Generic URIs and URI-Based Libraries

URIs are widely used in Android to identify resources. For instance, content providers use different components of URIs to reference data in their tables. However, a URI might contain components that trigger vulnerabilities in the recipient code which interprets the URI. The syntax for a "generic URI" which conforms to RFC 2396 [BLFIM98] is as follows:

```
⟨scheme⟩://⟨authority⟩⟨path⟩?⟨query⟩
```

where only the existence of *scheme* part is mandatory. The scheme component identifies a resource and defines the semantics for the remainder of the URI string. For example, the scheme component of the URIs used by the content providers has to be `"content"`. The next element, *authority*, is a hierarchical element specifying where the URI is governed by. An authority can consist of *userinfo*, *host* and *port* where *userinfo* and *port* might not be present. The *path* component contains data specific to the authority and *query* is a string consisting of key-value pairs. A URI reference may have additional information attached in the form of a fragment identifier.

The Android and Java libraries that implement the URIs are `android.net.Uri` and `java.net.URI` respectively which conform to RFC 2396 [BLFIM98]. Android apps use the first class for implementing URIs more often. Even though we explain our models mainly based on the `android.net.Uri`, most of the methods of the `java.net.URI` class are handled in a similar way. URI instances and their op-

erations are not directly supported by SMT solvers. There are a number of classes which are frequently used in Android programs and provide methods to manipulate URIs.[7] A content URI[8] is simply a URI that identifies data in a provider. Every data access method of `ContentProvider` class has a content URI parameter which allows developers to determine the table, row, or file to access. Content URIs include the name of the entire provider which has to match the *authority* attribute of the content provider in the manifest file. The `android.net.ContentUris` class declares methods for working with the *id* part of a content URI. This class has utility methods useful for working with the `android.net.Uri` objects that use the `"content"` scheme. The `java.net.UriMatcher` class helps in choosing which action to take for an incoming content URI. This class has methods which map content URI patterns to integer values. For example, the developer can use the integer values in a switch statement that chooses the desired action for the content URI or URIs that match a particular pattern (Line 11 in Listing 5.1).

Symbolic Representation for URIs

Our analysis keeps a separate pool of URIs to trace the states of the URI instances. We call the model that we have created to represent URIs, *summarized URI.* A summarized URI object can be altered by the methods of the classes listed above. Basically, our symbolic model for the URI instances follows the original URI class semantics and stores symbolic values for the URI fields. The states of fields of summarized URIs change during symbolic execution. A summarized URI also contains summarized methods which are modeled in one of the following ways: (i) directly translated to SMT formulas; and (ii) SFTs. Summarized URIs are further used as the building blocks of other related classes. For instance, the `java.net.UriMatcher` class stores the summarized URIs which are added using the `UriMatcher.addURI(Uri)` method (Line 6 and 7 in Listing 5.1). A content URI is also modeled and used as a summarized URI.

Direct Translation of Methods to SMT Formulas

Table 5.1 shows some of the methods which are directly translated to SMT formulas. We present example symbolic representation of methods of libraries which are dependent on URIs. `Uri.getLastPathSegment()` which returns the last path segment of a URI can be modeled with the following self-explanatory constraint:

---

[7]E.g., `android.net.Uri.Builder`, `android.content.UriMatcher`, `android.content.ContentUris`, etc.

[8]Content URI syntax: `content://`*authority*/*path-prefix*/*id*

```
Input: String path
Output: String LPS
Local variable: String x
¬LPS.contains("/") ∧ (path = x."/".LPS ∨ path = LPS)
```

where `uri` is an `android.net.Uri` instance object, `path` is the path field of the `uri`, `x` is a string variable, `LPS` is the output variable for the result of the method which represents the last path segment of the `uri`, `"."` is the concatenation operation and `contains("/")` means that the base variable contains the "/" character. We actually work with the constraint in SMT format to be solved by the SMT solver:

```
(= (str.++ x LPS) path)∧
(= (str.indexof y "/" 0) -1)∧
(or (= (str.++ "/" y) LPS) (and (= (str.len x) 0) (= y LPS)))
```

The `getPathSegments()` method is another method which is modeled using the semantics of the bounded version of `String.split()` method (i.e., `splitN`) where N is the number of string tokens found in the base string variable. N is two for the particular example shown in Table 5.1.[9] Some of the methods are translated using models for other methods. For instance, `Uri.Builder.appendId(String id)` is translated using the `Uri.Builder appendPath(String x)` method. In what follows, we present the symbolic models for library methods which are more complex and therefore modeled using SFTs. We illustrate symbolic execution for these library methods through concrete examples.

Matching URIs Using SFT

The `UriMatcher.match(Uri)` method is often used in Android programs to compare content URIs. It performs a mapping operation, i.e., accepts an object as input and maps it to a unique output value. Modeling this method as an SMT formula is problematic because the formula might be unbounded. We represent our model using Symbolic Finite Transducers (SFT) [HLM+11]. Finite transducers are an extension of finite automata used to model operations on lists of elements. A SFT extends a finite transducer by allowing the transition labels to be predicates. In what follows, we mainly explain how SFTs are used using examples.

Figure 5.2 shows the SFT designed for the `UriMatcher.match(Uri)` method. In this diagram, $\mathtt{U}_i$ belongs to the set of URIs added to an `android.content.UriMatcher` object via `UriMatcher.addURI(Uri)`. Each transition has a constraint (c_i) which

[9]Our implementation supports formulas for higher values of N.

Table 5.1: This table shows SMT formulas for modeling methods of URI-based classes.

Class	Method	Constraint
`android.net.Uri.Builder`	`Uri.Builder.` `appendPath(String x)`	path′ = (str.++ path "/" x)
`android.net.Uri`	`String` `getLastPathSegment()`	(= (str.++ x LPS) path)∧ (= (str.indexof y "/" 0) -1)∧ (or (= (str.++ "/" y) LPS) (and (= (str.len x) 0) (= y LPS)))
`android.net.Uri`	`List` `getPathSegments()`	(or (= path (str.++ "/" x_1 "/" x_2)) (= path (str.++ "/" x_1 "/" x_2 "/" x_3))) ∧ (not (str.contains x_1 "/")) ∧ (not (str.contains x_2 "/"))
`android.content.ContentUris`	`Uri.Builder` `appendId(String id)`	`appendPath(id)`
`android.content.ContentUris`	`long` `parseId(Uri uri)`	last := `uri.getLastPathSegment()` ∧ (ID = (str.to.int last))

`android.content.ContentUris`	`Uri`
	`withAppendedId(Uri uri,`
	`long id)`
		`appendPath`((int.to.str id))

outputs `d`, the default value registered in the `UriMatcher` (e.g., `0` registered at Line 2 in Listing 5.1), if it is not satisfied. φ is the path constraint of the current execution path. If the URI passed as argument to `UriMatcher.match(Uri)` matches any of the $\mathtt{U}_i$, the transducer outputs the integer code stored for $\mathtt{U}_i$ (code(U_i)). The symbol `e`, used for the transition between q_2 and q_3 denotes the end of inputs. Content URI patterns can be matched using wildcard characters. Our symbolic model understands the `"*"` and `"#"` used as the path segment by the `UriMatcher` class where `"#"` is `([(0-9)+])` and `"*"` is `(.*)` as regular expressions.

In order to symbolically execute `uriMatcher.match(uri)` at Line 11 in Listing 5.1, our analysis employs the SFT in Figure 5.2 for the two URIs registered in `uriMatcher` at Lines 6 and 7. First the SFT examines whether the argument `uri` matches the URI at Line 6 which restricts the path segment of the `uri` to match `([0-9]+)` and returns `1` (the code registered for the first URI). Next, the analysis examines the second URI at Line 7 which restricts the path to be equal to `"/contacts/"`. However, due to the `<path-permission>` at Line 5 in Listing 5.3, this URI is protected and should not be reported as exploitable (unsatisfiable due to the path constraint). Hence the path at Line 18 in Listing 5.1 is infeasible.

Comparing URIs Using SFT

Another example method which is modeled using SFT is `Uri.compareTo()`. This method constructs the string representation of the base and argument `Uri` objects and compares them: it returns `0` if the base and argument `Uri` objects are equal; and less or greater than `0` if the base URI string is lexicographically less or greater than the argument URI string respectively. Figure 5.3 depicts our model

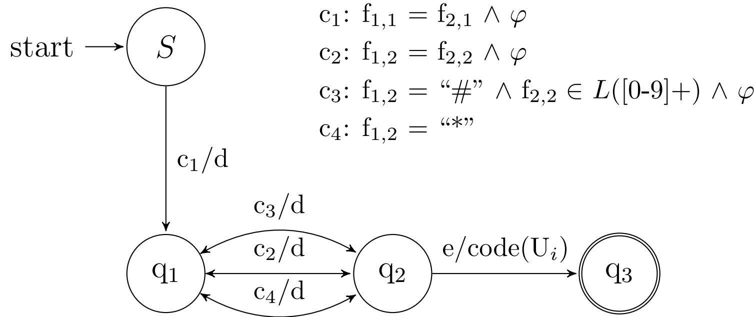


Figure 5.2: SFT for `UriMatcher.match(Uri)`: ϕ is the path constraint; the fields of the registered and argument URI are denoted by $f_{1,i}$ and $f_{2,i}$ respectively; c_1 checks the constraints for the authority and c_2, c_3 and c_4 check the constraints for the path segments of the URIs and d is the default integer registered in `UriMatcher` object.

for `Uri.compareTo(Uri)` as an over-approximation of this method: it returns `0` if the string representations of the two URIs are equal (i.e., transitions reach the accepting state q_7) and `1` otherwise. If the fields of the base `Uri` and the argument `Uri` are represented by $f_{1,i}$ and $f_{2,i}$ respectively and φ denotes the path constraint computed so far, c_i which is the constraint label of a transition is: $f_{1,i} = f_{2,i} \wedge \varphi$. For instance, the transition from the start state, S, to q_1 symbolically represents all possible scheme fields in the base URI ($f_{1,1}$) which match the scheme field of the argument URI ($f_{2,1}$) and also satisfy the path constraint.

We now explain how our symbolic model for `Uri.compareTo()` works using an example. If we add the two methods in Listing 5.5 to the `PublicDatabase` class in Listing 5.1 which is vulnerable to the public database pollution attacks, this class will also become vulnerable to the public database leakage attacks. In order to generate working exploits, our analysis needs to compare the two `Uri` instances, `u1` and `u2` at Line 4 in Listing 5.5. Our symbolic executor first creates a summarized URI instance for `u2`, the first argument of the `query` method which is a `Uri` object. Next, it analyzes `supportedUri()` at Line 2. This method builds and returns a new `Uri` object, hence the symbolic executor creates another instance of a summarized URI and initializes its fields with the corresponding values (e.g., `scheme` field will be `"content"`). Line 3 enforces constraints on the fields of `u2` as well and adds them to the path constraint. At this point, the path constraint for this part of the program[10] is:

[10] We do not present the path constraint for the previously analyzed parts of the program for simplicity.

```
public Cursor query(Uri u2, String[] projection, String selection, String[]
    selectionArgs, String sortOrder) {
  Uri u1 = supportedUri();
  if(u2.getScheme().contains("content") && (u2.getPathSegments().get(0) != null))
      {
    if(u1.compareTo(u2) == 0){
      SQLiteDatabase db = dbHelper.getReadableDatabase();
      db.query(u2.getPathSegments().get(0), projection, selection, selectionArgs,
           null, null, sortOrder, null);
    }
  }
  return null;
}
Uri supportedUri() {
  Builder builder = new Uri.Builder();
  builder.scheme("content").authority("com.example.app.PublicDatabase").
      appendPath("profile").appendPath("1").appendQueryParameter("id", "1").
      appendQueryParameter("type", "admin");
  Uri u = builder.build();
  return u;
}
```

Listing 5.5: Example Android program which calls the `Uri.compareTo(Uri)` method.

$$\phi = f_{2,1}.\texttt{contains("content")} \wedge f_{2,2} = \texttt{"com.example.app.PublicDatabase"} \wedge f_{2,3} = \texttt{"/"}.x_1.\texttt{"/"}.x_2 \wedge \neg x_1.\texttt{contains "/"} \wedge \neg x_2.\texttt{contains "/"} \wedge x_1 \neq \texttt{null}$$

where x_i is the (i-1)th path segment determined by the `Uri.getPathSegments()` at Line 3 and the fields of `u2` are denoted by $f_{2,i}$: $f_{2,1}$ is the scheme, $f_{2,2}$ is the authority and $f_{2,3}$ is the path field of `u2`. The authority `"com.example.app.PublicDatabase"` is obtained from the manifest file.

At each transition of the SFT, new constraints are concatenated to the path constraint as shown below and if they are satisfiable, the transducer moves to the next state:

$$c_1: f_{1,1} = f_{2,1} \wedge f_{1,1} = \texttt{"content"}$$
$$c_4: f_{1,2} = f_{2,2} \wedge f_{1,2} = \texttt{"com.example.app.PublicDatabase"}$$
$$c_6: f_{1,3} = f_{2,3} \wedge f_{1,3} = \texttt{"/profile/1"}$$
$$c_7: f_{1,4} = f_{2,4} \wedge f_{1,4} = \texttt{"id=1"}$$
$$c_7: f_{1,5} = f_{2,5} \wedge f_{1,5} = \texttt{"type=admin"}$$

where the fields of `u1` and `u2` are denoted by $f_{1,i}$ and $f_{2,i}$ respectively. The indices in this example are from 1 to 5 and refer to the scheme, authority, path, first query parameter and second query parameter respectively. If the analysis doesn't find any constraint for a transition, it moves to the next state. Since the constraints for all transitions in this example are satisfied, the SFT returns zero which means that `u1` and `u2` are equal. For example, c_1 enforces the $f_{1,1}$ (scheme) field of `u1`

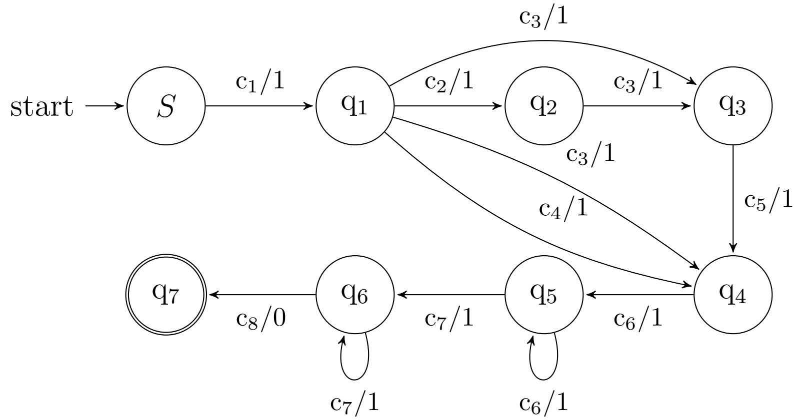


Figure 5.3: SFT for `Uri.compareTo(Uri)`. The label for each transition is a constraint (c_i) for a particular field of the base and argument URIs: c_1 for scheme, c_2 for userinfo, c_3 for host, c_4 for authority, c_5 for port, c_6 for path, c_7 for query pairs, c_8 for fragment.

to be equal to both `"content"` and $f_{2,1}$ and the path constraint also enforces the $f_{2,1}$ field of `u2` to contain `"content"`. The concatenation of the path constraint and c_1 is satisfiable and restricts the $f_{2,1}$ field of `u2` to be equal to `"content"`. Similarly, the constraints for the rest of transitions are satisfiable and as a result, the sink method at Line 6 can be reached by the malware on the device. The transducer for `Uri.compareTo()` deals with the multiple query parameters in the URI instance (Line 13) using c_7. In this case, the SFT iterates through the query parameters stored in the summarized URI and moves to the accepting state if all the constraints are satisfiable. Note that the query parameter pairs in URIs are implemented using Java container classes. In order to obtain the constraint for a pair, we keep track of individual loaded and stored elements during symbolic execution.

Integration of SFT to DBDroidScanner

One reason for choosing SFT to model the URI-based methods is their compatibility with SMT solvers. This allows us to construct symbolic models whose input are path constraints in the form of SMT formulas and reuse them all over the code-base. The labels of transitions in our implementation for SFT are SMT formulas. At each transition, a new constraint is checked whether it satisfies the existing path constraint. If the constraint is satisfiable and its variables have data dependency on the inputs, it is appended to the path constraint. These constraints can specially help us in generating more precise inputs at the end of the symbolic execution phase. One important characteristic of transducers which makes them

useful for modeling URI-based methods is that they can deal with unbounded inputs. This allows us to support URI fields which have recursive structure (e.g., query parameters). The number of iterations for transitioning between states might depend on the loops in the program. In this case, our framework employs a bounded symbolic execution, thereby transitioning for a limited number of times in the transducer. Our implementation for the SFTs is single-valued. Informally, this means that the value returned for a given transition is always a single value. We allow ϵ-transitions in our models by setting the predicate to `"true"` and mapping it to the appropriate value dependent on the states between which it transitions. In this chapter we have illustrated SFT models for two example URI methods. Other URI methods (e.g., `Uri.encode(String)`) can also be modeled using SFTs.

Parsing the URIs

Sometimes the analysis needs to construct a URI object for a given URI string (e.g., `Uri.parse(String)` returns a `Uri` object for the `String` argument). URI strings can be parsed using the POSIX regular expression in RFC 2396 to retrieve the scheme, authority, path, query components and fragment parts. In order to model the `Uri.parse(String)` method, first we use the SMT solver to compute a value for the `String` argument which satisfies the path formulas collected so far. Next we use the regular expression to retrieve the fields and construct a URI. If concrete values cannot be resolved for the fields of the URI, symbolic values are generated for them.

5.4.4 Database Attack Validation

In Section 5.1, we explained how database attacks can be classified into private and public categories. DBDroidScanner extends the validation component in the analysis framework to analyze Android apps for these categories of database vulnerabilities. Once the symbolic executor deploys static symbolic execution, it uses the CVC4 SMT solver [LRT+14] to solve the path constraints and generate values for the symbolic input variables identified by the source-sink identification phase.

However, generating such values is not adequate for exploiting the vulnerabilities and constructing working exploits using them is not straightforward as shown next. These generated values are processed by the validation component to generate concrete exploits that trigger the private and public database vulnerabilities. We have utilized and designed patterns for generating such exploits based on the source and sink methods of the reported vulnerable path.

Public Database Attacks

Public databases are accessible through content providers. Content providers can be reached from other apps (malwares) on the device by directly invoking standardized APIs (e.g., `insert()`). For this purpose, the malware can obtain the content model by calling `getContentResolver()` which allows calling APIs of content providers available to the system. The parameters of these APIs are the symbolic inputs for which the symbolic executor generates values. The validation component uses these generated inputs as well as the manifest file to derive concrete parameters and launch requests to a particular content provider. We explain the content provider exploit generation through an example:

One of the APIs of a content provider is `query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)` which returns a `Cursor` over the result set for the given URI. The `uri` parameter identifies a particular table of a content provider and `projection` is the list of columns to be queried. The `selection` parameter should be formatted as SQL WHERE clause to enforce constraints on the query and if it contains `?s`, they will be replaced by the `selectionArgs` parameter. Finally, the `sortOrder` determines how to order the rows in the result set.

The symbolic executor in the analysis framework generates values for each of the method parameters. The `uri` parameter starts with the `content` scheme which is fixed in content URIs. The authority segment is obtained by the manifest file and the symbolic executor checks whether its value is consistent with the authority value resolved from the analysis of the program. The reason is that sometimes developers make mistakes and apply conditions on the execution path which prevent the authority registered in the manifest file to be accepted by the content provider. In this case, the content provider cannot handle any request from other apps. Our analysis catches such mistakes to avoid reporting false positives.

The remaining segments of the `uri` identify the tables of a database which are generated by the symbolic executor. As you can see, some of the parameters of the `query` API have array types. We use the array models to resolve the elements. In practice, reasoning about arrays is not trivial and we only focus on the arrays which are the source method parameters. It is also possible to set all of the parameters of the `query` API except the `uri` to null. If `projection` is null, all of the columns and if the `selection` is null, all of the rows for the given URI will be returned. If `sortOrder` is null, results will return with the default sort order.

In order to perform the dynamic testing, we have created a malware skeleton app for invoking the APIs of vulnerable content providers with malicious argu-

ments which are resolved from the static analysis. Our malware does not have any permission granted from the user. Once the vulnerable content provider is invoked, the validator component logs the execution trace to obtain the concrete values. Using these concrete values, our analysis attempts to place them in the path constraints generated by the static symbolic execution to create more precise inputs if possible. If the generated inputs reach a fixpoint, our malware invokes the vulnerable content provider and validates the vulnerability using the following sample rules:

We assume the `openFile()` and `openAssetFile()` APIs of a content provider are exploited if they return non-null `ParcelFileDescriptor` and `AssetFileDescriptor` references respectively. Similarly, the `query()` API of a content provider should return a non-null `Cursor` reference; the `insert()` API of a content provider should return a non-null `Uri` reference; and the `update()` API of a content provider should return a non-zero integer.

Private Database Attacks

The difference between private and public databases is that public databases are accessed through content providers, while private databases are accessed via Intent messages received by any of the following components: activities, services or broadcast receivers. An input string obtained from an intent which triggers paths down to the `SQLiteDatabase` methods may allow attackers to manipulate the database or compromise the security of the app.

In order to trigger and validate the private database attacks, the attacker should generate intents which target the vulnerable component of the victim app. For this purpose, the values generated by the symbolic executor are embedded in an Intent message in the validation phase to construct an Intent exploit. A malware can send explicit malicious intents to a particular component of an app by explicitly setting the target class name using the `Intent.setClass()` API. Alternatively, it can construct an intent which conforms to the intent filter of the target component as shown at Line 13 in Listing 5.4.

The validation component collects information about the entry component by parsing the manifest file and combining the results from the static symbolic execution. It creates all possible data parameters that will match the intent filter. Path is one of the data elements in intent filters that will be checked for accepting an intent. The developer can specify a special form of regular expressions as the path pattern.[11] Some of these values might also be obtained from the symbolic

[11] We follow the same algorithm implemented in the Android framework to match against the paths of intents.

execution phase in which case we directly use the values generated by our symbolic executor.

Intent messages transmit data in the following ways: (i) a data URI which references the data resources consists of the scheme, host and path as well as query parameters which are the key-value mappings preceded by the "`?`"; (ii) Intent extras, the key-value pairs whose type can also be specified in the intent (e.g, int, string, etc); (iii) other Intent parameters such as categories, actions, etc., that can be sent as string values. An intent can be constructed as a Java object from a malware app as discussed in this chapter. It can also be represented as an intent hyperlink and invoked from web as explained in Section 4.3.3.

One difference between Intent objects and intent hyperlinks is that Intent objects can contain arrays and parcelable `extra` parameters while intent hyperlinks can only contain primitive type `extra` parameters. Hence, it is also possible for the attackers to send malicious data through parcelable key-value pairs and the victim receives the malicious inputs by invoking the `Intent.getParcelableExtra()`. We partially support this API for the parcelable types which have been modeled by our system (e.g., Intent). For this purpose, analysis should first resolve the type of the parcelable `extra` parameters received in the target app. For example, if a cast operation is applied to the parcelable parameter, the cast type will be used as the resolved type for the parameter. If the resolved type is supported by our analyzer, an object will be instantiated in the malware program and set as an `extra` parameter to the Intent object.

Once the analysis framework generates the key-value pairs as explained in Section 4.3.3 and other necessary inputs for the source-sink flows and the intent filter specifications for the target component, all of these elements are put together to generate an Intent message.

In order to perform dynamic testing, we configure our malware to send out an Intent message with malicious parameters. Once the target component gets invoked and receives the intent, the validator logs the execution trace to obtain the concrete values. Similar to the public vulnerability validation, our analysis attempts to place these concrete values in the path constraints generated by the static symbolic execution to create more precise inputs if possible. If the sink method is reached on the execution path and the malicious intent parameters are observed at the sink invocation site, we assume that the vulnerability is exploitable.

5.5 Experimental Results for Database Vulnerability Detection

We now analyze real-world Android apps to detect and exploit database attack vulnerabilities. Our main goal is not only to detect potential vulnerabilities but also to confirm the public and private database vulnerabilities with successful zero-day exploits. We analyze 924 apps in total which are the top 100 apps of all categories in Google Play. Of these apps, 133 apps have at least one exposed and unprotected content provider and all 924 apps have at least one exposed and unprotected component other than content providers.

We ran DBDroidScanner in Ubuntu 12.04 on an Intel Core i5-4570 (3.20GHz) with 16GB of RAM. Our analyzer is a prototype and not designed to be efficient or optimized. To get an idea of the analysis time, we have randomly chosen 50 apps, the static analysis (dataflow, symbolic execution and exploit generation) of DBDroidScanner takes on average 43.5 and 140.1 seconds for detecting public and private database vulnerabilities respectively. We can see that analysis of private database vulnerabilities is more complex because the number of paths that need to be traversed by symbolic execution can be high. Moreover, the execution paths triggered by intents which lead to vulnerable sink methods can be long and often contain many conditional statements which have to be solved by the SMT solver. To validate the database exploits, we configure our custom malware app to launch the components and to perform the privileged operations (e.g., inserting data into the app's database). Listing 5.4 shows the code-snippet of example malware used to send the requests to the vulnerable public and private database. The runtime execution of a single dynamic test varies depending on the app from seconds to 1-2 minutes. Although DBDroidScanner is a prototype, we see that the times are already reasonable.

5.5.1 Database Vulnerability Detection Results

We ran our analyzer on 924 apps where 133 apps have unprotected content providers in the manifest. Hence, we analyze 133 apps for public database attacks and all 924 apps for potential private database attacks which can be exploited via inter-app communication. As shown in Table 5.2, we detect and confirm 52 public and 23 private vulnerable apps and 153 vulnerabilities in total. We also classified our results based on the content leakage, pollution and file access categories. Our results show that modeling the URI-based libraries are necessary to generate accurate exploits for both public and private database attacks. Even

Table 5.2: Overall statistics of apps vulnerable to the database attacks.

Category	Sub-Category	# of Vulnerable Apps
Public Databases	Pollution	19
	Leakage	27
	File Access	26
Private Databases	Pollution	12
	Leakage	14
	File Access	5

though the mechanisms through which the private database attacks are launched (intents) are different from the public database attacks (content provider APIs), sometimes similar constraints are used by the developers to validate the incoming input. Next, we discuss two example apps vulnerable to the public and private database attacks to explain why a good model of such libraries is needed for exploit generation.

`chomp SMS` (version 6.07) is an SMS app vulnerable to public attacks. It requires accurate modeling of the `android.content.UriMatcher` and `android.content.ContentUris` libraries. A vulnerable content provider, `provider.ChompProvider`, accepts requests to update the scheduled messages if the URI parameter of the `update` API passes certain constraints. The goal is to generate specific values for each parameter of the `update` API (e.g., the URI parameter) to use in a working exploit. Our model for the `UriMatcher.match(Uri)` method tries to find a registered URI matching the given URI parameter: (1) our model checks if the URI parameter's authority is `com.p1.chompsms.provider.ChompProvider`; (2) it checks the path segment of the URI. If it is `"scheduled_messages"`, all the scheduled messages can be updated with the playloads crafted by the malware. In this case, DBDroidScanner generates the corresponding constraints using our model. Solving the constraints gives the attack URI parameter: `"content://com.p1.chompsms.provider.ChompProvider/scheduled_messages"`. Otherwise, if the path segment matches `"scheduled_messages/#"`, the `ContentUris.parseId(Uri)` method is invoked for the URI parameter to retrieve the last path segment and use it as the selection argument for the `SQLiteDatabase.update` sink method. In summary, the constraints generated using our models constrain the URI parameter to contain the `scheduled_messages` path segment and its last segment to be a number. Solving these constraints, DBDroidScanner generates a malicious URI, `"content://com.p1.chompsms.provider.ChompProvider/scheduled_messages/1"`, which triggers a different execution path.

We now discuss how we generate an accurate exploit for the `com.netease.cloudmusic`

(version 1.7.3) app which is vulnerable to private database attacks. The vulnerable component which the malware sends an intent to exploit is `LoadingActivity`. The victim app obtains the URI object set in the intent by invoking the `Intent.getData()` method and performs the following validations: (1) the scheme part of the URI object is checked whether it is `"content"`. Our analysis also generates the corresponding constraints using our model for the `android.net.Uri` library; (2) the `ContentUris.parseId(Uri)` method is called with the URI object passed as the argument to obtain its last path segment and it checks if the last path segment is a number. Our model for `android.content.ContentUris` generates the corresponding constraints and our tool generates `"content:///1"`[12] as the malicious URI part of the intent message. The intent is further configured by setting its `action` and `type` attributes to target the victim app and launch a private database attack.

5.5.2 Case Studies

We now discuss some findings for the representative vulnerable apps from our dataset. The public and private vulnerabilities are further categorized into pollution, leakage and file access attacks.

`Chomp SMS` (version 6.07) is an SMS app which provides several features such as sending scheduled SMS. This app is vulnerable to the pollution and leakage public attacks. Th exported content provider, `provider.ChompProvider`, allows the malware to steal or manipulate sensitive data such as scheduled and MMS messages. `CallApp - Caller ID & Block` (version 1.56) is a phone-call number identification app which is vulnerable to the public leakage attacks. The unprotected `.provider.CustomSuggestionsProvider` component allows attackers to query if a particular contact name is present in the contact list.

`Kii Keyboard` (version 1.2.22r6) is an alternative Android keyboard app vulnerable to public pollution and leakage attacks. This application has a content provider `.enhanced.BlacklistProvider` which is exported but not protected by any permission. This provider contains the blacklist of words which should not be shown as predictions. The attacker can manipulate or steal the content of the blacklist. `AppLoc` (version 1.99.2) is another app which allows users to lock apps with passwords. This app is vulnerable to the public leakage attacks due to an exported content provider which leaks information about the locks such as the process names and alarm times. The attacker can also apply SQL injection to retrieve information from the tables such as locations by setting the `projection`

[12] The authority part of the URI can be empty.

argument to `"* from locations;"`.

`ES File Explorer File Manager` (version 3.1.2), a popular file manager app, is vulnerable to public file access attacks. The exported content provider, `app.FileContentProvider` allows the attacker to obtain the file descriptors for arbitrary files in private internal or external storage which can lead to leakage of sensitive information.

`Lelong.my - Shop and Save` (version 1.2.5.3) is a mobile shopping app, vulnerable to the private attacks with two exposed activities. The `shoppingcart.PaymentOptionsActivity` activity allows the attacker to pop up order requests where the price and order ID can be manipulated. Another activity `product.ProductNewActivity` allows the attacker to perform `rawQuery()` and `insert()` to the database to search items. `Money Manager Expense & Budget` (version 2.4.6) is a money manager app vulnerable to the private leakage attacks. It exposes an activity allowing attackers to query and pop up bills for a specific account id. `Adidas World Football Live WP` (version 3.1) is a sport news app which is vulnerable to private pollution database attacks. It has an exposed broadcast receiver which can be exploited to insert scheduled advertisements into its private database.

5.5.3 Comparing DBDroidScanner and ContentScope

We also compare our system with ContentScope [ZJ13], the closest related work. As their data set is not available, we have tried to collect and analyze representative vulnerable apps mentioned in the paper. Some of these apps are removed or updated – the original versions are no longer available. Table 5.3 shows our results for the representative apps analyzed by ContentScope which are still available. Column three shows the closest APK version we could find. It is marked with "✓" if has been updated since the publication of [ZJ13]. Column four and five show the minimum and target SDK for which an app is compiled respectively. The last two columns show our findings for the public and private database attacks.

ContentScope only finds public database attacks. However, we also report private database vulnerabilities in these apps and show that 5 of these apps are vulnerable to both public and private database attacks. There are also updated versions of apps in which we do not find public database vulnerabilities but are vulnerable to the private database attacks. DBDroidScanner confirms the public database vulnerabilities in the apps which have the same version as those analyzed by ContentScope. In some cases (`Pansi SMS`, `mOffice - Outlook sync`), even though the vulnerabilities reported by ContentScope are not applicable any more in the updated versions of the apps, we find new public database vulnerabil-

ities. We also detect and confirm 8 apps vulnerable to the private database vulnerabilities. Surprisingly, there are some cases (`mOffice - Outlook sync`, `Dolphin Browser HD`, `Shady SMS 3.0 PAYG`) where the private database vulnerability allows the attacker to access the protected content providers and launch privilege escalation attacks. In what follows, first we compare our results with the findings of [ZJ13] for the public attacks. Next we present our private database attacks for these apps. We present results for all the available representative apps for both public and private attacks.

Public Database Attacks

DBDroidScanner did not find public database vulnerabilities in the updated versions of: `Shady SMS 3.0 PAYG`, `Nimbuzz Messenger`, `MiTalk Messenger`, `Youdao Dictionary`, `Netease Weibo`, `Dolphin Browser HD`, `Mobile Security Personal Ed.`, `Sina Weibo`, `Youni SMS` and `Tencent WBlog`. Our results are consistent with the results reported by ContentScope on the following apps (these apps have not been updated): `360 Kouxin`, `GO FBWidget`, `Boat Browser Mini`, `Droid Call Filter`, `GO TwiWidget`. Next, we present the public database attack case-studies for the remaining vulnerable apps in Table 5.3 by the ID column.

Table 5.3: Public and private database vulnerabilities in representative apps of [ZJ13].

ID	App Name	Version	SDK_M	SDK_T	Public	Private
1	Pansi SMS	3.6.0 ✓	7	13	✓	✓
2	Youni SMS	4.6.7 ✓	8	11		
3	mOffice - Outlook sync	3.7.7 ✓	5	-	✓	✓
4	Shady SMS 3.0 PAYG	3.38 ✓	8	18		✓
5	360 Kouxin	1.5.0	5	-	✓	-
6	GO SMS Pro	7.0.3 ✓	14	22	✓	-
7	Messenger WithYou	2.0.90 ✓	4	-	✓	✓
8	Nimbuzz Messenger	4.1.0 ✓	15	21		-
9	MiTalk Messenger	7.3.32 ✓	14	19		-
10	Youdao Dictionary	6.5.1 ✓	11	19		
11	GO FBWidget	2.2	5	-	✓	✓
12	Netease Weibo	2.4.0 ✓	8	10		✓
13	Dolphin Browser HD	11.5.3 ✓	14	17		✓
14	Maxthon Android Web Browser	4.5.8. 2000 ✓	8	21	✓	
15	Boat Browser Mini	3.0.2	7	7	✓	-
16	Mobile Security Personal Ed.	7.0 ✓	9	23		
17	Droid Call Filter	1.0.23	4	-	✓	
18	Tc Assistant	4.5.0 ✓	7	-	✓	
19	GO TwiWidget	2.1	5	-	✓	✓
20	Sina Weibo	6.3.1 ✓	14	23		
21	Tencent WBlog	6.1.2 ✓	10	-		

1. `Pansi SMS` is a messaging app which is vulnerable to the leakage and pollution attacks. ContentScope reports that the `.provider.MsgSummaryProvider` is vulnerable. However, this component is protected by both `readPermission` and `writePermission` in the updated version of the app. Therefore, malware will not be able to access this component. However, DBDroidScanner finds other content provider components which are not explicitly protected. The target SDK version for this app is 13 (<17), hence the content providers are exported by default and this application is vulnerable in all compatible Android platforms. The vulnerable content providers are: `.provider.PansiContactProvider`, `.provider.PhraseProvider` and `.provider.SmsDraftProvider` which may be compromised by attackers to steal or manipulate information such as contact details.

3. `mOffice - Outlook sync` is a productivity app which synchronizes private contact information with remote desktops. The updated app studied in our work does not export `.dao.DBProvider` anymore but `.dao.BackupProvider` is another content provider vulnerable to leakage and pollution attacks. Thus, attackers can steal and manipulate the sensitive data such as SMS and contact details.

6. The updated version of `GO SMS Pro`, which is an instant messaging app, is vulnerable under Android 16 and below. The `.StaticDataContentProvider` and `.golauex.smswidget.SmsProvider` components are unprotected content providers vulnerable to the public leakage and pollution attacks.

7. In the `Messenger WithYou` app, we find that the `openFile()` method in the `MiyowaExplorerContentProvider` content provider may return arbitrary database file descriptors, which is similar to ContentScope. In addition, we find the `openAssetFile()` API as another source method to trigger a similar vulnerability.

14. We could only find an updated version of the `Maxthon Android Web Browser` app. We find SQL injection and pollution vulnerabilities in the `.BrowserProvider` content provider if it runs in Android 16 and below. Even though ContentScope reports that the vulnerability in this app is fixed, the `exported` attribute of the vulnerable provider in the manifest file is not explicitly set to `"false"` in the updated version, nor is it protected by any permission. Hence, DBDroidScanner still reports vulnerabilities which are consistent with descriptions provided in [ZJ13]. These attacks can successfully be launched in Android <= 16.

18. The `Tc Assistant` app logs outgoing calls and traffic. The `.net.provider.TrafficProvider` and `com.wali.android.provider.LogsContentProvider` content providers in the updated version of this app are not protected resulting in sensitive data leakage.

Private Database Attacks

Now we present the private database attack case-studies for the vulnerable apps in Table 5.3. We remark that ContentScope only aims to detect the public database vulnerabilities, however, our analysis also finds the private database vulnerabilities in these representative apps analyzed by ContentScope. The last column in Table 5.3 shows our results for the private database vulnerabilities. The dynamic testing for a few apps could not be done due to some practical problems, e.g., we have to register a valid phone number for one app or the app crashes once launched due to incompatibilities. Thus, we do not confirm if these apps are vulnerable or not. We now discuss the case-studies by the ID column of Table 5.3.

1. `Pansi SMS` is vulnerable to private pollution and leakage attacks. The attacker can launch privilege escalation attacks using the vulnerable `MrBeanUpgradeMsgActivity` activity in this app to update the SMS content. It is also possible to exploit a vulnerability in `SearchActivity` activity to force this app to search the messages in the phone for a keyword.

3. The `mOffice - Outlook sync` app which has SMS read and write permissions can be reached by a malware through its `com.innov8tion.mobisynapse.core.SMS Receiver` component. By sending a malicious intent, the malware can choose an SMS ID and launch a privilege escalation attack by forcing the app to send an update request to the content provider of the default SMS app on the phone. Malware can also force the `mOffice - Outlook sync` app to send query requests to `.dao.DBProvide` even though this component is not exported and has been protected by `readPermission`. The malware can craft malicious intents and choose arbitrary task identifiers (e.g., `taskID`) to invoke the vulnerable component `.activity.task.EventEditActivity` to launch leakage attacks.

4. A malware on the phone can launch privilege escalation attacks and force the `Shady SMS 3.0 PAYG` app to manipulate a particular SMS content.

7. `Messenger WithYou` is vulnerable to the private pollution and leakage database attacks. For instance, the malware can send malicious intents to the `Capptain WebAnnouncementActivity` component to send query or delete requests using arbitrary `WHERE` clauses.

11. The `GO FBWidget` app has a vulnerability which can be exploited to send requests to manipulate Facebook authentication data.

12. An existing malware on the phone can force the `Netease Weibo` app to open an input stream for a given URI.

13. Although `Dolphin Browser HD` is updated to fix its public database vulnerabilities, it happens to be vulnerable to private leakage attacks. The vulnerable

component is the `AddBookmarkPage` activity which allows malware to force this component to send query requests to the `BrowserProvider` even though the latter is not exported in the updated version of the app. Therefore, the attacker can access the component which was reported by ContentScope [ZJ13] to be vulnerable in a new way.

19. `GO TwiWidget` is another app which can be forced by the malware to query the details of an account on the device. This app is vulnerable to the private leakage attacks.

5.6 Related Work

In the past few years, many different aspects of Android security have been studied. Even though many works aim to detect vulnerabilities in benign apps [GZWJ12, CHY12, HUHS13, ARF+14, LLW+12], few works also try to exploit them. Database vulnerabilities which may compromise the security of system has been partially studied in [ZJ13, LLW+12].

ContentScope [ZJ13] is designed for finding pollution and leakage attacks on content providers in the Android app. Their work finds public database vulnerabilities in Android apps built for the Android SDK 16 or lower in which the content provider components which by default are accessible by other apps on the phone. Our work is related as both detect database vulnerabilities and generate exploits for them. However, we focus on apps whose SDK target is 17 and higher where the default assumptions have been fixed to provide more security. We only analyze the representative apps in [ZJ13] which have lower SDK targets to compare our results with ContentScope under their lower SDK assumption.

Moreover, ContentScope only focuses on the vulnerabilities in public databases triggered through standard content provider APIs. We also study a new class of attacks involving private database vulnerabilities triggered via inter-app communication. We also improve the precision of our symbolic execution by modeling and presenting the URI semantics and several Android libraries. However, [ZJ13] does not present any details of library modelling.

CHEX [LLW+12] is an information flow analysis tool which is tailored to detect component hijacking vulnerabilities in benign Android apps. Component hijacking attacks happen when an unauthorized app, issuing requests to one or more exported components in a vulnerable app, seeks to read or write sensitive data. CHEX reports the potential vulnerabilities pertaining to the private databases. Compared to this work, our analysis system takes one step further and generates working exploits for the detected vulnerabilities.

Privacy leakage and privilege escalation [DDSW10] attacks are two more broad classes of attacks that have attracted researchers during the past few years. Privacy leakage might happen as the result of over-privileged malwares or privilege escalation attacks due to application bugs or flaws in the system design. A few systems have been proposed to mitigate these classes of attacks [OMEM09, DSP+11, BGH+13, NKZ10, HHJ+11, ZY14, XSA12, JMV+12].

Even though these mitigation mechanisms make such attacks more difficult to succeed, they can only prevent a subset of attacks. Moreover, they impose additional overhead at runtime or require changes in the underlying framework, thus not directly applicable to the existing systems. Detection of vulnerabilities that lead to such attacks is an alternative approach which is the focus of this chapter. Our work differs from these systems by focusing on detecting and exploiting specific classes of vulnerabilities in benign applications that lead to the privacy leakage or privilege escalation attacks. The target attacks in this chapter are database leakage, pollution and file access vulnerabilities.

Another line of research is the static analysis of apps to find potential vulnerabilities in Android applications [GZWJ12, GZJS12, GCEC12, EBFK13, CHY12, YY12, KYYS12, FHM+12, SSG+14, HUHS13]. The vulnerabilities reported by these works may have false positives.

Android apps pervasively use libraries such as `android.net.Uri` to perform operations on strings and structured data. There is a body of work which support string manipulation operations statically for different applications [SAH+10, CMS03, KGG+09, TCJ14]. We transform the utility methods of URI classes to SFTs [HLM+11, VHL+12] and leverage the CVC4 SMT solver [LRT+14] to support String, Integer and Boolean theories.

Static string analysis is another line of research which determines the values of string expressions at a given program point [CMS03, LLWH15, YBCI08]. Our approach differs from these works as we intend to generate strings that drive execution paths to reach a program point. Java String Analyzer [CMS03] models flow graphs of string operations to a context-free grammar which is over-approximated to a finite state automaton. Instead, we use the existing SMT solvers which support numeric and string constraints simultaneously to check the satisfiability of string constraints and generate test inputs.

Li et al. [LLWH15] present a general-purpose string analysis framework for Java and Android apps. The framework works based on an IR that represent the control flow and data flow relationships among string variables and string operations in the program. Programmers can implement interpreters to model the semantics of string operation and provide analysis properties such as context-

sensitivity and loop handling. In contrast, we use symbolic execution, a different approach which is a context-sensitive analysis but might have scalability problems. We have tried to improve the scalability of our analysis by several optimizations presented in this chapter and Chapter 3. Moreover, [LLWH15] does not address the more complex URI-based libraries which rely on string operations but also introduce more semantics.

Yu et al. [YBCI08] use deterministic finite automaton (DFA) to represent the set of values string expressions can take. In this work, first a control flow graph is constructed where nodes are string operations. Given an attack pattern, a string variable and a program point, it computes the DFA that accepts the language that corresponds to all the string values that the variable can take at a program point. Their analysis can be used to prove the correctness of sanitizers. Since we need to generate exploits which satisfy constraints imposed by libraries (e.g., URIs), we combine SMT formulas with automata models.

5.7 Summary

In this chapter, we have studied the database attacks targeting Android apps which can compromise the security of the system. While existing works [ZJ13, LLW+12] partially study and detect the database attacks, we present a comprehensive classification of these attacks: private and public database attacks.

We have extended and customized our analysis system introduced in Chapter 3 to detect and exploit the database vulnerabilities. More specifically, we have provided symbolic models for URI-based Android libraries such as `android.net.Uri` to generate more precise exploits automatically. For this purpose, we leverage SMT formulas and symbolic finite transducers to achieve better precision while maintaining the scalability. As a future work, it can be investigated whether our proposed models for the URI-based libraries can be generated automatically from the library code-base.

We have analyzed apps for both private and public database vulnerabilities. We have evaluated our system on 924 of Android apps from Google App store and found 52 apps vulnerable to public and 23 apps vulnerable to private database attacks (153 database vulnerabilities in total). We have compared our results with ContentScope [ZJ13] and surprisingly some of the apps are still vulnerable.

Unfortunately, the existing protection mechanism for private databases is too inflexible and developers might not be able to specify effective permissions. We recommend that instead of private databases, developers should use public databases for sharing data with other apps as the latter can be protected with more fine-

grained access control mechanisms.

Even though Android provides a relatively fine-grained protection mechanism for public databases, developers do not protect them carefully. We recommend more effective use of the protection mechanism available for public databases. Furthermore, developers can avoid private and public database vulnerabilities by validating the input data coming from the untrusted sources and protect the entry point components by using proper permissions.

Chapter 6

Conclusion and Future Work

In this thesis, we have introduced a novel analysis framework to detect and confirm data injection vulnerabilities in benign Android apps. We have studied two important classes of such vulnerabilities and developed analysis systems to detect and show proof-of-concept exploits in real-world apps.

First, we have developed an automated vulnerability detection system for Android apps which not only finds data injection vulnerabilities but also confirms them. Our tool starts with static dataflow analysis to reduce the number of source-sink pairs and uses symbolic execution to generate inputs and avoid reporting infeasible paths. Next it constructs working exploits and confirms the potential vulnerabilities using dynamic testing. Integrating these phases, we have designed an analysis framework which can be used as part of the application vetting process in App-stores. We show through experiments that this design significantly enhances the detection precision compared with an existing state-of-the-art dataflow analysis and maintains the efficiency.

Second, we have presented a detailed study of a new class of application vulnerabilities on Android platform that allows a malicious web attacker to exploit app vulnerabilities. It can be a significant threat as no malicious apps are needed on the device and the remote attacker has full control on the web-to-app communication channel. Using our analysis framework, we have conducted the first large-scale analysis on real apps from the official Google Play store – we have found many confirmed vulnerabilities which suggest that these attacks are pervasive and developers do not adequately protect apps against them.

Finally, we have conducted a systematic study of attacks targeting databases in benign Android apps. We have provided a more comprehensive classification for these attacks compared to the previous works. Our attacks can be triggered from a malware either by invoking content provider APIs or sending Intents which can be received throughout the victim app. In order to detect and exploit zero-day

database vulnerabilities, we have utilized our analysis framework and extended it with mechanisms for symbolically executing operations on the URI-based objects that are involved in database management. To the best of our knowledge, we are the first to analyze Android apps for these more comprehensive database attacks. We have found and confirmed many vulnerabilities and generated working exploits for them.

The focus of this thesis has been on analyzing data injection vulnerabilities in benign apps. As a future work, it would be interesting to generalize the analysis framework in Chapter 3 to also detect malware. Malware detection has additional challenges as attackers often use anti-analysis techniques to hide the malicious behavior of their apps. For instance, they may use dynamically loaded code, JNI or reflection to make it more difficult for analyzers to reach the malicious part of the code. The analysis framework presented in this thesis needs to be extended to deal with such challenges. One possible approach is to use existing tools like ConDroid [SFT15] which is based on concolic testing to reach the parts of the code which are not available statically and analyze them using our analysis framework.

While our analysis framework is able to generate complex data inputs (e.g., intents), generating event sequences might help to improve test coverage and detect more attack categories. Anand et al. [ANHY12] have designed a concolic testing framework for Android apps for generating event sequences. Integrating these two systems can be investigated as a future work to improve path coverage and find other types of attacks.

In this thesis, we have systematically studied security-critical Android vulnerabilities which can be exploited by attackers to launch devastating attacks. However, mitigating such attacks remains an open problem. Our results show that the W2AI vulnerabilities are prevalent in Android apps and abuse the web-to-app channel which is also used by Google app indexing. Considering the benefits of this channel, it is very likely that it gets even more popular and implemented by innumerous apps. Therefore, securing this channel seems to be crucial. On the other hand, database vulnerabilities which result in sensitive data leakage and pollution have inadequate protection mechanisms. In particular, our study shows that the protection mechanism for private databases is so coarse-grained that developers might give up the security altogether. More fine-grained mitigation techniques for database attacks can be another future research direction.

Bibliography

[ABL+10] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. Towards a Formal Foundation of Web Security. In *Computer Security Foundations Symposium (CSF)*, 2010.

[AFT11] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. A GUI Crawling-based Technique for Android Mobile Application Testing. In *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2011.

[AIN] Android Application Deep Linking. `https://developers.google.com/app-indexing/`.

[ANHY12] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Symposium on the Foundations of Software Engineering (FSE)*, 2012.

[ANU] Android Intents with Chrome. `https://developer.chrome.com/multidevice/android/intents`.

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[ARHB15] Steven Arzt, Siegfried Rasthofer, Robert Hahn, and Eric Bodden. Using Targeted Symbolic Execution for Reducing False-positives in Dataflow Analysis. In *Workshop on State Of the Art in Program Analysis (SOAP)*, 2015.

[ASH+14] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[BCE08] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. RWset: Attacking Path Explosion in Constraint-based Test Generation. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[BGH+13] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp Styp-Rekowsky. AppGuard – Enforcing User Requirements on Android Apps. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2013.

[BJM+15] Paulo Barros, Rene Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D. Ernst. Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents. In *Conference on Automated Software Engineering (ASE)*, 2015.

[BKLTM12] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In *Workshop on State Of the Art in Program Analysis (SOAP)*, 2012.

[BLFIM98] T. Berners-Lee, R. Fielding, U.C. Irvine, and L. Masinter. Uniform Resource Identifiers (URI), 8 1998. RFC 2396.

[BOU] Android and Security. `http://googlemobile.blogspot.sg/2012/02/android-and-security.html`.

[BST+10] Clark Barrett, Aaron Stump, Cesare Tinelli, Sascha Boehme, David Cok, David Deharbe, Bruno Dutertre, Pascal Fontaine, Vijay Ganesh, Alberto Griggio, Jim Grundy, Paul Jackson, Albert Oliveras, Sava Krsti, Michal Moskal, Leonardo De Moura, Roberto Sebastiani, To David Cok, and Jochen Hoenicke. The SMT-LIB Standard: Version 2.0. Technical report, 2010.

[CAS] Android Application Deep Linking Case Study. `https://developers.google.com/app-indexing/case-studies`.

[CCF03] Weihaw Chuang, Brad Calder, and Jeanne Ferrante. Phi-Predication for Light-weight If-conversion. In *Symposium on Code Generation and Optimization (CGO)*, 2003.

[CCK11] Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. Symbolic Cross-checking of Floating-point and SIMD Code. In *European Conference on Computer Systems (EuroSys)*, 2011.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *USENIX Security Symposium*, 2008.

[CGP+06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Conference on Computer and Communications Security (CCS)*, 2006.

[CHY12] Patrick P.F. Chan, Lucas C.K. Hui, and S. M. Yiu. DroidChecker: Analyzing Android Applications for Capability Leak. In *Conference on Security and Privacy in Wireless and Mobile Networks (WISEC)*, 2012.

[CK03] Edmund Clarke and Daniel Kroening. Hardware Verification Using ANSI-C Programs As a Reference. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2003.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004.

[CMS03] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise Analysis of String Expressions. In *Conference on Static Analysis (SAS)*, 2003.

[COR] Apache Cordova. `https://cordova.apache.org/`.

[CPC+14] Eric Y. Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. OAuth Demystified for Mobile Application Developers. In *Conference on Computer and Communications Security (CCS)*, 2014.

[CQM14] Qi Alfred Chen, Zhiyun Qian, and Z Morley Mao. Peeking Into Your App Without Actually Seeing It: UI State Inference and Novel Android Attacks. In *USENIX Security Symposium*, 2014.

[CS13] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 2013.

[CVWa] (CVE-2014-1939) Android WebKit Vulnerability. http://www.cvedetails.com/cve/CVE-2014-1939/.

[CVWb] (CVE-2014-6041) Android WebView Same Origin Policy Bypass Vulnerability. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6041.

[CW14] Erika Chin and David Wagner. Bifocals: Analyzing WebView Vulnerabilities in Android Applications. In *Workshop on Information Security Applications (WISA)*, 2014.

[DDSW10] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege Escalation Attacks on Android. In *Conference on Information Security (ISC)*, 2010.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-oriented Programs Using Static Class Hierarchy Analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, 1995.

[DSP+11] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *USENIX Security Symposium*, 2011.

[Due96] Evelyn Duesterwald. A Demand-driven Approach for Efficient Interprocedural Data Flow Analysis. *PhD thesis, Pittsburg*, 1996.

[EBFK13] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *Conference on Computer and Communications Security (CCS)*, 2013.

[EOMC11] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *USENIX Security Symposium*, 2011.

[FADA14] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based Detection of Android Malware Through Static Analysis. In *Symposium on the Foundations of Software Engineering (FSE)*, 2014.

[FDG] FlowDroid Wiki Page. https://github.com/secure-software-engineering/soot-infoflow-android/wiki.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, 6 1999. RFC 2616.

[FHM+12] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Conference on Computer and Communications Security (CCS)*, 2012.

[FW11] Adrienne Porter Felt and David Wagner. Phishing on Mobile Devices. In *Web 2.0 Privacy and Security (W2SP)*, 2011.

[FWM+11] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission Re-Delegation: Attacks and Defenses. In *USENIX Security Symposium*, 2011.

[GCEC12] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Conference on Trust and Trustworthy Computing (TRUST)*, 2012.

[GJS14] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. Breaking and Fixing Origin-based Access Control in Hybrid Web/Mobile Application Frameworks . In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[GKP+15] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. Information-flow Analysis of Android Applications in DroidSafe. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[GLM08] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[GPL] Leading App Stores. `http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/`.

[GZJS12] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe Exposure Analysis of Mobile In-app Advertisements. In *Conference on Security and Privacy in Wireless and Mobile Networks (WISEC)*, 2012.

[GZWJ12] Michael C. Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[HHJ+11] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These Aren'T the Droids You'Re Looking for: Retrofitting Android to Protect Data from Imperious Applications. In *Conference on Computer and Communications Security (CCS)*, 2011.

[HJY+15] Behnaz Hassanshahi, Yaoqi Jia, Roland H. C. Yap, Prateek Saxena, and Zhenkai Liang. Web-to-Application Injection Attacks on Android: Characterization and Detection. In *European Symposium on Research in Computer Security (ESORICS)*, 2015.

[HLM+11] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and Precise Sanitizer Analysis with BEK. In *USENIX Security Symposium*, 2011.

[HUHS13] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing Droids: Program Slicing for Smali Code. In *Symposium on Applied Computing (SAC)*, 2013.

[HZT+14] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. In *International Conference on Software Engineering (ICSE)*, 2014.

[INA] Intents and Intent Filters. `http://developer.android.com/guide/components/intents-filters.html`.

[IOS] Apple URL Scheme Reference. `https://developer.apple.com/library/ios/featuredarticles/iPhoneURLScheme_Reference/Introduction/Introduction.html`.

[JDW] Java Debug Wire Protocol. `http://developer.android.com/tools/debugging/index.html`.

[JHY+14] Xing Jin, Xunchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation. In *Conference on Computer and Communications Security (CCS)*, 2014.

[JMF12] Jinseong Jeon, Kristopher K. Micinski, and Jeffrey S. Foster. Symdroid: Symbolic Execution for Dalvik Bytecode. Technical report, 2012.

[JMV+12] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *Workshop on Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS (SPSM)*, 2012.

[Kap] David Kaplan. (CVE-2014-3500/1/2) Apache Cordova for Android - Multiple Vulnerabilities. `http://seclists.org/fulldisclosure/2014/Aug/21`.

[KGG+09] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A Solver for String Constraints. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[Kin76] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 1976.

[KKBC12] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient State Merging in Symbolic Execution. In *Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[Kor90] B. Korel. Automated Software Test Data Generation. *IEEE Trans. Softw. Eng.*, 1990.

[KYYS12] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications. In *Workshop on Mobile Security Technologies (MOST)*, 2012.

[LBHD11] Patrick Lam, Eric Bodden, Laurie Hendren, and Technische Universitt Darmstadt. The Soot Framework for Java Program Analysis: a Retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, 2011.

[LH03] Ondřej Lhoták and Laurie Hendren. Scaling Java Points-to Analysis Using SPARK. In *Conference on Compiler Construction (CC)*, 2003.

[LHD+11] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on WebView in the Android System. In *Annual Computer Security Applications Conference, ACSAC*, 2011.

[LLW+12] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Conference on Computer and Communications Security (CCS)*, 2012.

[LLWH15] Ding Li, Yingjun Lyu, Mian Wan, and William G. J. Halfond. String Analysis for Java and Android Applications. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015.

[LLZW14] Chia-Chi Lin, Hongyang Li, Xiaoyong Zhou, and XiaoFeng Wang. Screenmilker: How to Milk Your Android Screen for Secrets. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[LRT+14] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions. In *Conference on Computer Aided Verification (CAV)*, 2014.

[MABR12] Amiya Kumar Maji, Fahad A. Arshad, Saurabh Bagchi, and Jan S. Rellermeyer. An Empirical Study of the Robustness of Inter-component Communication in Android. In *Conference on Dependable Systems and Networks (DSN)*, 2012.

[MMP+12] Nariman Mirzaei, Sam Malek, Corina S. Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. Testing Android Apps Through Symbolic Execution. In *SIGSOFT Softw. Eng. Notes*, 2012.

[MTN13] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An Input Generation System for Android Apps. In *Symposium on the Foundations of Software Engineering (FSE)*, 2013.

[NKZ10] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010.

[OMEM09] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically Rich Application-centric Security in Android. In *Annual Computer Security Applications Conference (ACSAC)*, 2009.

[PHO] PhoneGap. `http://phonegap.com/`.

[RAB14] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Symposium on Principles of Programming Languages (POPL)*, 1995.

[SAH+10] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A Symbolic Execution Framework for JavaScript. In *Symposium on Security and Privacy (SP)*, 2010.

[sea] SeattleClouds. `http://seattleclouds.com/`.

[SFT15] Julian Schütte, Rafael Fedler, and Dennis Titze. ConDroid: Targeted Dynamic Analysis of Android Applications. In *Conference on Advanced Information Networking and Applications (AINA)*, 2015.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In *European Software Engineering Conference Held Jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2005.

[SR14] Raimondas Sasnauskas and John Regehr. Intent Fuzzer: Crafting Intents of Death. In *Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, 2014.

[SSG+14] D Sounthiraraj, J Sahs, G Greenwood, Z Lin, and L Khan. Smv-hunter: Large scale, Automated Detection of SSL/TLS Man-in-the-middle Vulnerabilities in Android Apps. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[TCJ14] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *Conference on Computer and Communications Security (CCS)*, 2014.

[Ter] Takeshi Terada. Whitepaper Attacking Android Browsers Via Intent Scheme URLs. `http://www.mbsd.jp/Whitepaper/IntentScheme.pdf`.

[TKFC15] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[VHL+12] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Bjorner. Symbolic Finite State Transducers: Algorithms and Applications. In *Conference on Implementation and Application of Automata (CIAA)*, 2012.

[WHI] Whitelist Guide. `http://docs.phonegap.com/en/3.5.0/guide_appdev_whitelist_index.md.html`.

[WXWC13] Rui Wang, Luyi Xing, XiaoFeng Wang, and Shuo Chen. Unauthorized Origin Crossing on Mobile Platforms: Threats and Mitigation. In *Conference on Computer and Communications Security (CCS)*, 2013.

[XSA12] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *USENIX Security Symposium*, 2012.

[XSS] Cross-Site Scripting (XSS). `https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`.

[YBCI08] Fang Yu, Tevfik Bultan, Marco Cova, and Oscar H. Ibarra. Symbolic String Verification: An Automata-Based Approach. In *Workshop on Model Checking Software (MCS)*, 2008.

[YCZJ13] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. DroidFuzzer: Fuzzing the Android Apps with Intent-filter Tag. In *Conference on Advances in Mobile Computing and Multimedia (MOMM)*, 2013.

[YY12] Zhemin Yang and Min Yang. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In *World Congress on Software Engineering (WCSE)*, 2012.

[ZJ12] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Symposium on Security and Privacy (SP)*, 2012.

[ZJ13] Yajin Zhou and Xuxian Jiang. Detecting Passive Content Leaks and Pollution in Android Applications. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[ZLZ+14] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *Symposium on Security and Privacy (SP)*, 2014.

[ZWZJ12] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[ZY14] Mu Zhang and Heng Yin. AppSealer: Automatic Generation of Vulnerability-specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

Appendices

Appendix A

Translating Jimple IR to SMT-LIB Format

We use Jimple as the intermediate representation (IR) in our analysis. Jimple is the three-address IR used in Soot [LBHD11]. In Chapter 3, we introduced an analysis framework which uses symbolic execution as its core analysis. Our symbolic execution relies on CVC4 SMT solver [LRT+14] to check the satisfiability of path constraints and to generate inputs that run an execution path.

The CVC4 used in this thesis supports SMT-LIB(v2) [BST+10] for input and output languages. We use `QF_S` as the basic logic which is the theory of strings. The `produce-models` option is used to activate model generation and `incremental` or multiple query solving is enabled to check more than one problem. We also use `strings-exp` option which allows us to handle a wide range of string operations (e.g., `str.to.int` which converts a string to integer).

Jimple IR supports primitive and reference types. Sometimes, these types cannot directly be used in the SMT-LIB(v2) format. For example, we need to understand where integer types are numeric types and where they are boolean types (represented as 0 and 1) and convert the integer to boolean type (represented as true and false) in the latter case. We try to resolve the types and perform a type checking before translating any Jimple statement to the SMT format. We use three SMT-LIB(v2) format types while translating from Jimple IR : `Int`, `String` and `Bool`. Our translator converts other numeric types such as long and double to `Int` if possible.

`String` types consists of several Jimple reference types, e.g., `StringBuilder`, `StringBuffer`, `CharSequence`. Our symbolic executor supports Java Reference types in equality conditions by generating unique identifiers for each Java object. These constraints are then translated to string equality operations supported by

Table A.1: Translating representative Java methods to SMT-LIB(v2) syntax.

Method	SMTLIB language
`X.contains(Y)`	`(str.contains X Y)`
`X.concat(Y)`	`(str.++ X Y)`
`X.append(Y)`	`(str.++ X Y)`
`X.startsWith(Y)`	`(= (str.++ Y T) X)`
`X.endsWith(Y)`	`(= (str.++ T Y) X)`
`X.length()`	`(str.len X)`
`X.equals(Y)`	`(= X Y)`
`X.charAt(N)`	`(str.at X N)`
`X.substring(I, J)`	`(str.substr X I J)`
`X.indexOf(Y, I)`	`(str.indexof X Y I)`
`X.replace(Y, Z)`	`(str.replace X Y Z)`
`String.valueOf(I)`	`(int.to.str I)`
`Integer.valueOf(X)`	`(str.to.int X)`
`Integer.parseInt(X)`	`(str.to.int X)`
`X.isEmpty()`	`(= (str.len X) 0)`
`X.split(Y)`	(or (= X (str.++ X_1 Y X_2)) (= X (str.++ X_1 Y X_2 Y X_3))) $\wedge$ (not (str.contains X_1 Y)) $\wedge$ (not (str.contains X_2 Y))

CVC4.

Table A.1 shows a summary of translations of some of the Java methods[1] to SMT-LIB(v2) syntax. For instance, `Integer.valueOf()` and `String.valueOf()` are representative methods for several Java classes (e.g., `Float`) which are handled in a similar way. While some of the Jimple instructions are directly translated to SMTLIB syntax, e.g., `String.contains()`, some of them are translated using a combination of existing SMTLIB operations, e.g., `startsWith()` and `split()`. We support a bounded version of `split()` method (`splitN`) which generates `N` new strings. In Table A.1, we show a translation of the `split()` method where `N` is two.

[1]We actually translate the invocations of these methods in the Jimple IR.