Spring 5-19-2017

Malware Analysis and Privacy Policy Enforcement Techniques for Android Applications

Aisha Ibrahim Ali-Gombe
University of New Orleans, aaligomb@uno.edu

Malware Analysis and Privacy Policy Enforcement Techniques for Android Applications

A Dissertation

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Engineering and Applied Sciences
with Concentration in
Computer Science
(Information Assurance)

by

Aisha I. Ali-Gombe

BSc, University of Abuja, 2005
MBA, Bayero University, Kano, 2011
M.S. University of New Orleans, 2012

May 2017

Dedication

In loving memory of my father who taught me hard work, perseverance, and resiliency; you are my hero. Like you always say success is not an option it is the only choice - indeed it is. To my husband Ibrahim; for our timeless friendship and deep affection. Without your endless support, this wouldn't have been possible. You are my rock. My mother Hajiya; your love and prayer kept me going through difficult times. Finally to my babies - Maryam, Fatima, and Mohammad Amin. The journey hasn't been easy for all of us, but we made it together. Thank you for all the sacrifices.

Acknowledgement

I would like to express an immense gratitude to my supervisors Dr. Golden G. Richard III and Dr. Irfan Ahmed for their guidance and mentorship. They have indeed made a difference in my life, and I will forever be grateful. I want to acknowledge the funding opportunity they gave me through the NSF (CNS #1409534) and faculty start-up grant. I would also like to thank members of my dissertation committee Dr. Vassil Roussev, Dr. Edit Bourgeois and Dr. Juliette Ioup for their time and effort in making this work a success. I want to thank my husband for all his encouragement and for giving me a shoulder to lean on when things were rough. To my kids, mom and my 21 siblings for their unconditional love. Finally, I want to thank my in-laws, my extended family, and friends for all their support.

Contents

List of Figures

List of Tables

Abstract

The rapid increase in mobile malware and deployment of over-privileged applications over the years has been of great concern to the security community. Encroaching on user's privacy, mobile applications (apps) increasingly exploit various sensitive data on mobile devices. The information gathered by these applications is sufficient to uniquely and accurately profile users and can cause tremendous personal and financial damage.

On Android specifically, the security and privacy holes in the operating system and framework code has created a whole new dynamic for malware and privacy exploitation. This research work seeks to develop novel analysis techniques that monitor Android applications for possible unwanted behaviors and then suggest various ways to deal with the privacy leaks associated with them.

Current state-of-the-art static malware analysis techniques on Android focused mainly on detecting known variants without factoring any kind of software obfuscation. The dynamic analysis systems on the other hand are heavily dependent on extending the Android OS and/or runtime virtual machine. These methodologies often tied the system to a single Android version and/or kernel making it very difficult to port to a new device. In privacy, access to the database system's objects are not controlled by any security check beyond overly-broad read/write permissions. This flawed model exposes the database contents to abuse by privacy-agnostic apps and malware. This research addresses the problems above in three ways.

First, we developed a novel static analysis technique that fingerprints known malware based on three-level similarity matching. It scores similarity as a function of normalized opcode sequences found in sensitive functional modules and application permission requests. Our system has an improved detection ratio over current research tools and top COTS anti-virus products while maintaining a high level of resiliency to both simple and complex obfuscation.

Next, we augment the signature-related weaknesses of our static classifier with a hybrid analysis system which incorporates bytecode instrumentation and dynamic runtime monitoring to examine unknown malware samples. Using the concept of Aspect-oriented programming, this technique involves recompiling security checking code into an unknown binary for data flow analysis, resource abuse tracing, and analytics of other suspicious behaviors. Our system logs all the intercepted activities dynamically at runtime without the need for building custom kernels.

Finally, we designed a user-level privacy policy enforcement system that gives users more control over their personal data saved in the SQLite database. Using bytecode weaving for query re-writing and enforcing access control, our system forces new policies at the schema, column, and entity levels of databases without rooting or voiding device warranty.

Android, Malware Analysis, Fingerprinting, Hybrid Analysis, Instrumentation, AspectJ, Similarity Matching, Privacy, Security, Obfuscation

Chapter 1

Introduction

Smartphones are powerful, high-tech devices designed with an operating system of a traditional computer. It's low-level system design together with other sophisticated hardware and sensor components has completely changed the face of handheld devices. This technology has not only revolutionized our telephony experience but has successfully integrated a vast amount of personal data, including our address books, calendars, diaries, pictures, etc., onto a single device. From a security perspective, the ease and convenience provided by this integration can have disastrous consequences, serving as a single point of exposure for a tremendous amount of personal data if not properly managed.

Android is a smartphone operating system developed by Google in alliance with 15 other tech companies. The objective was to design an open standard operating system for mobile devices [6]. This operating system, which was unveiled in 2007, has consistently enjoyed wide acceptance by many high-tech device manufacturers. According to a report by Statista 2016 [4], Android has maintained dominance in the global smartphone market for over five years in a row.

The Android system is built on top of a Linux kernel, which provides all the low-level management and access to the hardware components. Its security framework is designed to protect data and resources using two important concepts: Application sandboxing and a permissions model. With the permissions model, access is granted

by exclusive consent of the device user at installation time. However, its inflexibility in choices and irrevocability have left device user's vulnerable to privacy and security exposure by both malware and over-privileged or privacy-agnostic applications. Although the newest versions of Android are not designed with the all-or-nothing permission model, as of May 2016 [1], they constitute only about 13.1% of all global Android market share, leaving the remaining 86% vulnerable to extreme data abuse.

This thesis focuses on security monitoring of Android application for possible illicit behaviors via `malware analysis`. It also develops `privacy policy` enforcement techniques aimed at limiting access to very important device data.

1.1 Android Malware Analysis

The growing threat to user's security and privacy by Android applications has significantly increased the need for more reliable and accessible app analysis systems. Android apps are well-known for security and privacy violations and data leakage [42]. For instance, they transfer personal data outside the devices of end-users without their consent. Andrubis [78] performed an analysis on over a million (malicious, and benign) apps, and found that 38.79% of the apps have data leakage. The percentage further increases from 13.45% in 2010 to 49.78% in 2014, and is also noted by Yajin *et al.* [91].

In 2014, the Android platform is estimated to account for 97% of all malware on mobile devices [3]. According to GDATA report [71], over 2 million Android malware samples were detected in 2015, representing over a 50% increase from 2014. Modern malware is in use on an industrial scale by crime organizations and its development is often highly professional. In many respects, this presents an even greater threat to users than before, as mobiles are entrusted with the most private of information and mobile malware can very effectively spy on users in real time.

Traditionally, Android apps are analyzed using either static or dynamic approaches. Static analysis involves the use of predetermined signatures and/or other semantic artifacts such as API calls, strings etc. Enck et al. developed Kirin [33] which evaluates privacy risk based on the set of permissions requested, while [36, 93] analyzed Android applications by evaluating fine-grained API calls in addition to the permissions set. Other semantic-based analysis tools [38, 79] examine components and intents in addition to the permissions and API calls made within the application binary.

Dynamic analysis, on the other hand, executes a target application in a contained environment [84, 83, 15, 18, 19, 35, 51, 32, 31, 62] and monitors its behavior. In general, static analysis has the advantage of high performance and coverage. Conversely, simple obfuscation can hinder the extraction of important data such as API names. Dynamic analysis on the other hand provides a better view of an app's behavior, although, it is usually limited in scope to observed execution paths.

1.2 Data Privacy

While malware is a significant security threat on Android devices, privacy-related issues posed by over-privileged applications is equally threatening to mobile device users.

Android requires third-party applications to make explicit requests for resources and access to data at installation time and while this mechanism provides a general idea of what an application can access on a device, it does not provide the ability to institute fine-grained control over sensitive data. Essentially, it's an all-or-nothing model on most versions of Android under which the user has to approve all permissions or abort the installation of the application. Perhaps more disturbing is that the approved permissions remain a right of the installed application as long as it remains

on the phone.

Android extends the permission model to cover structured data stored in SQLite databases. However, it does not separate roles and privileges on the database, nor does it protect content data at the schema or entity levels. In fact, it does very little to protect the privacy of the stored user data and its associated metadata. Such a wide level of access is tantamount to giving the application administrative rights over the target provider. For example, this system does not distinguish accesses to a contact's phone number from the email and physical addresses. Other important information like the "last time contacted" as well as account type and names are also easily accessible with a simple READ permission. Similarly, write permission on the contacts provider allows an application to insert, delete and modify any contact at will. The application can also create groups and make them invisible. Such "perceived" benign access, however, can lead to malicious contacts being created and synched to restricted groups in major accounts like Google.

1.3 Contributions and Outline

This research work develops a fast and efficient static fingerprinting algorithm that can detect obfuscated malware variants with a high degree of accuracy. It scores similarity as a function of structural and behavioral features which are matched based on 3-level similarity matching algorithm.

Although fast, accurate, and resilient, this static analyzer is heavily dependent on known signatures and as such cannot detect unknown samples. To augment this drawback, we designed an app-level hybrid analysis system that monitors Android apps for possible illicit activities using bytecode instrumentation. This system performs taint-tracking, resource abuse tracing and analytics of suspicious activities independent of the Android runtime and/or kernel.

Finally, this thesis also addresses privacy issues on Android SQLite databases by enforcing access control at schema, column and entity levels, thus giving users absolute control over their data.

The outline of this work is organized as follows:

- Chapter 2 introduces software obfuscation techniques with emphasis on transformations that can hinder static analysis on Android. This chapter is largely based on existing work.

- Chapter 3 describes the first contribution of this thesis. *OpSeq* is a static malware fingerprinting algorithm based on statistical similarity that includes - feature extraction, signature generation, and matching processes. The chapter provides detailed description of its prototype implementation and the evaluation of the experimental results. The material from this chapter is drawn from a workshop paper [11] which appeared in the Privacy Protection and Reverse Engineering Workshop 2015. Since the original publication of this result, other researchers have proposed several alternatives to this approach [52, 65].

- Chapter 4 provides a brief background of software instrumentation and static bytecode weaving. It introduces the concept of Aspect-oriented programming and how it can be used to inject cross-cutting concerns in Android applications.

- Chapter 5 introduces *AspectDroid*, a hybrid analysis system for Android apps. This chapter highlights its workflow architecture and detailed implementation and evaluation of its prototype. The system and experimental results from this chapter are based on a published conference poster [12] and an extended version, currently under review by the Journal of Computer Virology and Hacking.

- Chapter 6 introduces SQLite databases as the RDBMS for Android systems. It also discusses the Android native content provider library as a security and

abstraction layer built on top of the SQLite engine.

- Chapter 7 discusses the last component of this thesis, called *priVy*, a privacy policy enforcement technique for Android applications that provides low-level access control for Android SQLite databases. In this chapter we explained how bytecode weaving can be used for privacy policy enforcement on Android SQLite database objects through access verification, query rewriting, and auditing. The prototype discussed here is adapted from a conference paper [13] published in WiSec 2016.

- Finally, Chapter 8 concludes with a summary of the contributions and possible future work.

Chapter 2

Software Obfuscation

Obfuscation involves the transformation of software code into an ambiguous form to hinder reverse engineering efforts without losing its functionality. Often obfuscation techniques are deployed to protect the intellectual propriety of software or by malware to circumvent static detection or fingerprinting algorithms.

Variants of the same malware can be created using simple or complex obfuscation techniques to evade detection. Such techniques make it difficult for analysts to manually understand the behaviors of a malware by thwarting disassembly and decompilation processes [68], e.g., through code packing, control flow redirection, etc.

Different obfuscation tools are available either as research prototypes, commercial products, or open source tools. How and where these schemes can be used depends on a number of factors:

- Software Programming Language: Languages with different intermediate representation often require different kinds of tools to obfuscate. Such tools have to understand the binary structure of the executable and disassembly representation. For example, tools designed for C/C++ compiled binaries may not work for Java executables. The former disassembles into assembly code with different kinds of instruction compositions (opcode and operand), while the latter disassembles into Java bytecode.

- Stealthiness: Depending on the need for obfuscation, malware may employ a trivial or non-trivial technique to make reverse engineering difficult. Simple techniques like identifier encryption and null pointer insertion can be used to deter understanding of the program's semantics. However, they may not necessarily make reverse engineering infeasible. Other high-end stealthy techniques use `Packers` to encrypt the malicious code or employ anti-disassembly techniques within the executable such as fake `jump instructions` to confuse the dissembler. Often the more complex the obfuscation is the more stealthy the malware will be.

- Cost: Although malware may want to block any access to its underlying code, it is vital that the program executes on a target system effectively with acceptable overhead.

- Detection Algorithm: There is always a correlation between the code analysis technique and obfuscation mechanism, especially in simple obfuscation. For example, string encryption can hinder semantic based detection algorithms but not opcode sequence-based tools. On the other hand, code insertion, substitution, and reordering will not have any devastating effect on semantic-based systems. However, complex obfuscation that involves more than one simple technique or employs stealthy code hardening like whole class encryption may completely hinder any form of static analysis.

2.1 Android Malware Transformation

Android malware are often created by injecting malicious payloads into benign applications. They employ various forms of obfuscation techniques to hide their presence from antivirus scanners. Recent studies have shown that common antivirus

software and static analysis tools are not resilient to such obfuscation techniques.

Most Android applications are written in Java, which are then compiled into bytecode in a class file. The compiled class files are further compressed using the `dx` utility into a `classes.dex` file. Although the execution of native code written in C/C++ is made possible using the Java native interface, very few Android apps use native code. Important application components such as package name, SDK version, component names etc., are found in the `AndroidManifest.xml`.

Various forms of software obfuscation on Android are employed on application files within the dex file, the Android manifest, resource files or in the native code. Based on the existing literature on Android obfuscation [63, 87, 58, 27], some of these schemes can be grouped as:

- Identifier Transformation: This involves a simple change in string identifiers and names from within the `dex` file or the `xml` files packaged in the Android application.

- Encryption: Algorithms such as simple cryptographic hashes or complex customizable algorithms can be employed by malware to deter analysis. Such encryption can include whole class encryption, identifier encryption (method names), and string encryption (URL names).

- Code Reordering: In this technique, chunks of code within the program body are repositioned through control flow changes. In practice, this will not change the functionality of the malware even though it may change the execution pattern.

- Junk Code Insertion: Null pointers and dead code through unreachable control flow can be added to the Android `dex` file to bloat the executable and completely break hash-based detection algorithms. This may also be employed to make program comprehension very difficult through manual examination.

- Java Reflection: This is a programming practice in Java that allows function calls to be resolved dynamically at runtime. Using this method, malware can make API calls inaccessible within the disassembled bytecode and as such techniques relying on API semantics will be thwarted.

Below is an example of how malware can use string encryption plus Java reflection to hide its request for Android IMEI.

Listing 2.1: Software Obfuscation

```
//Getting device ID without obfuscation
TelephonyManager mTelephony = (TelephonyManager)
    getSystemService(Context.TELEPHONY_SERVICE);
String imei = mTelephony.getDeviceId();
//Getting Android device ID (IMEI) using encryption and java reflection
//It first use the DecryptName function whose implementation decrypts the
    string encryptName to getDeviceId
String methName = DecryptName("encryptName");
Class nullParams []= {};
TelephonyManager mTelephony = (TelephonyManager)
    getSystemService(Context.TELEPHONY_SERVICE);
//Get Telephony class object
Class clazz = mTelephony.getClass();
//Initialize a new instance of Telephony object
Object myObj = clazz.newInstance();
//Get method name from Telephony class whose name = getDeviceId
Method meth = clazz.getDeclaredMethod (methName,nullParams);
//invoke getDeviceId on the new Telephony object
Object ret = meth.invoke(myObj, null);
}
```

Chapter 3

Static Malware Fingerprinting

3.1 OpSeq

Signature-based malware detection systems have long been used in identifying known malicious samples. On Android systems, malware is often introduced by repackaging benign applications with obfuscated malicious payloads, via a variety of transformations. These forms of obfuscation have been shown to trick commercial antivirus products [63, 87] and by extension the methodologies of many other Android research tools. Most of these common tools use algorithms that search application code for strings or other signatures, which can easily be subverted. Our aim is to develop a better system that can detect known malware, which have been obfuscated and repackaged within a new application.

In this task we present *OpSeq*–a new malware-variant detection approach that is resilient against common obfuscation techniques, including reflection, encryption, code reordering and junk insertion. It scores similarity as a function of normalized opcode sequences found in sensitive functional modules as well as app permission requests. This combination of structural and behavioral features results in a distinctive fingerprint for a malware sample, thereby improving our model's overall recall rate.

Given a malicious application, *OpSeq* performs static analysis of the application code and identifies functional level components (Java methods) referred to as *sensitive*

functional modules. These modules invoke vital APIs such as reflective, permission-guarded, resource/data access and network/file system activities. Based on the characterization of Android malware in well-known existing work [92], these sensitive APIs open a channel by which malicious apps can manipulate a victim's device.

OpSeq extracts the components from a known sample and creates corresponding signatures, which are used to search for similar components in target applications. Target applications are then classified as malicious or benign based on this evaluation. This approach is a significant improvement over existing work that targets opcode-sequence similarity [44, 90, 66], in that it filters out irrelevant application code (reducing noise in signatures) and focuses only on a small portion of code that has high potential to contain malicious code (making signatures more accurate). As a secondary feature, we use the list of requested permissions to improve detection accuracy, as app variants tend to have very similar permission sets.

3.2 Related Work

The first large-scale study of Android malware–the *Android Malware Genome Project*–was carried out by Zhou and Jiang [92]. Their work was aimed at characterizing existing Android malware but they did not detail their analysis and classification methodology. However, the corpus and characterization information they provided became the basis for a lot of follow up research, including ours. Their work identified that 86% of the malware is found as repackaged apps.

3.2.1 Opcode-sequence similarity

Santos et al. [66] developed a system of detecting malware using opcode-sequence frequencies on the *Intel x86* platform. On Android, Hanna and Zhou [44, 90] developed

methodologies for using opcode-sequences to detect repackaged applications in both primary and secondary app markets. Our work differs from theirs in terms of goals and approach: theirs is focused on detecting application repackaging in general, whereas we are interested in detecting malware, in particular. Their opcode-sequence-based detection can be disturbed with relative ease by injecting small amounts of noise; in contrast, we explicitly designed our approach to deal with different obfuscation techniques. Our system normalizes an opcode-sequence both on the known and target samples before comparison, which significantly reduces the effects of dead and junk code on our similarity measures.

For example, using the same target code snipped illustrated by [49] in their evaluation of Android repackaging detection algorithms, the original sequence is altered with series of junk instructions to form an obfuscated version as shown with the mnemonics in Listing 3.1. In the worst case, DroidMoss [90] can take the hash of the entire sequence (i.e., a 4-gram). In such cases, detection can be evaded completely. Conversely, in its best case, using a 2-gram rest point, DroidMoss can attain a maximum of ≈15% similarity.

However, our algorithm first normalizes both sequences into 2-gram sub-sequences, as shown in Listing 3.2. The target-overlap coefficient, which gives emphasis to the known profile, will be ≈ 67%. Furthermore, our small structured signatures target only sensitive functions, which can help eliminate most GUI code. This is useful because GUI code is almost always irrelevant for malware detection.

Listing 3.1: Original and obfuscated sequences

```
//Original sequence
invoke-static, move-result-object, const-string, invoke-interface
//Obfuscated sequence
invoke-static, move-result-object, move-object, const-string, move,
```

```
    invoke-interface, move, move-object
```

Listing 3.2: Original and obfuscated sequences

```
//Original sequence
const-string invoke-interface, invoke-interface invoke-static,
    invoke-static move-result-object
//Obfuscated sequence
const-string invoke-interface, invoke-interface invoke-static,
    invoke-static move, move move, move move-object, move-object
    move-object, move-object move-result-object
```

3.2.2 Semantic-based detection

Semantic-based detection uses information flows as features to detect similarity between Android applications [28, 30, 39, 67, 80, 88, 89]. PiggyApps [89] first identifies the code containing the main functionality (the primary module) in legitimate apps. Then, it extracts and organizes this semantic information from the module as a vantage point tree. The resulting signatures are used to search for "piggybacked" apps in Android markets. Apposcopy [39] provides a specification language that allows the manual creation of signatures for known malware. To find similarities, it extracts semantic features of a new app using inter-component call graphs and performs static taint analysis.

DroidLegacy [30] is an API-based static malware detection system that breaks an app into sub-modules. The set of API calls made in these modules are compared against the signatures of known samples. DroidAnalytics [88] uses a three-level signature that represents API calls made from within apps. API call sequences form signatures for methods, and the collection of all method signatures forms a signature

for each class. The collection of class signatures then forms the signature for the entire application. All of the techniques discussed above can be easily circumvented with simple obfuscation. Encryption alone can hinder data flow analyses while the combination of encryption and reflection will make it difficult to extract any meaningful information from the application code.

3.2.3 Permission-based certification

Kirin [34] detects dangerous behavior in applications by analyzing their permission requests. It uses a set of rules that defines which permissions combinations might be dangerous. Another permission-based behavioral fingerprinting is DroidRanger [94]. However, as detailed in [37], most Android apps are over-privileged in general and even benign apps have a tendency to request combinations of permissions that could be considered dangerous. SCanDroid [41] is a security certification tool that determines if specifications in the application manifest match what is requested within the app's components. RiskRanker [43] provides a systematic approach that measures the risk of dangerous behavior associated with an application based on native code, dynamic class loading, and callback handlers. VetDroid [86] uses dynamic analyses to reconstruct how permissions are used to access resources. All these techniques attempt to discover if dangerous behavior is present, while *OpSeq*'s primary goal is to measure similarity of unknown apps against known malware.

3.3 System Design

The *OpSeq* architecture consists of two major components as shown in Figure 3.1: feature extraction, and signature generation. Feature extraction identifies code in *sensitive functional modules* and extracts the corresponding opcode-sequences. It also

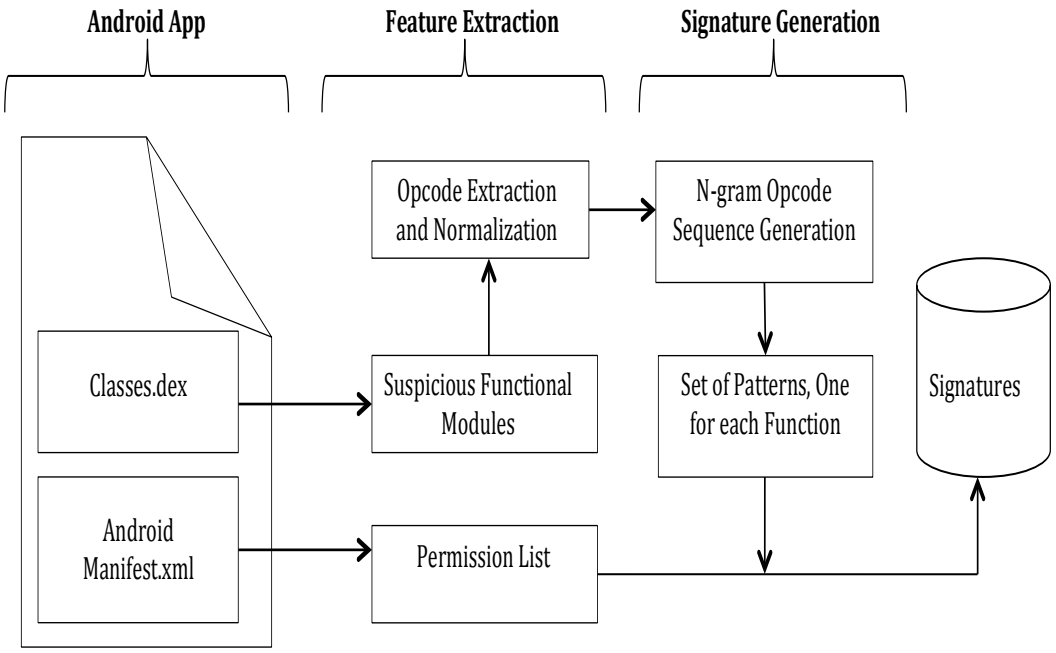

Figure 3.1: *OpSeq* Signature Generation Workflow

extracts the list of permissions used by the app to gain access to system resources. The signature generation step normalizes the sequences from feature extraction, and then slices each into a small chunk of n-gram opcodes, which constitute the signature used for similarity matching.

OpSeq's signature matching is a 3-step process, each illustrated in the algorithms 1, 2, and 3, respectively. We use a bottom-up approach consisting of three levels for similarity detection. First, we determine matches at the opcode level, and then their aggregate gives the similarity at the functional level. Finally, the result of functional-level and permission-overlap determines the final index. A similarity score is computed at each level where a subsequent level takes into account the score of its immediate last level, achieving substantial improvement in the overall accuracy of our system.

3.3.1 Feature extraction

In the feature extraction phase, permission requests and functional opcodes are extracted from the app's manifest and `classes.dex` files, respectively. The `classes.dex` file, denoted by cd, is a set of m Java classes, jc:

$$cd = \{jc^1, jc^2, \ldots, jc^m\}. \tag{3.3.1}$$

Each Java class $jc^k, 1 \leq k \leq m$ is made up of n functions.

$$jc^k = \{f_1^k f_2^k, \ldots, f_n^k\}. \tag{3.3.2}$$

We can simplify the notation by aggregating the set of functions in a dex file as:

$$cd = \sum_{j=1}^{m} \sum_{i=1}^{n} f_i^j. \tag{3.3.3}$$

An individual function f_i consists of set of instructions I: $f_i = \{I_1, I_2, \ldots, I_k\}$. Instructions are tuples containing an *opcode* o and a (potentially empty) list of operands. For the purposes of our analysis, we focus solely on the opcodes and we disregard the operands. In other words, we view a function f as a sequence of i opcodes:

$$f = o_1, o_2, \ldots, o_k. \tag{3.3.4}$$

To be included in the feature set, we filter the list of functions based on two criteria: a) the function must invoke at least one method from a *sensitive* API (a manually compiled list of selected system APIs); and b) its opcode sequence is not on the list of the most commonly found opcode sequences FS (determined empirically).

3.3.2 Signature generation

Our signatures are formed by taking into account the type of obfuscation that can affect opcode sequences; these include both junk code insertion and code reordering. Junk code insertion is an obfuscation technique which embeds pools of instructions that never execute at run time, such as instructions in an artificial `if-else` branch that never triggers. Code reordering permutes instructions that have no ordering dependencies; the main point is to subvert hash-based detection schemes.

The next step of the process is to normalize the extracted opcodes in each sequence *f*. This distorts the order of opcode arrangement and groups similar opcodes in the same cluster. Next, for each normalized opcode sequence, we generate sub-sequences of n-gram opcodes. In choosing the best value for n, we run our system with uni-gram, 2-gram and 3-gram. Empirical results (as shown in section 3.4.3) indicate that 2-gram gives the best accuracy. From now on, we refer to each sequence (representing one function) containing k number of 2-gram opcodes as a *pattern P*.

$$S = \{P^1, P^2, \ldots, P^m\} \tag{3.3.5}$$

where each

$$P^m = \{os_1^m, os_2^m, \ldots, os_i^m\} \tag{3.3.6}$$

Depending on the number of sensitive functions found, a signature for a known profile S is a set of m patterns P, where each pattern is a set of i 2-gram opcodes, denoted as *os*. This forms the structural features for the familial malware.

Our technique allows similarity to be measured from the basic unit of code upwards. Similarity between apps becomes an aggregate of individual functional similarities and as such the likelihood of determining relationships between two related codes increases.

We know that malware variants will not be exact copies of one another, but our assumption is that most of their malicious functionality and code structure remains similar. Malware can add, remove, or substitute code within a function. However, for it to retain vital key behaviors, some part of the code has to be consistent across variants. Thus by carefully analyzing each function as a single unit and normalizing its opcodes, our algorithm can ascertain whether a relationship exists between two functions of different applications.

3.3.3 Similarity Matching

Our similarity matching is a 3-step process that begins with Pattern-level similarity, then Function-level similarity to determine a score for all the matched functions. Lastly, the Final-similarity index scores similarity as a function of the Function-level similarity and permissions overlap.

Pattern-level similarity Given a reference piece of malware A with signature S_A and sample application B with signature S_B:

$$\begin{aligned} S_A &= \{P^1, P^2, \ldots, P^m\} \\ P^m &= \{os_1^m, os_2^m, \ldots, os_i^m\} \\ S_B &= \{P^1, P^2, \ldots, P^n\} \\ P^n &= \{os_1^n, os_2^n, \ldots, os_j^n\} \end{aligned} \tag{3.3.7}$$

For each P^m in S_A , we determine its best match in S_B using the **Targeted Overlap Coefficient** (TOC) technique[75]. TOC is derived from the Overlap Coefficient or Szymkiewicz-Simpson coefficient, which is defined as the ratio of the size of the intersection of two sets to the size of a *target* set.

In pattern-level similarity, the TOC measures the ratio of common 2-gram

opcodes found in the intersection of P^n and P^m to the size of P^m, given P^m as the target set. The result indicates the power of inclusion of P^m in P^n. The *TOC*, denoted by $R(P^n, P^m)$ is:

$$R(P^m, P^n) = |P^n \cap P^m|/|P^m| \tag{3.3.8}$$

Since our algorithm specifically leverages finding the relationship between a new pattern and a known target pattern, the TOC is our preferred similarity metric. To determine how P^m relates to P^n, we require a threshold value T define as *pattern-level threshold* (PLT). This denotes the minimum acceptable similarity ratio. In this step, If $R\ (P^m, P^n) \geq T$ then we write R to a buffer BUF and we eliminate both P^m and P^n. While if $R\ (P^m, P^n) < T$ then P^m is eliminated while P^n remains. This loop continues until all the patterns in S_A are compared to patterns in S_B (Algorithm 1).

The pattern-level similarity is measured by the value of R, which lies between 0 and 1. As R approaches 1, it means most 2-gram opcodes found in P^m are also present in P^n, hence P^m is similar to P^n. However as R approaches 0, the similarity between P^m and P^n diminishes. The similarity score calculated is not transitive–P^m is a pattern from our known profiles while P^n is a pattern in a test sample and the idea is to calculate how close P^m is to P^n and not vice versa. A positive outcome at this level of matching can be attributed to one of the following reasons. If P^m and P^n are modules with the same functionality then R will be close to 1. It is also possible P^n is a disguised version of P^m that has been obfuscated but still retains most of its original opcodes. In this situation, we can also derive a match. It is also possible for P^m and P^n to match to a certain degree even though neither is derived from same functional module, which will create a false positive. Our evaluation results have shown this is quite rare in practice.

This pattern level-matching algorithm has proven effective in overcoming the

effect of junk insertion and reordering obfuscation techniques.

Function-level similarity This step analyzes all the results generated in pattern-level matching. As shown above, for each matched pattern, the derived coefficient is stored in a buffer *BUF*.

$$BUF = \{R_1, R_2, \ldots, R_k\} \tag{3.3.9}$$

The set of ratios represents all the matched functions between S_A and S_B. Function-level matching calculates a score between two samples as an aggregate of their pattern-level matching. As preliminary testing, we tried 4 different similarity coefficients (Cosine, Jaccard, Edit Distance and Sorensen (Dice) Coefficient) on sample sets and measured the results. The *Dice Coefficient* gave us the best result. Briefly, the Dice coefficient is a measure of the intersection between two given sets scaled by their size. Although the choice of Dice Coefficient here is basically empirical, in general, it gives more weight when there is an intersection than the Jaccard thus strengthening similarity. The Dice Coefficient D is defined as follows:

$$D(S_A,\ S_B) = 2 * |S_A \cap S_b| / |S_A + S_B| \tag{3.3.10}$$

A value of T (pattern-level threshold) that is close to 1 means each R in BUF is also close to 1. Thus:

$$|S_A \cap S_B| = \sum BUF \tag{3.3.11}$$

And therefore:

$$D = 2 * (\sum BUF) \ / \ |S_A + S_B| \tag{3.3.12}$$

The coefficient D (Algorithm 2) denotes structural similarity between extracted functions found in two applications.

Final-similarity index The final similarity score is calculated based on the result of function-level matching and permissions overlap. We first need to calculate the permissions overlap using the Targeted Overlap Coefficient:

$$\begin{aligned} permA &= permission\ list\ in\ A \\ permB &= permission\ list\ in\ B \end{aligned} \tag{3.3.13}$$

Thus the permission overlap from A to B, called PO, is given as:

$$PO\ (A,B) = |permA \cap PermB|/|permA| \tag{3.3.14}$$

This gives the ratio of similar permissions found in A and B against the length of set of permissions for A . The permissions overlap is a weight that strengthens the result of the function-level matching and indicates (at a coarse-grained level) what behavior is common between A and B . If two apps contain the same malware footprint, they normally should have some common permissions. The overlap coefficient is a value between 0 and 1.

The final similarity index given as SS and is a function of function-level similarity D and permissions overlap PO. This is defined as:

$$SS = D\ *\ PO \tag{3.3.15}$$

The value of SS is a coefficient that indicates the strength of similarity between two applications based on our extracted features. A minimum similarity index MSI is required to determine if the coefficient SS is good enough. Thus when $SS \geq MSI$, we report that a known footprint for malicious code is present in the target app.

The summary of all the design notation is shown in Table 3.1.

Algorithm 1 Pattern-Level Matching

function :(PLM(S_A, S_B, T))
  for P^m in S_A **do**:
    for P^n in S_B **do**
      inter=multi_intersect(P^m, P^n)
      coef = len(inter) / len(P^m)
      if $coef \geq T$ **then**:
        append(coef, BUF)
        remove(P^n, S_B)
      end if
    end for
  end for
  return *BUF*
end function

Algorithm 2 Function-Level Matching - Dice coefficient

function :(FLM(S_A, S_B, BUF))
  suM=sum(BUF)
  funAvg= 2/ (len(S_A)+len(S_B)
  *D = funcAvg * suM*
  return *D*
end function

Algorithm 3 Final-Similarity Index

function :(FSI($perm_A$, $perm_B$, D))
  inter= multi_intersect ($perm_A$, $perm_B$)
  perm = inter/len($perm_A$)
  if perm > 0 **then**
    *SS=perm*D*
  end if
  return *SS*
end function

Table 3.1: Design Notation Table

Notation	Definition
cd	classes.dex
jc	Java Class
f	Java Function
o	Instruction opcode
S	Signature
P	Pattern from a Signature
$TOC - R(P^m, P^n)$	Targeted Overlap
BUF	Set of Targeted overlap Coefficients
T	Threshold
$D(S_A, S_B)$	Dice Cofficient
$PermA$	Permission list from sample A
$P.O(A, B)$	Targeted Permission Overlap
SS	Final Similarity Index
MSI	Minimum Similarity Index
PLT	Pattern Level Threshold

3.4 Evaluation

We have implemented a prototype of *OpSeq* in Python to test its efficacy on obfuscation techniques typically employed by Android malware. This section presents our empirical results.

3.4.1 Focus of the Evaluation

The focus of the evaluation is twofold: 1) detection of known malware, and 2) further categorization of detected malware into their respective malware families. In particular, the evaluation targets the following two research questions:

1. **Malware/Benign apps detection (Mal/Ben):**

   Can *OpSeq* accurately detect a variant of repackaged malicious code without confusion with benign applications?

   - True Positive (TP): known malware code is correctly detected.

- False Positive (FP): benign code wrongly detected as containing malware code.
- True Negative (TN): benign apps are not flagged as having malware code.
- False Negative (FN): known malware code is not detected.

2. **Malware class detection (`Mal_Class`):**

   Can *OpSeq* categorize a known repackaged malware code into its respective family?

   - True Positive (TP): all malware code that are correctly categorized.
   - False Positive (FP): all malware code that are wrongly categorized as variants of different family.
   - False Negative (FN): all malware code that were not detected.
   - True Negative (TN): true negative is eliminated in this test because all the malware samples belong to at least one known class. The samples used in this test were derived from the true positives in Mal/Ben above.

Experimental Dataset: We use two datasets for experiments. The first dataset has 1,551 Android applications consisting of 359 benign applications downloaded from *Google Play*, and 1,192 malware samples (of 25 families) from the well-known *Android Genome project* [92]. The second dataset has 207 malware samples employing four obfuscation techniques: reflection, encryption, code reordering and junk insertion. DroidChameleon [63], and SandMark [45] are used to generate the samples. Specifically, malware sample distribution of DroidChameleon and SandMark are 167 variants (of 25 malware classes), and 40 variants (of 20 malware classes) respectively. If only one obfuscation technique is used by a malware, we refer it as *simple obfuscation*, otherwise, it is referred as *complex obfuscation*. Table 3.2 presents the distribution

Table 3.2: Malware sample distibution as per four obfuscation techniques: reflection, encryption, code reordering, and junk insertion

Obfuscation Type	Obfuscation Techniques	Number of Malware Samples
	DroidChameleon	
Simple Obfuscation	Encryption (E)	24
	Reflection (R)	25
	Reordering (O)	24
	Junk (J)	24
Complex Obfuscation	E & R	23
	E & R & O	24
	E & R & J	23
	SandMark	
	Transparent Branches	20
	Random DeadCode	20
Total		207

of malware in the dataset in accordance with obfuscation tool and type.

3.4.2 Optimum Variables

As mentioned in the previous section, we chose to slice our normalized sequences using 2-gram sub-sequences, based on results from empirical studies. The average accuracy of our approach tested with unigram, 2-gram and 3-gram was 91%, 98.9% and 96% respectively.

Furthermore, we have identified two variables necessary for our algorithm: 1) Pattern-level threshold (PLT) - the minimum overlap ratio required to assume similarity between two 2-gram patterns from different apps (in Pattern-level similarity), and 2) Minimum similarity index (MSI) - the minimum score that determines if a malware footprint is present in an application (in Final-similarity index). To get the optimal values of PLT and MSI, we choose arbitrary values and plugged them into our algorithm. The chosen values for PLT are 70, 80 and 90, all expressed as a percentage. MSI values are 3, 4, 5, 6, 7, again expressed as a percentage. Our

variables are chosen based on the statistics of Mal/Ben detection.

In choosing the optimal values, we use F-Score statistics. The F-Score measures the overall accuracy of a test, which depends on precision and recall. Recall is the measure of accuracy that a specific class has been detected (% of correct malware families detected out of all malware samples), whereas precision is the percentage of positive prediction (% of all malware detected out of all sample applications).

The combination of PLT and MSI that gives the highest F-score is the optimal solution for the test data. We ran *OpSeq* on our dataset using the above combination of PLT and MSI and the results of our execution is shown in Table 3.3. Each row (a model) represents one combination. We then calculate the statistics (false positive, false negative, true positive, true negative, precision, recall and F-score) for each model. The equation for F-Score statistics is:

$$F = 2 * ((Precision * Recall)/(Precision + Recall)) \tag{3.4.1}$$

The results in Table 3.3 above indicate the highest F-Score is attained with PLT equal to 80% and MSI equal to 3%. This model gives us 99.3% precision, 98.5% recalls and an F-Score of 98.9% for Mal/Ben detection. Furthermore, the false negative rate (given as FN/TP+FN) for this model was ≈1.5%. This indicates our system has a very small probability of "miss" detection for known malicious code. The false positive rate for the same model is ≈2.2%.

Using the same metrics on Mal/Class, our F-Score accuracy was 97.5%, 98% recall and 97% precision. Thus overall, our system is capable of accurately detecting malware 98.9% of the time and can categorize the malware into its correct family with 97.5% accuracy. The confusion matrix for Mal/Ben and Mal/Class based on the optimal values is shown in Table 3.4 and Table 3.5, respectively.

Table 3.3: Model Selection Result: F-Score, Precision, Recall

Model	False Positive	False Negative	True Positive	True Negative	Precision	Recall	Interpolated Precision	F-Score
T_90M_7	0	68	1124	359	1	0.943	1	0.971
T_90M_6	1	63	1129	358	0.999	0.947	1	0.973
T_80M_7	0	63	1129	359	1	0.947	1	0.973
T_90M_5	2	55	1137	357	0.998	0.954	0.998	0.976
T_80M_6	2	54	1138	357	0.998	0.955	0.998	0.976
T_90M_4	2	44	1148	357	0.998	0.963	0.998	0.98
T_80M_5	2	42	1150	357	0.998	0.965	0.998	0.981
T_70M_6	25	31	1161	334	0.98	0.97	0.997	0.983
T_70M_7	10	35	1157	349	0.99	0.97	0.997	0.983
T_80M_4	5	34	1158	354	0.996	0.971	0.997	0.984
T_90M_3	4	34	1158	355	0.997	0.971	0.997	0.984
T_70M_4	108	22	1170	251	0.92	0.98	0.993	0.986
T_70M_5	52	27	1165	307	0.96	0.98	0.993	0.986
T_80M_3	8	18	1174	351	0.993	0.985	0.993	0.989
T_70M_3	192	15	1177	167	0.86	0.987	0.971	0.979

The rationale for choosing our optimal values is to further buttress the precision/recall curve of our solution as shown in Figure 3.2. With each point representing one model, the points on the curve where precision and recall are around their maximum and are nearly equal indicate the point of maximum accuracy (interpolated precision is used to smooth the PR curve).

3.4.3 Empirical Results

Accuracy: Given the F-Score for Mal/Ben, our system can detect malware footprints in an app with 98.9% accuracy. However due to some reasons identified below, the Mal/Class test (i.e., identifying the class category) is slightly lower (97.5%). In comparison with some recent known malware detection tools, *OpSeq*'s Mal/Class accuracy (97.5%) did better than Apposcopy [39] with 90% accuracy. On the other hand DroidLegacy [30] recorded a membership test accuracy of 98%, slightly higher than *OpSeq*'s Mal/Class figure of 97.5%.

Table 3.4: Mal/Ben Best Model Confusion Matrix

	(Positive) Malware	(Negative) Benign	Total
(Positive) Malware	1174	8	1182
(Negative) Benign	18	351	369
Total	1192	359	

Table 3.5: Percentage of Mal/Class Prediction Result

Malware Family	False Negative	False Positive	True Positive	Total
ADRD	0	0	22	22
Anserverbot	7	12	286	305
BeanBot	0	0	8	8
Bgserv	0	0	9	9
CruseWin	0	0	2	2
DroidDream	1	1	14	16
DroidDreamLight	4	0	43	47
DroidKungFu	5	19	418	442
Geinimi	0	0	69	69
GingerMaster	0	0	4	4
GoldDream	0	0	47	47
Gone60	0	0	9	9
GPSSMSSpy	0	0	6	6
HippoSMS	0	0	4	4
jSMSHider	0	0	13	13
KMin	0	0	52	52
Pjapps	1	2	55	58
Plankton	0	1	10	11
RogueLemon	0	0	2	2
RogueSPPush	0	0	9	9
SndApps	0	0	10	10
Tapsnake	0	0	2	2
YZHC	0	0	22	22
zHash	0	0	11	11
Zsone	0	0	12	12
Total	18	35	1139	1192

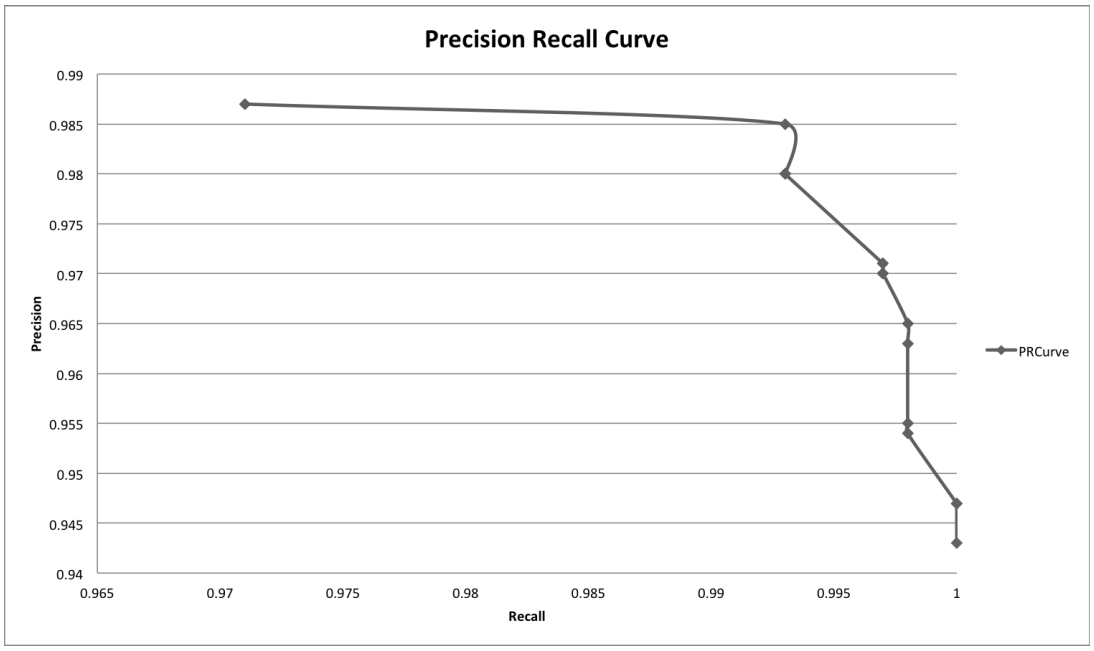

Figure 3.2: Precision & Recall Curve

However, comparing DroidLegacy's recall rate of 94%, which is the true positive rate, versus our figure of 98%, indicates our system is capable of better detection. Furthermore, our system has better coverage in terms of malware families processed (they processed 11 families against ours with 25 families) and resiliency to different obfuscators as shown in the next subsection.

False Negative: Eighteen (18) malware were incorrectly categorized as benign apps. Samples from DroidKungFu, DroidDreamLight and Anserverbot constitute the bulk of our false negative predictions as shown in Table 3.5. One reason for the false negatives can be traced to our signature generation process. This process randomly picks only one sample from a set to create a class signature; it is possible that the sample might be the oldest, newest, or even a variant that has more inclusion or exclusion of instructions within the malicious code. For instance, the sample we use to generate the signature for DroidDreamLight is a newer version than the rest of its variants. This sample has about 98 functions that were extracted for the signature

against 18 for the older version. Thus, since our similarity calculates overlap based on the target known profile, this sample was a miss.

Another reason for some false negatives can be attributed to native code exploits. Malware samples that have most of their malicious code written in native code will often result in a miss detection (e.g., some few variants of DroidKungFu and Anserverbot). Currently, *OpSeq* is only designed to handle the dex file (containing Java bytecode) and handling native code is the subject of ongoing research.

False Positives: For Mal/Ben, 8 applications out of 359 downloaded from Google Play were detected as malware by *OpSeq*, as shown in Table 3.4. In order to confirm the true nature of these 8 applications, we ran them through VirusTotal [7]. The output flagged 3 out the 8 apps as malware, reducing our true false positives to only 5 apps.

In Mal/Class, the false positives recorded were largely due to the use of common code snippets. This code ranges from 3rd party libraries to adware. Most of the malware families originate from the same location and have common targets, so it is not uncommon to find similar libraries and/or adware packaged within the applications. For instance, we analyzed one of the Anserverbot samples which *OpSeq* miscategorized as DroidKungFu. This miscategorization occurs as a result of the presence of an adware library called - Adware.waps in the new Anserverbot sample which was missing in the old version used to generate the familial signature. However, the adware on its own collects various user data from a device and invokes some sensitive APIs, like getRunningTask. Thus our system retrieved more features than were available for its family's signature. On the other hand, the DroidKungFu sample for signature generation contains the same adware. Since *OpSeq* assigned the malware class based on the strength of the final similarity index, that Anserverbot sample was flagged as DroidKungFu.

3.4.4 Evaluating Resiliency

Our approach can detect known malicious code obfuscated and repackaged within another app. In other to measure its resiliency, we employ two different open source obfuscators: *DroidChameleon* - newer and specifically designed to obfuscate Android apps and *SandMark* - a well known and tested Java obfuscator. In both systems, we gave special emphasis to those modules/algorithms that can severely hinder opcode-sequence based detection.

Test against DroidChameleon Using DroidChameleon, we generated 167 obfuscated variants of the malware in our sample set with varying degrees of transformations, ranging from simple (one method) to complex (combination of obfuscators). In this segment of evaluation we choose 4 common obfuscation techniques employed by Android malware:

1. `Encryption`: names, strings and field encryption
2. `Reflection`: permission =-based API calls are changed from direct invocation to using reflection API in helper classes
3. `Junk Insertion`: addition of "noise" instructions
4. `Code Reordering`: altering the flow of program execution by changing the positions of unrelated instructions

In simple obfuscation, an application is transformed with just one technique e.g., *getDeviceId* is transformed from direct call to invoking a helper class that calls *class.getMethod()* and then *method.invoke()*. In complex obfuscations applications get obfuscated with two or three techniques e.g., all strings, fields and names are encrypted and then all permission-based API calls are invoked using the reflection API.

For simple obfuscation, we introduced encryption into 24 samples, 25 samples had added reflection, junk instructions were added in 24 samples, and code reordering in 24 samples. For complex obfuscation our samples were transformed with a minimum of two different techniques. We modified DroidChameleon such that when an app is unpackaged, it is first encrypted, then the bytecode is run through more obfuscator modules before repackaging. We successfully repackaged 23 samples with encryption and reflection, 24 had (encryption, reflection and reordering) combined and finally 23 had (encryption, reflection and junk) combined. Our results for simple and complex obfuscation as shown in Table 3.6 and Table 3.7, respectively. *OpSeq* scores 100% average detection rate in simple obfuscation and 88% for complex transformations.

AntiVirus Results Using the same obfuscated samples mentioned above, we assessed the detection ratio of other antivirus products using the VirusTotal website. For simple obfuscation, out of the top 14 antivirus products, AVG recorded the highest detection rate with average detection of 65.83%, followed by Dr. Web, F-Secure, Kaspersky, AhnLab-V3 with slightly above 50%, and Panda, with the lowest detection rate of 4%. In the complex obfuscation cases, AhnLab-V3 led other antivirus products with a detection rate of 49.33%, then AVG with 27%. All the rest recorded less than 25%.

DroidLegacy Results We also used the same obfuscated variants described above to evaluate the performance of DroidLegacy. We set the optimum threshold of 0.7 as specified in their paper. For simple obfuscation, DroidLegacy had an average detection rate of 37% and 0% for complex obfuscation. Like many common malware detection tools, DroidLegacy depends on API names to create signatures for malware. In situations where the name is obfuscated using reflection, chances of detection

Table 3.6: Evaluation results for DroidChameleon's simple obfuscation.

	Encryption	**Reflection**	**Reordering**	**Junk**	**Average Detection Rate**
Total No. of Sample	24	25	24	24	
Research Tools					
OpSeq	24	25	24	24	100
DroidLegacy	6	0	15	15	37.5
AntiVirus Software					
AVG	8	20	18	18	65.98
DrWeb	17	5	17	17	57.73
F-Secure	6	16	17	16	56.7
Kaspersky	6	16	16	15	54.64
AhnLab-V3	14	14	10	12	51.55
Avast	6	15	13	14	49.48
Avira	5	7	12	12	37.11
Symantec	4	12	7	11	35.05
Ad-Aware	6	6	7	7	26.8
BitDefender	6	6	7	7	26.8
AVware	5	6	6	6	23.71
McAfee	5	5	5	5	20.62
Panda	2	2	2	2	8.25
Baidu-International	1	1	1	1	4.12

Table 3.7: Evaluation results for DroidChameleon's complex obfuscation.

	Encryption & Reflection	**Encryption Reflection & Reordering**	**Encryption Reflection & Junk**	**Average Detection Rate**
Total No. of Sample	23	24	23	
Research Tools				
OpSeq	23	24	15	88.57
DroidLegacy	0	0	0	0
AntiVirus Software				
AhnLab-V3	12	13	12	49.33
AVG	6	7	6	27.14
Kaspersky	6	6	4	22.86
Ad-Aware	6	7	3	22.86
BitDefender	6	7	3	22.86
F-Secure	6	7	3	21.33
Avast	6	6	3	21.43
Avira	5	5	3	18.57
AVware	5	5	3	18.57
DrWeb	5	5	3	18.57
McAfee	5	5	3	18.57
Symantec	4	4	2	14.29
Panda	2	1	2	7.14
Baidu-International	1	2	2	7.14

become very low. This explains why it recorded 0% for all apps that were repackaged using reflective method invocation. Furthermore, it performed poorly with encryption alone, detecting only 6 out of 24 encrypted apps.

Tests against Sandmark To further evaluate the resilience of *OpSeq* against major obfuscation techniques, we tested it against SandMark. SandMark is well known and highly documented tool used for watermarking, tamper-proofing and code obfuscation of Java bytecode [45]. For the purpose of our analysis, we used only some specific modules within *Obfuscation Executive* to transform the application bytecode. We generated 40 new variants by inserting: *Transparent Branches and Random DeadCode*. The resulting new jar files were repackaged, aligned and signed before analysis.

In this experiment, we tested *OpSeq*'s similarity detection capability by performing a one to one comparison between each of the 40 repackaged malware, and its original sample. The results indicates *OpSeq* can detect these obfuscations with 100% accuracy.

3.4.5 Measure of Performance

Our experimental system an Ubuntu Linux 64-bit system running on an Intel Xeon CPU at 2.5 GHz, with 16 GB RAM. We leverage an open source Android reverse engineering tool called *apktool* for the conversion of the Android dex file into an intermediate bytecode representation, called *smali*. Given a target application with 205 Java functions (28624 lines of Dalvik bytecode), for which 46 of these functions invokes one or more of the sensitive APIs, it takes an average of 4.5 seconds on our test machine to perform a one-to-one similarity matching with a known sample that has 118 Java methods (19307 lines of Dalvik instructions).

Furthermore, for detection purposes, it takes an average of 11.6 seconds to analyze a target app against known profiles of the 25 malware families (classification). This results outperforms Apposcopy, which took an average of 346 seconds per analysis of an app with 26786 lines of Dalvik bytecode.

The fact that *OpSeq* is designed to fingerprint apps based on small structured signatures extracted from vital points within the program code base helps to eliminate unnecessary noise in the matching process and thus improves our system's overall performance.

3.5 Discussion

The experiments in the previous section illustrate that for a large class of Android malware, *OpSeq* significantly outperforms state of the art commercial and research products that rely on signature-based detection algorithms. Our tests indicate that *OpSeq* is effective in detecting malware in the presence of both simple and complex obfuscation techniques, many of which compromise the accuracy of existing detection techniques.

In the malware families we analyzed, the largest stored signature has 118 patterns (slices) to be matched while the smallest has 3 patterns. Theoretically, if we compare *OpSeq* with other opcode-sequence based detection like [90, 44] which slice the entire app's opcode-sequence into n slices, our app's signature sizes are guaranteed to be smaller and therefore much more efficient. Furthermore, our average processing time of 11.6 seconds per 25 family comparison indicates our algorithm is fairly scalable.

During the course of our analysis, our findings reveal extensive code reuse amongst some of the malware families. For instance, Zhou et al has categorized DroidKungFu into 5 major families [92]. However we found malware from these classes

to contain a considerable amount of common code segments. Thus we categorize them as one class.

Also, Anserverbot and Basebridge were found to contain a similar main package com.keji.danti. They differ slightly where BaseBridge loads an extra payload that leads to privilege escalation while some Anserverbot variants do not. But since *OpSeq* only processes the dex file, our analyses flag one as a variant of the other. Information from Foresafe encyclopedia [8] and analysis results of some antivirus products in VirusTotal also affirm their relationship; hence we merge them into one class. Some common adware and external libraries were also found to be present in malware of different families. Such instances have increased the rate of our false positives and affect the overall accuracy for `Mal_Class` detection, but these issues also raise indicate that *OpSeq* has significant value in better malware classification as well as detection.

3.6 System Limitations

Like most malware detection schemes based on static analysis, *OpSeq* can be thwarted via whole class encryption and extensive dynamic class loading. This is because *OpSeq* only extracts features from the available classes.dex file. Extra classes that are fully encrypted or loaded at runtime cannot be processed.

Code-reordering that can split functions into multiple sub-functions may also negatively impact our approach. Also, very large numbers of junk instructions can introduce so much noise that it may affect the quality of our signatures. However these obfuscation techniques will only be problematic when more than one *sensitive functional module* is tampered with, which in practice will require significant human intervention. Furthermore, since our approach clusters common opcodes together by normalizing them before we slice the whole sequence into 2-gram patterns, excessive noise can only affect our signature when unique opcodes are introduced viz-a-viz

the normalization pattern. These unique opcodes must vary significantly from those normally used within the functions.

Finally, like all static fingerprinting algorithms, *OpSeq* is designed based on signature of a known sample to detect its variant. This system cannot analyze unknown samples. Thus in the next two chapters we will discuss a further contribution of this thesis, involving design of a hybrid analysis system that will augment some of the limitations of *OpSeq*.

Chapter 4

Instrumentation

Instrumentation is the process of analyzing programs by adding trace code to their source code, binary code, or execution environment. This provides mechanisms for an analyst to define concerns related to program verification, enforcement, monitoring, error-checking, performance, debugging, or tracing. Instrumentation techniques do not necessarily modify code but rather tamper with the execution or behavior based on defined constructs. In recent years, instrumentation techniques have gained momentum in the cybersecurity community for vulnerability [83], malware [15, 31, 32, 35, 51, 62] and privacy analysis [18, 19, 50, 84].

Aspect oriented programming (AOP), first introduced by [55], is a modularized programming model allowing the separation of cross cutting concerns [73], which are difficult to capture in traditional programming models. AOP encapsulates the concerns, defined as `aspects`, by instrumenting extra behavior in the existing code. These aspects are special constructs forming the building blocks of AOP. Their designs can be generic, which allows for reuse throughout program execution. Implementation of AOP can be performed in two distinct ways:

1. Static instrumentation allows for code to be injected at compile time. This technique merges both the aspects and the original code into one binary, which then executes in the execution environment of the original code.

2. Dynamic instrumentation, on the other hand, injects code at runtime. In most instances, this requires a custom classloader to enable the interpreter to understand and implement the AOP features.

In 2001 [54], PARC developed an extension for AOP designed specifically for the Java programming language, called AspectJ. Its Java-like syntax, coupled with its ease of use, makes AspectJ a very popular instrumentation tool for Java programs. Aspects in AOP are defined by some key terms:

1. Pointcuts are defined by `kinded` constructs such as function `call`, method `execution`, `within` class, `cflow` etc., which match some specified `signatures` or `modifiers`.

2. Signatures are semantic definitions which can be decoded by the AspectJ compiler during joinpoint creation. It can encapsulate both broad and narrow definitions, giving an analyst ample flexibility.

3. Joinpoints are points within the execution of the program that are interesting and/or defined by the concerns of an analyst. These are chosen based on constructs defined in a pointcut.

4. Advice is the piece of code that gets executed when a certain joinpoint is reached during program execution

In addressing security concerns, advice defined for a joinpoint adds some functionalities such as logging, code injection, value manipulations, execution rerouting, skipping execution, etc. to an instrumented program. Advice to be executed can target `before, after` and `around` the execution of a particular joinpoint. As the name implies, execution of `before` advice precedes the execution of the target joinpoint. In this advice, parameters and the target object can be retrieved, in

addition to signatures, source location, etc. For `after` advice, in addition to the information extracted in before advice, the return value can also be retrieved and evaluated. The most interesting is `around` advice which, although potentially expensive to use, allows code injection and modification of arguments, variable values, and return values.

The code snippet in Listing 4.1 shows a sample aspect that defines a pointcut which picks a joinpoint at the call to `getDeviceId`. When instrumented, this aspect will pick the method `getDeviceId` from the class `TelephonyManager`. The joinpoint is picked because the signature matches the method in that class and it is the only class in the Android SDK with such a method. However, if within the application there also exists a library class with such a method, our broad signature will automatically capture such a joinpoint, too.

Listing 4.1: Simple Aspect

```
public aspect Logger{
  pointcut myId():
     call(* *..*.getDeviceId());
  after()returning(String id): myId(){
     log.v("AspectJ", "DeviceId="+id);
  }
}
```

4.0.1 Bytecode Weaving

In the Java compilation process, an intermediate representation called `bytecode` is generated when the original source code is compiled. This bytecode is contained in .class files representing each source class. More specific to Android, the system has added another level of abstraction to its compilation process, where the class files

are further compressed into one dex file. Bytecode weavers are tasked with weaving together class files (both Java classes and aspect classes). In this research, our chosen bytecode weaver is AspectJ [55]. Its robust framework allows us to define and inject security concerns related to Android apps for the purpose of logging and monitoring. Furthermore, its programming syntax and semantics are identical to that of the Java language, allowing us to tie and weave the monitoring code into a target Android application with better precision.

AspectJ compilers (ajc) can accept both raw sources and class files for compile-time weaving and thus have the capability to compile and weave the aspect/Java sources and/or class files to produce a new woven class. The resulting merged Java bytecode has to be compatible with the execution platform's VM. However, in load-time weaving AspectJ exposes an interface that facilitates the weaving process between the target bytecode and a custom `classloader`.

The requirements of our system involve analyzing unknown binaries where there is no available source code. Thus we limit the discussions in this thesis to only compile-time bytecode weaving. This form of static instrumentation takes the advice defined in an aspect and weaves them at specified jointpoints as ilustrated in Figure 4.1. For Android apps, the resulting binary is dexed and re-packaged into a new apk. Since this new application does not need a custom classloader, it has the flexibility of executing on any device emulator without changes to the underlying OS.

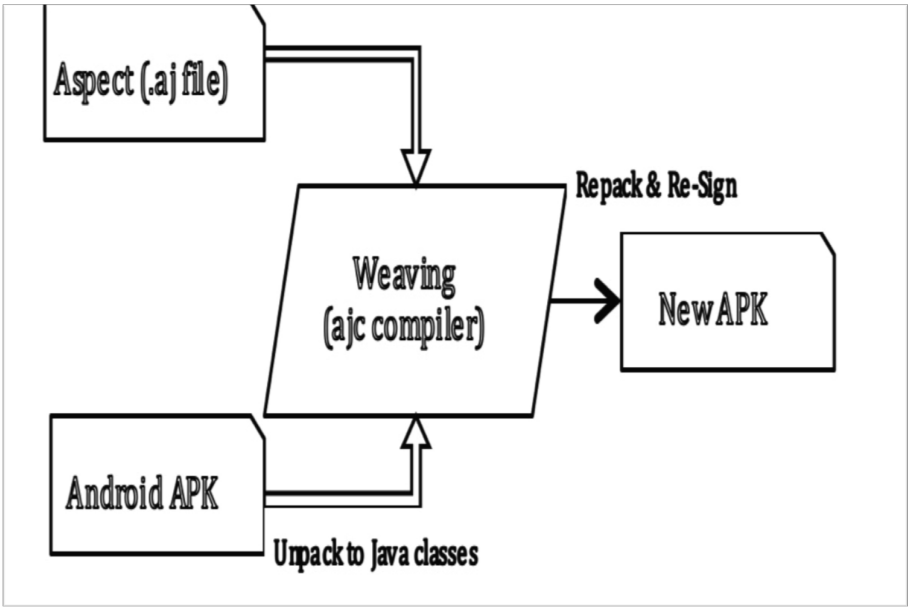

Figure 4.1: *OpSeq* Signature Generation Workflow

Chapter 5

Hybrid Analysis

5.1 AspectDroid

As mentioned in the previous chapter, static analysis involves detecting known malicious applications using predetermined signatures and other semantic artifacts, while dynamic analysis understands program behavior by executing it in a contained environment. As malicious apps evolve over time, signature-based static analysis techniques alone are not sufficient to detect stealthy obfuscated variants and/or new malware samples. Hence, we explored the concept of bytecode weaving to develop a better analysis system.

Most comprehensive dynamic analysis techniques either require instrumentation of the underlaying operating system code [32, 62, 31] or involve virtual machine introspection [82]. They provide effective sandboxing for the analysis of the target applications, but unfortunately, such techniques are heavily dependent on OS versions and the Android runtime. Porting and flashing a new build on real devices for various versions of Android is not an easy task, which can limit the number and kind of applications that can be analyzed. Existing application-level techniques like [15, 18, 19, 35, 51] are mostly constrained to performing only API monitoring. Although systems like Capper[84, 83] can perform app-level taint analysis, the heavy reliance on static analysis for the extraction of taint slices makes it equally vulnerable to

simple obfuscation.

In this chapter, we present *AspectDroid*, a hybrid analysis system for Android applications based on the AspectJ instrumentation framework. *AspectDroid* performs static bytecode instrumentation at the application level, and does not require any particular support from the operating system or the Dalvik virtual machine. It weaves in monitoring code at compile time using a set of predefined security concerns, such as data/resource abuse and other non-traditional behaviors like reflective calls and native code execution. The target application is then executed on any Android platform of choice for which behavioral patterns are monitored and logged dynamically.

5.2 Related Work

5.2.1 Application-level instrumentation

The first application-level dynamic taint tracking on Android was developed by [84, 83]. Their system (Capper) is designed to monitor exfiltration of sensitive data from source to sink. However, their work requires significant static analysis to refactor the Java bytecode and compute taint slices which are used at runtime as the taint propagation map. This system is prone to most of the inaccuracies of static taint tracking that can result from simple obfuscation techniques. More so, data sources, propagation, and sinks that pass through reflective call invocation are not processed if the invoked class and method names cannot be statically resolved. And finally, like most app-level analysis systems, Capper does not handle dynamic class instrumentation. Our system *AspectDroid* on the other hand can perform better data flow analysis since it can handle Java reflection and runtime class instrumentation.

Other research efforts that exist which target app-level instrumentation are [15, 18, 19, 35, 51]. The authors of Droidox developed APImonitor[15] to counter

its numerous porting issues. ApiMonitor like [18], [19], RV-Droid [35] and [51] all use static bytecode instrumentation to analyze method calls in target applications at runtime. Although they use different instrumentation frameworks, these systems are all limited to sensitive API monitoring during program execution. In contrast, *AspectDroid* is a complete analysis system that targets security concerns such as data flow analysis, sensitive API monitoring, as well as analytics of suspicious behaviors.

5.2.2 Low-Level Instrumentation

Most Android dynamic analysis tools are developed by instrumenting the operating system code and/or the underlying framework. TaintDroid [32] is a real time dynamic taint tracking system that monitors the flow of `sensitive` data. It uses some basic data flow rules to track the movement of tainted variables, method files and IPC messages from `sources` until they reach a specified Java library `sink`.

Several extensions to TaintDroid [31, 62, 78] were built with added functionalities. DroidBox [31], for instance, logs an app's activities related to starting services, broadcast receivers, SMS and calls made, cryptography operations performed using the Android API, and file read/write operations, irrespective of taint marking. Andrubis [78] is an automated analysis system that combines both static and dynamic approaches to an app's analysis. Applications submitted via an online link are dynamically examined on a QEMU-based emulation environment for method tracking, system level analysis, and data exfiltration using TaintDroid. Other systems like AppsPlayground [62] added more functionality such as kernel level-monitoring and automated testing to TaintDroid. The critical design rule for these approaches relies on low-level instrumentation, thus making them very OS version-dependent and in some cases platform-dependent. It is important to note that TaintDroid based systems depend fully on the Dalvik virtual machine and as such will require a complete

make-over to port to the new Android runtime. More so, stealthy malware can often detect emulation environments which may result in inaccurate analysis. Lastly, due to significant requirements for expert knowledge to port from version to version, the capacities of such systems for long term analysis is very limited.

Other host-based dynamic analysis tools are DroidScope, AASandbox and Crowdroid. DroidScope [82] uses virtual machine introspection to monitor the activity of untrusted applications. This system performs API tracing, native instruction and Dalvik tracing, and taint tracking. AASanbox [22], on the other hand, evaluates system call logs by placing hooks between kernel space and user space. These hooks hijack the system calls made and log information such as process ID, syscall name and execution time. CrowDroid [26] analyzes system calls performed by an application based on logs collected using the strace debugging utility in a lightweight CrowdClient. This system is limited to extracting only Linux-specific information like open files, but cannot give broad information on IPC and Android specific data.

Dynamic binary instrumentation (DBI) systems like DynamicRIO [24], PIN [57], Spike [76], and Dyninst [25] which performs runtime monitoring, are mostly dependent on low level system C instructions. Furthermore, with the exception of PIN, the remaining are not applicable to ARM systems. DBI techniques are also very dependent on the underlying hardware architecture and as such will require modification of the operating system in the case of Android app analysis.

5.3 System Design

AspectDroid is a hybrid system that uses static instrumentation to inject monitoring code into the target app based on some specific cross-cutting concerns. A requirement of our system is the ability to analyze unknown binaries where there is no available source code. The core of *AspectDroid* is built based on compile-time bytecode

weaving. This form of static instrumentation takes the advice defined in an aspect and weaves them at specified joinpoints in a target class file. For Android apps, the resulting binary is `dexed` and re-packaged into a new apk. Since this new application does not need a custom `classloader`, it has the flexibility of executing on any device emulator without changes to the underlying OS.

With *AspectDroid*, the new injected code executes alongside the original code and performs custom logging and other analytical functions. The instrumentation engine (IE) which is the primary component, forms the backbone of *AspectDroid* and is designed to address three objectives:

1. Dataflow Analysis
2. Resource Abuse Tracing
3. Analytics of Suspicious Behavior

Our instrumentation code is encapsulated in an aspect and is tailored for each of the objectives mentioned above. The aspects are weaved into the target app using AspectJ's `ajc` compiler, producing the instrumented version used to perform the analysis. The instrumentation process is done in-vitro on a host machine. After successful re-compilation the target app is then pushed onto the test bed for dynamic execution.

5.3.1 Dataflow Analysis

AspectDroid performs application-level tainting of target data source(s). Our approach is built around the fact that standard Java and Android libraries use specific method naming conventions to express common types of operations. Thus, we utilize the consistent use of specific verbs, such as `read, open, write, put, connect`, and `execute`, to define broad signatures to capture actions such as file/stream/network

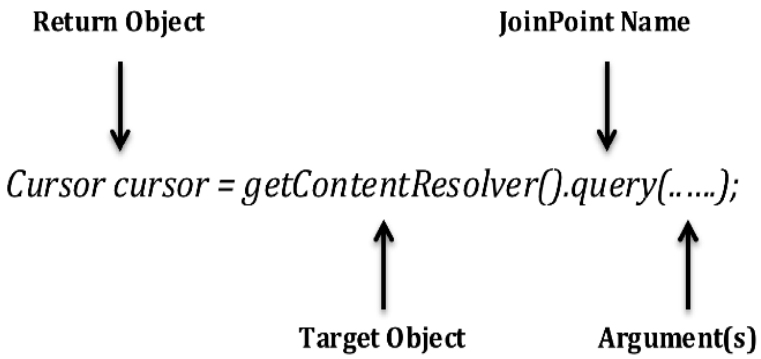

Figure 5.1: Parts of a Method Joinpoint

access. More specific signatures, such as `getLongitude` are used to define narrower joinpoints. Based on all the signatures, we define pointcuts to select various `source, sink, propagation` joinpoints. With the help of AspectJ APIs, a joinpoint's data, such as the `target object`, `parameters`, `return values`, etc. (as shown in Figure 5.1) can be extracted at runtime. Java programming semantics categorize data types as primitive, object, and arrays. Although beyond the scope of this thesis, it is important to note that the JVM stores and processes these data types very differently. Therefore, our data sources, propagation and sink for each data type are handled differently. To simplify the terminology, we refer to `primitive` data types, such as `string` and `character`, as `low-level data types`.

Our dataflow analysis is limited to explicit propagation, where the tainted data must be in the sink call, as shown in Listing 5.1. On the other hand, Listing 5.2 illustrates an example of an implicit flow which exfiltrates inferred data based on the real tainted data.

Listing 5.1: Explicit Data Flow

```
TelephonyManager telephonyManager = (TelephonyManager)
   getSystemService(Context.TELEPHONY_SERVICE);
String IMEI = telephonyManager.getDeviceId();
```

```
if (!IMEI.equals("00000000"){
   String id = Base64.encodeToString(myString.getBytes(), 0);
   SmsManager sms = SmsManager.getDefault();
   sms.sendTextMessage("5556", null,id, null, null);
   }
```

Listing 5.2: Implicit Data Flow

```
TelephonyManager telephonyManager = (TelephonyManager)
   getSystemService(Context.TELEPHONY_SERVICE);
String IMEI = telephonyManager.getDeviceId();
if (!IMEI.equals("00000000"){
   String val = "Device not emulator";
   SmsManager sms = SmsManager.getDefault();
   sms.sendTextMessage("5556", null,val, null, null);
   }
```

Although, AspectJ API used by our system is not designed to create joinpoints on conditional/branch instructions thus making it hard to capture implicit flow. Nonetheless, in the analysis of Android applications, sensitive data leaving the device is the real threat, not inferred data. As illustrated in Listing 5.1, the real device IMEI was exfiltrated compared to "Device not Emulator" in Listing 5.2.

Taint sources

We are interested in sources that are relevant to the privacy and security of the user. We define `vital` sources as phone-related data, content provider objects, file reads, and user input. In Android, most important data are guarded by permissions and only accessible to the user through specialized Android API calls. Other relevant data not guarded by permissions, such as data read from files and user input from text

boxes, are also accessed via the standard Java/Android APIs. Specialized pointcuts are created using signatures to intercept these vital API calls. After execution, the return value is stored as a key in a `taint map` with a corresponding special tag for each unique source as the value. Depending on the return type, low-level data types are stored in raw form, while every other object is stored in hash form. This storage design is very significant in reducing the overhead associated with checking if tainted data is part of an object. It allows us to check if the object is tainted using its hashcode at propagation or sink joinpoints.

Taint sinks

Taint sinks are defined as points where the target application communicates with an external component, either within the device or the outside world. In our data flow analysis, we seek to monitor only those sinks that form a `possible` exfiltration point for the data sources defined above. The data sinks are broadly categorized as network, e.g., writing to a `Socket`, `URLConnection`, etc.; SMS sends; file writes (both ordinary files and shared preferences); and IPC. We use the same signature semantics to pick the sink joinpoints. We also leverage the `around advice` of such joinpoints to check if its arguments, or target, contain tainted data.

This process is straightforward for parameters. For example; if the sink call is a `sendTextMessage(..)`, the tainted data will be checked against the parameters of this joinpoint. However, for a target object we need to parse it and check the associated fields against the keys in the taint map. For example; if the tainted data is appended to a URL, and then a URLConnection is created from that URL object, which then invokes its `getOutputStream()` method. Our system parses the URLConnection object to get the URL field and compare that against the data in our taint map. Overall, data exfiltration is detected if a tainted piece of data is found either within

the sink joinpoint's parameters/parameter's fields or within the target/target's fields.

Taint Propagation

Knowing data sources and sinks alone cannot accurately determine data exfiltration; we also need to identify the data propagation process as represented by the sequence of variable assignments along the path from source to sink. The tainted data can be part of an object's fields and the object can be manipulated in different ways. For every joinpoint, we determine if it contains tainted data; if so, the appropriate propagation rule is picked based on the respective joinpoint's return type, as enumerated in the 7 point rules below:

1. Rule 1: Joinpoint that returns a low-level data type and contains a tainted argument.
2. Rule 2: Joinpoint that returns a low-level data type and contains a tainted target.
3. Rule 3: Joinpoints that convert a tainted array to other data types.
4. Rule 4: Joinpoints that create an array from other tainted data types.
5. Rule 5: Object constructor joinpoint that contains a tainted argument.
6. Rule 6: All joinpoints with object return type that contains tainted arguments.
7. Rule 7: All joinpoints with object return type that contains a tainted target.

For joinpoints targeting low-level data types, their return values and target object are the same. However, for object joinpoints, the target is always a reference to the location of the object in memory while the return type could be anything. For example, an object's joinpoint could return a `Boolean` indicating success of a method

Table 5.1: Flow rules examples for updating taint/tag map

Rules	joinpoint Example	Taint Data	Taint Tag/Map Update
Rule 1	`Int myInt = System.identityHashCode(val)`	`valueOf(val), tag = DeviceID`	`valueOf(myInt), tag = DeviceID`
Rule 2	`String str1 = myInt.toString()`	`valueOf(myInt), tag = DeviceId`	`valueOf(str1), tag = DeviceID`
Rule 3	`char arr[] = str1.toCharArray()`	`valueOf(str1), tag = DeviceID`	`Hashcode(arr), tag = DeviceID` Elements of arr, tag = DeviceID
Rule 4	`Str str2=Arrays.toString(arr)`	`Hashcode(arr), tag = DeviceID`	`valueOf(str2), tag = DeviceID`
Rule 5	`StringBuilder stb = new StringBuilder(str2)`	`valueOf(str2), tag = DeviceID`	`Hashcode(stb), tag = DeviceID`
Rule 6	`stb.append(val2)`	`Hashcode(stb), tag = DeviceID` `valueOf(val2) tag = LineNum`	`HashCode(stb)` `tag = DeviceID and LineNum`
Rule 7	`Vector vec = new Vector()` `vec.add(str2)`	New empty vector is created `valueOf(str2), tag = DeviceID`	`Hashcode(vec), tag = DeviceID`

call, `void`, a low-level data type, or other objects. This distinction forms the basis of how our taint tag/map is updated after the execution of the joinpoint. Table 5.1 gives a taint propagation example for each of the flow rules, and shows the taint tag/map update after the joinpoint's execution. Propagation rule 7 can create a weaving process that might get out of hand, thus we included some optimizations for joinpoints associated with that rule, based on the object's class.

To optimize the weaving process and reduce the complexity of the instrumentation, the propagation's joinpoints for every source are created along the control flow path of its enclosing method. For example, if the data source IMEI returned by `getDeviceId` is found within the body of an Activity's `onCreate` method, then the propagation joinpoints will only be created for methods that satisfy the propagation rules above and are in that control flow. This optimization greatly enhances our weaving process and eliminates the need for redundant joinpoints.

5.3.2 Resource Abuse Tracing

Access to some vital functionalities such as Telephony (SMS and Calls) on the mobile devices are requested through specialized API calls. According to a 2012 Trend Micro report [74], resource abuse is the most common category of Android malware. Thus it is imperative for an analysis system to trace and report such abuse. With

AspectDroid, the system instruments the telephony method's invocations and have their target object, parameters and return value are logged. This information is significant in determining the phone number used (premium service or device contact), the message content, settings, and format.

5.3.3 Analytics of Suspicious Behaviors

Programming practices such as native call invocation, dynamic class loading, native code execution, and reflective call invocation add flexibility to software development. Although these concepts may be benign, malware can often hide its behavior using these practices to hinder static analysis or for malicious purposes such as privilege escalation.

Reflection, for instance, allows method calls to be resolved dynamically at runtime. Malware can use this technique to hide calls to sensitive APIs. With *AspectDroid*, we instrument reflective calls and analyze the target object at runtime for possible tainted data sources, propagation, sinks, or any sensitive API calls. We also check parameter arrays for possible taint propagation.

Android apps can load extra classes at runtime using a dynamic class loading API. One of the drawbacks of static bytecode instrumentation (and by extension all static analysis) is that only available classes are processed at compile time; extra classes loaded at runtime are not affected by the weaving. To address this, *AspectDroid* implements `dynamic class instrumentation`: at the joinpoint where `Dexclassloader` loads the new dex file, the weaved advice captures the absolute path to the file, sends it to the host machine via an Asynchronous task, and waits for notification to proceed. On the host machine, *AspectDroid* has a server side component that receives the dynamic class, instruments it, and pushes it back to its original path on the testbed. Although this wait time slows down the process, it

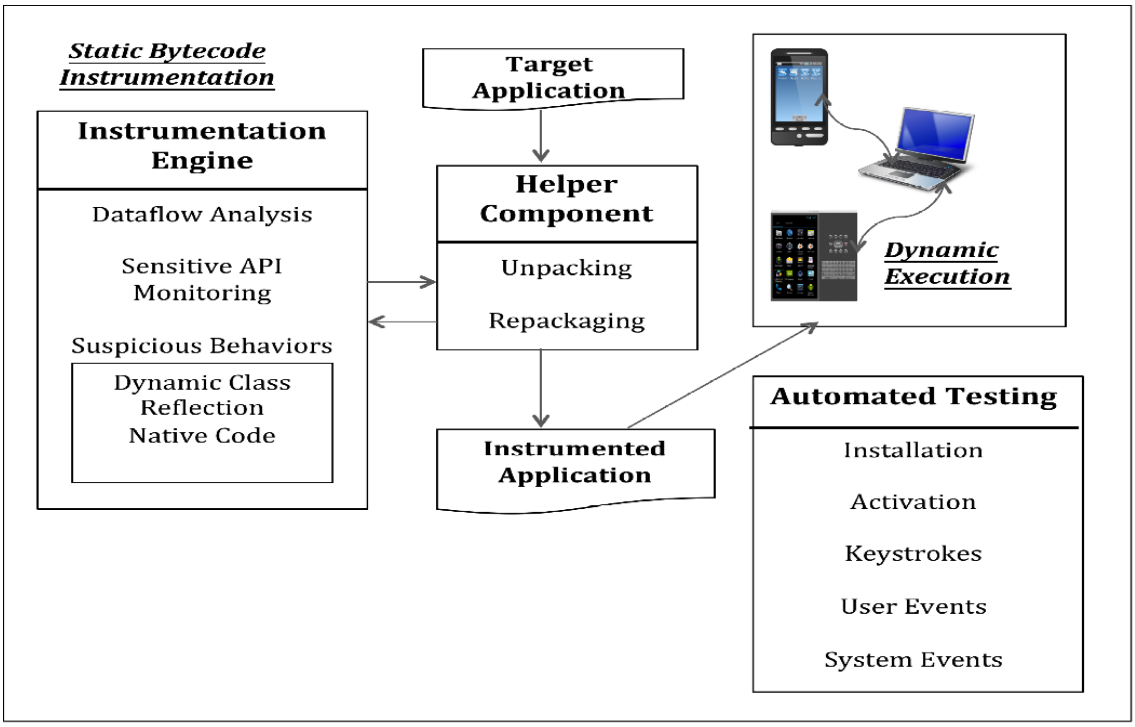

Figure 5.2: *AspectDroid* Implementation architecture

considerably expands the code coverage of our analysis.

AspectDroid also logs native code invocation, both for simple processes like `Logcat` or through the Java native interface. Although it does not trace the activities within the native code, it does log the name, object, parameters, and return value.

5.4 Implementation

5.4.1 Prototype implementation

We implemented a working prototype of our system in Python, Java, and PHP. The instrumentation engine is setup on a host machine (64-bit Ubuntu system) for the initial `dex` weaving and dynamic class instrumentation. Our software dependencies includes external tools; `dex2jar`[23], `AspectJ-ajc` [10] compiler version 1.8, and

external libraries; `Apache Web Server`[2], `aspectjweaver`[10], `Apache Commons`[9] and `Android SDK`[14]. Our experiments were carried out on both a physical device (a rooted Motorola Droid2 with Android 2.2) and two emulated devices (Android 4.1.2 and 4.4.2). The execution environments are loaded with text messages, calls, contacts, one Gmail account, and some browser history.

Helper component

AspectDroid includes a "helper" component containing modules that automate key actions. In particular, it implements unpacking, re-packaging and application signing. Android applications are written in Java and compiled into a compressed class called `classes.dex`. However, the AspectJ compiler does not understand the `dex` file format, thus the need for decompression before weaving. We use a popular open source tool called `dex2jar`, which takes an application file (.apk) or classes.dex as input and outputs a jar file containing individual .class files. When the target application is unpacked, it can be weaved together with desired aspects. After the instrumentation process, the class files are repackaged (dexed and zipped) and re-signed into an Android-compatible app using `jar2dex` and `versign` respectively.

Automated testing

Unlike many traditional applications, smartphone apps are mostly event-driven and exhibit their true functionalities based on user interactions and in response to system events. For example, forcing an SMS to be received so that a broadcast receiver can be activated is an important system event that needs to triggered for us to observe SMS abuse.

In the case of bulk analysis, manual execution of apps and triggering such events can be time-consuming. One of the drawbacks of dynamic analysis is code

coverage and a single execution path corresponding to a single app execution, whereby information obtained may not necessarily represent the complete behavior of the target app. Our assumption is the more a tool can explore an app, the more information about the app's behavior can be obtained. For that reason, we build into *AspectDroid* an automated testing module as Python scripts which trigger a series of system and user events to more fully exercise an app's functionality. This module combines some open source tools together with custom-built instrumentation programs. These events are designed to mirror real-life events on a regular Android device. They include:

1. App installation and activation of its main activity, as specified in the manifest, using `adb`.

2. Random keystrokes that simulate user touch and gestures on the app using `monkey`.

3. A user's input is simulated where necessary within the instrumentation framework. `EditText` user inputs can be associated with different input types. Most developers specify input types as provided by the Android API – email, password, etc. We make a best effort to generate data to match its possible input type. This program is attached to the body of the instrumentation code.

4. SMS, calls and device settings are generated and manipulated using `uiautomator` while GPS coordinates are simulated and triggered on the emulator by `telnet`.

Independent testing frameworks like Android Monkey are limited to only random application touches and gestures. With our automated testing, the simulated user input built on the `EditText-SetText` method automatically creates the needed `textbox` data during analysis, which proves to be very important. For example, if an

EditText is expecting an email, if the Ok button is hit using Monkey, the application may return an error and program execution may not proceed due to an empty text. But with our injected input text, execution will proceed without an error.

Other vital parts of this testing module built with uiautomator help with forcing various system's event like `calls`, which would otherwise have to be done manually.

5.5 Testing and Evaluation

Our approach seeks to provide analysts with an easier to use and more flexible system for application analysis. It is capable of examining and monitoring Android applications without restriction based on version and/or platform while still maintaining a very high level of accuracy. The objectives of the evaluation were to quantify the following aspects of the system's performance:

1. *Accuracy.* We tested the accuracy of our data flow algorithm on 105 applications from the DroidBench corpus.

2. *App Analysis.* We further evaluate the effectiveness of our system by comparing the behavioral patterns in 100 real malware families from the Drebin dataset and a set of 100 apps downloaded from Google Play. We examine data exfiltration, telephony abuse, reflective invocation, dynamic class loading, and native code execution.

3. *Execution overhead.* We measured the cost associated with dynamic execution of the target app post-instrumentation.

5.5.1 Accuracy of Data Flow Algorithm

DroidBench 2.0 [17] is an open source project consisting of 120 simulated Android applications used for testing analysis tools. These applications evaluate the accuracy

of an algorithm in detecting data flow between a source and a sink. The authors employ different methods of data manipulation, such as callbacks, arrays, application lifecycle, inter-application communication, loops, reflection, threading, and implicit flows to hide the flow of sensitive data. The apps are relatively small and they may not necessarily be representative of real life apps and/or malware in terms of size. However, they contain a wide spectrum of diverse, tricky data flow paths that can be employed by malicious and/or over-privileged applications, thus making them a corpus of interest to test *AspectDroid*.

Before executing the apps with *AspectDroid*, we execute the untampered dataset to determine if they are running correctly and producing expected results. Out of 120, 15 apps failed to execute correctly in our environment due to either permission errors or other bugs and were excluded from the analysis. The remaining 105 apps were instrumented using our *AspectDroid* prototype. All apps were tested on emulator version 4.4 using the automated testing module except for the three apps from the emulator detection group, which were retested on the physical phone.

Based on the original source code for the 105 apps, the ground truth indicates 86 apps have data leaks and 19 apps have no leaks. Our experiments show that *AspectDroid* yielded 80 true positive (TP) results, 16 true negative, 3 false negative, and 6 false positive. Thus, the *AspectDroid*'s recall is 96.4%, precision is 93.02%, and the standard F-measure stands at 94.68%. Subsequent analysis showed that in the three false negative cases, tainted and untainted data were added to a data structure, then the app sinks only the untainted data. Our algorithm taints an object that contains a tainted field, entry or element and does not handle removal of that data/object from the taint map once it is written. Since the untainted data is still part of a tainted object, we recorded a false positive. With respect to the six false positives, four were apps with the following propagation paths: `Public API`

`Field1`, `StartProcessWithSecret` and `Implicit Flows`. Our data flow algorithm taints by means of data comparison (possible taint with items on taint map), thus data exfiltration that is not explicit cannot be detected. The remaining two under tainting were a result of an optimization added to our propagation rule in order to reduce the effect of over-weaving (which results in too much additional code added to the application). This optimization is a tradeoff between the effect of over-weaving and a possible false negative; hence, these two false results are avoidable.

5.5.2 App Analysis

To test the effectiveness of *AspectDroid* for analyzing Android applications for violations of security and privacy concerns, we used malware samples from the Drebin [16] dataset, a corpus comprising 179 malware families. In our experiments, we picked one sample per family from the top 100 families. For the non-malicious samples, we downloaded 100 Android apps from Google Play. All 200 samples are instrumented, recompiled and executed using our automated testing module.

In our prototype we tagged 27 important data sources, including phone related data (IMEI, IMSI, ICCID, line Number and location data), user data (database queries), and input data. We also created joinpoints on some sensitive APIs that perform telephony functions, native code execution, dynamic class loading, and reflection invocation. The data flow and sensitive API traces created after each app execution are then parsed using a Python script to obtain the aggregated result. We categorize the analysis result into 4 groups: data exfiltration, telephony abuse, reflection and dynamic class loading, and native code execution.

Data Exfiltration

Malware and to a large extent privacy-agnostic applications often target user and/or phone related data either with malicious intent, for advertisement or identification purposes. Most sensitive data are guarded by one-time permissions (for Android versions 1-5) that gives an app open access to quite a large group of data on a device e.g., Phone-State permission. We define exfiltration as unauthorized writes of sensitive data to a file (log, sharedprefs, user-defined files), network, and SMS that are not explicitly granted by the user at the point of transfer. Our analysis of the 100 malware samples showed 127 explicit data exfiltration paths of the 27 tainted sources carried out by 23 samples. Our results showed IMEI, IMSI, ICCID, and LN are the most widely exfiltrated phone data. This is followed by contacts, call logs, and SMS from user-related data. SharedPref and network are the most common sink calls we noted while SMS seems to be the least. For the Google Play apps, we observed 25 exfiltration paths, most of which are location and phone IMEI. Network is the sink path for all these data leaks.

Telephony Abuse

SMS is one of the most widely abused resources on Android smartphones. Out of the 100 malware families we evaluated, 8 families were recorded to have some level of SMS abuse. The apps in the Pirater family send SMS to all contacts on the user's phone, posing as the user. The socially engineered, "friendly" SMS generated by Pirater contains a link that downloads the same malware to the receiver's phone if clicked. The MobileTX family, on the other hand, does not just abuse SMS functionality, but also transmits the phone's ICCID to a private number via SMS. The remaining 6 families send specially crafted SMS to premium numbers. We have not recorded any phone call interceptions, spoofing, or recording in any of the analyzed malware.

We observed the use of SMS in 2 apps and CALLs from 3 apps which belong to the communication category on Google Play. In all these instances, the SMS and CALLs were authorized by the user, based on user-supplied input.

Reflection and Dynamic Class Loading

The Reflection API is part of the standard Java environment and allows method calls to be resolved dynamically at runtime. It is a powerful tool that can be employed by malware to evade static detection. We have observed 5 malware families that use reflection in different ways. We then examine if such invocation exhibits some element of malicious intent. The Mobsquz and FakeDoc families reflectively check if the device has support for telephony-related services (phone calls and SMS). Although this may not necessarily constitute malicious behavior, given the functionality of the applications as an antivirus scanner and battery optimizer, it requires further analysis. The FaceNiff family uses reflection to invoke the methods of a background service that spoofs user accounts and passwords after it has successfully executed the super user command. The 2 other remaining families, BaseBridge and DroidDream, are not suspicious as they both invoke methods from GUI-related classes. We observed 34 instances of reflective call invocation on the Google Play apps. Surprisingly, this is higher than the malware. However, none of the API calls invoked are from our sensitive API call list. We also observed that the BaseBridge family dynamically creates 3 jar files (bootablemodule.jar, moduleconfig.jar, mainmodule.jar) and 2 dex files (mainmodule.dex and bootablemodule.dex). Within the timeframe for our automated testing and even with an extended manual execution afterwards, the app did not load these new classes dynamically as expected. Thus, we rewrote the binary to force the app to load the new dex files. This enables our dynamic instrumentation to trace the loading joinpoint and the newer classes were instrumented using our

dynamic instrumentation engine. For the Google Play apps, 7 dynamic classes were loaded in 5 apps within our testing time. We were able to successfully instrument and execute all the dynamic classes.

Native Processes

Android applications are commonly written in pure Java code, although quite a number of them include an embedded C/C++ binary. Over the years, Android malware has exploited this capability to embed mostly root exploits that trigger privilege escalation. In other instances, Linux commands that communicate with the underlying Android kernel are becoming increasingly common. In our data set, 9 out of the 100 malware families invoke native processes 72 times. Commands like `su, chmod, ps, mount` and Android's `logcat` are the most widely executed native processes. We have also noted the execution of an unknown binary (myicon) in DroidKungFu family. Since *AspectDroid* does not instrument native code, we log the code path and then manually extract the code using adb. An Md5Sum later verified that the native binary is a root exploit belonging to the family `RageAgainstTheCage`. We've noticed native code execution in 6 out of the 100 Google Play apps. In comparison with the malware apps, the Google Play apps all executed ".so" libraries vs. starting other processes like `chmod` or `su`. Beyond exploring that a particular native code has been called within the Java execution, *AspectDroid* does not monitor its content as the instrumentation engine works only on Java. Thus it is inconclusive what some unknown native libraries do.

5.5.3 Runtime Overhead

The most important costs of instrumentation occur at runtime, since both CPU and memory usage are vital on a resource constrained machine. It is especially important

that apps limit their resource usage to avoid possible garbage collection. Though uncommon in foreground processes, this does occur when apps consume too many resources.

The CPU usage is the percentage of CPU time used by a process. We measured the value given the system uptime (uTime), processes start time (startTime) and the CPU time spent in both user and kernel code for the main process and any of its child processes (uTime, sTime, cuTime, csTime). The formula is given below:

$$\begin{aligned} seconds &= upTime - (startTime/Hertz) \\ tTime &= uTime + sTime + CuTime + csTime \\ cpu_usage &= ((tTime/Hertz)/seconds) * 100 \end{aligned} \tag{5.5.1}$$

We carried out this experiment by re-running the 100 malware families using automated testing on the same platform, keystroke seeds, and number/pattern of system and user events. Using the `procrank` utility, we obtained the process memory size from each app both before and after instrumentation as well as the CPU indices above. The experiment was executed 5 times and an average for each metric (Memory and CPU) was computed.

The dark portion of the stacked bar chart illustrated in Figure 5.3 shows the memory usage for each malware pre-instrumentation, while the lighter shade shows the overhead after instrumentation. The data illustrates that the `MemSize` difference is uniform and on average, 1MB of additional memory is required to execute the instrumented application. This translates to approximately 16% more memory usage on average. In our tests, this overhead caused no issues with any of the apps.

Figure 5.4 on the other hand shows the percentage of CPU needed to render and execute each malware. The dark portion indicates the CPU usage before instrumentation while the lighter portion stacked showed the CPU usage overhead.

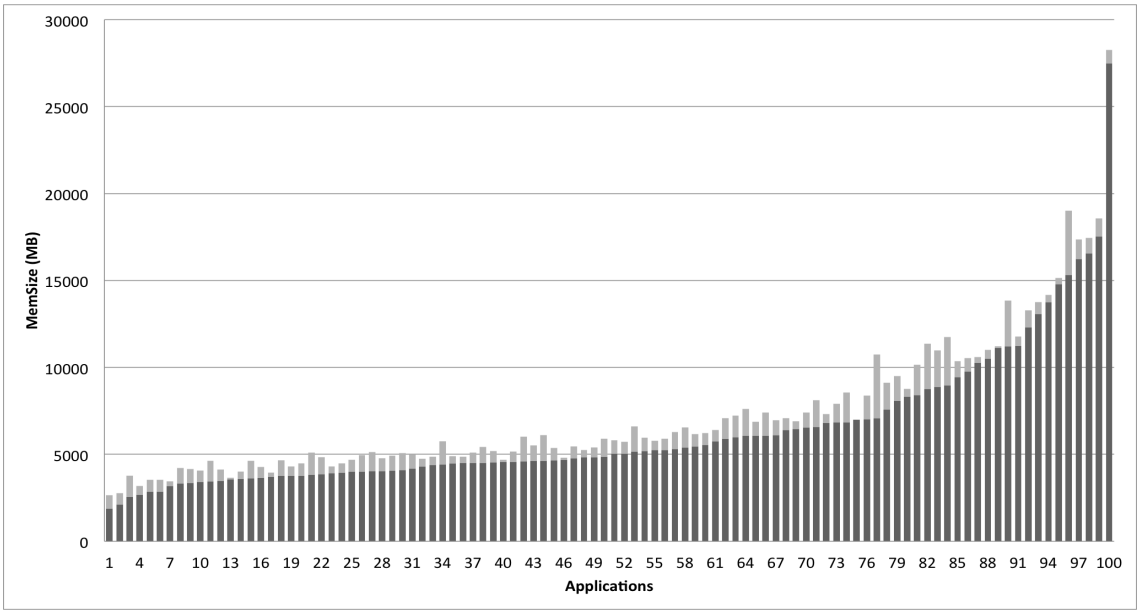

Figure 5.3: MemSize Overhead (MB)

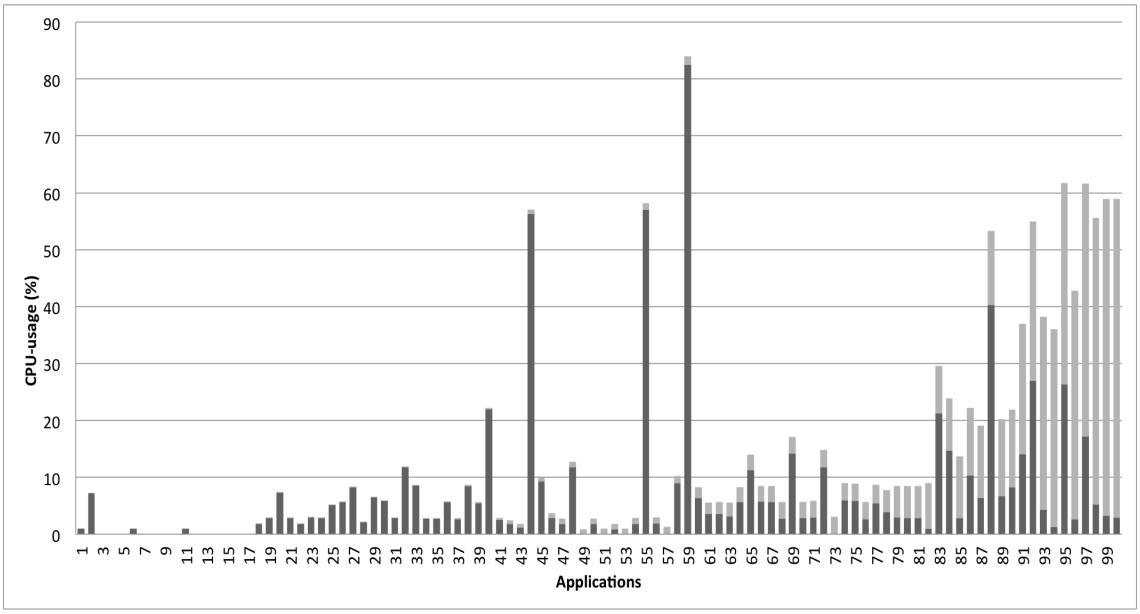

Figure 5.4: CPU usage Overhead (%)

Although the results are not uniform, the average CPU overhead is approximately 5.91%. Some apps tend to have significantly more overhead than others. We manually examined these apps and found two important factors: the number of data sources tagged and the propagation path can have a varied and compounded impact on the

CPU usage overhead. CPU intensive apps like games that requested a lot of tagged data, and especially if the request is along the path of an Activity, tend to require more CPU time to load the activity, thus increasing the percentage of CPU usage (e.g., the PJApps and Jifake families). Although the Fujacks malware family has the highest CPU usage pre-instrumentation, its overhead is negligible since it did not request any tagged data.

5.6 Challenges and Discussion

In the evaluation section we discussed the accuracy of the *AspectDroid* algorithm in detecting data leaks, the importance of tracing resource abuse and detection of suspicious behaviors like reflection, native code and dynamic class loading. Furthermore, we also highlighted the overhead associated with our system. Naturally, some challenges remain. In bytecode weaving, the compiler has to make a best effort adjustment to registers, fields, methods and instructions of the weaved class. Some applications can be sensitive to this kind of intrusion and as such can pose a setback in our re-compilation process. We make a best effort to optimize the weaving process, especially in data flow aspects, while at the same time keeping false negatives as low as possible. Specifically, we ensure that:

1. Propagation rule 1, which handles primitive returns, excludes void and boolean values. Allowing boolean values in our taint map significantly increases false positives.

2. GUI-related classes that handle graphics, views, and activities are also excluded from propagation in propagation rule 7.

3. Well-known public libraries from Android, Amazon, Google, Samsung, and Apache are excluded from the scope of weaving, however their calls are included

within the signatures, if necessary.

4. We use abstract methods within advices to reduce the number of instructions added directly to the weaved class.

Overall, our system effectively uses these optimization techniques to boost its accuracy and performance. In our testing, AspectDroid has proven to be effective in analyzing data flow paths, sensitive API monitoring, and analysis of suspicious behaviors. It provides a flexible and efficient system for assessing Android applications and does so with relatively low overhead.

5.6.1 Limitations

The main limitations of every static bytecode instrumentation are anti-unpacking and anti-repackaging obfuscation mechanisms. Developers can include obfuscated bytecode in their compiled dex files that decompilers cannot parse correctly. However, in most cases this obfuscation does not affect method invocation, which is what *AspectDroid* uses to create joinpoints. Furthermore, malware can detect instrumentation code and/or change the package signatures, which can negatively affect analysis with *AspectDroid*.

As opposed to variable level tainting, AspectDroid's data flow compares the hashes of raw data and as such cannot be affected by simple manipulations through variable re-assignment. However, arithmetic instructions can have an adverse effect on our taint propagation (though we have not encountered such in our analysis). As mentioned in Section 3, joinpoints on conditional instructions cannot be created due to the limitation in the Aspectj's APIs. This limits our data flow analysis to explicit data exfiltration.

Another limitation of our approach is analysis of native code. At this point, AspectDroid can only trace to the point where a native class is loaded and executed

and it can return the name and parameters for the execution. However, it cannot trace inside native code. Very few Android applications use native code and even with malware, the native code is typically used only for privilege escalation which is heavily dependent on system vulnerabilities. Addressing this issue is the subject of future work.

Chapter 6

Android Data Storage

6.0.1 SQLite Database

SQLite is a single-user relational database management system (RDMS) used for storing structured data. Unlike a traditional RDMS, SQLite is a server-less database engine that stores data in normal files. It manages access and concurrency based on direct file reads and writes and operating system-level file locks, respectively. SQLite is lightweight and efficient and requires little configuration, making it the database engine of choice on many operating systems, such as Android and Apple's iOS.

On Android, SQLite is used to store both private data at the application level and system-wide data like contacts and calendars. SQLite does not provide a facility for enforcing access restrictions on stored data and Android layers security on top of SQLite by prohibiting applications from directly accessing native databases. Instead, applications must use the Content Provider library, which enforces mandatory access control through the permissions model. Read and write access to these content providers must be explicitly granted at application installation time and permissions are checked at runtime when the providers are accessed.

6.0.2 Android Native Providers

Android offers built-in native content providers that store a variety of user data maintained by the system. Each is associated with at least one SQLite database that contains various tables, columns and entities. Some of the Android native providers are: Contacts, CallLog, VoiceMail, Browser, Settings, Media, and Dictionary.

These providers together with the content resolver provide the basis for Android CRUD (Create, Read, Update, Delete) operations, corresponding to SQL insert, query, update, and delete operations on database objects. The chain of events for data access occurs at two levels. At a high level, access begins with the resolver object's invocation of one of the CRUD functions, passing at least a *Uri* parameter, which identifies the location of the required data. Other parameters for CRUD functions include column name(s), a WHERE clause, and order information. The resolver validates the *Uri* and then passes the request to its provider. The provider performs permission checks and if the requesting application has the required permissions, it uses the function parameters to construct an appropriate *SQLiteStatement*.

At a lower level, the SQLiteStatement is passed to the native content provider through the binder parcel. The native library translates the parcel and sends the request to the database engine, which then performs syntax and semantic checks, expansion and code generation. The result is sent back through the same route. In the case of read operations, a database *Cursor* is returned, For write operations, an integer indicating the number of entries affected is returned. The diagram in Figure 6.1 illustrates CRUD operations on Android's native databases.

As discussed above, each of the CRUD operations triggers permission checks by the content provider. Queries are protected by READ permission while insert, delete, and update are guarded by WRITE permission. However, these coarse-grained permissions do not distinguish database roles for applications or privileges

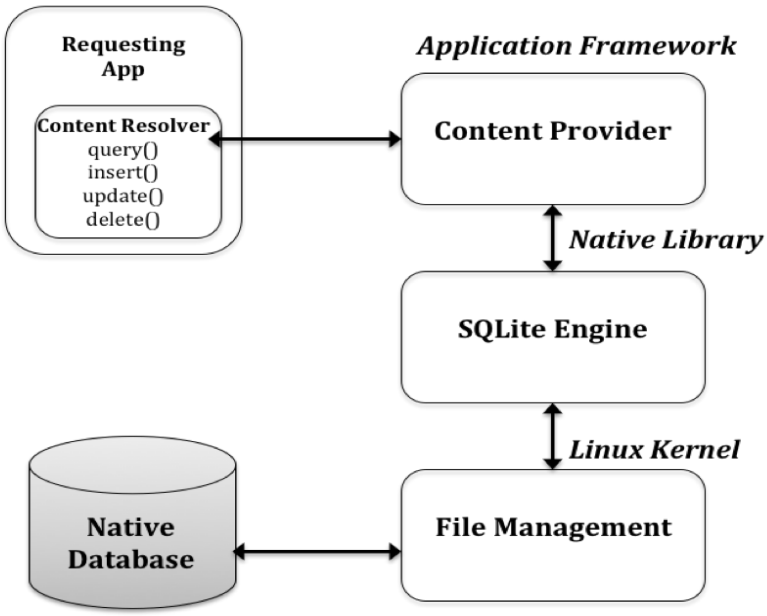

Figure 6.1: CRUD Operation on Android Content Providers

for individual data items. A simple READ permission allows access to all the tables, columns and rows in the entire database, while a simple WRITE permission allows manipulation or deletion of any database entities.

6.0.3 Threats and Vulnerabilities

The coarse-grained access control for databases under Android has serious security and privacy implications. In our preliminary research, we analyzed the contacts database and explored some issues associated with providing arbitrary applications with READ and WRITE access to this database.

Security Implications

Denial of Service: We explored a vulnerability with account types based on a malformed SQL statement that can crash the *acore* process, resulting in denial of service on the phone. A malicious app with WRITE permission can create a new contact without the user's knowledge under the "com.google" account type with a malformed account-name which contains a SQL terminator ";". The system will accept the malformed account name at the time of insertion, but after a while, Android will try to synchronize and delete bad account names. When this occurs, the malformed account-name will trigger a SQL exception in the SELECT statement and crash both the contacts application and the *acore* process. This key process is designed to automatically restart after it is killed, however, the malformed name will cause it to die once again. The repeated restart followed by crash of *acore* results in a denial of service attack on the phone and the only solution is to delete the entire contacts database, causing a loss of all local contacts if the user has no backups at hand.

Permission Leak: We also discovered that applications with the READ-CONTACTS permission can infer the user accounts on a device without requesting the GET-ACCOUNT permission. If a contact belongs to an account, the account name will be written alongside the contact in the RAW-CONTACTS table. And since there is no restriction on schema or column, an application can read the account name and type for all the contacts on the phone.

Malicious Contact: Finally, an application with WRITE access to the contacts can add a new contact under a particular account and group without restriction. When that account is synced, the contact gets pushed on to the server. This becomes a serious problem if, for example, the contact is pushed into an important work group that shares confidential information or if a contact's email address is secretly updated,

to facilitate a targeted attack.

Privacy and Attribution

Applications with appropriate coarse-grained permissions can read clearly mapped data containing names, phone numbers, email and physical addresses, and even IM status. This data can clearly distinguish an individual and be used for annoying advertisement or targeted social engineering attacks. Worse, information such as "last time contacted" can provide inferential information about call logs without having the CALL-LOG permission.

Furthermore, with WRITE permission, update, insert, and delete database operations can be performed by an application with very little data available to support attribution, since Android produces no audit logs associated with database operations. This is primarily because SQLite is a single user system and is not designed to keep track of who performs which operations on a system. For forensics investigation, this makes it very hard to ascertain if a particular entry in the database is added or updated by the user or by a malicious application.

Chapter 7

Privacy Policy Enforcement Techniques

7.1 Fine-grain Access Control

As discussed in the last chapter, Android applications access native SQLite databases through their Universal Resource Identifiers (URIs), exposed by the Content provider library. By design, the SQLite engine used in the Android system does not enforce access restrictions on database content nor does it log database accesses. Instead, Android enforces read and write permissions on the native providers through which databases are accessed via the mandatory applications permissions system. This system is very coarse grained, however, and can allow applications far greater access to sensitive data than a user might intend. For instance, in the case of the contacts database, write permission through the associated content provider allows writing new contacts and manipulating and deleting contacts at will. Of equal concern is the fact that read permissions allow access to the entire database, including phone numbers, email addresses, physical addresses, account information, etc. Ultimately, this large corpus of data that is clearly associated with a particular individual can be both accessed in malicious fashion and even transferred to a third party server with almost no restrictions.

Looking at the bigger picture, the privacy violation scales beyond the device user alone. It also exposes data associated with third parties to the prying eyes of malware and other privacy-violating applications. Clearly mapped information like phone numbers, email and physical addresses provide sufficient information about third parties that they could be used to support targeted advertisement, social engineering, surveillance, data brokerage, and physical attacks.

Furthermore, due to the interconnectivity of the different providers and their data, we have found the current approach to result in various forms of inferential permission leaks. Other security breaches like denial of service due to malformed SQL data as mentioned in the previous chapter are also possible.

To reduce the propensity of these problems and provide users with additional control over the Android content providers, Mutti et al [59] proposed an integration of SQLite and SELinux. Based on context security, this system enforces fine grain access control at the lowest level in the database. Unfortunately, the solution requires extensive changes in the operating system code to accommodate the security context schema table and its corresponding library code. Given how long it takes for Android to effect changes and for manufacturers to integrate such solutions on existing systems, techniques that require extensive OS modifications are not viable, practical options for an average user.

To solve these problems, we developed a new privacy enforcement technique that enforces access restrictions, query-rewriting and database access logging via static bytecode weaving. Our system, called *priVy*, does not require any change to the Android kernel and middleware. Furthermore, *priVy* does not treat SQLite databases as a single information store with a single set of access permissions; instead, it enforces restrictions for different schema and entity levels by re-implementing content provider library code using instrumentation at the application level, based on user-provided

access restrictions. The new weaved checking code forces the application to access only user-approved schema or entities, while maintaining application integrity.

7.2 Related Work

7.2.1 Android SQLite

The sensitivity of data and the disastrous effect of its breach has led to more rigorous research on database security in the recent past. Access control, auditing, authentication and encryption are all paramount at the server, network, and application level. Object and system privileges ensures users have the right access level to perform both administrative and simple data access on the system respectively.

Traditional relational database management systems (RDBMSs) has evolve over the years, incorporating different levels of security granularity at schemas, column, and entity level. SE-PostgreSQL [56], for instance, integrates PostgreSQL with SELinux such that every database object has a security context indicating its privilege and attributes. Oracle's virtual private database [46] and INGRES [72] on the other hand support fine grained access through runtime query modification.

SQLite is an RDBMS which by design is a server-less jumbo file attached to an application. Its security layer is completely provided by the Android OS through the READ/WRITE permission system on the file. Although the file is protected from unauthorized access, privileges on the individual objects (schema, column, entities) are not segregated. SE-SQLite [59] like SEPostgreSQL is developed by integrating SELinux into Android SQLite. It provides low-level access control on database schema and tuples. Our work, though very much related in objective, differs completely in implementation. Theirs integrated the access policies in the database engine, while we enforced the access constraints at the application level by hooking the CRUD

method calls and forcing query-rewriting where necessary. Our system does not require flashing a customized ROM, thus it is a more accessible and easily deployable solution.

7.2.2 Instrumentation

Instrumentation has been a vital tool for enforcing secure policies on Android systems both in static and dynamic contexts. Largely due to ease of application repackaging, static bytecode weaving at the application level has garnered a lot of attention. Dr. Android [50] retrofits Android permissions using bytecode instrumentation. Capper [85] tracks sensitive information flow from source to sink while RetroSkeleton [29] enforces various flexible security policies at runtime. Appguard [18] and [19] both provide customizable user-policies through on-the-device application repackaging. Most of these solutions have the same aim of reducing permissions associated with an Android app in general, whereas *priVy* is very specific to access control on SQLite databases, which permission control alone cannot achieve.

Dynamically, FireDroid [64] and NJAS [21] use ptrace to attach their policy monitor to the target process. In both of these solutions, security policies are defined at a lower level by re-mapping the system calls to higher level API calls. The Android PIN project also supports dynamic binary instrumentation. TISSA [95], Aurasium [81] Apex [60] all developed different security policies, mostly with respect to reducing permissions, by extending the Android framework. COMPAC [77] segregates permissions within the components of an application. AdDroid [61] segregates advertisement and the Android framework by introducing new advertisement APIs and permissions while AdSplit [69] executes the advertisement code in a different process.

ASM [47], SEAndroid [70], MockDroid [20], and AppFence [48] are operating system-centric solutions that developed integrated security policies at the kernel and

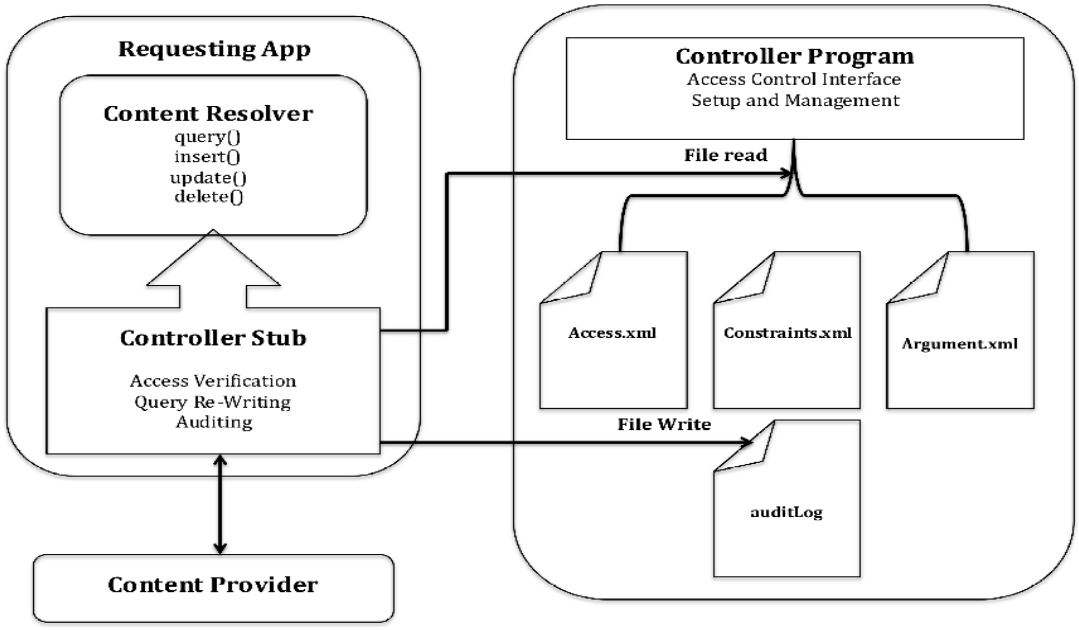

Figure 7.1: *priVy's* System Architecture.

Dalvik code. While the security policies suggested above can be used to either allow or deny access to the database file, they cannot address the issue of access control on the database object.

7.3 System Design

Our goal is to define low-level access controls for Android's native content providers and enforce these access controls for third party applications. This will ensure that users have tight control over read/write accesses on sensitive data for instrumented applications.

priVy is comprised of two components, a Controller app and Controller stub. The Controller app is an independent application running in a different process that registers an instrumented application and sets up and manages its access levels. The stub provides the weaved code that forces the instrumented app to verify access levels

at startup, enforce access constraints, perform query-rewriting as necessary, and effect database auditing. The architecture of *priVy* is illustrated in Figure 7.1.

7.3.1 Controller Stub

Our approach uses the AspectJ instrumentation framework [53] to insert and enforce fine level access verification, query re-writing and database auditing for Android's native providers.

Depending on the cross cutting concern, signatures can be made very broad using wildcards or specific with direct package names, return types and parameter types. In *priVy*, we designed signatures for Android packages related to data access on SQLite databases. The three most important are *Database, Content Provider and Resolver*. The database package hosts the main SQLite database object and corresponding methods to query it in raw form. It also provides the *Cursor* interface for reading the results of database queries. It is important to note that Android does not support direct raw access for databases associated with an application with a different *uid*. Access to such data can only be provided via the *Uri* of the target Content provider. The provider classes expose data of one app to the code executing in a different process.

Generic AspectJ advices were then developed around the methods in the relevant classes from the packages discussed above. These advices are encapsulated in an aspect which is then statically recompiled into an Android binary. The result is the same Android binary extended with our controller stub. This static instrumentation process intercepts the resolver CRUD functions and inserts the controller code where necessary. As mentioned in Section 2, direct access to native databases is completely prohibited by Android and access is only available through the exposed native content. Thus, it is relatively convenient for us to develop specific signatures corresponding to

Table 7.1: Joinpoints Picked by *priVy*'s Pointcut Signatures

Target Object	Insert	Update	Query	Delete
Content Resolver	insert(..)	update(..)	query(..)	delete(..)
Content Resolver	bulkInsert(..)			
ContentProviderOperation	newInsert(..)	newUpdate(..)	newAssertQuery(..)	newDelete(..)

only the resolver and provider packages.

As shown in Table 7.1, insertion operations can be performed in three different ways, either via a single insert, bulk insert, or using a content provider operation. Delete, update, and query operations can each be performed in two different ways. Our signatures take into account all these and we target the respective joinpoints accordingly.

With the exception of adding auditing log entries, where code is inserted using "after" advice, all other code that performs constraint checks and query-rewriting uses "around" advice, which can perform code injection in the middle of method execution and allows manipulation of target object, parameter(s) and return value. This enables us to generate the correct return values in case a query is blocked or restricted. It also allows us to enforce constraints and reflectively perform new method invocation on an already created object residing in memory.

Access Verification

At runtime, when the instrumented app begins execution, the controller stub performs the access verification as illustrated in Figure 7.1. It reads and parses the assigned access control for the target application from the world readable, shared preferences XML file for the controller app. It sets up the global variables for the access level, schema, and column as well as entity privileges for each provider. The global variables are used by the CRUD operation's joinpoint to determine how the method call will proceed when invoked.

The CRUD function's access level can be ALL_ALLOW, ALL_BLOCK and RESTRICT. The ALL_ALLOW access level, as the name implies, does not impose sanctions on the joinpoint and simply allows it to proceed with its original parameters. ALL_BLOCK, on the other hand, completely blocks the execution of a joinpoint. For ALL_BLOCK, our controller stub must ensure that the query is actually blocked while not affecting application stability. We are able to achieve this by ensuring the affected functions return appropriate values as shown in Figure 7.2 for Query joinpoint. Specifically, depending on the type of access functions, different actions must be taken:

1. Query and Insert: We transform the given *Uri* parameter of the function into an *Entity Uri* with appended zero. For queries, the system proceeds with this special entity *Uri* which in turn will force the return of an empty cursor with at least header information. For insert operations, the entity *Uri* gets returned and the parameters are discarded.

2. Update and Delete: Functions performing update and delete operations expect an integral return type indicating how many rows were affected. We simply return 0, indicating that no rows were affected and thus the program will continue to execute smoothly.

The RESTRICT option regulates access to database schema, columns, and entities. In a relational database, a schema represents a logical group of objects. In this paper we restrict the schema definition to the database tables available through the content *Uri*, e.g., the contact table in "Contacts.db" or the events table in "Calendar.db". Also, we logically include all objects in a table that can be grouped by the same MIME types, like emails, phone numbers, and addresses, as different schemas.

```
pointcut getCurObj(Uri uri, String[] Projection,
      String Selection, String[] Selection_Args):
      call(*..*Cursor* *..*.query(..))
      || call(*..*Cursor* *..*.*Query(..)))
      && args(uri, Projection, Selection,
      Selection_Args,..) && NotNewLogger();

Object around(Object tar, Uri uri, String[] Projection,
      String Selection, String[] Selection_Args):
      target(tar) && getCurObj(uri, Projection,
      Selection, Selection_Args){
      //...
      //...
      ContentValues cont = getAccess();//From SharedPrefs
      if (cont.containsKey(uri.getAuthority())){
            start = System.nanoTime();
            String level =
            cont.get(uri.getAuthority()).toString();
            if (level.equals("ALL_ALLOW")){
                  ret = proceed(tar, uri, Projection,
                  Selection, Selection_Args);
            }else if (level.equals("ALL_BLOCK")){
                  ret =  proceed(tar, getEntityUri(uri),
                  Projection, Selection, Selection_Args);
            }else if (level.equals("RESTRICT")) {
                  checkSRestrict(..);//Schema Restriction
                        //...
                  checkCRestrict(..);//Column Restriction
                        //...
                  checkERestrict(..);//Entity Restriction

            }else{

            }
      }
```

Figure 7.2: Advice on a Query Joinpoint that Shows How the Controller Stub Performs Access Verification

Thus, the schema restriction ensures an app only queries from the approved tables or MIME type(s). Since most of these MIME types have individual Uris assigned to them through the *CommonKind Uri*, their schema restriction must ensure a entity restriction on the main table as well. The controller stub makes a decision to ALLOW, BLOCK or REWRITE the query based on the schema restriction established by the user. For example, if a user has a schema restriction set up to only allow access to email information and and the app requests both email and phone numbers, *priVy* must re-write the query such that only the email table gets projected on the SQL statement. Furthermore, restrictions can be imposed on database columns so that certain columns are prohibited from being viewed by apps, e.g., account-type and name, or an entity based on a column value.

Aside from the two mandates mentioned above for BLOCK option, a third condition becomes necessary here, specifically, that *priVy* must ensure that any other part of the application that depends on the return value of the function does not crash. This is mostly an issue with query functions, because they return a cursor and the program may have been designed to access a particular column which may not be available due to restrictions. To solve this problem, we instrument all the functions that access cursor information directly. The advice on these joinpoints tests if the column requested is available and if it isn't, the column will return a empty string. This has proven to work well in practice to ensure that applications do not crash due to the imposed access restrictions.

Query Re-Writing

Android creates a proper SQLstatement after the request has passed the permission checks. Since our system operates at the highest level, we rewrite the intended query by altering and/or supplying new CRUD function argument(s). These functions con-

```
// qSRestrict contains list of restricted Uri
public Uri checkSRestrict(ArrayList<String> qSRestrict,
     Uri uri, ContentResolver resolver){
     if (qSRestrict.contains(resolver.getType(uri))){
          uri= getEntityUri(uri);
     }else{
          //...
     }
          //...
}
```

Figure 7.3: Schema Restriction Check on a Query Function

tains *Uri*, *Projection*, *Selection*, *Selection-Arguments* and *Content Values* parameters.

In SCHEMA restriction, *priVy* compares the query Uri with the restricted schema; if matched, the query is simply blocked otherwise the system allows it to execute. The code snippet is shown in Figure 7.3.

The query-rewriting module is triggered when the initial query is projected on column(s) and/or entities outside its access restrictions. In a query function a projection argument can take an array of column names or null (indicating all rows in a table should be returned). Armed with the column-level access restriction, the Controller stub executes the *checkCRestrict* function and re-writes the query based on the following rules;

1. **If projection is not null** - the stub checks for the intersection of the *projected column(s)* and the restricted column(s) and then removes them from the projection list as shown in Figure 7.4.

2. **If the prohibited column is the only column to be projected** - the function will be blocked completely. This is because exchanging the prohibited list with null will return all the columns including the prohibited ones.

3. **If projection is null** - For query, the stub checks the intersection of the columns of the *return cursor* and the restricted column(s). If found, the

```
if (Projection!=null){
     if(myMap.keySet().contains(resolver.getType(uri))){
          String val = myMap.get(resolver.getType(uri));
          for(String str: Projection){
               if(!(resolver.getType(uri)+str).equals
                    (resolver.getType(uri)+val)){
                    newProj.add(str);
               }
          }finProj = newProj.toArray(new
          String[newProj.size()]);
     }else{
          finProj= Projection;
     }
}
```

Figure 7.4: Column-level Restriction with Not-Null Projection

intersected column(s) are removed and the query continues with the remaining column as shown in Figure 7.5. For Update and Insert, restricted column are prevented from database write thus key sets of the content values are compared against the restricted column and removed if there is an intersection. Delete operations do not require column projection.

Selection and Selection-Arguments indicate the WHERE clause column(s) and value(s). The user can restrict access on some predefined values, e.g., to certain account types, whitelisted contacts, etc. For the most part, we don't test or nullify these arguments, but rather we enforce the new specification by concatenating our restriction to an already established WHERE clause. For instance, an application might be restricted to only query contacts from account-type *"com.google"* and we simply ensure that this is enforced by influencing the WHERE clause. If this restriction involves only one entity, the controller stub appends it with an "AND" operator to the function's WHERE clause, if not null. If the WHERE clause is null, however, the stub then substitutes the null with the new restriction, and the function

```
if (ret instanceof Cursor){
      Cursor cur =(Cursor)ret;
      if(cur.getCount()>0){
            String[] pNames = cur.getColumnNames();
            if(myMap.keySet().contains
                  (resolver.getType(uri))){
                  String val =
                        myMap.get(resolver.getType(uri));
                  for(String str: pNames){
                        if(!(resolver.getType(uri)
                              +str).equals(resolver.getType
                              (uri)+val)){
                              newProj.add(str);
                        }
                  }finProj = newProj.toArray(new
                        String[newProj.size()]);

            }else{
                  finProj= null;
            }
      }
}
```

Figure 7.5: Column-level Restriction with Null Projection

proceeds with this new value. On the other hand, the situation is more challenging when there are more than one entity restriction and it applies to different tables (e.g., account type (Raw_Contacts) and lookup key (Contacts)). In typical SQL we can perform complex joins on the different tables. However, on content providers such operations are very limited. To solve this, we extract the primary key (and foreign key where necessary) from each of the tables and use them as the parameter(s) for the target query's WHERE clause as shown in 7.6.

For example, consider the query "_ID from contacts WHERE lookup key = value" and the query "_ID and RAW_CONTACT_ID from raw_contacts WHERE account_type = value". The intersection of _ID and RAW_CONTACT_ID in these query results will be the new WHERE clause for the target CRUD operation.

For delete and update functions, a developer may or may not supply the WHERE clause and/or its argument. According to a user's preferences, our system

```
public String checkERestrict(Uri uri, ContentResolver resolver){
     String finSel= null;
     if (uri.getAuthority().contains("contacts")){
          // for each entity restriction,
          // getContactIds(..) gets its primary key column.
          //The intersection of the results is return in fin
          String fin = getContactIds(resolver);
          if(fin!=null){
               if(uri.equals(ContactsContract.Contacts.
                    CONTENT_URI)){
               finSel = ContactsContract.Contacts._ID +
               " IN ( "+fin+" )";
               }else if (uri.equals(ContactsContract.Data.
                    CONTENT_URI)){
                    finSel = ContactsContract.Data.
                         RAW_CONTACT_ID + " IN ( "+fin+" )";
               }else{
                    finSel = ContactsContract.
                    RawContactsEntity.CONTACT_ID + " IN (
               "+fin+" )";
               }
          }
     }
     return finSel;
}
```

Figure 7.6: Code Snippet Showing Entity Restriction for Contacts Provider

can enforce restrictions on when and where delete operations can occur by reflectively invoking a new delete function within the joinpoint on the target object. After it returns, the new return value is supplied as the return value of the joinpoint's advice as shown in Figure 7.7.

Database Auditing

SQLite database is a single user RDBMS. But nevertheless, on Android, important native databases are often accessed by multiple applications which are considered individual users with different user IDs. It is important to keep track of which applications perform which actions on system resources, especially since these applications are typically created by different developers and may manipulate the same data with few restrictions. Currently, Android does not support this kind of fine-grain auditing.

In our prototype implementation of *priVy*, we introduce auditing using a file attached to the Controller app called the auditLog. We implement this by injecting the auditing function after the CRUD function has executed and returned a desired result. For insert, an "after" advice will request for the returned *Uri* and then parse it get the row id. This package name, row id, together with Uri name, Content Values and time stamp are written to the auditLog file.

On update, the return value is the number of rows affected rather than the *Uri*. Thus, we need a global variable to keep track of the row lookup ids (rowid) affected by the update function. We use this global rowid, together with package name, Uri name, selection and its arguments (if any), content values and time stamp as an audit file entry. This also applies for Delete operations. Query operations return a Cursor, thus we keep the audit of the query parameters as well as the number of rows in the cursor. We do not track the IDs of the columns because it may or may not be part of the projection list. The code snippet for auditing query operations is shown in Figure

```
pointcut deleteInst(Uri uri):call(* *..*.delete(..))
     && NotNewLogger() && args(uri,..);

Object around(Uri uri, ContentResolver tar):
     deleteInst(uri) && target(tar){
     if (access.containsKey(uri.getAuthority())){
            //
          }else if (level.equals("RESTRICT")){
               ContentResolver resolver = null;
               if (tar instanceof ContentResolver){
                    resolver = (ContentResolver)tar;
               }
               if (resolver!=null){
                    //check Schema Restriction
                    uri = checkSRestrict(qSRestrict,
                         uri, resolver);
               }
               if (!uri.toString().contains("/0")){
                    Log.d(uri.toString(), "here3");
                    //check Entity Restriction
                    String finSel = checkERestrict(uri,
                      resolver);
                    if(finSel!=null){
                         //populate selection
                    }
               String[] selArgs = (String[])args[2];
               Object[] params = new Object[]{uri,
                    sel, selArgs};
               Class clazz = thisJoinPoint.getSignature().
                    getDeclaringType();
               //Reflectively Recreate Delete
               //function with new selection and
               //return number of rows deleted
               try {
                    String methName = thisJoinPoint.
                    getSignature().getName();
                    Class[] paraTypes
                         =getMeth(thisJoinPoint);
                    Method method
                         =clazz.getDeclaredMethod(methName,
                         paraTypes);
                    ret = method.invoke(tar, params);
                    delRet= true;
               }catch (Exception e){
                  //
               }
         //
     }
     return ret;
}
```

Figure 7.7: Code Snippet Showing Query Re-writing for Delete Function

7.8.

Apart from its major objectives, our controller stub further checks for malformed strings in arguments passed to the CRUD function. This is important so as to prevent the *denial of service* attack mentioned in Section 6.0.3. This functionality checks for special characters in the content value(s) of an insert or update function. It then triggers a warning to the user and he/she can opt to remove any special character.

7.3.2 Controller App

The controller app running on a separate process coordinates the content provider restrictions for targeted applications. Our aspects are written as generically as possible to integrate into any Android app as well as work for all the native providers. The controller app provides an interface for choosing the access levels and further access restrictions on schema, column or entity, thus saving the cost of re-instrumentation in case changes need to be made. This significantly improves the usability of our approach.

When an application is installed, the user needs to register it with the controller. Its user interface (UI) exposes the available access level/restrictions for the user to choose from. After selection, the values is stored for the target application in a shared preferences XML file maintained by this controller app. The Controller stub queries these files at runtime. The controller app maintains three different XML files for the access level, constraints, and the arguments, as shown in Figure 7.1.

1. Access.xml - this file contains entries for all registered instrumented apps. It takes the concatenation of package name, provider names, and CRUD function name as the key which is also the record identifier *RID*, while the value contains the access level as ALL_ALLOW, ALL_BLOCK, or RESTRICT. Listing 7.1 shows an example of a key:value pair in Access.xml file.

```
after(Uri uri, String[] Projection, String Selection,
      String[] Selection_Args) returning (Cursor ret):
      getCurObj(uri, Projection, Selection, Selection_Args){
      //...
      if (ret.getCount()>0 ){
            String vals=null;
            StringBuilder stb = new StringBuilder();
            if(Projection!=null){
                  for(String str: Projection){
                        stb.append(str);
                        stb.append(",");
                  }

            }
            vals = stb.toString();
            String args=null;
            stb = new StringBuilder();
            if(Selection_Args!=null){
                  for(String str: Selection_Args){
                        stb.append(str);
                        stb.append(",");
                  }
            }
            args = stb.toString();
            String audit = "Time"+Long.toString(System.nanoTime())
            +" Uri "+uri.toString() +" Values "+vals + "Selection"
            +Selection + " Selection_Args "+ args+
            thisJoinPoint.getSignature().getDeclaringTypeName()+
            "."+thisJoinPoint.getSignature().getName();
            Log.d("R-DAC","Query Audit- "+audit);
            //…
}
```

Figure 7.8: Instrumentation Code Snippet for Auditing Query Operations

2. Constraint.xml - If the access level is set to RESTRICT, the schema, column, or entity constraint has to be provided. This constraint is registered in constraint.xml. Its entries are the RID as provided in Access.xml file and the values are the constraints separated by commas as shown in Listing 7.2. Empty brackets indicate there is no constraint on the element. The SCHEMA constraint has to take complete Uri string names, while the COLUMN and ENTITY constraints contain the Uri and column name, each of which can have zero or more constraints.

3. Argument.xml - As mentioned above, the ENTITY constraints are enforced in the WHERE clause of the SQLStatement. Thus for each entry in the Constraint.xml file that contains a record for an ENTITY constraint, there must exist an record in the Argument.xml file that provides the argument value(s). For example, if Constraint.xml contains a record as shown in Listing 7.2, the Argument.xml file will have a corresponding record as shown in Listing 7.3. This constraint ensures an app is restricted from querying contacts using clauses like "WHERE account_type is *"com.google"*".

Listing 7.1: Entry in Access.xml

```
key - com.bbm:contacts:query:
value <RESTRICT>
```

Listing 7.2: Entry in Constraint.xml

```
key - com.bbm:contacts:query:
value < SCHEMA(),
      COLUMN(vnd.android.cursor.dir/contact:
         display_name),
      ENTITY(vnd.android.cursor.dir/raw_contacts:
```

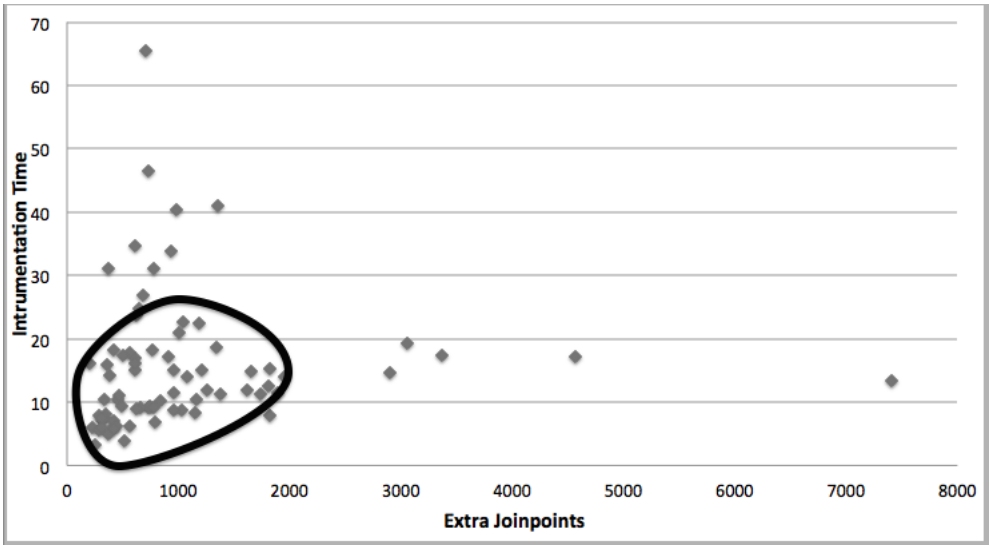

Figure 7.9: Relationship Between Instrumentation Time and Extra Joinpoints

```
        account_type)
    >
```

Listing 7.3: Entry in Arugument.xml

```
key - com.bbm:contacts.query:
value < ENTITY((vnd.android.cursor.dir/raw_contacts:
        account_type) :com.google)>
```

Apart from creating and managing access verification information, the controller app also manages the auditLog file.

7.4 Implementation

We implemented the prototype of our approach in Python, Java and AspectJ as the weaving framework. The Controller app is written as a standalone Android application with three shared preference files that store the access level, constraints, and its arguments for an instrumented app. This app does not require any permissions

to install. For the Controller stub, the instrumentation process is implemented using Python scripts which automate application unpacking, repacking and signing, while the weaving aspect is written using Java/AspectJ.

Android apps are shipped as a single zip file called an *apk*, which contains the main *classes.dex* file and other resource files. The classes.dex is a highly optimized compressed file that contains the Dalvik bytecode which is parsed and interpreted by the Dalvik Virtual machine at runtime. It is created by removing redundant information from the app's compiled Java classes. The AspectJ framework, on the other hand, does not understand the Dex file format. Thus, to weave-in the Controller stub, we need to unpack from Dex to Java class files.

The automated instrumentation processing makes use of an open source Dalvik translator called *"dex2jar"* [23] for the unpacking, repackaging, and app signing. This processing is set up on a Linux system with Java and "ajc" compilers installed and the AspectJ library and the Android SDK on its class-path. The weaving module takes the unpacked classes as input and after recompilation executes the repackaging and resigning modules, respectively.

We developed generic aspects that can be woven into any Android application to enforce access control on any of the 7 native providers. However we limited our testing and evaluation to contacts and calendar providers. These two providers contain valuable and sensitive data for both the device user and any third party associated with the user. According to [40], the contact information is by far the biggest privacy concern of all the sensitive data found on smartphones. The contacts provider exposes different kinds of data via its numerous Uris. These data are contained in three tables (Contacts, Raw_Contacts, Data) under the contacts database, while the calendar provider exposes five tables (Calendars, Events, Attendees, Reminders, Instance) from the calendar database through its URIs.

Our aspect has a total of 11 pointcuts which corresponds to the 9 method calls as shown in Table 7.1, one pointcut for application context, and one for the aspect itself. It also has a total of 16 advices for these pointcuts and numerous Java-related methods that support the functionality of the advices.

7.5 Evaluation

The target of our evaluation covers two main objectives; overhead and application crash. *priVy* is developed as a user-centric solution with the aim of providing a reliable means of securing and restricting access to native database objects. The goal is to ensure *priVy* works on a diverse group of applications with minimal overhead. Furthermore, we want to ensure the instrumented app does not crash as a result of the weaved controller stub.

We downloaded the top 350 applications from Google Play [5] and choose 76 apps with read/write permission to either the contacts or calendar providers. Our samples are instrumented and repackaged with new sign-in keys. We assess their static overhead in terms of weaving time and number of joinpoints created.

We also measure the runtime overhead, which is the time it takes to execute each of our joinpoint's advice. We developed a test application that triggers all the advices in our aspect (since most of the sample apps trigger only one or two of them) and measured their execution time. Finally we evaluate app crashes by executing each of the instrumented applications. Our testbed is a Samsung tablet running on the Android 4.4.2 kernel. It has some saved contacts under the device's main gmail account, local phone numbers, and others imported from one extra Microsoft Exchange account.

7.5.1 App Execution

We then executed all the 76 instrumented apps using a three round testing scheme (total of 228 executions) on the test bed and monitored them for app crashes that can be directly linked to the instrumentation stub. The testing involves changing the different access levels - ALL_ALLOW, ALL_BLOCK and RESTRICT. Each app is manually installed, executed and profile created for those requiring one. We interact with them using touch events, text inputs, and various system events like calls. The testing period for each app ranges between 15 to 20 minutes, depending on the initial setup required by the app.

In the first round of testing, we set the access policies for all the 76 apps to ALL_ALLOW. Our first observation is five apps (Chase, SendHub, All State, BlueBird, Citizens) fail to execute or connect to their server in the first round of testing. An examination into these groups revealed that mostly they are vendor apps like mobile banking. Such apps, for security reasons, do not execute when the signature changes or they have broken resources. They fail to execute not because of our instrumentation stub but because of a change in the application file, thus we eliminate these from further testing.

The second group of seven apps (Sirma Bible, Docusign, Autodesk, Faith-ComesbyHearing, Zillow, Backgrounds, Jiffy) did not make any attempt to query the contacts or calendar database, even though they requested READ and/or WRITE permission. Thus, they were eliminated too.

The final group executed correctly in all 3 rounds of testing. We randomly changes the combination of access restrictions that will trigger the query re-writing module during manual execution and check for app failure. Within the execution period, we observed that all the 64 apps in this group invoked one or more of our joinpoints. For instance, the BBM app requested READ CONTACT permission and

it also asks users explicitly for access to contacts on the setup window. When we BLOCK all contacts, it was not able to access any. Similarly for RESTRICT, it was only able to view contacts from the gmail account. This is the same for other apps like KIK, Pinterest, Mr. Number, AVG AntiVirus, AutoCard, Vine etc. The Sunrise app uses the calendar provider to manage and organize events. We successfully limited the events that can be viewed by this app based on Event_ID.

We have observed that $\approx 82\%$ of the apps in our sample perform only database read (query) even though 60% of them request both READ and WRITE permissions.

7.5.2 Static Overhead

Instrumentation entails weaving new code into a binary and optimization becomes essential in order to avoid bloating the existing code. Specifically, the nature of pointcut signatures can have adverse effects on the number of joinpoints created and wildcards that designate all (*) either in the parameter(s), names, or return type broaden the scope of joinpoint matching. Android has restricted native database access to very few libraries, and as such we avoided using wildcards where necessary and used more specific signatures instead.

In this test, we measured the time it takes to perform bytecode weaving as well as the number of joinpoints that are created. The bytecode weaving involves class parsing, joinpoint matching and insertion of advices for every class on the jar path as specified by the weaving aspect. On average it takes 15 seconds weaving time on our test platform to process each sample app, with the highest and lowest being 65.5 and 3.3 seconds respectively. The plot in figure 7.9 showed no correlation between instrumentation time and the number of joinpoints created. Nevertheless, we can see from the cluster that almost all the apps are woven in less than 30 seconds. Manual investigation into the packages of the outlier applications indicates they contain a

very large number of classes, thus requiring more time for the compiler to parse and match the joinpoints. Overall, we find the instrumentation time to be acceptable as the maximum is slightly above 60 seconds.

In AspectJ weaving, for every matched joinpoint, the compiler adds a call to its corresponding advice. This is in addition to any Aspect-specific and Java-based method attached to the aspect class. Based on our sample set, we recorded an average of 1032 joinpoints created, with the highest being 7407 and the lowest 199. We find our joinpoint's overhead of approximately 1000 is on the high side considering there are about 16 advices. On investigation we find the pointcut that gets *application context* is the culprit. Though not part of the main functionality of our aspect, this pointcut serves as a helper that assigns application context to our query rewriting joinpoints.

The Aspect class in not part of the traditional Android API and thus cannot instantiate a context. On the other hand, our advices requires it to get a ContentResolver for nested SQLstatements and the processing of ContentProviderOperations functions. Thus, we created this pointcut around the *onCreate* method of every Activity to get its context. This ensures at every point during the app's execution, a context is available to the aspect class. We find this method to be very reliable since even if one activity dies, the next activity will provide the needed context for the aspect, but not necessarily efficient.

7.5.3 Runtime Overhead

In this evaluation, we examined the impact of the introduced code on the app performance on the device at runtime. This is measured as the extra time it takes to run the advice on a joinpoint. Our advices verify access levels and enforce restrictions where necessary.

Table 7.2: *priVy*'s Average Runtime Overhead Given Various Access Restrictions

Restrictions	Insert (ns)	Update (ns)	Query (ns)	Delete (ns)
1 Schema	59	40	64	47
1 Column	53	45	62	50
1 Entity	23	38	92	54
1 Schema , 1 Column	57	48	58	49
1 Schema , 1 Entity	76	47	70	69
1 Column , 1 Entity	74	50	73	67
1 Schema, 1 Column , 1 Entity	62	70	75	54
Average	57.71	48.29	70.57	55.71

As mentioned in section 3, *priVy* provides three access levels, and when the access level is set to RESTRICT, zero or more schema, column and entity restrictions can be enforced. Thus, considering all this criteria, we expect different possible combinations for each of the CRUD functions. It is important to note that this runtime overhead remains the same irrespective of the application executing because the advice code does not change. However, it is only affected by the access level and constraints enforced by the user. The more constraints there are, the more instructions are traversed and the greater the size of the SQLstatement.

To make this experiment possible we develop a testing app that triggers all our joinpoints and we run it many times with different combinations as shown in Table 7.2.

Each point on the table represents the average time in nanoseconds (ns) required to execute each CRUD function based on at least one restriction.

ALL_BLOCK and ALL_ALLOW incurs zero runtime overhead to process thus these are excluded from the table. The times are computed by Java's *System.nanoTime()*. We set the start time at the beginning of the "around" advice execution and an end time at the beginning of its "after" advice. This ensures we take the time after the method has returned and before the next instruction. The difference between the start and the end time is the runtime overhead per joinpoint.

Our experiments indicates it takes an average of 70 nanoseconds to execute the joinpoint on update, 55 for delete, 48 for insert, and 57 for query, with imposed restrictions. Overall our joinpoints take an average of 58 nanoseconds to execute any of their advice.

Since the static overhead gave us an average of 1032 joinpoints created, our instrumentation can incur a maximum runtime overhead of 0.06 milliseconds if all the joinpoints are executed in a single app. We find this overhead to be negligible and not be noticeable by users.

7.5.4 Access Policies

The access policies exposed by our Controller app enable users to protect the security and privacy of their devices and its data. For instance, entity restrictions can help protect devices from adding malicious contact on protected account types such as google.com or Exchange. The entity restriction is set with an appropriate argument e.g., "com.google". Denial of service attacks due to malformed SQL can also be checked and special characters removed in the content values of an insert or update function.

Both schema and column restrictions can help reduce the exposure of clearly mapped data, e.g., enforcing schema restrictions on an email Uri can block a target app with access to Contacts from mapping contacts phone numbers and their email addresses. However, to attain absolute protection for some chosen contacts, the user can set up entity restrictions on individual names, IDs or groups. This will completely safeguard whole record(s).

Column restrictions can also protect against permission leaks. As mentioned in Section 3, enumerating account names and types can give apps access to all accounts on the system just like those provided by the *getAccount* permission. Thus, a user

can set a policy to restrict these columns. This policy can be further optimized by enforcing entity restrictions such that apps can access only their account name and types.

7.5.5 Limitations

priVy performs instrumentation via application repackaging and thus depends on the fact that the app will be correctly translated before and after instrumentation. However this may not be true for apps with:

1. Anti-repackaging techniques that detect and crash the translation process, often detected at compilation time. To deal with such problems, we use a well documented and widely used open source utility *dex2jar*. So far we have not encountered this issue, but it remains a possibility.

2. Signature verification that detects changes in the developer's original signature at runtime. Such an app may not necessarily crash but will fail to either render its activity or connect to its server. We have encountered some of these apps, which are mostly banking applications. Support for such applications is outside the scope of our research.

Chapter 8

Conclusions

8.1 Summary

This research work has focused on developing new techniques for static and hybrid malware analysis. It ends with a user-level privacy policy enforcement system that can help protect user data on the device which are otherwise not protected by the current Android system.

We have presented a novel resilient approach for statically detecting Android malware variants based on three-level similarity matching. This system generates signatures for known malicious code as a function of the normalized opcode sequence found in *sensitive functional modules* and the permissions app requests. Malware belonging to the same family often reuses considerable portions of their codebase and possesses common behavioral characteristics. Permissions requested by an application gives a hint of what the resulting behavior might likely be. Thus the combination of these two distinctive features creates a unique and robust signature for known malware.

The result of our analyses illustrated that we can correctly detect and categorize malware variants with an F-measure of 98.9% and our system is resilient to some complex obfuscation schemes, such as reflection, name and string encryption, junk code insertion, and code reordering. In comparison with current state-of-the-art

Android malware detection tools, including both commercial antivirus and research tools, *OpSeq*'s outperformed them with an average of 35% detection ratio.

We have also supplement our static analyzer with a hybrid Android app analysis system that can detect new malware samples. The system, called *AspectDroid*, provides an efficient and flexible alternative for detecting suspicious and illicit behavior independent of Android runtime and or system release. Our goal is to ease analysis and avoid the numerous problems associated with porting between versions and/or building customized device kernel. It comprises a primary component called the instrumentation engine and the secondary components - helper and automated testing modules.

The instrumentation engine which is at the heart of *AspectDroid* is designed to achieve three main objectives: data flow analysis, resource abuse tracing, analytics of suspicious behavior like native code, and reflective call invocation. *AspectDroid* leverages on AspectJ instrumentation framework to inject monitoring code. The instrumented app is then executed dynamically to trace and log runtime activities at specific joinpoints. It also has the capability to instrument runtime classes for further analysis, thus increasing code coverage. We have demonstrated that *AspectDroid* can achieve up to 94.68% F-score accuracy in detecting data leaks. Further analysis of 100 malware families for the Drebin dataset and 100 apps from Google Play showed our system can effectively analyze a diverse set of apps including stealthy malware with very minimal CPU and memory overhead.

Finally, this research work also presents a privacy policy enforcement technique via fine-grain access control on the SQLite database. The system called *priVy* is a user-centric approach to enforcing object level privilege on Android native providers. Currently, database objects are not treated differently from their main source, meaning when access is granted to the SQLite database file, that access

extends to all the objects encapsulated within it. The native databases contain very important sensitive data that should not be lumped together as a single entity, hence our motivation to segregate their access control. Our system *priVy* is designed to guarantee user's privacy is secured in an accessible and highly usable way. It does not require operating system extensions nor does it tamper with the framework code, making it a much more practical solution than its contemporaries like SE-SQLite.

priVy leverages static bytecode instrumentation to weave in controlling code in database CRUD functions. The controller stub ensures that only user-approved schema, column, and/or entities are accessed by an instrumented application. When these CRUD operations are intercepted, the attached stub performs access level verification, query-rewriting where necessary, and proceeds with the function execution. It also performs database auditing when the attached app accesses any of the encapsulated objects. Our evaluation results demonstrated *priVy* incurs a minimal overhead of 15 seconds instrumentation time and a very negligible execution time overhead.

8.2 Future Work

One important area to investigate further in our malware/app analyzers - *OpSeq* and *AspectDroid* is the analysis of native code. Both of these systems are designed to process only the Dalvik instructions. As part of our future work, we aim to extend the static analyzer to parse and create signatures for the native code. We also intend to include a debugging mechanism in a future revision of *AspectDroid* which gets started via the bytecode rewriting architecture. During native code execution, a debugger can be started with the ID of the new process to collect lower level syscalls made by the native code.

Another area of significant interest in *AspectDroid* is working to improve the

automated testing module such that all control flow paths are forced to execute. As we have seen from the analysis result of *AspectDroid*, its taint tracking is limited to explicit data exfiltration, thus, via code refactoring we intend to inject simple methods after arithmetic and conditional instructions that take the preceding instructions' parameters. Then jointpoints will be created for the new method call at weaving time. This will take care of the possible mis-propagation thereby improving our data flow analysis.

Bibliography

[1] Android Version Market Share 2016. https://www.statista.com/statistics/271774/share-of-android-platforms-on-mobile-devices-with-android-os/. [Online; accessed 05-September-2016].

[2] Apache HTTP SERVER PROJECT. [Online; accessed 11-December-2015].

[3] F-Secure Labs H2 2013 Threat Report. [Online; accessed 30-September 2014].

[4] Global Mobile OS Market Share 2009-2016, By Quarter. [Online; accessed 11-September 2016].

[5] Google Play. [Online; accessed 04-Jan 2013].

[6] Industry Leaders Announce Open Platform for Mobile Devices. http://www.openhandsetalliance.com/press_110507.html. [Online; accessed 07-June-2014].

[7] VirusTotal - online malware scan service. [Online; accessed 26-February-2015].

[8] Forsafe Mobile Security - Android Anserver, 2012. [Online; accessed 04-June-2015].

[9] Apache Commons - Common Lang, 2015. [Online; accessed 30-August-2015].

[10] Eclipse - AspectJ Compiler, 2015. [Online; accessed 26-August-2015].

[11] Ali-Gombe, A., Ahmed, I., Richard III, G. G., and Roussev, V. Opseq: Android malware fingerprinting. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop* (2015), ACM, p. 7.

[12] Ali-Gombe, A., Ahmed, I., Richard III, G. G., and Roussev, V. Aspectdroid: Android app analysis system. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy* (2016), ACM, pp. 145–147.

[13] Ali-Gombe, A., Richard III, G. G., Ahmed, I., and Roussev, V. Don?t touch that column: Portable, fine-grained access control for android?s native content providers. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks* (2016), ACM.

[14] Android Studio. Android developers, 2015. [Online; accessed 11-August-2015].

[15] APIMonitor. Installation and usage of DroidBox APIMonitor, 2012. [Online; accessed 14-Dec-2015].

[16] Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., Rieck, K., and Siemens, C. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)* (2014).

[17] at the European Center for Security, S. S. E., and by Design EC SPRIDE, P. DroidBench Benchmarks Project, 2015. [Online; accessed 02-May-2016].

[18] Backes, M., Gerling, S., Hammer, C., Maffei, M., and von Styp-Rekowsky, P. Appguard–enforcing user requirements on android apps. In

Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2013, pp. 543–548.

[19] Bartel, A., Klein, J., Monperrus, M., Allix, K., and Le Traon, Y. Improving privacy on android smartphones through in-vivo bytecode instrumentation. Tech. Rep. 978-2-87971-111-9, uni. lu, 2012.

[20] Beresford, A. R., Rice, A., Skehin, N., and Sohan, R. Mockdroid: Trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications* (New York, NY, USA, 2011), HotMobile '11, ACM, pp. 49–54.

[21] Bianchi, A., Fratantonio, Y., Kruegel, C., and Vigna, G. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices* (New York, NY, USA, 2015), SPSM '15, ACM, pp. 27–38.

[22] Blasing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S. A., and Albayrak, S. An android application sandbox system for suspicious software detection. In *Malicious and unwanted software (MALWARE), 2010 5th international conference on* (2010), IEEE, pp. 55–62.

[23] Bob, P. Dex2jar, 2014. [Online; accessed 13-December-2014].

[24] Bruening, Derek Zhao, Q., and Amarasinghe, S. Transparent dynamic instrumentation. In *International Conference on Virtual Execution Environments* (March 2012), VEE-12.

[25] Buck, B., and Hollingsworth, J. K. An api for runtime code patching. *Int. J. High Perform. Comput. Appl. 14*, 4 (Nov. 2000), 317–329.

[26] Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. Crowdroid: Behavior-based malware detection system for android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (2011), SPSM '11, pp. 15–26.

[27] Cai, Z., and Yap, R. H. Inferring the detection logic and evaluating the effectiveness of android anti-virus apps. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy* (New York, NY, USA, 2016), CODASPY '16, ACM, pp. 172–182.

[28] Crussell, J., Gibler, C., and Chen, H. Attack of the clones: Detecting cloned applications on android markets. In *Computer Security ESORICS 2012*, vol. 7459 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 37–54.

[29] Davis, B., and Chen, H. Retroskeleton: Retrofitting android apps. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2013), MobiSys '13, ACM, pp. 181–192.

[30] Deshotels, L., Notani, V., and Lakhotia, A. Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014* (2014), PPREW'14, pp. 3:1–3:12.

[31] DroidBox. Droidbox - Android Application Sandbox, 2011. [Online; accessed 01-July-2015].

[32] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: An information-flow tracking system for

realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), OSDI'10.

[33] Enck, W., Ongtang, M., and McDaniel, P. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009), CCS '09, pp. 235–245.

[34] Enck, W., Ongtang, M., and McDaniel, P. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009), CCS '09, pp. 235–245.

[35] Falcone, Y., Currea, S., and Jaber, M. Runtime verification and enforcement for android applications with rv-droid. In *Runtime Verification*, vol. 7687 of *Lecture Notes in Computer Science*. 2013, pp. 88–95.

[36] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), CCS '11, pp. 627–638.

[37] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), CCS '11, pp. 627–638.

[38] Feng, Y., Anand, S., Dillig, I., and Aiken, A. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), FSE 2014, pp. 576–587.

[39] Feng, Y., Anand, S., Dillig, I., and Aiken, A. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of*

the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (2014), FSE 2014, pp. 576–587.

[40] Ferreira, D., Kostakos, V., Beresford, A. R., Lindqvist, J., and Dey, A. K. Securacy: An empirical investigation of android applications' network usage, privacy and security. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks* (New York, NY, USA, 2015), WiSec '15, ACM, pp. 11:1–11:11.

[41] Fuchs, A. P., Chaudhuri, A., and Foster, J. S. Scandroid: Automated security certification of android applications. Tech. rep., University of Maryland, 2009.

[42] Gibler, C., Crussell, J., Erickson, J., and Chen, H. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Trust and Trustworthy Computing*, vol. 7344 of *Lecture Notes in Computer Science*. 2012, pp. 291–307.

[43] Grace, M., Zhou, Y., Zhang, Q., Zou, S., and Jiang, X. Riskranker: Scalable and accurate zero-day android malware detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2012), MobiSys '12, pp. 281–294.

[44] Hanna, S., Huang, L., Wu, E., Li, S., Chen, C., and Song, D. Juxtapp: A scalable system for detecting code reuse among android applications. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2013), DIMVA'12, pp. 62–81.

[45] Heffner, K., and Collberg, C. The obfuscation executive. In *Information Security*, vol. 3225 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004, pp. 428–440.

[46] Heimann, J., and Needham, P. White paper :the virtual private database in oracle9ir2 - understanding oracle9i security for service providers. Tech. rep., Oracle Corporation, 2002.

[47] Heuser, S., Nadkarni, A., Enck, W., and Sadeghi, A.-R. Asm: A programmable interface for extending android security. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 1005–1019.

[48] Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 639–652.

[49] Huang, H., Zhu, S., Liu, P., and Wu, D. A framework for evaluating mobile app repackaging detection algorithms. In *Trust and Trustworthy Computing*, vol. 7904 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 169–186.

[50] Jeon, J., Micinski, K. K., Vaughan, J. A., Fogel, A., Reddy, N., Foster, J. S., and Millstein, T. Dr. android and mr. hide: Fine-grained permissions in android applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (New York, NY, USA, 2012), SPSM '12, ACM, pp. 3–14.

[51] Karami, M., Elsabagh, M., Najafiborazjani, P., and Stavrou, A. Behavioral analysis of android applications using automated instrumentation. In *Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on* (2013), IEEE, pp. 182–187.

[52] Karbab, E. B., Debbabi, M., and Mouheb, D. Fingerprinting android packaging: Generating dnas for malware detection. *Digital Investigation 18* (2016), S33–S45.

[53] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. Getting started with aspectj. *Commun. ACM 44*, 10 (Oct. 2001), 59–65.

[54] Kiczales, G., Lamping, J., Lopes, C., Hugunin, J., Hilsdale, E., and Boyapati, C. Aspect-Oriented Programming, Oct. 15 2002. US Patent 6,467,086.

[55] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In *ECOOP'97 ? Object-Oriented Programming*, vol. 1241 of *Lecture Notes in Computer Science*. 1997, pp. 220–242.

[56] Kohei, K. Security enhanced postgresql, 2013.

[57] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2005), PLDI '05, pp. 190–200.

[58] Maiorca, D., Ariu, D., Corona, I., Aresu, M., and Giacinto, G. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security 51* (2015), 16–31.

[59] Mutti, S., Bacis, E., and Paraboschi, S. Sesqlite: Security enhanced sqlite: Mandatory access control for android databases. In *Proceedings of the 31st Annual Computer Security Applications Conference* (New York, NY, USA, 2015), ACSAC 2015, ACM, pp. 411–420.

[60] Nauman, M., Khan, S., and Zhang, X. Apex: Extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2010), ASIACCS '10, ACM, pp. 328–332.

[61] Pearce, P., Felt, A. P., Nunez, G., and Wagner, D. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2012), ASIACCS '12, ACM, pp. 71–72.

[62] Rastogi, V., Chen, Y., and Enck, W. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy* (2013), CODASPY '13, pp. 209–220.

[63] Rastogi, V., Chen, Y., and Jiang, X. Catch me if you can: Evaluating android anti-malware against transformation attacks. *Information Forensics and Security, IEEE Transactions on 9*, 1 (Jan 2014), 99–108.

[64] Russello, G., Jimenez, A. B., Naderi, H., and van der Mark, W. Firedroid: Hardening security in almost-stock android. In *Proceedings of the*

29th Annual Computer Security Applications Conference (New York, NY, USA, 2013), ACSAC '13, ACM, pp. 319–328.

[65] Sadeghi, A., Bagheri, H., Garcia, J., et al. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering* (2016).

[66] Santos, I., Brezo, F., Nieves, J., Penya, Y., Sanz, B., Laorden, C., and Bringas, P. Idea: Opcode-sequence-based malware detection. In *Engineering Secure Software and Systems*, vol. 5965 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 35–43.

[67] Schmidt, A.-D., Bye, R., Schmidt, H.-G., Clausen, J., Kiraz, O., Yuksel, K., Camtepe, S., and Albayrak, S. Static analysis of executables for collaborative malware detection on android. In *Communications, 2009. ICC '09. IEEE International Conference on* (June 2009), pp. 1–5.

[68] Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., and Weippl, E. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Comput. Surv. 49*, 1 (Apr. 2016), 4:1–4:37.

[69] Shekhar, S., Dietz, M., and Wallach, D. S. Adsplit: Separating smartphone advertising from applications. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (Bellevue, WA, 2012), USENIX, pp. 553–567.

[70] Smalley, S., and Craig, R. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS* (2013), vol. 310, pp. 20–38.

[71] Software, G. GDATA Mobile Malware Report: Q4/2015, 2016. [Online; accessed 3-October-2016].

[72] Stonebraker, M., and Wong, E. Access control in a relational data base management system by query modification. In *Proceedings of the 1974 annual conference-Volume 1* (1974), ACM, pp. 180–186.

[73] Team, A. The aspectj tm programming guide, 2002-2003. [Online; accessed 21-Jan-2015].

[74] Trend-Micro. Android Malware: How Worried Should You Be?, 2012. [Online; accessed 4-October-2016].

[75] Tustison, N. J., and Gee, J. C. Introducing dice, jaccard, and other label overlap measures to itk. *The Insight Journal* (July-December 2009), 1–4.

[76] Vasudevan, A., and Yerraballi, R. Spike: Engineering malware analysis tools using unobtrusive binary-instrumentation. In *Proceedings of the 29th Australasian Computer Science Conference - Volume 48* (2006), ACSC '06, pp. 311–320.

[77] Wang, Y., Hariharan, S., Zhao, C., Liu, J., and Du, W. Compac: Enforce component-level access control in android. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy* (New York, NY, USA, 2014), CODASPY '14, ACM, pp. 25–36.

[78] Weichselbaum, L., Neugschwandtner, M., Lindorfer, M., Fratantonio, Y., van der Veen, V., and Platzer, C. Andrubis: Android malware under the magnifying glass. *Vienna University of Technology, Tech. Rep. TRISECLAB-0414-001* (2014).

[79] Wu, D.-J., Mao, C.-H., Wei, T.-E., Lee, H.-M., and Wu, K.-P. Droidmat: Android malware detection through manifest and api calls tracing. In

Proceedings of the 2012 Seventh Asia Joint Conference on Information Security (2012), ASIAJCIS '12, pp. 62–69.

[80] Wu, D.-J., Mao, C.-H., Wei, T.-E., Lee, H.-M., and Wu, K.-P. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on* (Aug 2012), pp. 62–69.

[81] Xu, R., Saïdi, H., and Anderson, R. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 27–27.

[82] Yan, L.-K., and Yin, H. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX Security Symposium* (2012), pp. 569–584.

[83] Zhang, M., and Yin, H. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium* (2014), NDSS 2014.

[84] Zhang, M., and Yin, H. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security* (2014), ASIA CCS '14, pp. 259–270.

[85] Zhang, M., and Yin, H. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *Proceedings of the*

9th ACM Symposium on Information, Computer and Communications Security (New York, NY, USA, 2014), ASIA CCS '14, ACM, pp. 259–270.

[86] Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X. S., and Zang, B. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (2013), CCS '13, pp. 611–622.

[87] Zheng, M., Lee, P., and Lui, J. Adam: An automatic and extensible platform to stress test android anti-virus systems. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, vol. 7591 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 82–101.

[88] Zheng, M., Sun, M., and Lui, J. C. S. Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware. In *Proceedings of the 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications* (2013), TRUSTCOM '13, pp. 163–171.

[89] Zhou, W., Zhou, Y., Grace, M., Jiang, X., and Zou, S. Fast, scalable detection of "piggybacked" mobile applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy* (2013), CODASPY '13, pp. 185–196.

[90] Zhou, W., Zhou, Y., Jiang, X., and Ning, P. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy* (2012), CODASPY '12, pp. 317–326.

[91] Zhou, Y., and Jiang, X. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on* (May 2012), pp. 95–109.

[92] Zhou, Y., and Jiang, X. Dissecting android malware: Characterization and evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy* (2012), Oakland '12.

[93] Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the Network and Distributed System Security Symposium* (2012), NDSS2012.

[94] Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 19th Network and Distributed System Security Symposium* (2012), NDSS '12.

[95] Zhou, Y., Zhang, X., Jiang, X., and Freeh, V. W. Taming information-stealing smartphone applications (on android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing* (Berlin, Heidelberg, 2011), TRUST'11, Springer-Verlag, pp. 93–107.

Vita

The author was born and raised in the city of Gombe, Northeastern Nigeria. She earned a B.Sc. in Computer Science from the University of Abuja in 2005 and an MBA degree from Bayero University, Kano in 2011. She joined the University of New Orleans graduate program in 2011, where she earned an M.S degree in Computer Science in 2012. She continued on to pursue Ph.D. in Engineering and Applied Science with a concentration in Computer Science and research emphasis in cyber security and digital forensics.