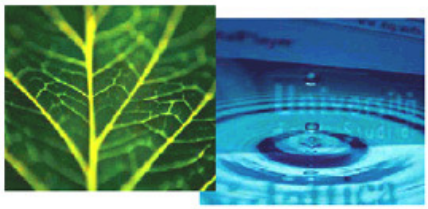
PhD Dissertation



International Doctoral School in
Information and Communication Technologies (ICT)
University of Trento, Italy

Mobile Application Security in the Presence of Dynamic Code Updates

Maqsood Ahmad

SUBMITTED TO THE DEPARTMENT OF
INFORMATION ENGINEERING AND COMPUTER SCIENCE (DISI)
IN THE PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Advisor:	Associate Prof. Dr. Bruno Crispo, University of Trento, Italy
Examiners:	Prof. Luigi Mancini, University of Rome "La Sapienza", Italy
	Prof. Fabio Massacci, University of Trento, Italy
	Dr. Giovanni Russello, The University of Auckland, New Zealand

April 2017

To my Mom, Dad and my late Grandmother!

Abstract

The increasing number of repeated malware penetrations into official mobile app markets poses a high security threat to the confidentiality and privacy of end users' personal and sensitive information. Protecting end user devices from falling victims to adversarial apps presents a technical and research challenge for security researchers/engineers in academia and industry. Despite the security practices and analysis checks deployed at app markets, malware sneak through the defenses and infect user devices. The evolution of malware has seen it become sophisticated and dynamically changing software usually disguised as legitimate apps. Use of highly advanced evasive techniques, such as encrypted code, obfuscation and dynamic code updates, etc., are common practices found in novel malware. With evasive usage of dynamic code updates, a malware pretending as benign app bypasses analysis checks and reveals its malicious functionality only when installed on a user's device.

This dissertation provides a thorough study on the use and the usage manner of dynamic code updates in Android apps. Moreover, we propose a hybrid analysis approach, StaDART, that interleaves static and dynamic analysis to cover the inherent shortcomings of static analysis techniques to analyze apps in the presence of dynamic code updates. Our evaluation results on real world apps demonstrate the effectiveness of StaDART. However, typically dynamic analysis, and hybrid analysis too for that matter, brings the problem of stimulating the app's behavior which is a non-trivial challenge for automated analysis tools.

To this end, we propose a backward slicing based targeted inter component code paths execution technique, TeICC. TeICC leverages a backward slicing mechanism to extract code paths starting from a target point in the app. It makes use of a system dependency graph to extract code paths that involve inter component communication. The extracted code paths are then instrumented and executed inside the app context to capture sensitive dynamic behavior, resolve dynamic code updates and obfuscation. Our evaluation of TeICC shows that it can be effectively used for targeted execution of inter component code paths in obfuscated Android apps. Also, still not ruling out the possibility of adversaries reaching the user devices, we propose an on-phone API hooking

based app introspection mechanism, AppIntrospector, that can be used to analyze, detect and prevent runtime exploitation of app vulnerabilities that involve dynamic code updates.

Keywords: Android Security, Malware Analysis, Dynamic Code Updates

Acknowledgements

I would like to extend my heartfelt gratitude to my supervisor, Prof. Bruno Crispo, for his guidance, support and advice on academic research.

I would also like to thank the co-authors of our publications for their untiring and honest effort to make our collaborative research possible.

Last but not least, I would always be indebted to my family and friends for their best wishes, prayers and unconditional support.

Maqsood Ahmad
Trento, Italy
April 2017

Contents

List of Tables

List of Figures

Listings

List of Algorithms

List of Acronyms

OS	Operating System
DCL	Dynamic Code/Class Loading
API	Application Program Interface
ICC	Inter Component Communication
SAAF	Static Android Analysis Framework
AOSP	Android Open Source Project
UID	User Identification
GID	Group Identification
APK	Android Application Package
SMS	Short Message Service
ART	Android RunTime
AOT	Ahead Of Time
DEX	Dalvik Executable
DVM	Dalvik Virtual Machine
SDK	Software Development Kit
JSON	JavaScript Object Notation
XML	Extensible Markup Language
MCG	Method Call Graph
MOI	Method Of Interest
TeICC	Targeted Execution Of Internet Component Communication
POI	Point Of Interest
SDG	System Dependency Graph
JNI	Java Native Interface
ELF	Executable and Linking Format
ARM	Acorn RISK Machine
ADBI	Android Dynamic Binary Instrumentation
LOC	Lines Of Code

Chapter 1

Introduction

In recent years, technological advancements in every field of life have been observed, adopted and have now become integral parts of our daily lives. The field of communication has revolutionized the way man lives and the world has effectively become a global village. The advent of the Internet in conjunction with modern day smart handheld devices, i.e., smartphones and tablets, etc., has brought the world to a user's fingertips.

Mobile devices are more pervasive and ubiquitous than ever before. Traditional computing has transformed, and rightly so, the number of active mobile devices surpassed the world population in 2014 [54]. Smartphone vendors, framework providers, and mobile application developers contribute to this mobile computing ecosystem. With computing gone mobile, users are empowered to perform office work, undertake banking transactions, remain active on their social networks, and much more, on the move. As smartphones provide a wide range of services of varied nature, they access, store, process, send and receive users' personal information, which if compromised can harm the user financially, socially and psychologically. The nature and the sensitivity of the information handled by a smartphone requires protection from both inside and outside adversarial access.

1.1 Motivation and Problem Statement

Mobile devices are shipped with one of the many operating systems (OS) available; proprietary or open source depending upon the vendor. Typically, vendors allow users to extend the functionality of their devices by downloading applications (shortly apps) from various app markets. Although, this feature is more or less common to a varying degree in all the mobile platforms, in this text we would mainly focus on the Android platform.

Since its introduction in 2008, Android has emerged as the leading operating system used for handheld devices. Dominating the smartphone market for the last few years, it reached 86.2% of the smartphone market share in 2016 [97]. During a 30-day active user

monitoring in September 2015, Google confirmed that Android has 1.4 billion active users globally [50]. Android is an open source operating system with open architecture where apps are published at numerous third-party market along with its official app market, i.e., Google Play store [25]. Google Play store surpassed 2.6 million apps in December 2016 [92].

These are some of the stats that provide an idea about the popularity of the Android framework. The pervasiveness, popularity and capability of mobile devices to collect and store users' private and sensitive information makes them very attractive for malware developers too. Hence, the number of mobile malware samples increases as the days go by. Among others, here too, Android based devices are in the firing line and recent reports suggest that 99% of all the mobile malware are targeted towards Android based devices [60].

To counter mobile malware, app markets deploy different kind of vetting mechanisms, e.g., Bouncer at the Google Play store. Generally, apps are analyzed before being made available for the users at the app markets. Apps failing the analysis check are rejected and Google claimed to have reduced the number of malware by 40% the year immediately following the introduction of Bouncer [81].

Researchers in both academia and industry contribute to strengthen the analysis process and detect malicious apps. Analysis techniques are broadly divided into static and dynamic analysis, both having their own advantages and disadvantages. As analysis techniques evolve, malware developers also come up with new ways of evading these analysis tools and infecting user devices. With the passage of time, malicious apps have also evolved and a variety of evasion techniques, such as anti-emulation, anti-debugging, code obfuscation, evasive use of dynamic code updates, etc., are being widely used to thwart analysis tools.

1.1.1 App Analysis in the Presence of Dynamic Code Updates

Mobile app developers use dynamic code updates to extend their apps' functionality at runtime. The use of techniques, such as dynamic code loading (DCL) and reflection, is getting mainstream in mobile apps in the quest to develop feature-rich and adaptive solutions to meet the needs of providing sophisticated user experiences. On the other hand, the inherent nature of these techniques makes apps analysis a challenging task, and therefore, they are often used by malware developers to evade analysis tools deployed at the app markets.

To make the analysis even more challenging, these techniques are used in conjunction with other analysis evasion techniques, such as code encryption, parameter encryption and dynamic provisioning of the parameters used, to name a few. State-of-the-art research

on static analysis finds it extremely daunting to properly analyze code in the presence of dynamic code updates [95]. Indeed, the dynamic nature of DCL and reflection do not allow static analysis tools to properly infer the behavior of an app and, as a result, make them clueless when detecting malicious functionality.

Moreover, most of the static analysis tools are based on the assumption that the code base of an app remains the same and does not change dynamically [41,57,74]. Obviously, this assumption is far from reality and a simplification made to cover the limitations of the current analysis tools. On the other hand, researchers have demonstrated that dynamic code updates can be used to bypass the analysis check at app markets [51,90,103]. Nevertheless, a number of static analysis approaches are found in literature to analyze apps in the presence of dynamic code updates [72, 80, 106] . However, static analysis can always be hindered with app features of dynamic nature. At the same time, solutions found in the literature for enhancing static analyzers of Java code to analyze dynamic code updates rely on loadtime code instrumentation which is not available for Android, and therefore, these solutions cannot be directly applied to Android apps [48]. Similarly, solutions based on instrumentation of apps rely on repackaging the app, which breaks the app signature and allows malicious apps to conceal their malicious functionality.

A research challenge in this regard is to study what makes dynamic code updates so hard to analyze when only static analysis is used. Moreover, as it can be concluded from the above discussion that the possibility of dynamic analysis coupled with static analysis can make the job of an analyst simpler, a challenge would be to design a hybrid approach that can be deployed to analyze mobile apps in the presence of dynamic code updates.

1.1.2 Ensuring Execution of Targeted Code Paths during Dynamic Analysis

To cover the shortcomings of static analysis and resolve analysis issues created by features like dynamic code updates, dynamic analysis is often the go-to solution. On the one hand, dynamic analysis provides solution to these problems. On the other hand, it requires test cases which could execute a major/required portion of the code which leads to another challenging problem. Execution of certain code paths in mobile apps depends upon a combination of various user/system events. Generally, it is hard to predict inputs which can stimulate the required behavior in these apps. This feature of mobile apps is widely used by malware developers to conceal malicious functionality.

State-of-the-art research shows a number of triggering solutions, ranging from black-box to grey-box, for Android apps with a varied degree of code coverage [83,85,94,96,113]. Code coverage is a well-known limitation of dynamic analysis approaches. However, for the purpose of security analysis rather than testing, it is required to stimulate/reach only specific points of interest in the code rather than stimulating all the code paths in an

app. In literature, researchers have focused mainly on providing inputs to make an app follow a specific path. Providing the exact inputs and environment becomes very hard as different apps may require different execution environments. Moreover, not all inputs can be predicted statically, because of obfuscation or other hiding techniques. In addition, existing target triggering solutions, such as [93] and [43], are generally limited to code execution inside a signal component of the app or do not handle the dynamic code updates well.

The key research challenge in this direction is to design an automated and effective solution for dynamic targeted execution of Android apps. Moreover, the triggering mechanism should also cover inter component communication as it constructs an essential part of Android apps behavior and analysis would be incomplete without it.

1.2 Research Contributions

The main research aim of this thesis is to move forward the state-of-the-art app analysis research in the presence of dynamic code updates. Here we briefly discuss the main research contributions of this work.

1.2.1 Reflection-Bench and Empirical Analysis on Real World Apps

In order to understand what causes the difficulty in analyzing apps in the presence of dynamic code updates, it is important to perform a study on the manner in which dynamic code updates are used in real world apps. Moreover, it is essential to have a set of benchmark apps that could be used by the research community to test the capability of their static analysis tools to handle dynamic code updates.

- We design and develop reflection-bench, a set of benchmark apps that use reflection to conceal information leakage, and use it to test some of the state-of-the-art static analysis tools. We plan to make reflection-bench public so that it can be used by other researchers to test the effectiveness of their analysis tools in the presence of reflection.

- We develop an automated static analysis tool which can perform analysis on Android apps, detect information flow between given source and sink APIs, and produce statistics about the presence of such information flow paths between source/sink APIs in individual apps as well as the whole market.

- We collect and analyze a dataset of real world apps containing 16,528 benign and 3,645 malicious apps in order to investigate the sources of the parameters used in

reflection/DCL APIs. To the best of our knowledge, this is first study focusing on the sources of the parameters of reflection/DCL APIs. The analysis results would help in understanding the behavior of apps that use dynamic code updates and designing more effective analysis procedures and policies to detect malicious apps in the market.

1.2.2 Handling Dynamic Code Updates using a Combination of Static and Dynamic analysis

We propose a hybrid approach combining static and dynamic analysis to cover for the inherent inability of static analysis to deal with dynamic code updates in Android apps.

- We propose, design and implement StaDART, a system that interleaves static and dynamic analysis in order to reveal the hidden/updated behavior. By utilizing ArtDroid, we avoid modifications to the Android framework and make it largely framework independent. StaDART downloads and makes available for analysis the code loaded dynamically, and is able to resolve the targets of reflective calls complementing app's method call graph with the obtained information. Therefore, StaDART can be used in conjunction with other static analyzers to make their analysis more precise.
- We integrate StaDART with DroidBot to make it fully automated and to ease the evaluation. Moreover, we analyze a dataset of 2,000 real world apps (1,000 benign and 1,000 malicious). Our analysis results show the effectiveness of StaDART in revealing behavior which is otherwise hidden to static analysis tools.
- We plan to release our tool as open-source to drive the research on app analysis in the presence of dynamic code updates.

1.2.3 Triggering Problem: Targeted Execution and Runtime Analysis

One of the main challenges associated with solutions based on dynamic analysis is the triggering problem, i.e., apps require certain user/system events to follow specific paths. In this direction, the key research goal is to advance the state-of-the-art research in triggering mechanisms and design an intelligent and scalable solution for execution of targeted inter component code paths in Android apps. The main contributions in this regard are enlisted here.

- We extend the backward slicing mechanism to support inter component communication (ICC), *i.e.,* extract slices across multiple components. Moreover, we enhance

a backward slicing tool, SAAF, to perform data flow analysis with context-, path- and object-sensitivity.

- Targeted execution of the extracted inter-component slices without modification to the Android framework.
- We design and implement a hybrid analysis system based on static data-flow analysis and dynamic execution on a real-world device for improved analysis of obfuscated apps featuring dynamic code updates.

Keeping in mind the severity of the problem and the increasing use of anti-analysis techniques in recent malware that infiltrated the official Google market, we go a step ahead and aim to move part of the analysis from the analysis environment to real user devices. Key contributions in this regard are:

- We introduce a paradigm shift by moving part of the analysis of Android apps from an artificial analysis environment to end user devices. Careful design, implementation and deployment of this type of solutions could pave the path to solving problems like de-obfuscation, triggering and avoiding vulnerability exploitation at runtime.
- We investigate and provide a theoretical overview of some of the well known hooking tools in security research community and techniques found in the literature.
- We design and implement an app introspection mechanism that leverages API hooking to analyze, detect and prevent malicious activities that involve dynamic code updates. Our analysis solution relies on minimal collaboration from app developers and does not require any modification to the Android framework or rooting the device.

1.3 Structure of the Thesis

This dissertation is organized as follows:

Chapter 2 provides a brief background of Android security. It describes the basics of dynamic code update techniques and their usage motivations in Android apps. Furthermore, it makes a case of the prevalence of dynamic code updates by presenting the results of our analysis on the use of dynamic code updates in real world Android apps, both benign and malicious.

Chapter 3 goes deeper into the problem by presenting an analysis on the manner in which real world apps use dynamic code updates, i.e., how various dynamic code

update APIs get their parameters. It also presents a set of benchmark apps and use them to demonstrate the inability of static analysis tools to analyze apps in the presence of dynamic code updates.

Chapter 4 provides the design, details of the implementation and evaluation results of our proposed hybrid solution based on interleaving static and dynamic analysis techniques to handle dynamic code updates in Android apps.

Chapter 5 presents the design, implementation and evaluation results of our proposed backward slicing based mechanism for targeted execution of inter component communication in Android apps. It also discusses the enhancement made to the backward slicing mechanism and the tool used for backward slicing.

Chapter 6 shifts the analysis from an artificial analysis environment to real user devices. It presents a runtime analysis approach to avoid exploitation of benign, but vulnerable, apps that involve dynamic code updates. It relies on an API hooking based app introspection mechanism that analyzes dynamic code updates as they appear.

Chapter 7 draws the main conclusions of this research work and discusses the possible future directions.

Chapter 2

Background

Android is globally accepted as the most widely used operating system for handheld devices. It supports a wide range of devices, such as smartphones, watches, smart TVs, etc. Android provides an open platform for the developers as well the device manufacturers. As a result, device manufactures ship customized variants of Android with their devices. On the other hand, any third-party developer can develop applications to extend the Android platform. These apps are usually published to app stores from where users can download, purchase and install them. This openness of the platform makes it more attractive for developers and as a result, the number of apps developed for Android based devices are always on the rise. However, it also attracts adversaries to develop malicious apps and infect user devices. Hence, the number of malware directed towards Android based devices is also the highest among other peer platforms.

To counter the problem of malware penetration into the Android ecosystem, Android incorporates a wide of range of security features. Android security team collaborates with developers, device manufacturers and researchers to ensure that the best security practices are followed and the Android platform/apps are free of bugs and vulnerabilities. The goal of these security features and practices is to stop malware reach a user device and, in case a malware infects a user device, minimize the damage, i.e., to protect a user's private data and resources. Android uses a layered security architecture to counter malware that targets various levels of the Android stack. In order to protect sensitive data and resources from malicious apps, Android relies on a Unix-like sandboxing model and app permissions. Moreover, Android uses an app scanning process at the Google Play store which blocks apps that can be harmful for user devices.

Despite the robust security architecture of the Android platform, malware developers still find ways to bypass the scanning process and infect user devices. Hence, Android ecosystem has malicious apps in abundance. There has been an ongoing competition between the good and the bad since the day existed and the area of Android security

is no different. Researchers/developers in both industry and academia propose the best security practices and tools to avoid being victims of malicious apps. On the other hand, malware developers use an array of techniques to evade analysis tools and infect user devices.

2.1 Android Security

An open architecture resulted in Android being the leading OS in smarphone community. Android is open source where the source code is provided as part of the Android Open Source Project (AOSP) for developers, researchers and device manufacturers [39]. The open architecture allows for a repetitive and rigorous research-attack-fix cycle, and as a result a security hardened framework which is a basic step towards a vigorous Android ecosystem. We briefly discuss some of the key security features that Android incorporates to ensure app security.

2.1.1 App Sandboxing

The heart of Android platform is based on a Linux kernel. Over the years, Linux kernel has been exposed to rigorous research and testing. As a result, it has become secure and mostly free of bugs. Therefore, it is widely adopted in both academia as well as industry. Based on the Linux kernel, Android inherits some key security features which help in running each app in an individual sandbox. The main purpose of app sandboxing is to prevent harmful apps from damaging other apps on the device as well the Android framework.

Linux provides a user-based permission model to protect one user's resources from the other. Android leverages this user-based permission model to restrict apps to their own sandboxes. Contrary to Linux, however, Android uses a separate user ID (UID) for each app. So, each app run as a separate user in a separate process. This in turn creates a kernel-level sandbox for each app where there are virtual walls between different apps and the Android framework itself. Each process has its own privileges which determines the data and resources that can be accessed by this process. Moreover, processes are assigned group IDs to enforce permissions to access sensitive resources. Consequently, an app can not access the data of another app or resource unless and until explicitly permitted.

Kernel level sandboxing ensures that apps at all levels, i.e., native apps, user apps and system apps, abide by the restrictions imposed upon them by the Android framework. The security model makes sure that, in a properly configured device and without compromising the Linux kernel level security, a harmful app does not damage any of the other apps on the device or the device itself.

2.1.2 Android Permissions

An Android app is delivered to users as an APK file which is a `.zip` archive containing multiple code, resources and configuration files. One of these files, `AndroidManifest.xml`, determines the app's capability. It provides information about the various components of the app, i.e., activities, services, content providers, etc. Listing 2.1 shows an example Manifest file.

Listing 2.1: Example Manifest File

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3      package="com.test" >
4
5      <uses-permission android:name="android.permission.READ_PHONE_STATE" />
6      <uses-permission android:name="android.permission.SEND_SMS" />
7
8      <application
9          android:allowBackup="true"
10         android:icon="@drawable/ic_launcher"
11         android:name="com.test.MyApplication"
12         android:label="@string/app_name" >
13         <activity
14             android:name="com.test.MyActivity"
15             ...
16         </activity>
17     </application>
18
19 </manifest>
```

In the Android sandboxing model, an app is restricted to access limited resources by default. Access to sensitive resources, which if compromised can adversely affect the user or device, is generally provided through higher level framework APIs. These resources are guarded with permissions and an app must declare the required permission in its Manifest file if it has to access the resource. Android framework defines a list of permissions corresponding to various sensitive resources, e.g., accessing camera, network, telephony services, messaging services, etc.

In Listing 2.1, Line 5 and Line 6 represent the declared permissions by the app. The permission `android.permission.SEND_SMS` guards the SMS sending API. Any app without this permission cannot send text messages. The list of permissions declared in the Manifest file of an app are displayed to the user, along with a brief description of what they are used for, at the time of installation. Before Android 6.0, named Marshmallow,

Android used an all accept or reject model where a user had to accept all the permissions in order to proceed with the installation. Starting from Android 6.0, users can selectively allow and/or deny permissions and install the app. Furthermore, a user can later on revoke allowed permissions or allow denied permissions. Current permission model makes app installation more flexible.

Permissions provided by the Android framework are called System permissions. These permissions are known to app developers and they guard standard framework features or resources. In addition, developers can define their own permissions to guard some of the functionality their apps provide to other apps on the device. As a result, only apps having these user-defined permissions can avail the exported functionality. Based on the nature of the resources they guard, permissions are divided into various protection levels.

- **Normal:** Normal permissions are required for apps that require access to data or resources where a user's privacy or operation of other apps are not exposed to high risk. Normal permissions are granted by the system without notifying the user.
- **Dangerous:** In contrast, dangerous permissions guard data or resources where a user's privacy or operation of other apps is at stake. Dangerous permissions are displayed to users and only granted to an app after an explicit user approval. Example of a dangerous permission is `android.permission.SEND_SMS`.
- **Signature:** Signature level permissions are granted by the system only if the declaring and requesting apps are signed with the same signature.
- **SignatureOrSystem:** SignatureOrSystem level permissions are granted by the system to apps which are in the Android system image or in case where declaring and requesting apps are signed with the same signature. Android documentation discourages use of permissions with this protection level.

In most of the cases, a `SecurityException` is raised when an app tries to access data or resources guarded with a permission that is not granted to the app. Permissions allow apps to operate within the boundaries explicitly defined by the user. It enables a user to have control over the app, i.e., revoke permissions or uninstall the app if the app shows malicious behavior.

2.1.3 App Vetting

Android supports an open app market where anyone can develop apps and publish them to app stores. There are a number of third-party app stores along with the standard Google Play store. On the one hand, openness of the app market provides users plenty of

options to extend the functionality of their Android based devices. On the other hand, it makes it hard to distinguish between legitimate apps and malicious apps disguised as benign apps. Hence, the job of security engineers and researchers to maintain a vigorous Android ecosystem has become more difficult.

At the Google Play store, Android introduced an app scanning mechanism named Bouncer in 2012. Bouncer is the vetting process Google Play store uses to detect and block malicious apps. With the introduction of Bouncer, they claimed to have reduced the number of malware by 40% the following year. Nevertheless, proper vetting process at most of the Android app stores is close to non-existent. Therefore, these app stores contribute to a major portion of the malware samples.

The vetting process usually checks apps for malicious contents, looks for suspicious patterns and observes the behavior of the submitted app before publishing it to the app store. Two broad categories used for app analysis are static and dynamic analysis, which analyze the app without and with executing it, respectively. Using such analysis techniques, Google Play store blocks most of the apps with obvious malicious behavior. Therefore, incidents of malicious apps being reported at the Google Play store are a minor portion of all the malware. Higher number of malicious apps found at the third-party app stores are largely due to the lack of app vetting process at these app stores. To ensure security and prevent malicious apps at third-party stores infect users' devices, Android introduced a new security feature in Android 7.0, i.e., Verify Apps. The verify apps feature scans apps downloaded from sources other than Google Play before installation. It also keeps on monitoring the app after installation and warns the user if the app shows malicious behavior.

As security researchers and engineers strive to develop robust security mechanisms to ensure a clean Android ecosystem, malware developers come up with more stealthy ways to go through the security walls and infect user devices. Every new malware sample reveals that malware developers are well equipped to capitalize on every little security bug and loophole that might be there in the Android framework or apps. They make use of a number of evasion techniques which makes it extremely hard for automated analysis tools to understand the behavior of an app. Some of these techniques include code encryption, various forms of obfuscation, reflection and dynamic code loading, anti-emulation and anti-debugging, etc. Since the focus of this dissertation is app analysis in the presence of reflection and dynamic code loading, we elaborate them more in the following section.

2.2 Dynamic Class Loading and Reflection in Android

Dynamic code updates play a very important role in modern day feature-rich and dynamically changing mobile apps. Here, we provide some background of dynamic class loading and reflection, their usage and implementation in Android. Android uses Android Runtime (ART) to run apps and system services. ART uses ahead of time (AOT) compilation using a dex2oat tool to convert DEX files into `.oat` binaries. However, ART is backward compatible with its predecessor Dalvik and can execute apps compiled for the Dalvik Virtual Machine (DVM). Moreover, the dynamic code update functionality, i.e., DCL and reflection, is supported by ART as it was supported by the Dalvik runtime.

2.2.1 Overview of Dynamic Class Loading

DCL provides flexibility to a developer to load classes at runtime to extend the functionality of apps. Similar to Dalvik, ART allows a developer to load additional code obtained from alternative locations at runtime [53]. It allows apps to load `.zip`, `.jar` and `.apk` files containing a valid `classes.dex` file from outside the app code base, such as files stored on the internal storage or downloaded from the network.

Android provides a set of class loaders, arranged in a hierarchical manner, which are used to load classes to memory from internal storage. Every child class loader holds a reference to its parent class loader where the root of the tree is the `BootStrap ClassLoader`, which has a `null` reference to its parent.

A common interface required by all the class loaders is implemented by an `abstract` class named *ClassLoader* whereas other specific class loaders are then derived from *ClassLoader*, such as *DexClassLoader*, *PathClassLoader*, etc. *ClassLoader* provides methods such as `loadClass()`, `findLoadedCalss()` and `defineClass()`, which allows a developer to load a class, search for loaded classes and define a class from a byte sequence at runtime, respectively. Android also provides a class *DexFile* whose methods can be used to load classes directly. However, these methods require a reference to a class loader as an argument.

In case of a class loading request, the current class loader first checks whether the class has already been loaded or not. If it fails to find the class in the list of the loaded classes, it requests its parent class loader to find out if the class has already been loaded. This process continues until the request reaches the root of the tree which tries to find the class. If the root of the tree is unable to find the requested class, a `ClassNotFoundException` is thrown, which propagates back to the initial class loader. This necessarily means that the class has neither been loaded by the current class loader nor by its parents up till the root of the class loaders tree. The current class loader then tries to load the class by

itself. If it fails to load the requested class, the `ClassNotFoundException` is released.

2.2.2 Overview of Reflection

Reflection is the ability of a program to treat its own code as data and manipulate it during execution [47]. Using reflection, an app can reason about and modify its execution state during runtime. The dynamically loaded code is usually accessed using reflection. Android uses the same reflection APIs as used in Java. In the following, the functionality provided by reflection APIs is outlined:

- **Retrieving Class objects:** All of the reflection operations start from `java.lang.Class`. Objects of this class represent all the classes and interfaces in a running app. Classes and interfaces that could be used to obtain reflective information about other classes and objects are provided by the `java.lang.reflect` package. Classes in the `java.lang.reflect` package are usually without any public constructor. However, these classes can be instantiated by calling different methods on *Class*. Based on the information, an object of *Class* can be retrieved in different ways. It is clarified that an instance of *Class* is referred here as object while an instance of the corresponding *Class* object is referred to as 'instance'. If an instance of a *Class* object is available, its *Class* object can be retrieved by calling `getClass()` method on the instance. If the type information of an object is available, the corresponding *Class* object can be retrieved by appending `.class` to the class type (and `.TYPE` for primitive types). A very common way to obtain *Class* objects, however, is to call `Class.forName(className)` where the string `className` represents the name of the *Class* object. Once a *Class* object is retrieved, other related classes can also be retrieved using methods such as `getSuperClass()`, `getClasses()`, `getDeclaredClasses()`, etc.

- **Accessing Members:** Once a *Class* object is retrieved, its members can also be accessed using reflection APIs. These members can be fields, methods or constructors. *Field* objects can be retrieved using `getField(fieldName)`, where the string `fieldName` represents the name of the field, and `getDeclaredFields()`, which retrieves all the declared fields. Similarly, there are APIs to obtain the type information of fields, and obtain and change field values as well. Having a *Class* object, *Constructor* objects of this class can be retrieved using the `getConstructor(Class[] params)`, `getConstructors()`, or `getDeclaredConstructors()`. Similarly, *Method* objects of a retrieved class can be obtained using methods such as `getMethod(methodName, params)`, `getMethods()`, and `getDeclaredMethods()`, which return objects of the

specific *Method* represented by the string `methodName`, all the public methods of the class, and all the declared methods in the class, respectively.

- **Instance creation and Method Invocation:** An instance of a specific class type can be created if the corresponding object of *Class* or *Constructor* is available. A default zero argument constructor of the class can be called using the `newInstance()` method on the *Class* object whereas the constructor with parameters can be called using the `newInstance(params)` method on the *Constructor* object. Both of these methods return instances of the given *Class* object. Similarly, the methods obtained from the *Class* objects can be invoked using the `invoke(objectRef)` method where the string `objectRef` represents a reference to the object on which the method is invoked.

2.2.3 Usage Motivation of Dynamic Class Loading

Dynamic class loading is usually used for the following purposes:

- **Extensibility:** As shared library does help developers in building modular software, DCL permits to easily extend the app's capabilities such that developers can programmatically get new code running by loading it via different sources (i.e., network, persistent storage, etc...) at runtime.

- **App updates:** Instead of distributing updated versions of the same app, functionality provided by the current app are extended using updates downloaded through the network and loaded dynamically using class loaders.

- **Common Frameworks:** Some of the apps depend upon a common framework. For example, an advertisement framework, which shows advertisements to the user. Such a framework is most of the times installed as a separate app and the apps which rely on it load its code dynamically when needed. If this would not have been the case, the functionality provided by the framework must have been implemented in every app dependent upon the framework. Similarly, in the case of updating that common functionality provided by the framework, only the framework needs to updated rather than updating all the dependent apps.

2.2.4 Usage Motivation of Reflection

Some of the reflection APIs are discussed in this section earlier. In the following, we provide an overview of what reflection offers to a developer [106].

- **Hidden API method invocation:** Developers of the Android operating system may mark some methods as hidden. In this case, the declaration of these methods does not appear in the SDK library and, thus, they are not available for app developers. At the same time, app developers, who want to use these undocumented features of Android, may use reflection APIs to invoke them.

- **Access to the private API methods and fields:** During the compilation, the compiler ensures that the rules for accessing fields and methods hold according to the specified modifiers. Unfortunately, using the reflection API at runtime it is possible to manipulate modifiers and, therefore, gain access to private variables and methods.

- **Conversion from JSON and XML representation to Java objects:** Reflection is heavily used in Android to automatically generate JSON and XML representation from Java objects and vice versa.

- **Backward compatibility:** It is advised to use reflection to make an app backward compatible with the previous versions of the Android SDK. In this case, reflection is exploited either to call the API methods, which have been marked as hidden in the previous versions of the Android SDK, or to detect if the required SDK classes and methods are present.

- **Plugin and external library support:** In order to extend the functionality of an app, reflection APIs may be used to call plug-ins or external library methods provided during runtime.

In general, we can conclude that dynamic code loading and reflection are both essential parts of apps, specifically Android apps. To reinforce the fact further, we provide an analysis of real world apps on the usage of reflection and DCL in the following section.

2.3 Analysis of Dynamic Code Update Features in Android Apps

To understand the significance of the use of reflection and DCL in Android apps, we performed a study of 13,863 apps from Google Play store and 14,283 apps from several third-party markets gathered in July 2013, along with 1260 malware samples from [116]. In this analysis, we consider reflection calls that influence the app method call graph (MCG), i.e., method invocation (`invoke`) and object creation (`newInstance`) functions, and do not study other reflection API capabilities like field modification.

The aggregated results of the analysis with our modified version[1] of AndroGuard [2]

[1]We found out that AndroGuard does not discover all possible cases of reflection and DCL.

Table 2.1: Usage of DCL and Reflection in Applications

Markets	**Total**	**DCL used by**		**Refl. used by**	
	Apps	**Apps**	**%**	**Apps**	**%**
Google Play	13863	2573	18.5%	12233	88.2%
Androidbest	1655	35	2.1%	1088	65.7%
Androiddrawer	2677	379	14.1%	2596	96.9%
Androidlife	1677	117	6.9%	1368	81.5%
Anruan	4230	162	3.8%	2868	67.8%
Appsapk	2664	112	4.2%	1907	71.5%
F-droid	1380	11	0.07%	792	52.8%
Malware	1260	251	19.9%	1025	81.3%
Total	**29406**	**3640**	**12.3%**	**23877**	**81.1%**

are shown in Table 2.1. It is evident that dynamic code update features are widely used by application developers.

On Google Play we downloaded approximately 500 top free applications from each category. Results of the analysis reveal that on average 18.5% of analyzed apps from Google Play contain DCL and 88% use reflection. On average, apps with DCL contain 1 DCL call and apps with reflection incorporate around 22 reflective calls. The categories *"BUSINESS"*, *"SHOPPING"* and *"TRAVEL_AND_LOCAL"* show minimal DCL rates (at most 10% of apps use DCL). The most "dynamic" category is *"GAME"*; 38.3% of applications in this category use DCL[2].

We further downloaded apps from 6 third-party markets, namely, *androidbest* [5], *androiddrawer* [6], *androidlife* [7], *anruan* [9], *appsapk* [10] and *f-droid* [21]. The first 5 markets distribute only provided APK files, while the latter (*f-droid*) along with the final packages also provides links to the source code of the apps. The lowest fraction of applications with DCL calls were observed on the *f-droid* market that contains only open-source apps. In terms of individual usage, the average number of reflection calls is around 19 per app package across all third-party markets (with *f-droid* exhibiting again the lowest number of reflection calls at around 14).

Besides the analysis of benign applications, we studied malware samples provided in [116]. The average usage of DCL across all malware samples is 19.9%, whereas 81% of all samples use reflection. However, this dataset is old, and DCL usage rates in more recent malicious applications are expected to be significantly higher [90] because this

[2]Mobile games can be very sophisticated and include realistic physics and a lot of graphics. Thus, developers often develop the original app as an installer that dynamically fetches additional code during the first run.

functionality is used to conceal malicious payloads [64] from static and dynamic analyzers like Google Bouncer.

Listing 2.2: DCL and Reflection Usage in `AnserverBot`

```
1  [com.sec.android.providers.drm.Doctype]
2  public static Object b(File pFile, String pStr1, String pStr2, Object[]
       pArrOfObj) {
3    String s3;
4    if (pFile == null) {
5      String s1 = a.getFilesDir().getAbsolutePath();
6      //get the name of the file to be loaded
7      //9CkOrC32uI327WBD7n__ -> /anserverb.db
8      String s2 = Xmlns.d("9CkOrC32uI327WBD7n__");
9      s3 = s1.concat(s2);
10   }
11   for (File locFile = new File(s3); ;locFile = pFile) {
12     String s4 = locFile.getAbsolutePath();
13     String s5 = a.getFilesDir().getAbsolutePath();
14     ClassLoader locClassLoader = a.getClassLoader().getParent();
15     //get the class specified by "pStr1" from anserverb.db
16     Class locCls = new DexClassLoader(s4, s5, null, locClassLoader).loadClass(
       pStr1);
17     Class[] arrOfCls = new Class[5];
18     arrOfCls[0] = Context.class;
19     arrOfCls[1] = Intent.class;
20     arrOfCls[2] = BroadcastReceiver.class;
21     arrOfCls[3] = FileDescriptor.class;
22     arrOfCls[4] = String.class;
23     //get the method specified by "pStr2"
24     Method locMtd = locCls.getMethod(pStr2, arrOfCls);
25     //create new instance of the class
26     Object locObj = locCls.newInstance();
27     //invoke the method through reflection
28     return locMtd.invoke(locObj, pArrOfObj);
29   }
30 }
```

Example Malware: Listing 2.2 is a code snippet of the `AnserverBot` Trojan [115], which illustrates how reflection and DCL are used to thwart static analyzers from detection of malicious functionality. Line 16 shows an example of a dynamic class loading call in Android using the `DexClassLoader` class. The name of the file from which the code is loaded is computed at runtime in Line 8. Line 26 exhibits how to create an object of the loaded class using a reflective call to the default constructor. Line 28 demonstrates a method invocation through reflection; the name of the invoked method is passed as a

parameter and, thus, may not be available for static analysis.

2.4 Chapter Summary

This chapter provided a brief background on how security engineers and researchers strive to ensure a clean Android ecosystem. We discussed the basics of Android security architecture and some of the techniques used by malware developers to evade security checks. An introduction to dynamic code updates and an analysis on its usage in real world apps is provided at the latter part of the chapter. The results of our analysis reveal that the dynamic code updates are used widely in both legitimate as well as malicious apps, however, their usage in malicious apps is on the higher side.

Chapter 3

Bypassing Analysis tools Using Dynamic Code Updates

Dynamic code update techniques, such as reflection and dynamic class loading, enable apps to change their behavior at runtime. These techniques are heavily used in Android apps for extensibility. However, malware developers misuse these techniques to conceal malicious functionality, bypass static analysis tools and expose the malicious functionality only when the app is installed and run on a user's device. Although, the use of these techniques alone may not be sufficient to bypass analysis tools, it is the use of reflection/DCL APIs with *obfuscated parameters* that makes the state-of-the-art static analysis tools for Android unable to infer the correct behavior of the app. This chapter demonstrates this fact further by testing some of the state-of-the-art static analysis tools with *Reflection-Bench*, our suite of benchmark test applications that use reflection in various ways to perform malicious activities. Moreover, using a test malware app, InboxArchiver, that is based on dynamic code loading to conceal malicious functionality, we demonstrate how dynamic code loading can be used to evade online analysis systems.

To understand the current trends in real apps, it is important to perform a study on the sources of the parameters used in reflection/DCL APIs. In this chapter, we describe how malicious apps bypass analysis tools using reflection/DCL with parameters provided by sources, such as network, files, encrypted strings, etc., which are hard to analyze statically. We further develop a tool to analyze a dataset of 3,645 real world malware samples and 16,528 benign apps in order to investigate the sources of the parameters used in reflection/DCL APIs. The results of our analysis indicate the presence of such programming practices in both legitimate and malicious apps. However, malicious apps tend to obfuscate the parameters of reflection/DCL APIs more often. The use of *Crypto* related APIs as sources of the parameters of reflection/DCL APIs is significantly higher in

malicious apps, which endorses the fact that malicious apps try to thwart static analysis tools.

3.1 Introduction

Malware developers use an array of techniques to evade analysis tools deployed by app markets and execute malicious code on users' devices. Code obfuscation, anti-debugging, emulator detection, time bombs, reflection and DCL are some of the techniques found in modern mobile malware. In this dissertation, we are particularly interested in the latter two techniques. Reflection and DCL enable development of flexible apps which can change their behavior at runtime after being installed on a user's device. The same feature serves well for malware developers as they develop seemingly benign apps at installation time that can load additional malicious code at runtime using DCL and access it using reflection APIs. Doing so, they evade static analysis tools that rely on the availability of all the information before the analysis starts. Reflection/DCL APIs operate on string parameters representing code files, classes, methods, etc. When these parameters are not readily available in the code at analysis time (i. e., encrypted and only decrypted at runtime, read from a file provided via network), state-of-the-art static analysis tools find it impossible to infer the exact behavior of the app. Therefore, the sources of these parameters become much more important from security point of view as they play a vital role in malicious usage of reflection/DCL APIs. While previous works use various techniques to analyze apps in the presence of dynamic code updates, this dimension of reflection/DCL is most often overlooked [80] [106] [110].

In order to further describe the problem, this chapter demonstrates the lack of effectiveness of the state-of-the-art tools when it comes to analysing apps that hide suspicious behavior using reflection and dynamic code loading. We develop a set of benchmark apps that use reflection in different ways to conceal information leakage. Our analysis of reflection-bench using some of the state-of-the-art static analysis tools shows their ineffectiveness to handle apps that use reflection. We plan to make reflection-bench public to enable researchers to test their analysis tools with it. Furthermore, we develop InboxArchiver, a seemingly benign app that uses dynamic code loading to hide its suspicious functionality, and use it to test some of the most well known online analysis systems. The analysis reports show that InboxArchiver easily bypasses these analysis systems.

Moreover, we analyze the sources of the parameters of reflection/DCL APIs in real world apps that allow them to conceal malicious behavior and evade static analysis tools. We develop a tool, based on SAAF [74], to track information flow to reflection and DCL APIs. It uses backward program slicing to determine the sources of the parameters used in

reflection/DCL APIs. The tool takes an Android APK file (or a directory containing APK files), performs analysis on it and generates statistics about the usage of reflection/DCL APIs and the corresponding sources of their parameters. The results of our analysis on real world apps show that it is more common in malicious apps to take parameters of reflection/DCL APIs from sources, such as *Crypto* related APIs, which help thwart static analysis.

Contributions:

- We design and develop reflection-bench, a set of benchmark apps that use reflection to conceal information leakage, and use it to test some of the state-of-the-art static analysis tools. We plan to make reflection-bench public so that it can be used by other researchers to test the effectiveness of their analysis tools in the presence of reflection.

- We develop an automated static analysis tool which can perform analysis on Android apps, detect information flow between given source and sink APIs, and produce statistics about the presence of such information flow paths between source/sink APIs in individual apps as well as the whole market.

- We collect and analyze a dataset of real world apps containing 16,528 benign and 3,645 malicious apps in order to investigate the sources of the parameters used in reflection/DCL APIs. To the best of our knowledge, this is first study focusing on the sources of the parameters of reflection/DCL APIs. The analysis results would help in understanding the behavior of apps that use dynamic code updates and designing more effective analysis procedures and policies to detect malicious apps in the market.

3.2 Motivating Examples

Evidence of obfuscated parameters of reflection/DCL APIs used in real world malware motivates this part of the work. To explain it further, we consider three concrete samples of mobile malware: BrainTest, Fakenotify and AnserverBot [61,91,115]. In BrainTest, the strings representing the code files to be downloaded, classes to be instantiated and methods to be invoked using reflection/DCL APIs are provided through a file downloaded from the Internet at runtime; in Fakenotify, the strings representing the classes to be instantiated and methods to be invoked are provided as encrypted strings and only decrypted at runtime; and AnserverBot is a malware family where strings representing code files to be loaded are provided as encrypted strings.

BrainTest: Check Point Mobile Threat Prevention detected an Android malware in August 2015, which is packaged inside a game app known as BrainTest and has 100,000-500,000 downloads at Google Play Store. As reported, the malware infected up to 1 million users.

The malware uses a number of techniques to bypass Google Bouncer. It conceals its malicious activity if the IP or domain in which the app is being executed is mapped to Google Bouncer. It uses a combination of time bombs, dynamic code loading, reflection, encrypted code files, and malicious code (root exploits) downloaded from the Internet, to harden reverse engineering and evade analysis tools.

Once the app is installed on a user's device, it decrypts an encrypted file `start.ogg` from the app's assets directory and loads it using `DexClassLoader`. The dynamically loaded file starts communicating with a Command&Control (C&C) server. The server responds with a `.json` file that contains a link to a `.jar` file which the app downloads and dynamically loads using `DexClassLoader`. In addition, the `.json` file also contains names of the classes and methods which are to be invoked by the app using reflection APIs. The malware then drops root exploits and installs/uninstalls other APKs as the C&C server directs.

Listing 3.1: FakenotifyA - SMS Trojan

```
1 SmsManager localSmsManager = SmsManager.getDefault();
2 String str2 = paramString1;
3 String str3 = paramString2;
4 localSmsManager.sendTextMessage(str2, null, str3, null, null);
```

Fakenotify: It is noticed that Android malware evolves to harden analysis and reverse engineering. Listing 3.1 shows an excerpt from an SMS trojan named FakenotifyA [61]. The Listing shows how FakenotifyA uses a standard SMS sending procedure to send messages to premium numbers. Although, the message, `paramString2`, and the number, `paramString1`, to which the message is sent are provided at runtime, the SMS sending mechanism is pretty obvious and easy to detect for the analysis tools.

After some time, a new version of the same malware, FakenotifyB, surfaced. FakenotifyB is exactly similar to FakenotifyA when it comes to its malicious functionality, however, FakenotifyB makes use of reflection to dynamically create an instance of the `SMSManager` class, retrieves objects of its `getDefault` and `sendTextMessage` methods and invokes them. In addition to using reflection, the parameters representing the names of `SMSManager` class and its methods are provided in encrypted form and only decrypted at runtime. The SMS sending routine is shown in Listing 3.2 and it is much harder for analysis tools to infer its behavior unlike FakenotifyA [61].

AnserverBot: The presence of such evasive usage of reflection/DCL APIs is not an

Listing 3.2: FakenotifyB - Version 2 of FakenotifyA

```
Class class1 = Class.forName(StringDecoder.decode("&nd}D%d.(!x!ejDn5.SmsM&n&g!}"
    ));
Object obj = class1.getMethod(StringDecoder.decode("g!(?!f&wx("), new Class[0]).
    invoke(null, new Object[0]);
class1.getMethod(StringDecoder.decode("s!ndz!4(M!ss&g!"), new Class[] {java/lang
    /String, java/lang/String, java/lang/String, android/app/PendingIntent,
    android/app/PendingIntent}).invoke(obj, new Object[] {s, null, s1, null,
    null});
```

isolated incident in the Android malware. There are many examples where whole malware families rely on loading code dynamically and using encrypted strings in reflection/DCL APIs to evade detection by analysis tools. Listing 3.3 shows a piece of code taken from a sample of the AnserverBot family. It uses an encrypted string (`9CkOrC32uI327WBD7n__`) to hold the file name which is then decrypted (`str2`) at runtime and concatenated with another string (`str1`) to get the file name (`str3`). The absolute path is then retrieved in `str4` and provided to `DexClassLoader` to load the file dynamically.

Listing 3.3: Excerpt from AnserverBot

```
//9CkOrC32uI327WBD7n__ -> /anserverb.db
String str2 = Xmlns.d("9CkOrC32uI327WBD7n__");
str3 = str1.concat(str2);
for (File localFile = new File(str3); ; localFile = paramFile){
  String str4 = localFile.getAbsolutePath();
  String str5 = a.getFilesDir().getAbsolutePath();
  ClassLoader localClassLoader = a.getClassLoader().getParent();
  //get the class specified by "paramString1" from anserverb.db
  Class localClass = new DexClassLoader(str4, str5, null, localClassLoader).
    loadClass(paramString1);
```

The point worth noticing in these three examples is the use of parameters in reflection/DCL APIs that are not readily available for the analysis tools. Consequently, static analysis tools find it impossible to construct the exact behavior of these apps. Therefore, the focus of the analysis in this chapter is not only reflection/DCL APIs, but also the manner in which the parameters are provided to these APIs.

3.3 Reflection-Bench and InboxArchiver

This section demonstrates how malware developers can evade static analysis tools and the available online analysis systems using dynamic code updates. Both the cases, reflection

and DCL, are discussed separately. In the first subsection, we discuss Reflection-Bench (our benchmark of Android applications to test static analysis for reflection resolution), whereas in the second subsection, we discuss our sample test malware, InboxArchiver, which makes use of dynamic code loading to evade current available online analysis systems.

3.3.1 Reflection-Bench

Reflection is a very useful and heavily used dynamic code update technique in ever changing Android apps. However, it comes with an inherent ability to harden static analysis of apps which makes it attractive for malware developers. Although, researchers have worked on trying to resolve reflection in Android apps, there has not been a benchmark of apps which could be used as a test suite for reflection in Android. We present Reflection-Bench, a set of Android apps, which use reflection to conceal information leakage so that it cannot be detected by static anlyzers. We use Reflection-Bench to test some of the very recent state-of-the-art static analysis tools.

Overview: The purpose of developing Reflection-Bench is to provide a set of Android based applications which could be used by analysis tools to test their capabilities in resolving reflection. These apps use reflection in various forms to conceal information leakage and make the flow of the program ambiguous. We have developed a set of 14 apps based on how reflection is used and how the reflection APIs get their arguments.

Before describing the apps in Reflection-Bench, we go through the different cases of reflection that this benchmark covers. We divide the apps into different categories and try to make detection harder as we move from one case to the other. The hardness of reflection resolution depends upon the nature of the arguments used in the reflection APIs. We can broadly categorize their nature into static strings and dynamic strings. By static strings, we intend to refer to those string arguments which are provided as part of the application package, e.g., strings defined inside the program, read from a file which is part of the application, etc.

In Reflection-Bench, we do not consider the case of dynamic strings/arguments, e.g, those received over the network, read from files on disk, received from other apps, etc. The case of such dynamic strings makes it almost impossible for static analysis tools to resolve reflection. We focus on those cases where all the information regarding the arguments of the reflection APIs are provided as part of the application. However, with each case the complexity is gradually increased.

In the first few cases, the arguments of reflection APIs are constant strings assigned to program variables. In the latter cases, we consider reading the arguments from a properties file (part of the APK file) and from a hashtable defined inside the program.

Moreover, we also consider the cases where the string arguments are formed from the concatenation of multiple strings or decrypted from encrypted strings using crypto APIs. In addition, we consider two levels of complexity where in level one, reflection is used to call only the methods defined inside the app and in level two, both the methods defined inside the program as well as the senstive APIs, which are responsible for leaking sensitive information, are called through reflection.

Reflection-Bench is designed so that it can be used to test tools which perform taint analysis as well as those which only generate call graphs for other forms of static analysis.

Implementation: We have two major classes in most of the cases, i.e, BaseClass and MainActivity. BaseClass has two methods, where Method1 gets the device ID using the `getDeviceID` API and stores it in a local field Str. Method2 gets a string and sends it out using the `sendTextMessage()` API.

MainActivity calls Method1() of BaseClass, gets its field Str and sends it to the Method2 of BaseClass which leaks it out.

We tried to cover different combinations of reflection APIs which could make it hard for static analyzers to detect the information leakage. In the following, we describe how reflection APIs are used in each case.

- ① MainActivity retrieves the the field Str of BaseClass using `getField()` reflection API.
- ② MainActivity retrieves an instance of BaseClass using the reflection API `forName()`, creates its object using the `newInstance()` API and gets its field Str using the `getField()` reflection API.
- ③ MainActivity retrieves an instance of BaseClass using the reflection API `forName()`, gets its Constructor using the `getConstructor` API, creates its object using the `newInstance()` API and gets its field Str using the `getField()` reflection API.
- ④ MainActivity retrieves an instance of BaseClass using the reflection API `forName()`, creates its object using the `newInstance()` API and gets its field Str using the `getField()` reflection API. It also retrieves the methods of BaseClass using the `getMethod()` reflection API and calls them using the `invoke()` reflection API.
- ⑤ MainActivity retrieves an instance of BaseClass using the reflection API `forName()`, gets its Constructor using the `getConstructor` API, creates its object using the `newInstance()` API and gets its field Str using the `getField()` reflection API. It also retrieves the methods of BaseClass using the `getMethod()` reflection API and call them using the `invoke()` reflection API.

In the above cases, the names of the class "BaseClass", its methods and its field are provided as static strings in the MainActivity class. In the following, we try to acquire/generate these names at runtime in addition to Case (4).

- (6) Reads the names of BaseClass, its methods and its field from a file.
- (7) Reads the names of BaseClass, its methods and its field from a Hashtable.
- (8) Constructs the names of BaseClass, its methods and its field from multiple strings in the program.
- (9) Decrypts the encrypted names of BaseClass, its methods and its field using Crypto APIs.

  In all of the above cases, reflection APIs are only used in MainActivity and the sensitive APIs, i.e., `getDeviceId()` and `sendTextMessage()`, are called directly in BaseClass. In the following cases, we introduce reflection in BaseClass too in addition to Case (4).

- (10) BaseClass retrieves an instance of the TelephonyManager class using the reflection API `forName()`, creates its object using the `newInstance()` API, gets the sensitive APIs using the `getMethod()` reflection API and calls them using the `invoke()` reflection API.

  In the above case, we use static strings for the names of the class TelephonyManager and the methods `getDeviceId()` and `sendTextMessage()`. In the following we acquire/generate these names at runtime in addition to Case (10).

- (11) Reads the names of TelephonyManager class, methods `getDeviceId()` and `sendTextMessage()` from a file.
- (12) Reads the names of TelephonyManager class, methods `getDeviceId()` and `sendTextMessage()` from a Hashtable.
- (13) Constructs the names of TelephonyManager class, methods `getDeviceId()` and `sendTextMessage()` from multiple strings inside the app.
- (14) Decrypts the encrypted names of TelephonyManager class, methods `getDeviceId()` and `sendTextMessage()` using Crypto APIs.

Tools analysis results: We report the results of analysis on recent state-of-the-art tools, e.g., Flowdroid [41], Androguard [2], Amandroid [104], SAAF [74], SCandroid [66] and IccTa [78].

Table 3.1: Analysis with State-of-the-art tools

Apps	Taint Analysis				Call Graphs	
	Flowdroid	IccTa	Amandroid	SCandroid	Androguard	SAAF
DataFlow1	✗	✗	✗	-	NA	NA
PlainStringsL1-1	✗	✗	✗	-	✗	✗
PlainStringsL1-2	✗	✗	✗	-	✗	✓
PlainStringsL1-3	✗	✗	✗	-	✗	✓
PlainStringsL1-4	✗	✗	✗	-	✗	✓
FileStringsL1-1	✗	✗	✗	-	✗	✗
HashtableStringsL1-1	✗	✗	✗	-	✗	✗
MultipleStringsL1-1	✗	✗	✗	-	✗	✗
EncryptedStringsL1-1	✗	✗	✗	-	✗	✗
PlainStringsL2-1	✗	✗	✗	-	✗	✗
FileStringsL2-1	✗	✗	✗	-	✗	✗
HashtableStringsL2-1	✗	✗	✗	-	✗	✗
MultipleStringsL2-1	✗	✗	✗	-	✗	✗
EncryptedStringsL2-1	✗	✗	✗	-	✗	✗

A summary of the results is provided in Table 3.1. Those tools which perform taint analysis, such as Amandroid, etc., are analyzed by performing taint analysis of the apps in Reflection-Bench. However, for those tools which do not perform taint analysis, such as Androguard, etc., we analyze them by generating call graphs of the apps using these tools. In Table 3.1, a ✓ in column X, indicates that the app is successfully analyzed by tool **X**, whereas, a ✗ in the same column indicates otherwise.

- **Amandroid, Flowdroid, IccTa and SCandroid**

  To analyze Reflection-Bench with Amandroid, Flowdroid, IccTa and SCandroid, we performed taint analysis of the apps using these tools. These tools analyze APK files and report the presence of sources/sinks of information as well as the tainted paths between these sources and sinks, if any.

  As shown in Table 3.1, in the analysis of Reflection-Bench with Flowdroid, Flowdroid did not report any information leakage in any of the apps as represented by ✗ in the Flowdroid column. Although, it did report the presence of sources and sinks in some of the apps. Similar is the case with Amandroid and IccTa too. None of these tools

could detect the information flows obfuscated using reflection in Reflection-Bench. With IccTa, it is understandably so, because it relies on Flowdroid for information flow analysis. For SCanDroid, we could not get any meaningful results as all the experiments ended with an error. We also could not get any help to fix it as the tool is not well supported.

- **Androguard and SAAF**

  Since Androguard and SAAF are not taint analysis tools and only generate method call graphs of apps, we analyze Reflection-Bench with these tools by generating the MCGs of the apps. In each of the generated MCGs, we look for the app's methods and APIs called through reflection.

  The first application of Reflection-Bench is only for those tools which perform taint analysis. It only uses reflection to make the data-flow ambiguous. The rest of the apps can be used to test both kinds of apps, those which only generate MCGs and those which perform taint analysis too. As shown in Table 3.1, Androguard does not correctly identify any method called through reflection in any of the 13 apps.

  SAAF's results are relatively better than Androguard's results. As column 'SAAF' shows, SAAF is able to correctly identify the targets of reflection calls in four of the applications in Reflection-Bench. In these four apps, the arguments provided to the reflection APIs are plain strings. SAAF does not resolve the targets in other cases where the arguments are either read from a file or hashtable, encrypted strings and formed from multiple strings inside the apps. It is important to remember here that none of the applications get any arguments from outside the application.

These analysis results show that with a bit of tweaking using reflection, static analysis tools find it extremely hard to properly analyze apps.

3.3.2 InboxArchiver: Test Malware using DCL

App developers use dynamic code loading for various legitimate purposes, mainly extending the functionality of the app. However, this feature can be used by malware developers to bypass analysis tools deployed at the app markets. A malware developer can submit a seemingly benign app with hidden malicious functionality, i.e., obfuscated functionality to load additional code provided once the app is installed on a user's device. We demonstrate with our InboxArchiver app how a malware developer can bypass analysis tools using DCL.

Overview: InboxArchiver is a simple app that reads the SMS inbox and sends some statistics to a number provided by the user. These statistics include the number of SMS

messages sent to and received from certain numbers. A user can configure InboxArchiver to receive a daily, weekly or monthly SMS message containing these statistics.

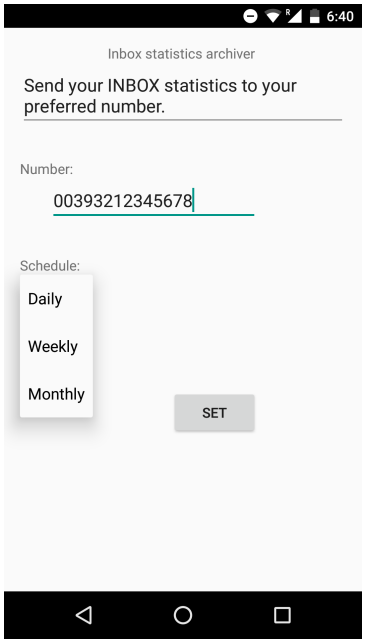


Figure 3.1: InboxArchiver - Screenshot

The malicious part of the app, however, downloads some additional code from the Internet which contains other numbers potentially owned by an adversary, loads this code using the DCL APIs and leaks these SMS inbox statistics.

Implementation: The main features of InobxArchiver are the use of DCL and reflection having encrypted strings representing the code paths, class names and method names. This helps InboxArchiver evade static analysis tools. In order to evade dynamic analysis, it makes use of a simple delay technique where again the APIs are called using reflection with encrypted parameters. Although there are other more sophisticated techniques available, the use of a mere delay technique signifies the role of DCL/reflection in evading analysis tools.

InboxArchiver consists of three main classes, i.e., a `MainActivity` class, a `MessageSender` class and a `Loader` class. The `MainActivity` class presents an interface to the user as shown in Figure 3.1. The `MessageSender` class, which is a Service and runs in the background, is responsible for retrieving the inbox statistics and sending it periodically to a pre-configured number. After a certain delay, the `MessageSender` class instantiates an object of the `Loader` class which handles the downloading of additional code from the Internet and dynamically loading it using DCL APIs. It makes use of encrypted parameters and encryption/decryption functionality provided by other auxiliary classes.

Table 3.2: InboxArchiver Analysis using Online Analysis Systems

Analysis System	**Analyzed**	**Obfuscation**	**DCL**	**Malware**
VirusTotal [35]	✓	✗	✗	✗
UnDroid [13]	✓	✓	✗	✗
AndroTotal [8]	✓	✗	✗	✗
ds-andrototal [30]	✓	✗	✗	✗
MobiSec Lab [28]	✓	✗	✗	✗
CopperDroid [102]	Queued	-	-	-
SandDroid [32]	✓	✓	✓	✗

Analysis results: We uploaded InboxArchiver to a number of online Android app analysis systems. Table 3.2 shows a summary of the results from the online analysis systems. Column *Analyzed* shows whether the app is properly analyzed or not. The next two columns, *Obfuscation* and *DCL*, show if the analysis systems detect obfuscation and the use of dynamic code loading, respectively. The last column in the table represents the final remarks about the app.

Among the online analysis tools shown in Table 3.2, we did not receive any results from CopperDroid and the app is still in the queue for more than a year now. All other tools were unable to detect that the submitted app is malicious. VirusTotal scanned the app with 54 antivirus tools, including BitDefender [15], GData [24], AVG [14], Avast [12] and Kaspersky [26], etc., and none of them labeled it suspicious. UnDroid and SandDroid termed the app as obfuscated, while SandDroid could also detect dynamic code loading in the app. However, it could not detect the loaded file and analyze it.

3.4 Analysis Tool: Design and Implementation

The architecture and workflow of the analysis tool is shown in Figure 3.2. It is composed of two main modules represented by the dotted rectangles, i. e., a slice extraction module and a slice analysis module.

Slice Extraction: Most android app analysis tools transform Android's Dalvik bytecode to Java bytecode or source code in order to use the already available tools for Java program analysis. However, the translation from Dalvik bytecode to Java source code cannot always be accurate, specifically in apps that use some obfuscation techniques [101]. We perform analysis on disassembled Dalvik bytecode, i. e., smali code, which does not suffer from this limitation.

The slice extraction module takes an APK file or a directory, where APK files are

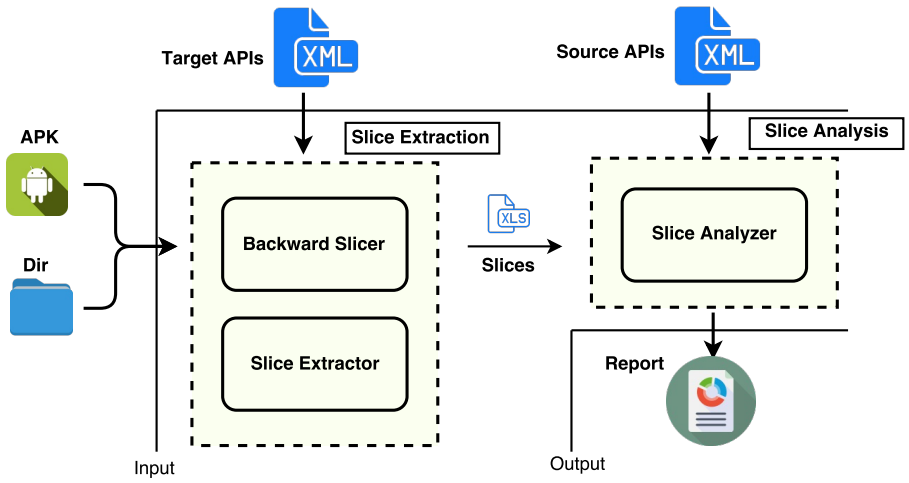


Figure 3.2: Analysis Tool Design

located, and a list of target APIs as input. A target API along with its specific parameter is the starting point of the analysis. The *Backward Slicer* searches for all the occurrences of the target API in the app's smali code after disassembling the APK file and backtracks them. The backtracking process starts with the register that stores the value of the parameter used in the target API and tracks backward in the code to find all other registers that have a direct or indirect effect on the value of this register. Consequently, the analysis tool captures the information flow to the target API. The set of code statements involved in the information flow to a target API is called a *Backward Slice*. The *Backward Slicer* is based on SAAF which can perform backward program slicing on smali code [74]. However, original SAAF does not consider the information flow performed through Android Intents and may miss some information flows. Intents are messaging objects used for inter-component and inter-app communication. Typically, they are used to start activities, services or invoke broadcast receivers. In order to extend this functionality, we modified SAAF to track information flow performed through explicit Android Intents. The *Slice Extractor* extracts all the code instructions that form a particular slice, marked by the *Backward Slicer*, in the form of a `.csv` file. Listing 3.4 shows an example of a code slice for the method `forName` of the class `Ljava/lang/Class;`. Once the slices are extracted, the analysis process moves to the next module, i. e., slice analysis.

Slice Analysis: The next step in the analysis is to detect information flow from a source API to the target API. This module takes the slice files generated in the previous step and a list of source APIs as input. It consists of a set of Python scripts, which we call *Slice Analyzer*, collectively. The *Slice Analyzer* traverses through each code instruction in

Listing 3.4: Backward Code Slice. TargetLine: 63, TargetClass: Ljava/lang/Class;, TargetMethod: forName

```
34: invoke-virtual {p0}, Ldisi/test/app/MainActivity;->getResources()Landroid/content/res/Resources;
36: move-result-object v12
38: const/high16 v13, 0x7f05
40: invoke-virtual {v12, v13}, Landroid/content/res/Resources;->openRawResource(I)Ljava/io/InputStream;
42: move-result-object v10
46: new-instance v9, Ljava/util/Properties;
48: invoke-direct {v9}, Ljava/util/Properties;-><init>()V
52: invoke-virtual {v9, v10}, Ljava/util/Properties;->load(Ljava/io/InputStream;)V
55: const-string v12, "class"
57: invoke-virtual {v9, v12}, Ljava/util/Properties;->getProperty(Ljava/lang/String;)Ljava/lang/String;
59: move-result-object v1
63: invoke-static {v1}, Ljava/lang/Class;->forName(Ljava/lang/String;)Ljava/lang/Class;
```

the slices corresponding to the target APIs and locates the presence of source APIs. The purpose of such traversal is to infer a source/sink relationship between the source and the target APIs. The *Slice Analyzer* not only reveals such source/sink relationships, but also provides statistics regarding the number of apps containing the target/source APIs, occurrences of the target/source APIs and their relationship in each individual app and in all the analyzed apps in a market. A report containing these statistics is generated in the form of a `.json` file.

3.5 Application Analysis

We analyze a dataset of apps for potential dangerous usage of reflection/DCL APIs. Potential dangerous usage refers to the usage of reflection/DCL APIs in conjunction with certain sources of their parameters that complicate the overall analysis and might be used by malware developers to evade static analysis tools. There are two distinct entities in this analysis, i. e., 1) reflection/DCL APIs, 2) source APIs. The purpose of the analysis is to infer the presence of information flow from the source APIs to certain parameters of reflection/DCL APIs. In this section, we describe the target reflection/DCL APIs, the corresponding source APIs of their parameters and the apps dataset.

3.5.1 API Selection

Reflection and DCL APIs: Table 3.3 contains a representative list of DCL and reflection APIs that are tracked for analysis and considered as the target APIs. The first two columns represent the class and method names, whereas the last column represents the specific parameters of interest in these API calls. The APIs are divided into three categories. The first category, *Dynamic Code Loading*, contains APIs that are used to load code in the form of `.jar/.apk/.dex` files at runtime. APIs in the second category, *Class Retrieval*, are used to load classes and create their objects. The last category, *Method Retrieval and Invocation*, contains APIs that are used to retrieve method objects and invoke them.

We have included only those APIs which can potentially help conceal malicious behavior and they require essential parameters when using either DCL or reflection. For instance, `loadDex` method of the class `Ldalvik/system/DexFile` is tracked for its first parameter, `sourcePathName`, which is of type `String` and represents the path to the `.jar/.apk/.dex` file to be loaded. A malware developer can obfuscate the parameter provided to the `loadDex` method and make it hard for analysis tools to determine the location of the code which is loaded dynamically. Similar is the case with the constructors of `PathClassLoader` and `DexClassLoader`. Obfuscation of the parameters of these class constructors makes it hard for analysis tools to infer the location of the dynamically loaded code. Here, `libraryPath`, represents the path to the directory containing native libraries.

Moreover, static analysis can be led to unsound results through the use of obfuscated parameters passed to the `loadClass` method of the `Ljava/lang/ClassLoader` class or the `forName` method of the `Ljava/lang/Class` class. In both cases, static analysis tools will be unable to know the exact class which is to be loaded at runtime. Using obfuscated parameters in methods, such as `getDeclaredMethod`, `getMethod` and `invoke`, can leave the analysis tool clueless about the methods being retrieved or called. So, even if the location of the code and class to be loaded is known, the behavior of the app can not be completely determined as static analysis can not correctly identify the method or the order in which the methods are being called, which is pivotal to understanding an app's behavior.

Source APIs: Hard coded strings inside the code are easy to analyze for static analysis tools even if they are used as parameters to reflection/DCL APIs. However, when these strings are not readily available inside the code, static analysis tools are completely ineffective in inferring an app's behavior. To evade static analysis tools, the string parameters to reflection/DCL APIs can be retrieved from various sources at runtime. Table 3.4 provides a list of APIs that are used to access such sources. The sources are chosen

Table 3.3: The List of Tracked APIs and Their Parameters

Class	Method	ParamNo	Params
Dynamic Code Loading			
Ldalvik/system/PathClassLoader;	<init>	1,2	dexPath, libraryPath
Ldalvik/system/DexClassLoader	<init>	1,3	dexPath, libraryPath
Ldalvik/system/DexFile;	loadDex	1	sourcePathName
Class Retrieval			
Ljava/lang/ClassLoader;	loadClass	1	className
Ljava/lang/Class;	forName	1	className
Method Retrieval and Invocation			
Ljava/lang/Class;	getDeclaredMethod	1	methodName
Ljava/lang/Class;	getMethod	1	methodName
Ljava/lang/reflect/Method;	invoke	1	methodObject

based on their potential capability to evade static analysis tools specifically when they provide parameters which are to be used in reflection/DCL calls. The first column in the table represents the classes and the second column represents the corresponding methods which are considered as potential sources. Classes are grouped in categories, e. g., Crypto, Telephony, etc. X in the second column indicates that there are several methods in the corresponding class which are considered to be potential sources, thus, we simply did not enumerate all of them in the table. Similarly, X* represents that all the subclasses are also considered, e. g., subclasses of `InputStream` such as `FileInputStream`, `BufferedInputStream`, etc.

Some of the categories, such as *Telephony* and *Internet*, are purely dynamic and cannot be analyzed by static analysis tools. A malware developer can use these sources to communicate important parameters to the target APIs from a C&C server and thus a static analysis tool has no way to determine the behavior of the app. Other categories, such as *InputStreams*, *Readers* and *Crypto*, etc., include APIs that access resources which might be available at the time of analysis. However, their use hardens analysis. For instance, an app can retrieve the required parameters, using APIs from *InputStreams*/*Readers* categories, from a file which can be in any format while the analysis needs to know in advance which format to expect.

As discussed in §3.3.1 earlier, to test the ability of existing tools to analyze such apps, we developed a set of apps that leak sensitive information. These apps use reflection APIs to call various sensitive APIs where the names of these APIs and their classes are

provided as strings by some of the sources listed in Table 3.4, such as *Hashtable*, *Crypto*, and *InputStream*. We analyzed these apps using Flowdroid [41], IccTa [78], SAAF [74], Androguard [2], SCandroid [66] and Amandroid [104]. We observed that none of these tools were able to successfully detect the concealed malicious functionality, which reflects that the sources of the parameters used in reflection/DCL APIs play an important role in complicating their analysis. It is worth mentioning here that some of the tools, such as Flowdroid, SAAF, etc., can determine the targets of reflection calls to a certain extent when the parameters used in reflection APIs are string constants provided in the code.

3.5.2 Dataset Description

For the analysis process, we created a dataset of real world apps containing both benign and malicious samples.

Table 3.4: Sources of Parameters

Class	Methods
Map, Hashtable	
Ljava/util/Map;	X*
Crypto	
Ljavax/crypto/Cipher;	doFinal
Ljavax/crypto/Cipher;	update
Ljavax/crypto/CipherInputStream;	read
Ljavax/crypto/Mac;	doFinal
Ljavax/crypto/Mac;	update
Ljavax/crypto/SealedObject;	getObject
Telephony	
Landroid/telephony/TelephonyManager;	X
Landroid/telephony/SmsManager;	X
Internet	
Ljava/net/URLConnection;	X
Ljava/net/HttpURLConnection;	X
Ljava/net/ssl/HttpsURLConnection;	X
Ljava/net/JarURLConnection;	X
Input Streams	
Ljava/io/InputStream;	X*
Readers	
Ljava/io/Reader;	X*
Content Resolver	
Landroid/content/ContentResolver;	X

Google Play Store: The dataset consists of 13,223 apps downloaded from the Android official Google Play Store [25]. Although, there are instances of malicious apps been published to the official app store, Google Play Store uses Google Bouncer as a vetting mechanism for the apps submitted to the store. Hence, one can safely assume that the probability of a malicious app at the Google Play Store is considerably lower as compared to other markets.

F-droid: We added 3,305 apps downloaded from an online third party market, i. e., F-droid [21]. F-droid also provides the source code of the apps. Third-party app markets usually contain a higher number of malicious apps as these markets, most of the times, do not analyze the apps before publishing them. However, these samples are assumed to be benign in our work as they are flagged benign by most of the antivirus tools on VirusTotal [35].

To complement the downloaded benign apps, the dataset also consists of 3,645 malware samples, in the form of `.apk` files.

Genome Project: 1,260 malware samples, divided into 49 families, are taken from

the Malware Genome Project [116].

AndroidSandbox: 1,875 of the malware samples in our dataset are downloaded from AndroidSandbox [4]. AndroidSandbox is an online malware analysis service, unfortunately, out of service temporarily.

Contagio Mobile Malware Dump: The rest of the malware samples are downloaded from Contagio Blog [17]. Contagio Blog is a repository for collecting malware samples. These samples are also downloadable for research purposes.

3.6 Analysis Results and Discussion

Experiment Design: We performed the experiment separately for the various app sources as discussed in the previous section. Doing so, we had more control over when to stop/start the analysis in case there is some problem. Moreover, this design of the experiment later on helped in two ways, i. e., 1) comparing results from different app sources, and 2) aggregating the results into two categories (malicious and benign).

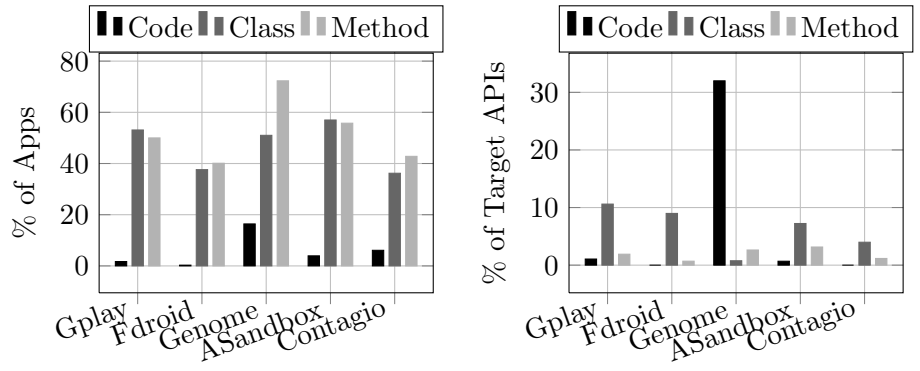
We used two machines for the experiment. The first one is a desktop, Dell Precision T1700, with a Quad-Core Intel(R) Xeon(R) 3.10GHz CPU and 8GB memory. The second machine is an HP laptop having an Intel Core i7-2630QM 2.00GHZ CPU and 4GB of memory. Analyzing all these apps with our tool on the two machines running in parallel took roughly a month.

The analysis provides an idea about the prevalence of reflection/DCL usage, in real world apps, in a manner which can be used to conceal malicious behavior and bypass app vetting process deployed at app markets.

The goal of the analysis is to answer the following research questions:

- **Q1:** What is the distribution of different categories of reflection/DCL APIs (as mentioned in §3.5) in both legitimate and malicious apps?
- **Q2:** How often do reflection/DCL APIs receive their parameters from one or more source APIs (as mentioned in §3.5)?
- **Q3:** What is the share of individual source APIs among all the mentioned source APIs in providing parameters to the target APIs?
- **Q4:** What is the highlight of the analysis results which is distinguishable in benign and malicious apps?

Q1. Presence of Reflection/DCL APIs: It is important to mention that we are only concerned with the developer's code and do not consider the occurrences of the target



(a) % of Apps with Target APIs

(b) % of Target APIs getting parameter from the Source APIs

Figure 3.3: Prevalence of Target APIs in the analyzed apps and Source APIs providing the parameter passed to Target APIs

APIs in the Android framework itself. Figure 3.3(a) shows a graphical representation of the prevalence of the three categories of the target APIs, i. e., *Code*, *Class*, *Method*, in the analyzed apps. It shows that class loading and method invocation using reflection is widely used in both legitimate as well as malicious apps. At the same time, usage of additional code loading in the form of `.jar/.dex/.apk` is comparatively lower. The Black bars in the graph show that the use of code loading in the form of `.jar/.dex/.apk` is negligible in legitimate apps, whereas malicious apps tend to use this feature which helps them evade static analysis tools.

Q2. Parameters from Source APIs: A small fraction of the total occurrences of the target APIs in the analyzed apps receive their parameters from the source APIs, mentioned in §3.5, which could potentially hinder static analysis tools. This fraction is less even in malicious apps as shown in Figure 3.3(b), except for the Genome malware dataset. The obvious reasoning behind these low numbers (or almost equal numbers in legitimate and malicious apps) can be the fact that most of the malware samples are repackaged versions of benign apps and, therefore, would use reflection/DCL in the same manner in general. This necessarily implies that apps usually provide class names and method names to reflection/DCL APIs as string constants, which is a good news for static analysis tools. However, in order to evade static analysis tools, it is not necessary to obfuscate the parameters of all the reflection/DCL calls, rather obfuscating those calls which perform malicious behavior is enough. Moreover, the trend in malicious apps is to provide a significant amount of benign functionality to lure the user into installing the app and, also, surreptitiously perform some malicious functionality.

Q3. Contribution of Individual Source APIs: Apart from the bar representing

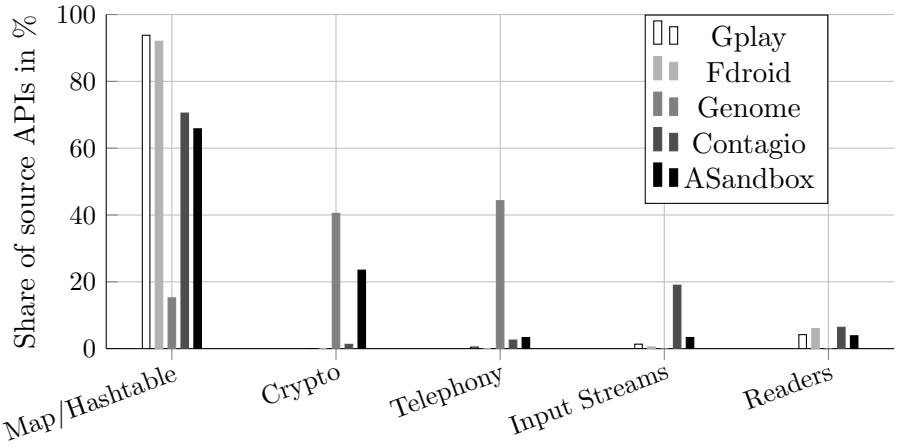


Figure 3.4: Contribution of source APIs in providing arguments to Target APIs. X-axis represent the various categories of source APIs.

the percentage of *Dynamic Code Loading* APIs taking its parameters from the source APIs for the Genome malware dataset in Figure 3.3(b), the rest of the bars for all the apps datasets hardly reach 10% . This behavior is more or less identical at this coarse level in both legitimate as well as malicious apps. However, Figure 3.4, which provides a finer view of the contributions of individual sources, reveals more about the behavior of legitimate and malicious samples. Figure 3.4 shows the graph for the top 5 contributing source API categories in each dataset. It reveals that most of the apps in both datasets, benign and malicious, retrieve class and method names from `Map/Hashtable` and, therefore, it is the prime contributor in providing parameters to reflection/DCL calls. One reason for the high usage of `Map/Hashtable` can be the usage of DexGuard (a commercial Android app obfuscator) [19]. Its string encryption mechanism uses byte-arrays and `Maps` for obfuscating strings.

The other major contributing sources are `Input Streams` and `Readers`, which are used to retrieve class and method names from configuration files either provided along with the `.apk` package or provided at runtime. Both of these categories can be used to conceal behavior and, therefore, their usage in malicious apps is slightly on the higher side. The use of `Telephony`, however, is mostly found in malicious apps only. Apparently, there are not many benign reasons for receiving class and method names via an SMS message. However, for malicious apps, this mechanism could be used as a communication channel to a C&C server.

Q4. Crypto APIs: In the initial experiment, we found very few instances of the standard `Crypto` APIs being used as the sources of parameters for reflection/DCL in our analyzed dataset. However, as shown in §3.2, malicious apps do use encrypted strings,

which are only decrypted at runtime, as parameters of reflection/DCL calls. Therefore, we further manually analyzed the *AnserverBot* family of the Genome dataset by disassembling the `.apk` files and looking into the Smali code. We found out that *AnserverBot* stores the code file names as encrypted strings and decrypts them at runtime when passing them on to `DexClassLoader`. However, it does not use the standard `Crypto` APIs to decrypt these strings, but rather uses its own logic for decryption. We could not look into all the apps for such encryption/decryption techniques, which could be another interesting study, but understandably, using non-standard encryption/decryption techniques might be more attractive to malware developers.

These results show that a wide range of real world apps, specifically malicious apps, use reflection/DCL in a manner that enables them to bypass state-of-the-art automated analysis tools. The increasing number of apps and the rapid evolution of anti-analysis techniques found in modern day malware demand for more effective and sophisticated automated analysis tools.

3.7 Considerations on Analysis Tools for Android

The combination of code update techniques along with anti-debugging, emulator detection techniques and the ability to reveal malicious behavior only when particular conditions (i. e., temporal) are met enables malware developers to bypass analysis tools. We propose some recommendations that could be useful in detecting malware even in the presence of evasive techniques.

Modern analysis tools need to have an efficient and effective dynamic analysis part due to some inherent limitations of static analysis. We recommend to push for targeted dynamic analysis where target APIs, such as those of reflection/DCL, are identified and the application is triggered with inputs which make it follow the target paths. A targeted triggering solution coupled with other solutions, such as those combining static and dynamic analysis, will help in revealing malicious behavior otherwise concealed by a malicious app.

Considering the problem of stimulating apps' behavior during an analysis/debug environment, loadtime analysis of the code other than that contained in the standard `.dex` file of an app can help detecting malicious code loading. Android framework can have an analysis module which performs some lightweight on-device analysis of the code loaded from arbitrary locations before loading it. [62] provides a library for secure class loading, but they only check for the integrity of the code. Adding other forms of security analysis to their solution would be more helpful.

According to Google's policy, all the apps must use the Google Play store for their

updates. However, this policy is not always enforced, as the BrainTest example shows. An effective enforcement of this policy would result in catering the problem of malicious code updates to an extent. Therefore, any app which downloads code from any location other than the Google Play store should be deemed malicious and not allowed to do so.

3.8 Limitations

A non trivial number of apps were analyzed in the work discussed in the chapter and should provide a fair view of reflection/DCL usage. However, we understand that the same experiment on a much larger scale, possibly performed by app markets such as Google play store, would result in providing a much better picture of the situation regarding how benign and malicious apps use reflection/DCL.

The malware datasets, in particular, the one from the Genome Project, is a bit old now keeping in view the rapid increase in the number of mobile malware samples. The trend towards more obfuscation and sophistication in malware implies that the evasive behavior would be more prevalent in newer malware samples.

We do not analyze native code, therefore, the sources of parameters coming from native code are not considered in the analysis presented in this chapter. Moreover, the analysis tool does not capture information flow to reflection/DCL calls obfuscated through other reflective calls.

3.9 Related Work

Literature shows that there have been efforts to analyze apps in the presence of reflection and DCL in Java as well as in Android. Livshits *et al.*'s work uses points-to analysis and cast analysis to statically resolve the targets of reflection [80]. Similarly, Christensen *et al.* use Java string analyzer to statically track the arguments passed to reflection APIs to resolve their targets [52]. A static analysis tool for Android apps, Flowdroid, performs data flow analysis and resolves the targets of reflection only when the parameters are string constants [41]. However, none of them provides an analysis on the sources of the parameters passed to these APIs and their possible contribution in concealing malicious behavior. Moreover, Flowdroid also present a workbench of Android apps, Droidbench, which can be used to test static analysis tools. However, the part of Droidbench focusing on reflection is very basic and contains fewer apps in comparison to Reflection-Bench.

Hirzel *et al.* extend pointer analysis to resolve reflection, DCL and native code using online (dynamic) analysis [73]. They instrument the virtual machine service that handles reflection and DCL with handlers, which dynamically updates a constraint database

during the program execution. Similarly, Bodden *et al.* propose TamiFlex which complements static analysis of Java apps by resolving DCL and reflection [48]. TamiFlex executes a Java app, which is modified using `java.lang.instrument` API, and logs the information about DCL and reflection. However, similar to dynamic analysis, both these techniques suffer from the triggering problem.

Poeplau *et al.* [90] have tried to solve the problem of dynamic code loading, potentially malicious, using a whitelisting approach. Their whitelists are based on hashes of codes to be loaded. They propose that only those pieces of code could be loaded dynamically, which have their hashes available in the mentioned whitelist. They also developed a sample malicious app and practically evaded Google Bouncer using DCL. Similarly, Canfora *et al.* present composition malware where they present a model for evading analysis tools. However, their focus is more on downloading the code from different places and combining them at runtime to create malicious app logic [51]. However, in our work we present a more generic evasion process used by malicious apps focusing on the underlying reflection and DCL APIs and the sources of their parameters.

3.10 Chapter Summary

Dynamic code update features, such as reflection and DCL, are widely used in Android apps to make them extensible. These features, however, attract malware developers due to their potential capability of evading analysis tools when their parameters are obfuscated or provided only at runtime. In this chapter, we demonstrated this fact by analyzing a set of benchmark apps, Reflection-Bench, that conceal information leakage using reflection with some of the state-of-the-art static analysis tools. As expected, the results of our analysis reveal the ineffectiveness of static analysis tools in such situations.

We also developed a tool that analyzes Android apps and finds source/sink relationships between certain potentially dangerous source APIs and reflection/DCL APIs. Moreover, to emphasize the importance of the parameters used in reflection/DCL APIs, we analyzed a dataset of real world apps. The results of our analysis show that malicious apps do try to hide the parameters of reflection/DCL APIs, by encrypting them or receiving them at runtime from the outside world, in order to bypass static analysis tools. The results of our analysis combined with the study of the static analysis tools available today for Android apps highlight the need for further research and development of analysis tools that efficiently combine static and dynamic analysis.

Chapter 4

StaDART: Combining Static and Dynamic Analysis

Abstract

Static analysis of Android applications is inherently susceptible to be evaded by applications using dynamic code update techniques, i.e., dynamic class loading and reflection. These techniques, now heavily used in modern real-world malware, thwart even the latest of static analysis tools. Chapter 3 demonstrated this fact by testing some of the state-of-the-art static analysis tools with *Reflection-Bench* and using InboxArchiver to evade online analysis systems. In this chapter, we present StaDART, an extented version of our previously proposed solution Stadyna [110], which combines static and dynamic analysis of Android applications to reveal the concealed behavior of malware. Unlike Stadyna, StaDART utilizes ArtDroid to avoid modifications to the Android framework. Furthermore, we integrate it with a triggering solution, DroidBot, to make it more scalable and evaluate it with more Android applications. We present our evaluation results with a dataset of 2,000 real world applications; containing 1,000 legitimate applications and 1,000 malware samples.

4.1 Introduction

Ensuring users' privacy and security is a major concern and requires adequate measures from all the involved parties, such as application developers, framework providers, application stores, etc. Android applications go through a vetting process, at some of the app markets at least, before being published. For this purpose, Google makes use of Google Bouncer at the official Google Play store. The vetting process generally uses some form of static/dynamic analysis to scrutinize apps for malicious content and Google Bouncer

is no different [89].

However, growing number of malware samples found in the Android echosystem reveals that malware developers bypass such vetting processes using various evasion techniques. Previous research shows that the use of dynamic code update techniques along with various forms of obfuscation makes it extremely hard for the state-of-the-art analysis tools to understand the behavior of an app [37, 90]. Thus, the use of these techniques in newly found malware is not surprising [91].

This is particularly daunting when static analysis is used in order to check the security of mobile applications (i.e., to detect the presence of malicious behavior). Indeed, Rastogi et al. [95] mention reflection among the techniques that make most of the current static analysis tools unable to detect malicious code. Additionally, static analysis is hindered by the code that evolves dynamically, because some parts of the code are impossible to discover or to analyze at installation time as they appear only at runtime. As a matter of fact, existing state-of-the-art static analyzers for mobile applications (e.g., [41, 57, 74]) assume that the code base does not change dynamically and the targets of reflection calls can be discovered in advance. This is a clear simplification of what happens in the real world, where many apps rely on code base updated at runtime.

At the same time, previous approaches that enhanced static analyzers of Java code in the presence of dynamic code update techniques (e.g., [48]) cannot be directly applied to Android due to the differences in the platforms (in Android, load-time instrumentation of classes is not available). Moreover, offline instrumentation also cannot solve the problem because this approach breaks the application signature, while some apps check it at runtime. If the signature does not correspond to some hardcoded value they may refuse to work. In case of malicious apps this check may be used to conceal illicit behavior.

In this chapter, we present StaDART - an analysis system that combines static and dynamic analysis to analyze apps in the presence of dynamic code updates. Instead of relying on modifications to the Android framework, StaDART utilizes ArtDroid which is an API hooking tool [56]. Furthermore, we integrate StaDART with DroidBot, a triggering tool for Android apps, to make the analysis system fully automated. StaDART is evaluated using a dataset of 2,000 real world apps and the results of our evaluation are presented here.

Contributions:

- We propose, design and implement StaDART, a system that interleaves static and dynamic analysis in order to reveal the hidden/updated behavior. By utilizing ArtDroid, we avoid modifications to the Android framework and make it largely framework independent. StaDART downloads and makes available for analysis the code loaded dynamically, and is able to resolve the targets of reflective calls complement-

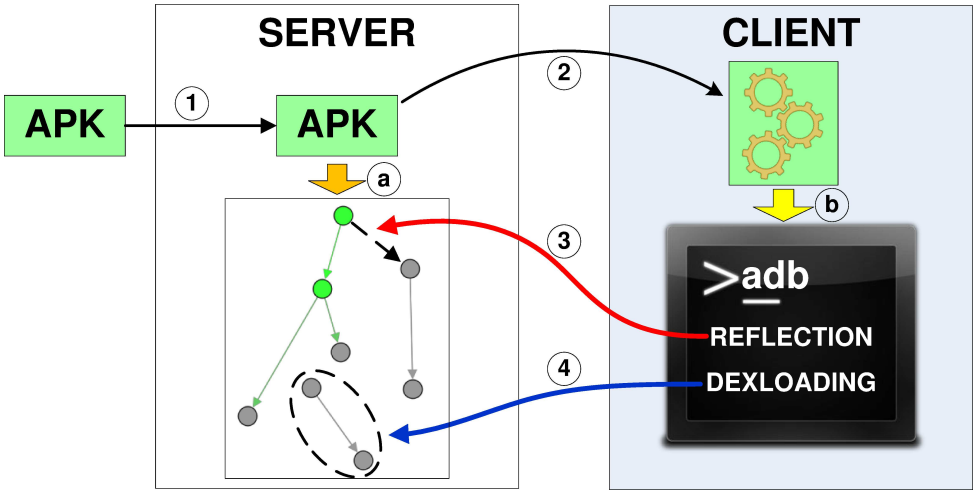


Figure 4.1: System Overview

ing app's method call graph with the obtained information. Therefore, StaDART can be used in conjunction with other static analyzers to make their analysis more precise.

- We integrate StaDART with DroidBot to make it fully automated and to ease the evaluation. Moreover, we analyze a dataset of 2,000 real world apps (1,000 benign and 1,000 malicious). Our analysis results show the effectiveness of StaDART in revealing behavior which is otherwise hidden to static analysis tools.

- We plan to release our tool as open-source to drive the research on app analysis in the presence of dynamic code updates.

4.2 An Overview of StaDART

The architecture of StaDART presented in Figure 4.1 comprises two logical components: a server and a client.

The static analysis of an application is performed on the server. In this respect, StaDART allows an analyst to easily plug in and use any static analyzer in its architecture. The static analyzer on the server builds the initial *method call graph* (MCG) of the app, integrates the results of the dynamic analysis coming from the client, and stores the results of the scrutiny. The client part of StaDART is based on an API hooking technique, ArtDroid, that intercepts dynamic code update APIs and captures dynamic behavior. The client part can be hosted either on a real device or an emulator. The client runs the application whenever the dynamic analysis is required.

In action, our system interleaves the execution of the static and dynamic analysis phases. However, to simplify the presentation, we describe them sequentially.

Preliminary analysis The server statically analyzes an app package and builds a MCG of the application (see Step *a* in Figure 4.1; solid arcs denote edges resolved statically). Dynamically loaded code cannot be analyzed during this phase and, thus, the corresponding nodes and edges are not present in the MCG. Further, the names of methods called through reflection may also not be inferred if they are represented as encrypted strings or generated dynamically. Still, a static analyzer can effectively detect the points in the MCG where the functionality of an application may be extended at runtime. Indeed, the usage of reflection and DCL requires to use specific API calls provided by the Android platform. The server detects these calls during the static analysis phase by searching for methods where DCL and reflection API calls are performed. We call these methods *methods of interest (MOI).*

Dynamic execution If any MOIs are detected in the application, StaDART installs the app on the client (Step *2*) and launches the dynamic analysis. The dynamic phase is exercised to complement the MCG of the app and to access the code loaded dynamically. The dynamic analysis is performed on a device (or an emulator) which uses ArtDroid for API interception and adding StaDART client side functionality. The added functionality logs all events when the app executes a call using reflection, or when additional code is loaded dynamically. Along with these events, the client also supplies some additional information, e.g., in case of a reflection call, the information about the called function, its parameters and the stack trace (that contains the ordered list of method calls, starting from the most recent ones) is added. In case of a DCL call, the path to the code file and the stack trace are supplied. The information collected by the client is passed back to the server side (Step *3*).

Analysis consolidation The server performs an analysis of the obtained information. In case a reflection call happens, the server complements the MCG of the app with a new edge (in Figure 4.1, it is represented by a dashed arc). This edge connects the node of the method that initiated the call through reflection (the node at the beginning) with the one corresponding to the called function (the node at the end).

When DCL is triggered, the client captures the location of the code file. Using this evidence, the server downloads the file (Step *4*) containing the code, and performs the static analysis on it. The MCG of the app is then updated with the obtained information (see the part of the MCG in the dashed oval in Figure 4.1). Additionally, for each

downloaded file the server analyzes whether it contains other MOIs. If it does, the list of the MOIs for the application is updated. This allows StaDART to unroll nested MOIs. The stack trace data for both the reflection and DCL cases is used to detect which MOI initiated the call.

Marking suspicious behavior In Android, some API calls are guarded by permissions. Since APIs protected by the permissions could potentially harm the system or compromise a user's data, the permissions must be requested in the `AndroidManifest.xml` file. However, there is no actual check on the permissions required to execute the written code and sometimes developers request more permissions than they actually use. In this case, those apps are called overprivileged. Many researchers, e.g., Bartel et al. [45], identified that malware, adware and spyware exploit additional permissions to get access to security sensitive resources at runtime.

Based on these considerations, we classify the following app behavior patterns as *suspicious*:

- An application dynamically loads code that contains API functions protected with permissions. Indeed, malware may use this approach to evade detection by static analyzers, as the security-sensitive code is loaded dynamically.

- An application uses reflection APIs to call an API method protected with a *dangerous* permission[1]. This functionality can be used, for instance, to send malicious SMS, which cannot be detected by static analysis tools because the name of the SMS sending function is encrypted and decrypted only at runtime.

StaDART automatically detects such suspicious patterns and raises a warning if such patterns occur during the analysis. Section 4.5 shows that indeed malware samples do expose such suspicious patterns.

In addition, we further analyze the parameters passed to methods called using reflection APIs. Indeed, a suspicious pattern, i.e., a reflective call to an API guarded with dangerous permission, in conjunction with suspicious parameters, e.g., a premium number in case of the `sendTextMessage()` API, helps in identifying malicious behavior concealed using reflection.

[1]Google classifies as "dangerous" permissions with higher-risk level that guard access to private user data or device controls [3].

4.3 Method Call Graph

Method call graphs (or function call graphs) identify the caller-callee relationships for program methods. These structural representations of programs are widely used for different purposes. In the scope of Android, method call graphs are used, e.g., to detect malware [67,70,76], to identify potential privacy leaks in applications [59,68,114], to find vulnerabilities [101] and execution paths for automatic testing [113].

StaDART extends the initial MCG generated with a traditional static analyzer with the information detected at runtime. Thus, if an application exposes dynamic behavior, all mentioned approaches can benefit from the expanded MCG obtained with StaDART.

Example To visualize the capabilities of StaDART and the process of method call graph expansion, we show the evolution using an example of a *demo_app*. Figure 4.2(a) shows the MCG of the app obtained with the AndroGuard static analyzer [2]. Figure 4.2(b) shows the one gained with StaDART before dynamic execution phase, and Figure 4.2(c) presents it with dynamic execution phase. The *demo_app* dynamically loads some code from an external `.jar` file at runtime and calls the loaded methods through reflection.

Figure 4.2(a) illustrates that AndroGuard identifies only the presence of ordinary methods and DCL calls (Ellipse 1) but no further analysis is done about those. Yet, Figure 4.2(b) shows that after preliminary analysis StaDART selects 3 paths, which are surrounded by dashed ellipses. Ellipse 1 shows that a MOI (the dark grey node) invokes a constructor (the dark green node) through reflection. Similarly, Ellipse 2 displays a method invocation through reflection. Ellipse 3 depicts that a DCL call (the red node) is performed in a MOI (the dark grey node).

During the dynamic analysis, StaDART adds the edges that are outlined by Ellipses 4-7 (see Figure 4.2(c)). These ellipses show the cases when the MOIs are resolved and corresponding nodes and edges are added to the MCG. Ellipse 4 shows that as a result of a DCL call (the red node) a new code file has been loaded (the pink node). Ellipse 7 shows that a class constructor (the grey node) is called through reflection. Ellipse 5 shows a method invoked through reflection. This method contains an API call protected by the Android permission indicated by the blue node in Ellipse 6. There are also nodes and edges that appear as a result of the analysis of the code file (the pink node) loaded dynamically. These nodes and edges are connected with the rest of the graph through the reflection *new instance* call (see Ellipse 7).

Ellipses 2, 3, 8, 9 show other types of connections possible among nodes in a MCG obtained with our tool. Ellipse 2 shows the connection between the class and its constructor, Ellipse 3 shows an ordinary relation between two methods, Ellipse 9 connects

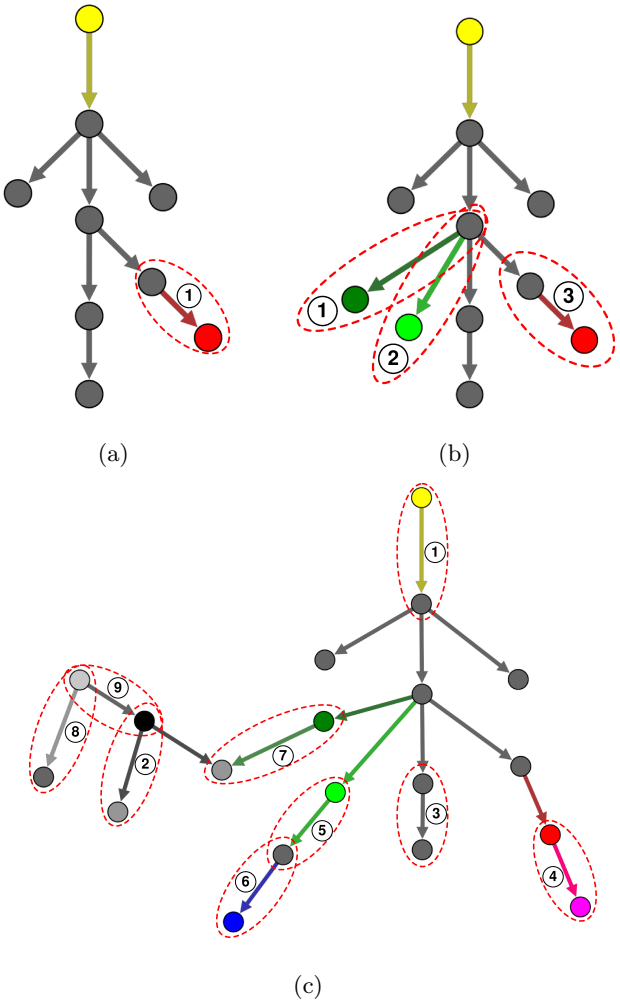


Figure 4.2: MCG of *demo_app* Obtained with a) AndroGuard b) StaDART after Preliminary Analysis c) StaDART after Dynamic Analysis Phase

the static initialization block and the class, and Ellipse 8 shows that the method is called from the static initialization block.

Each node type is assigned with a set of attributes, not shown in the figures. The analysis of values of these attributes can facilitate dissection of Android applications accompanied by the expanded method call graph. For instance, each method node is assigned with attributes, which correspond to a class name, a method name and a signature of this method. A permission node is assigned with a permission level along with the information about the API call that it protects.

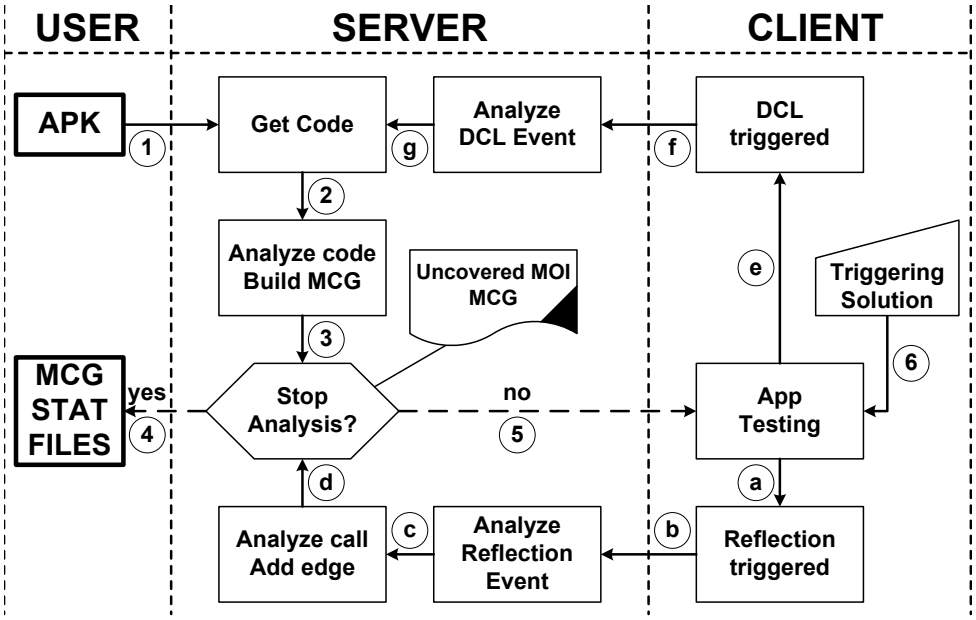


Figure 4.3: StaDART Workflow

4.4 Implementation

This section provides the implementation details of some key aspects of StaDART. The workflow of our system operation is shown in Figure 4.3. App analysis starts at the server side. All occurrences of reflection and DCL methods are identified in the code of the application under analysis. In case neither of them is found, StaDART builds a MCG of the app and exits. Otherwise, it starts the dynamic analysis on a device, which utilizes ArtDroid to intercept dynamic code update APIs and constitutes the client part of StaDART.

4.4.1 The server

The server side of StaDART is a Python program that interacts with a static analysis tool. Currently, StaDART uses AndroGuard [2] as a static analyzer. AndroGuard represents compiled Android code as a set of Python objects that can be manipulated and analyzed. However, StaDART can work with any static analysis tool that is able to analyze `apk` and `dex` files. To improve suspicious behavior detection we substituted the permission map embedded in AndroGuard (built for Android 2.2 in [63]) with the one generated by PScout [42] for Android 5.1.1, which is the latest API-permission mapping available in the research community.

The pseudo-code of the main server function is presented in Algorithm 1. The server starts the analysis of the provided app by extracting the `classes.dex` file (see Step *1*, *2*

Algorithm 1 App Analysis Main Function Algorithm

```
1: function PERFORM_ANALYSIS(inputApkPath, resultsDirPath)
2:     makeAnalysis(inputApkPath)
3:     // Check if there are MOI
4:     if !containsMethodsToAnalyze() then
5:         performInfoSave(resultsDirPath)
6:         return
7:     end if
8:     dev ← getDeviceForAnalysis()
9:     package_name ← get_package_name(inputApkPath)
10:    dev.install_package(inputApkPath)
11:    uid ← dev.get_package_uid(package_name)
12:    messages ← dev.getLogcatMessages(uid)
13:    loop
14:        msg ← dequeue(messages)
15:        // analyzeStadartMsg contains a switch statement
16:        // that selects a corresponding processing routine
17:        // shown in Algorithms 2 and 3 based on the msg type
18:        analyzeStadartMsg(msg)
19:
20:        // Quit if a user finishes analysis
21:        if finishAnalysis then
22:            performInfoSave(resultsDirPath)
23:            return
24:        end if
25:    end loop
26: end function
```

and *3* in Figure 4.3; Line 2 in Algorithm 1), and then dissects the extracted code. During this step StaDART searches for all the occurrences of reflection and DCL calls in the code. The list of searched patterns for these API calls is presented in Table 4.1.

If MOIs are found, StaDART selects a device (a real phone or an emulator) to perform the dynamic analysis on (Line 8) and installs the app under analysis (Line 10) onto the client (Step *5* in Fig. 4.3). After that the server obtains the UID of the installed package (Line 11) and starts a loop (Lines 13-25) that analyzes, one by one, the messages (Line 12) obtained using the *logcat* utility from the *main* log file of the Android system. Basically, each obtained message is represented in the JSON format and contains values for the following fields: *UID* (required), *operation* (required), *stack* (required), *class* (optional), *method* (optional), *proto* (optional), *source* (optional), *output* (optional). The value of the `UID` field is used to select the messages produced by the analyzed app. If the user stops the analysis, StaDART saves the results and finishes its execution.

The function `analyzeStadartMsg` (Line 18) analyzes the selected StaDART messages obtained from the client. It extracts the value of the `operation` field and based on this value selects the appropriate routine to analyze the message.

The routines for the reflection messages analysis are similar, so we consider them on the example when operation corresponds to *reflection invoke*. The algorithm for analysis of the *reflection invoke* messages is shown in Algorithm 2[2]. Lines 2 - 4 extracts the

[2]The algorithm for analysis of *reflection newInstance* messages is very similar so we do not show it.

Table 4.1: The List of Searched Patterns

Class	Method	Prot.
Dynamic class loading		
Ldalvik/system/PathClassLoader;	*< init >*	.
Ldalvik/system/DexClassLoader;	*< init >*	.
Ldalvik/system/DexFile;	*< init >*	.
Ldalvik/system/DexFile;	*loadDex*	.
Class instance creation through reflection		
Ljava/lang/Class;	*newInstance*	.
Ljava/lang/reflect/Constructor;	*newInstance*	.
Method invocation through reflection		
Ljava/lang/reflect/Method;	*invoke*	.

Algorithm 2 Analysis of the Reflection Invoke Message

```
1: function processReflInvokeMsg(message)
2:     cls ← message.get(JSON_CLASS)
3:     method ← message.get(JSON_METHOD)
4:     prototype ← message.get(JSON_PROTO)
5:     stack ← message.get(JSON_STACK)
6:     invDstFrCl ← (class, method, prototype)
7:     invPosInStack ← findFirstInvokePos(stack)
8:     thrMtd ← stack[invPosInStack]
9:     invSrcFrStack ← stack[invPosInStack + 1]
10:    for all invPathFrSrcs ∈ sources_invoke do
11:        invSrcFrSrcs ← invPathFrSrcs[0]
12:        if invSrcFrSrcs ≠ invSrcFrStack then
13:            continue
14:        end if
15:        addInvPathToMCG(invSrcFrSrcs, thrMtd, invDstFrCl)
16:        if invPathFrSrcs ∈ uncovered_invoke then
17:            uncovered_invoke.remove(invPathFrSrcs)
18:        end if
19:        return
20:    end for
21:    addVagueInvoke(thrMtd, invDstFrCl, stack)
22: end function
```

method name along with its class name and the prototype, which has been called through reflection. Line 5 gets the stack from the message. Line 7 searches for the first *reflection invoke* occurrence in the stack. The next stack entry corresponds to the method that has performed the reflection call `invSrcFrStack` (Line 9). Then in the loop StaDART compares this method with the list of MOIs extracted from the application executable (Lines 10 - 20). If the method is found StaDART complements the MCG with the obtained information (Line 15), and deletes it from the list of uncovered invoke MOIs (Line 17). Otherwise, it adds this method to the list of vague methods (Line 21). This information is later analyzed to see why the method calling reflection was not found in the application executable during the static analysis phase.

The processing function for the DCL messages is slightly different (see Algorithm 3). From the message received from the client the server extracts the source path of the

Algorithm 3 Analysis of the DCL Message

```
1: function processDexLoadMsg(message)
2:     source ← message.get(JSON_DEX_SOURCE)
3:     stack ← message.get(JSON_STACK)
4:     newFile ← dev.get_file(source)
5:     newFilePath ← processNewFile(newFile)
6:     dlPathFrStack = getDLPathFrStack(stack)
7:     if dlPathFrStack then
8:         srcFrStack ← dlPathFrStack[0]
9:         thrMtd ← dlPathFrStack[1]
10:        if dlPathFrStack ∈ uncovered_dexload then
11:            uncovered_dexload.remove(dlPathFrStack)
12:        end if
13:        addDLPathToMCG(srcFrStack, thrMtd, newFilePath)
14:        if !fileAnalyzed(newFilePath) then
15:            makeAnalysis(newFilePath)
16:        end if
17:        return
18:    end if
19:    addVagueDL(newFilePath, stack)
20: end function
```

file containing the code loaded dynamically (Line 2). Using this information, StaDART downloads the file locally (Line 4), and processes it (Line 5). This process includes computation of the file hash and copying the file into the results folder with a new filename, which includes the computed hash. The file hash allows us to check whether the file has been already loaded and avoid analysis of already checked code. Otherwise, the code analysis for MOIs is performed for the loaded code (Line 15). Function `getDLPathFrStack` (Line 6) searches for a pair of a DCL call and a MOI in the stack corresponding to the one extracted from the app executable. If this pair is found, then it is removed from the list of uncovered DCL calls (Line 11). Otherwise, StaDART adds the information about the dynamic class loading call into the list of vague calls (Line 19).

Notice that the presented algorithms are simplified versions of the ones actually implemented in the server part. For instance, in a real application it is possible that the same MOI acts like a proxy used to call different targets (e.g., the same method could be used to load different code files). The real algorithms implemented in StaDART are able to process these cases.

4.4.2 The client

The client side can run either on a real device or on an emulator. Using the emulator is more convenient because one can run the client and server on the same machine. The main drawback is that currently the Android emulator is quite slow. Moreover, mobile applications may suppress some functionality if they detect they are running in an emulated environment. With these limitations in mind, we implemented and tested our client on a real device. However, the code is not device-dependent so it can be easily ported to

an emulator or another device.

To capture the dynamic behavior offered by reflection and DCL, we intercept a number of Android API methods that provide an interface to DCL and reflection capabilities. A brief overview of these APIs is provided the earlier part of §4.4. Some of them have been modified across different Android versions moving their implementation to the native side (e.g., `java.lang.Class.newInstance` has only a native implementation in Android 6). To achieve dynamic instrumentation of Java-level APIs we used the approach proposed in ArtDroid [56] to intercept Java virtual methods. It intercepts all calls to monitored Java virtual methods including calls via Java reflection, native code or dynamically loaded code without any modification to both Android OS and the target app. In addition, we integrated native function hooking capabilities in ArtDroid by means of **inline hooking** technique. The client side employed by StaDART is completely Android version-agnostic and it is able to interpose custom code on both Java methods and native functions. Therefore, it can be used to analyze Android apps on any Android version intercepting DCL and reflection calls irrespective of the actual code representation (i.e., Java or native).

To support all available Android versions, we included in StaDART the capability of intercepting DCL and reflections calls according to the running Android version. In the following we describe methods intercepted by StaDART on both Dalvik and ART runtime. The code added by StaDART to perform requested analysis is not influenced by the underlying Android version.

To obtain the information related to DCL we added a hook to the method `openDexFile` of the `DexFile` class. This method is called when a new file with the code is opened. It gets three parameters as an input, where `sourceName` is of our interest. Moreover, we added a hook to the constructor of `DexClassLoader` class that is used to create a class loader that loads classes from JAR and DEX files. It gets four parameters as an input, where `dexPath` and `optimizedDirectory` are of our interest. The former specifies the complete path of the DEX file that is being loaded while the latter is the directory where the optimized version will be written to as a result of the compilation step. The added code forms a *JSON* message that contains the path to the file, from which the code is loaded (`sourceName`). Along with this information, the stack trace data and the *UID* of the process are also added into the message, which is then printed out to the *main* log file of Android.

To get the information about method invocation through reflection, a hook was placed into the `invoke` method of the `Method` class. As of the release of Android version 6, this method is defined as `public native`, thus the client will hook the appropriate function by means of the proper hooking engine, according to the running Android version. Each `Method` object has `declaringClass`, `name` and `parameterTypes` member fields, which

represent class name, method name and prototype of the invoked method, respectively. Moreover, `invoke` gets an array of Object type as input which represents the arguments intended for the target method. This information along with the stack trace is put into the StaDART message. Similarly, to log the information about new class creation through reflection, we put our hooks into the `newInstance` method of the `Class` and `Constructor` classes. As for the `invoke`, different hooks were added targeting `newInstace` code representation for both DVM and ART runtime.

Each StaDART message contains the stack trace information. Stack trace is a sequence of method calls performed in the current thread starting from the most recent ones. The information from a stack trace is usually used to find the origin of an exception in a program. In our case, the stack trace information is used to detect the MOI, which calls the reflection or DCL methods. In essence, a stack trace is an array of stack trace elements. Each stack trace element contains information about the class name, the method name and the line number of the method call in the source code. Unfortunately, using only this information it is not possible to uniquely identify the MOI, because we do not have access to the source code of the application. Moreover, due to function overloading it is possible to have several methods in a class with the same name. In the previous version of StaDART (i.e., StaDyna), we had modified the `StackTraceElement` class so that it can store the information about the method prototype, but this approach is not feasible when it comes to dynamic instrumentation. To overcome this limitation and detect MOIs from stack trace data even when they appear multiple times with same name but different prototype, we employed a hybrid approach. First, we statically detect potentially ambiguous methods declared in the target app and for each method found we store its prototype information. Then, we dynamically instrument the app to insert a shadow method that is basically an empty wrapper to distinguish the ambiguous method. It is named as the concatenation of original method name plus its prototype stored by the previous step. In this way, we are able to distinguish target MOIs by looking for them into the stack trace data as it is normally returned by the Android OS. In fact, method name and its prototype allow us to uniquely identify a method in a class.

A StaDART message has a header and a body. To distinguish StaDART messages from other log messages we add a special marker to the header. The second part of the message header is the part number. Currently, there is a limit on the length of the Android log entries specified by the constant `LOGGER_ENTRY_MAX_PAYLOAD`. To overcome this problem, we added the functionality to the client that allows it to split a message into several parts. The server takes care of assembling the original message.

4.5 Evaluation

Experiment Setup and Test Suite: This section describes our application test suite and reports on the results of our experiments. In order to evaluate StaDART, we tested it on a dataset of real world applications, both benign and malicious. The server runs on a machine with 3.2 GHz Intel Core i7 processor and 8 GB DDR3 memory. The client is a Google Nexus 6 smartphone running stock Android OS version 7.1.1 connected to the server using a standard USB cable. The evaluation test suite consists of a set of 1,000 benign and 1,000 malicious applications. The benign applications were downloaded on December 2016 and selected based on their popularity. The malware samples were selected from Drebin [40] dataset populated by 5,560 applications from 179 different malware families collected in the period of August 2010 to October 2012.

Evaluation Goal: Inline with the aim of StaDART, i.e., uncovering dynamic behavior, we set certain research questions that this evaluation should answer as our evaluation goal.

- **RQ1:** How widespread is the use of these dynamic code update features in the analyzed dataset and does StaDART reveal dynamic behavior in each of the analyzed app?
- **RQ2:** How effective is StaDART in expanding the MCGs? How expansion of MCGs due to dynamic behavior differ in the malicious and benign dataset?
- **RQ3:** Does StaDART reveal potentially dangerous behavior, i.e., reveal nodes guarded with permissions? How do they differ in benign and malicious apps?
- **RQ4:** Is there a correlation between the captured dynamic behavior and the APIs used for dynamic code updates, e.g., DCL or reflection?
- **RQ5:** Do the analyzed apps show suspicious behavior, i.e., use additional new permissions which are not used in the initial MCG? How does this behavior differ in malicious and benign apps?

Analysis Results: Figure 4.4 illustrates the prevalence of dynamic code update APIs in the analyzed dataset and the effectiveness of StaDART in expanding the MCGs. It shows the percentage of apps with invoke, newInstance and DCL among both benign and malicious app dataset. The right most bar represents the percentage of apps where StaDART expanded the MCG. In the dataset, close to 90% of the apps use `invoke` and/or `newInstance` APIs. Similarly, 48% of the apps use DCL feature which is considerably higher to previous analysis results [110] (first part of RQ1). Increase in the number of

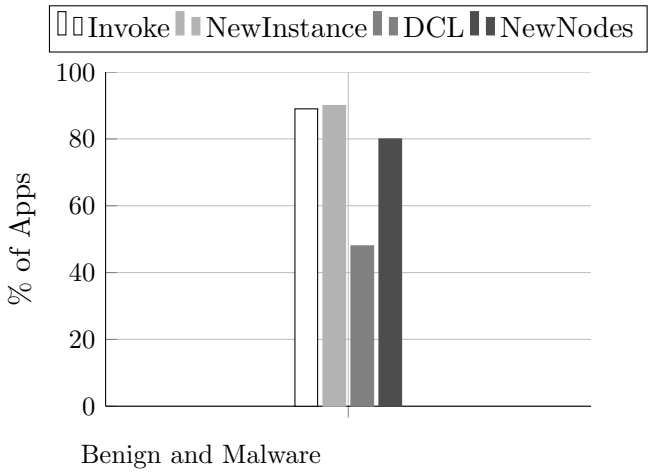


Figure 4.4: Prevalence of Reflection/DCL and StaDART effectiveness in expanding MCG

apps using DCL could largely be related to the increasing complexity of the Android apps. StaDART was able to expand the MCG by at least one node in 80% of the analyzed apps (second part of RQ1).

In order to answer the question whether StaDART expands the MCGs over the entire dataset and by how much?, Figure 4.5 shows the effectiveness of StaDART for analyzed dataset, both benign and malicious. It shows the average percentage increase in the number of nodes, edges, nodes with normal permission and nodes with dangerous permissions. Clearly, the lower percentage increase is attributed to apps that use only reflection as dynamic code update feature. The MCG expansion in these apps, which do not use DCL, is minimal and more or less similar in benign and malicious apps (RQ2).

To clarify the role of DCL in MCG expansion and dynamic behavior, we extracted the results from apps that use DCL. Figure 4.6 shows the effectiveness of StaDART when the apps use DCL. It shows the average percentage increase in the number of nodes, edges, nodes with normal permissions and nodes with dangerous permissions. It clearly shows a considerably higher increase in the number of nodes, edges and nodes guarded with permissions (both normal and dangerous). In addition, it can be seen that the malicious apps hugely increase their code base when they use DCL (RQ4). Similarly, the number of nodes guarded with permissions for malicious apps doubled or in some cases quadrupled (RQ3). This clearly indicate that malicious apps make use of sensitive APIs in the loaded code. We also check the added nodes for Signature level permission and SignatureOrSystem level permission. However, we did not observe a noticeable increase in the number of nodes guarded with these permissions.

Although, the high increase in the number of nodes guarded with dangerous permissions is indeed suspicious, we investigate the analysis results further for a more suspicious

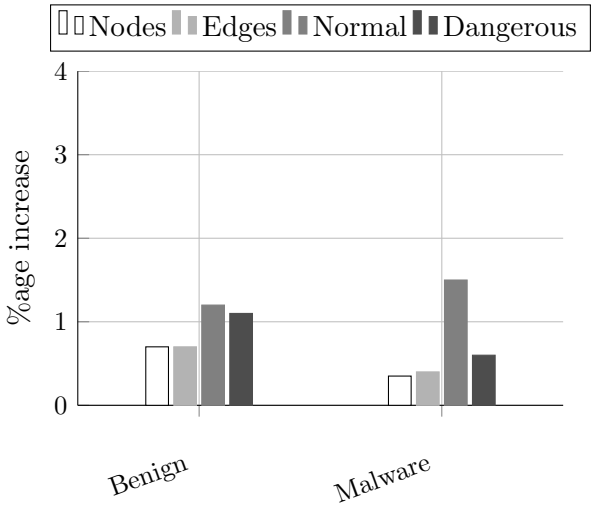


Figure 4.5: MCG Expansion

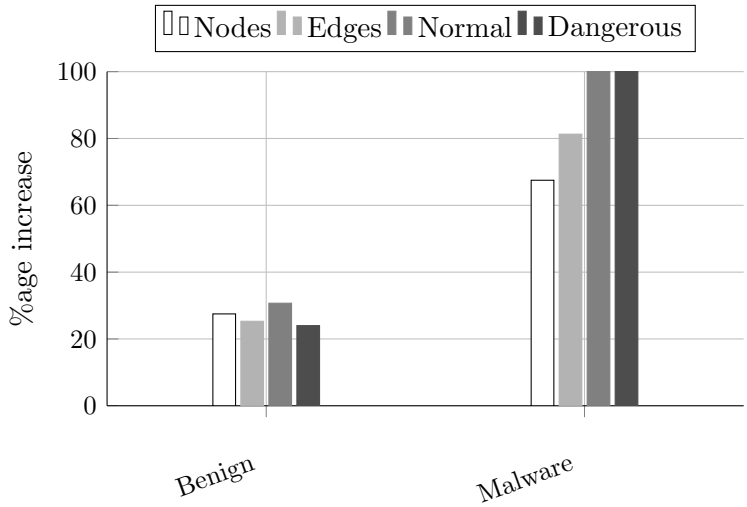


Figure 4.6: MCG Expansion when apps use DCL

malware behavior. In practice, malicious payloads are packaged inside legitimate apps and their manifest files are modified to cover for the extra permissions needed by the payload. In this scenario, the final MCG of the app contains nodes guarded with new permissions, i.e., those not found in the initial MCG. Figure 4.7 and Figure 4.8 show the distribution of apps based on increase in the number of nodes guarded with permissions in the form of pie-charts, in benign apps and malware, respectively. Here we discuss only those apps which use DCL. The white part shows the percentage of apps with no increase in the number of nodes guarded with permissions, whereas the grey part represents the percentage of apps with increase in the number of nodes guarded with permissions. The darker grey part shows the percentage of apps where new permissions are used in the dynamically added part using StaDART.

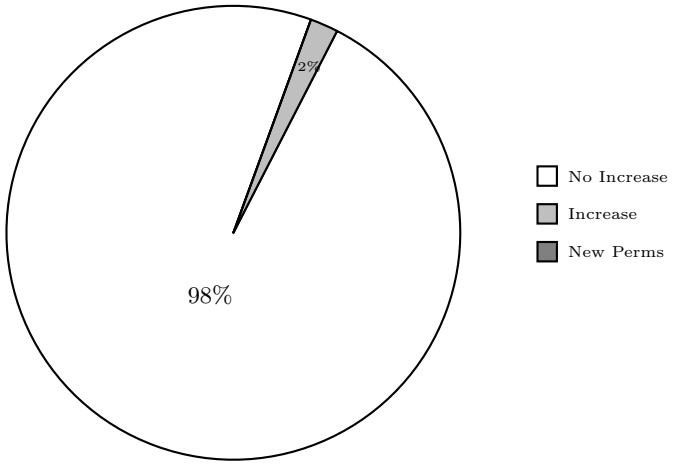


Figure 4.7: Benign Apps with increase in permission nodes

The pie-chart for the benign apps shows that a very small fraction of the apps observe an increase in the number of nodes guarded with dangerous permissions. In contrast, a considerably higher number of malicious apps reveal such behavior. Also, in none of the benign apps in the dataset, the loaded coded contained nodes guarded with new dangerous permission. However, all the malicious apps in the dataset that loaded code dynamically contained nodes guarded with at least one new dangerous permission (RQ5). Moreover, a further analysis of the loaded code in malicious apps reveals a pattern of dangerous permissions, e.g., `READ_PHONE_STATE` and `INTERNET`, that could be associated with malicious functionality, such as privacy leakage, etc.

Also, noteworthy here is the fact that the revealed behavior is only due to triggering of a small fraction of the total MOIs. Albeit the most advance automated triggering tool in the research community, DroidBot does not serve well for app exploration from a security point

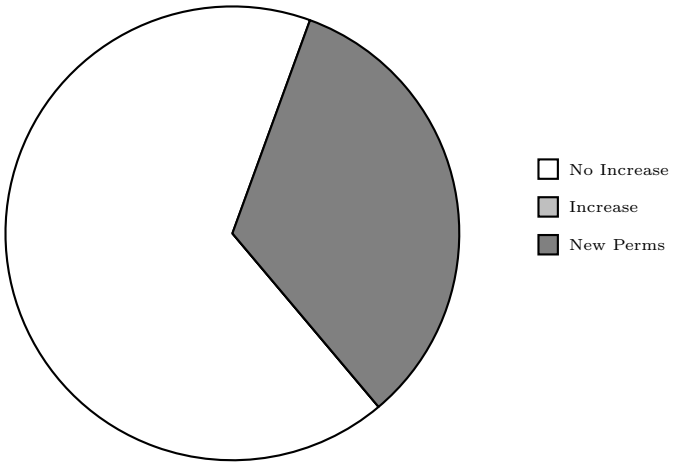


Figure 4.8: Malicious Apps with increase in permission nodes

of view. Taking into account the low exploration that DroidBot achieved in most of the apps and the suspicious results that we observed, the actual hidden suspicious/malicious behavior could be alarming.

Our results show evidence that malware samples are more overprivileged (they contain more permission types required for the code loaded dynamically), so it is valid to identify the apps as suspicious if they are overprivileged. Yet, as benign apps can be overprivileged too, more research is required to understand if an application is benign or malicious, and StaDART can be handy in exploration of this topic.

4.6 Discussion

For any dynamic (or hybrid for that matter too) analysis tool, coverage is the main limiting factor and StaDART is no different in that regard. For StaDART the coverage of MOIs (the ratio between the number of executed MOIs at least once and total number of discovered MOIs) is especially important. In order to achive higher MOI coverage, we explored if the tools like *monkey* [34] can be handy. However, in our experiments we found out that pseudo-random events generated by the tool do not produce tolerable coverage values for MOIs. Therefore, we opted for a more advance automated triggering tool, DroidBot, to trigger MOIs. However, as discussed in the previous section, even DroidBot did not achieve reasonable coverage of MOIs.

A possible approach to achieve satisfying values is to use systems like SmartDroid [113]. SmartDroid allows an expert to specify sensitive API methods required to be triggered. In case of StaDART the sensitive API methods correspond to reflection and DCL calls. Other possible tools, which may be useful in developing fully automatic approach, are [46,94,108].

Another possible direction to reduce the dependence on the triggering tool is to resolve as many targets of reflection calls as possible statically, at least those which are represented by constant strings [74]. The analysis performed in [63] has shown that it was possible to resolve automatically the targets of reflection calls in 59% of applications that used reflection. At the same time, the analysis was performed for the "closed world" scenario, which is not realistic, given that dynamic class loading is a popular technique for modern apps. Consequently, we can minimize the more expesive dynamic part of the analysis.

Usually, dynamic analysis allows an expert to explore only one execution path at a time. However, dynamic traces may differ depending on the context of the execution, e.g., some methods may contain calls invoked with parameters affecting the reflection call target. Therefore, another direction for improving StaDART is to incorporate information obtained during different runs of analysis.

StaDART has also other limitations. Its analysis is based on the UID of an application. However, it is possible in Android that several apps have the same UID. In this case, StaDART will also collect the information produced by other apps with the same UID. At the same time, this information will not be used to complement MCG, but will be added to the category of vague calls that need to be manually analyzed later.

4.7 Related Work

Being the most popular mobile OS, Android has won this position due to the openness of its ecosystem and the ease with which developers can publish apps on Google Play and third-party markets. Yet the openness comes at the price of large volumes of malware apps polluting the ecosystem. One approach to tackle security and privacy of mobile apps is to extend the security controls of the platform to detect misbehaving apps or to enforce the desired security policy [55, 112]. Solutions following this approach, often require to modify the system image.

Another approach, more relevant to StaDART, consists in the analysis of the mobile application code. Many static and dynamic analysis techniques have been proposed for Android. The ded system [59] re-targets Dalvik bytecode into Java class files that can be analyzed by the variety of tools developed for Java. In the original paper [59] the FortifySCA static analysis toolset was used for detecting vulnerabilities and dangerous functionality, like leaking the device IMEI. DroidAlarm [114] performs static detection of privilege-escalation vulnerabilities in apps by constructing paths in inter-procedural call graphs from a sensitive permission to a public interface accessible to other apps. StaDART complements these static analysis techniques by completing inter-procedural call graphs.

Hu et al. proposed to explore functional call graphs (FCG) and rely on graph similarity

metrics to detect malware based on known malware graph patterns [76]. Gascon et al. continue this research direction for Android with a technique to detect malware apps based on comparing FCGs that are mined with AndroGuard [67]. StaDART can complement these techniques by providing more precise graphs required for analysis.

TaintDroid was among the first dynamic analysis tools for Android apps [58]; it allows to track propagation of information via the TaintDroid infrastructure-equipped smartphone software stack. Sources of sensitive information are typically the device sensors or private user information, and sinks are network interfaces; thus the main scope of TaintDroid is detection of privacy leaks. This approach is followed by DroidScope [108]. DroidScope allows to emulate app execution and trace the context at different levels of the Android software stack: at the native code level, at the Dalvik bytecode level, at the system API level, and at the combination of both native and Dalvik levels. While executing an app in DroidScope a security analyst can track events at different levels and instrument parameters of invoked methods to discover a malicious activity.

Dynamic analysis techniques are especially difficult to automate due to the need of emulating a comprehensive interactions of applications with the system and a user (UI interactions). Several approaches are proposed to automate the triggering of UI events, from random event generation [75] to more advanced approaches like AppsPlayground [94] and SmartDroid [113]. However, all of them still have many limitations on the type of events they can handle and the coverage.

Poeplau et al. [90] have identified the problem of dynamic code loading in Android apps. The authors selected possible vulnerable patterns of dynamic code loading and built a tool that can analyze Android apps for the found patterns. Moreover, they propose to use whitelists to prevent dynamic code loading that can potentially expose dangerous behavior. Whitelisting prevents unauthorized code from running. To get authorization the code must either signed [111] and its signature has to be included into a special list distributed by trusted authorities. However, as mentioned in the article [90], extraction of the dangerous behavior is a difficult problem by itself, especially when the protected API is called through reflection. In contrast, StaDART aims not at preventing this loading (because a lot of legitimate apps use it and extra complications will not be welcomed by the developers) but at its analysis.

Reflection and Dynamic Class Loading in Java Gaps in the static analysis techniques in the presence of dynamic class loading, reflection and native code were previously studied for Java. For example, similarly to our approach, in [73] a pointer analysis (based on program call graphs) technique for the full Java language is extended by addressing dynamic class loading and reflection via an “online” analysis, when a call graph is

built dynamically based on the program execution, and dynamic class loading, reflection and native code are treated in real time by modifying the pointer analysis constraints accordingly.

A run-time shape analysis for Java is investigated in [49]. Traditionally a shape analysis operates based on the call graph of a program, and it allows to conclude how the heap objects are linked to each other (e.g., if a variable can be accessed from several threads). Yet in Java the call graph produced from a program can be incomplete; and [49] suggests how to execute an incremental shape analysis when the call graph evolves dynamically. Our proposal does not involve a shape analysis, yet the ideas behind our proposal and [49] are similar.

Livshits, Whaley and Lam have studied the reflection analysis for Java [80]. They propose refinement for the static algorithms to infer more precise information on approximate targets of reflective calls, as well as to discover program points where user needs to provide a specification in order to resolve reflective targets.

Relevant to StaDART is TamiFlex [48] that complements static analysis of Java programs in the presence of reflection and custom class loaders. Using the load-time Java instrumentation API TamiFlex modifies the original program to perform logging of class loading and reflection call events. This information is used to seed a tool that performs static analysis of the program having the information obtained during the dynamic analysis phase. This work differs from StaDART in several aspects. First, TamiFlex uses a special Java API that is not available in Android. Second, although in Android it is possible to instrument an app before loading it on a device (offline instrumentation), some Android apps check the application signature in its code that is changed during the patching. Thus, for these applications the TamiFlex approach will not work in Android. Third, TamiFlex requires some debug information (the line number of the function call) to be present. In Android during the obfuscation phase this kind of information may be deleted from the final package. Therefore, the TamiFlex approach will not work, while StaDART is able to process correctly this case due to dynamic API hooking using ArtDroid.

4.8 Chapter Summary

Today mobile applications make an extensive use of dynamic capabilities, namely reflection and dynamic class loading, available in the Android OS. Being adopted from Java, these techniques in Android incur an additional threat because the loaded code receives the same privileges as the loading one. Malicious apps can leverage these facilities to conceal their malicious behavior from analyzers.

In this chapter we presented StaDART, a technique that interleaves static and dy-

namic analysis in order to scrutinize Android applications in the presence of reflection and dynamic class loading. Our approach makes it possible to expand the method call graph of an application by capturing additional modules loaded at runtime and additional paths of execution concealed by reflection calls. In order to produce the expanded call graph, StaDART relies on code interposition based on a dynamic API hooking technique. It does not require any modification to the Android framework or the application itself. As observed from the evaluation results malware apps were more inclined to exhibit a suspicious increase in dangerous permissions after dynamic loading of new code, proving that StaDART is an effective hybrid approach able to detect and capture apps' dynamic capabilities used at runtime.

The results produced by StaDART can then be fed to the state-of-the-art analyzers in order to improve their precision (for instance, a reachability analysis will be more precise over the expanded MCG than over the original one). Thus, StaDART may help malware analysts by increasing their ability to detect suspicious samples.

Chapter 5

TeICC: Targeted Execution of ICC

Effective analysis of applications is essential to understanding their behavior. Two analysis approaches, *i.e.*, static and dynamic, are widely used; although, both have well known limitations. Static analysis suffers from obfuscation and dynamic code updates. Whereas, it is extremely hard for dynamic analysis to guarantee the execution of all the code paths in an app and thereby, suffers from the code coverage problem. However, from a security point of view, executing all paths in an app might be less interesting than executing certain potentially malicious paths in the app. In this chapter, we present a hybrid approach that combines static and dynamic analysis in an iterative manner to cover their shortcomings. We use targeted execution of interesting code paths to solve the issues of obfuscation and dynamic code updates. Our targeted execution leverages a slicing-based analysis for the generation of data-dependent slices for arbitrary methods of interest (MOI) and on execution of the extracted slices for capturing their dynamic behavior. Motivated by the fact that malicious apps use Inter Component Communications (ICC) to exchange data [78], our main contribution is the automatic targeted triggering of MOI that use ICC for passing data between components. We implement a proof of concept, TeICC, and report the results of our evaluation.

5.1 Introduction

Mobile apps are analyzed for malicious contents before being published to app stores, such as Google Play Store. The analysis usually involves two categories, *i.e.,* static (reasoning about an app without executing it) and dynamic (executing apps in a controlled environment and understanding their behavior). Both of these analysis techniques have their pros and cons. While the former provides an over-approximation of what a piece of code actually performs, the latter misses certain execution paths due to limited duration of the analysis and the triggering problem. Static analysis also suffers from code obfuscation

problems and dynamic code updates. Dynamic analysis, on the other hand, provides a solution to these problems, but requires test cases which could execute a major/required portion of the code.

Execution of certain code paths in mobile apps depends upon a combination of various user/system events. Generally, it is hard to predict inputs which can stimulate the required behavior in these apps. This feature of mobile apps is widely used by malware developers to conceal malicious functionality.

Code coverage is a well-known limitation of dynamic analysis approaches. However, for the purpose of security analysis rather than testing, it is required to stimulate/reach only specific points of interest (POI) in the code rather than stimulating all the code paths in an app. In literature, researchers have focused mainly on providing inputs to make an app follow a specific path. Providing the exact inputs and environment becomes very hard as different apps may require different execution environments. Moreover, not all inputs can be predicted statically, because of obfuscation or other hiding techniques.

In this chapter, we propose a fully automated hybrid system which uses a slicing based approach for target triggering of a given MOI. It performs static data-flow analysis [38, 65] based on program slicing technique [105] to extract target slices which hold data-dependency with the parameters used by the given MOI. Moreover, our slicing approach permits slice extraction following the ICC flow across different app components. Importance of ICC in malware for sharing sensitive data is shown by Bodden *et al.* in [78]. However, to the best of our knowledge, none of the existing approaches [43,93] for targeted triggering support the extraction of interesting paths across different Android components.

In our proof of concept, TeICC, we leverage an enhanced version of SAAF to achieve program slicing [74]. We modified SAAF adding more sensitivity and support for ICC using a System Dependency Graph (SDG) (cfr. §5.3). Besides that, TeICC, employs a modified version of Stadyna [110] which integrates ArtDroid [56] to support dynamic execution of the extracted slices to resolve obfuscation and dynamic code updates. It runs on a real device/emulator with no modification to the Android framework.

TeICC operates in an iterative manner where a SDG helps extraction of slices across multiple components for targeted execution and targeted execution overcomes the limitations of static analysis by resolving obfuscation and dynamic code updates. It results in construction of an improved SDG and extraction of extended slices for better analysis of apps.

Contributions:

- We extend the backward slicing mechanism to support ICC, *i.e.,* extract slices across multiple components. Moreover, we enhance SAAF to perform data flow analysis with context-, path- and object-sensitivity.

- Targeted execution of the extracted inter-component slices without modification to the Android framework.
- We design and implement a hybrid analysis system based on static data-flow analysis and dynamic execution on real-world device for improved analysis of obfuscated apps.

5.2 Motivating example

The rising use of techniques such as obfuscation and ICC for information leakage by newly found malware motivates this work. Existing analysis approaches generally do not support information-flow analysis across multiple app components in obfuscated apps. As a result, malware use these features for evading these analysis tools. As reported by different antivirus companies [1, 27, 29, 31, 33], obfuscated malware has started to show up more frequently. This trend poses a strong challenge for the current static analysis tools, which are unable to automatically analyze apps in the presence of obfuscation or dynamic code loading. Furthermore, as demonstrated in [78], the ICC mechanism offered by Android is used by both normal and malicious apps for passing data between different Android components.

Listing 5.1: MessageReceiver

```
public class MessageReceiver extends BroadcastReceiver {
public void onReceive(Context context, Intent intent) {
    SharedPreferences v3 = ...
    Map v0 = this.retrieveMessages(intent);
    Iterator v6 = v0.keySet().iterator();
    while(v6.hasNext()) {
        Object v2 = v6.next();
        Object v5 = v0.get(v2);
        Intent v4 = new Intent(context, SendService.class);
        v4.putExtra("number", ((String)v2));
        v4.putExtra("text", ((String)v5));
        context.startService(v4);
    [...]
    }
} }
```

Listing 5.2: SendService

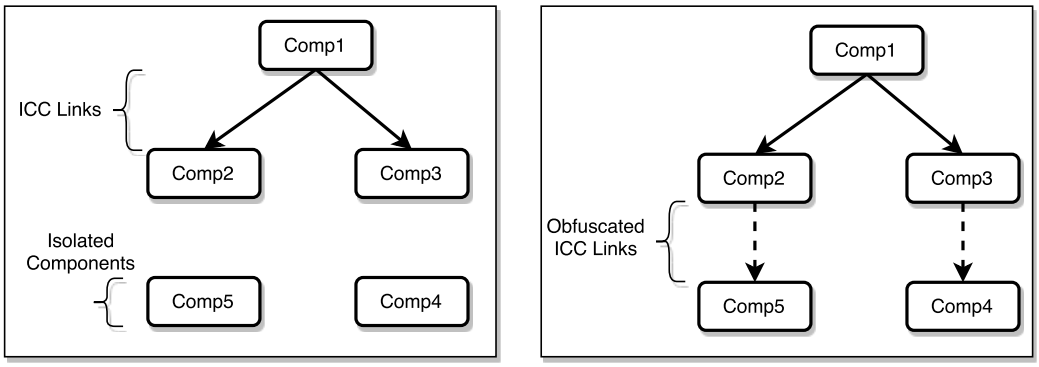
```
public class SendService extends IntentService {
    protected void onHandleIntent(Intent intent) {
        if(v1.equals("REPORT_INCOMING_MESSAGE")) {
            Sender.request(this.httpClient, "http://37.1.204.175/?action=command
    ", RequestFactory
                    .makeIncomingMessage(v2, intent.getStringExtra("number"),
    intent.getStringExtra(
                    "text")).toString());
            return;
        }
```

```
9  }}
10 public class Sender {
11     public static JSONObject request(DefaultHttpClient hc, String serverURL,
       String data) throws Exception {
12         HttpPost v1 = new HttpPost(serverURL);
13         StringEntity v3 = new StringEntity(data, "UTF-8");
14         HttpResponse v2 = hc.execute(((HttpUriRequest)v1));
15     }
16 }
```

To ease the understanding of our contributions, we are going to introduce a code snippet of a real-malware sample reported by FireEye in [22]. Listing 5.1 shows the de-obfuscated version of the code used to intercept and then report the incoming SMS. The forwarding process is defined in a service component. The *MessageReceiver* (line 2) is called for each incoming SMS and then an Android service is started by an Intent (line 12). The number and text data are stored within the Intent (lines 10, 11). Note that the original obfuscated malware uses string encryption on the constant string along with Java reflection for calling ICC methods. Then the started service, shown in Listing 5.2, gets data from the incoming Intent (lines 5, 6) and leaks (line 14) SMS number and text via a remote server connection (the server IP address string was obfuscated as well).

To the best of our understanding, static analyzers [69, 78, 87, 88], are not successful in analyzing such cases because of both encryption and reflection techniques used by this malware sample. Moreover, also hybrid approaches proposed in [93] and [43] cannot properly analyze the sample because they lack support for ICC.

5.3 Our approach



(a) SDG - First Iteration (b) SDG - Second Iteration

Figure 5.1: SDG during the first and second iteration. Comp: Component

During a normal execution of an Android app, the control transfers between various components based on certain user or system events. In order to trigger a specific piece of

code inside an app, it is important to provide the exact user/system events in a specific order to make it follow the target path. We take a slightly different approach based on isolating target execution paths from within the app and executing them; thereby avoiding to rely on user/system events. Target execution paths are isolated by means of a slice extraction mechanism that leverages backward program slicing across various components of the app.

5.3.1 Slice Extraction

Backward code slicing is a static analysis technique that identifies the data flow to a certain variable v at point p in the program while tracking the code in backward direction. In the process it identifies all the code instructions I which directly or indirectly affect the value of v at point p. This set of instructions I is called a backward slice. An important property of a backward slice is that it can execute independently of the rest of the program.

We leverage this property of the backward slice in our approach. Our backward slicing mechanism starts from a target point and traverses the code in backward direction until it reaches an entry point in the app. Instructions corresponding to each target point are marked accordingly and extracted from the program to be refined and executed separately. In simple apps, a backward slice may belong to a single app component. However, the complexity of apps these days demands for more inter component communication. Therefore, approaches based on extracting slices from only a single component might miss critical information passed through ICC.

5.3.2 Inter-Component Communication

Our approach extends backward slicing across multiple app components. We build a System Dependency Graph (SDG) before starting slice extraction. A SDG is a representation of the program highlighting the inter-connectivity and program flow among various components. Figure 5.1 provides a simplified representation of a SDG. The *nodes* in the SDG represent components which are connected to each other with directed *edges* where the direction shows the flow of execution from one component to the other. A SDG also provides information about the nature of the components, *i.e.*, activity, service, broadcast receiver, etc. This information is not shown in the figure where we simply refer to them as Comp$X$. The backward slicing assisted by the SDG then extracts slices which may contain instructions from multiple components.

Listing 5.3: Extracted and Refined Slice

```
1 public class MessageReceiver_fake extends BroadcastReceiver {
2     public void onReceive(Context context, Intent intent) {
3         Map v0 = MessageReceiver_fake.retrieveMessages(intent);
```

```
4           Iterator v6 = v0.keySet().iterator();
5           while(v6.hasNext()) {
6               Object v2 = v6.next();
7               Object v5 = v0.get(v2);
8               Intent v4 = new Intent(context, SendService_fake.class);
9               v4.setAction("REPORT_INCOMING_MESSAGE");
10              v4.putExtra("number", ((String)v2));
11              v4.putExtra("text", ((String)v5));
12              context.startService(v4);
13          }
14      }
15  }
16  public class SendService_fake extends IntentService {
17      public void onCreate() {
18          [...]
19          this.httpClient = new DefaultHttpClient();
20      }
21      protected void onHandleIntent(Intent intent) {
22          String v2 = SendService.settings.getString("APP_ID", "-1");
23          Sender.request(this.httpClient, "http://37.1.204.175/?action=command",
        RequestFactory
24                  .makeIncomingMessage(v2, intent.getStringExtra("number"), intent
        .getStringExtra(
25                  "text")).toString());
26  }}
```

Our approach uses an iterative mechanism which works in a CreateSDG-ExtractSlice-Execute cycle. Each phase in this cycle provides input for the next phase. SDGs help in extracting slices across multiple components and extracted slices simplify execution of target points in the app. Similarly, the execution phase helps in resolving obfuscation and dynamic code updates which leads to improved creation of the SDG in the next iteration. Figure 5.1(a) and 5.1(b) show a SDG in two iterations. In the first iteration, TeICC finds the obvious non-obfuscated ICC links only. Therefore, the SDG contains Comp4 and Comp5 which are isolated components. The second iteration reveals that the app has obfuscated ICC links from Comp2 to Comp5 and from Comp3 to Comp4 as shown in Figure 5.1(b). This process carries on until the SDG reaches a stable point. At this stage, all the obfuscated links are resolved and the slices are ready for the final execution to capture and analyze suspicious behavior.

Most of the state-of-the-art analysis tools would fail to extract the complete slice in the case of the sample described in §5.2. However, TeICC allows the extraction of such data-dependent slices because it can follow the ICC flow across multiple components. Listing 5.3 shows the resulting slice extracted and refined by TeICC; it shows the corresponding aggregated Java code to ease the understanding.

5.3.3 Slice Execution

The extracted slices are put together in one or more resultant components where the irrelevant instructions are removed as shown in Listing 5.3. Similarly, the `AndroidManifest.xml` file is also modified to include entries for these resultant components and remove irrelevant ones. The enriched app is then assembled and signed. The flow of the app is hijacked using a stub code so that it executes the resultant component after it is launched. The app is then installed and run on a real device or emulator. The target slice is executed once the resultant component is started. Similarly for each extracted slice, a resultant component is added to the app. The app is observed during execution of the resultant components to capture the target behavior of the app.

5.4 Design and Implementation

TeICC is a hybrid system composed of various static and dynamic analysis modules. Here we describe the design, implementation and work-flow of TeICC.

5.4.1 Overview

Figure 5.2 illustrates a high level design of TeICC. TeICC consists of a Static Analyzer, a Slice Analyzer and an App Executor module. The Static Analyzer further relies on a disassembler to convert an app's compiled Dalvik bytecode to Smali code [71]. The Smali files are then taken as input by the SDG Generator to create the first iteration of a SDG. The Slice Extractor assisted by the SDG performs backward program slicing on the Smali files to extract target slices, for the list of MOIs provided as an XML file, across multiple components. The Slice Analyzer module refines the slices by removing irrelevant instructions and merging them in the resultant components as shown in Listing 5.3. The Slice Assembler part of this module assembles the modified app Smali files and signs the APK file.

The App Executor module takes the app under analysis as input and installs it on a device for dynamic execution of the target slices. The purpose of the execution of target slices is two-fold. One for de-obfuscation and resolving the targets of dynamic code updates, such as reflection and dynamic class loading. The other purpose is to capture any sensitive/malicious behavior. For handling dynamic code updates, we utilize a modified version of Stadyna [110] that can resolve the targets of reflection and handle the code loaded dynamically. In order to capture sensitive behavior of app, we leverage an API hooking tool, ArtDroid [56], to hook sensitive APIs, such as dynamic code update APIs or the `sendTextMessage()` API, etc.

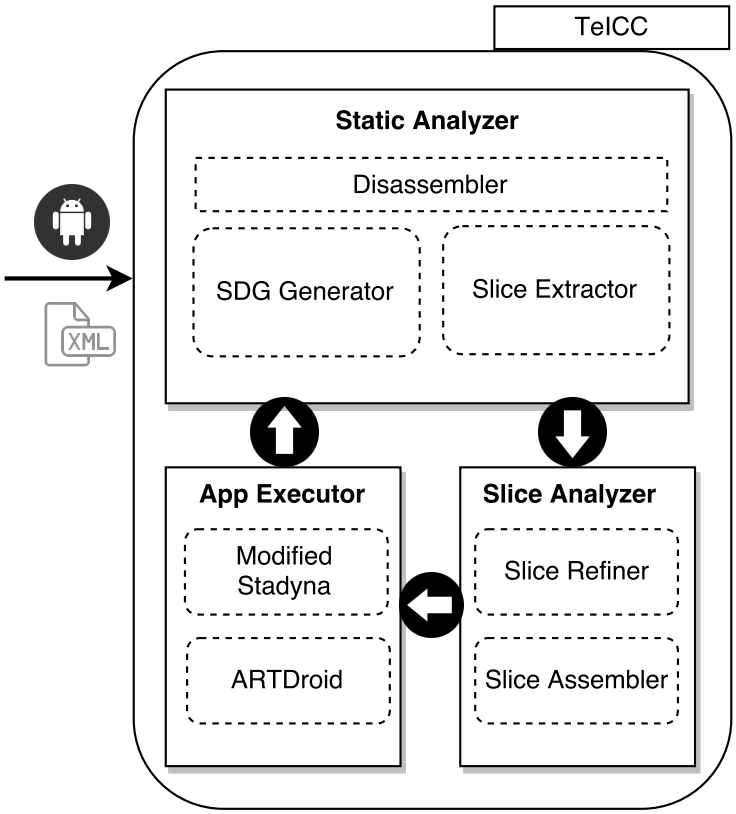


Figure 5.2: TeICC Design

5.4.2 Enhancement to Backward Slicer

Our backward slicing mechanism is based on an enhanced version of SAAF which performs static analysis of Android apps on Smali code [74]. We added certain features to it to overcome some of its limitations.

We extended SAAF to perform backward slicing across multiple components. This extended backward slicing is guided by a SDG when the start of a component is encountered. The backward slicing process continues until it reaches an entry point of the app according to the SDG. The entry point is a node in the SDG which has no predecessor. Moreover, we added a slice extraction feature to SAAF, *i.e.*, to mark all the instructions in the backward slice and write them to another file for further analysis.

Apart from extending backward slicing to cover ICC, we added other features which are important for the soundness of static analysis. The most important features we added are path-, context- and object-sensitivity [79]. Context- and object-sensitivity is essential to extracting slices across multiple components. We also utilize path-sensitivity where the conditions leading to different paths are resolvable. In cases where these conditions are not resolvable, we use an approach similar to the one used in [93].

5.4.3 Capturing Dynamic Behavior

The idea behind a multiphase iterative model is to overcome the shortcomings of both static and dynamic analysis. TeICC relies on a modified version of Stadyna to handle reflection and dynamic code loading [110]. Originally, Stadyna is based on modifications to Android framework (Android 4.2) to resolve the targets of reflection and integrate the code loaded dynamically to the original code base for further analysis. We re-implemented Stadyna removing the need of Android framework modification by using ArtDroid.

We used ArtDroid to hook framework APIs used for dynamic code updates as well as those responsible for sensitive behavior. By intercepting calls to the dynamic code APIs, App Executor provides a feedback to the Static Analyzer for improved creation of SDGs and extended backward slices. In addition, sniffing on sensitive API calls enables TeICC to put a check on suspicious app behavior.

5.5 Evaluation and Discussion

This section presents experimental results that characterize the effectiveness of TeICC to analyze apps that conceal sensitive information flow using obfuscated ICC. We evaluate TeICC on two benchmark test suites, DroidBench [20] and ICC-Bench [104], specifically crafted for testing tools to detect information flow concealed using ICC. ICC-Bench includes 9 test case apps and DroidBench contains 23 apps included in the *InterComponentCommunication* test case. The goal of evaluation of TeICC is to test its capability to extract slices across multiple components in obfuscated apps and execute them. Therefore, we obfuscated these ICC-based test suites using DexGuard [19] to evaluate TeICC.

Table 5.1 shows evaluation results for both DroidBench and ICC-Bench test suites. For brevity we group the apps in categories. The second column contains the number of ICC links found by TeICC while the third and fourth column show if the apps have been correctly analyzed by IccTA [78] and TeICC, respectively. The symbol ✘ means that the tool has failed to analyze the app and the symbol ✔ means that the app has been properly analyzed. Not surprisingly, TeICC outperforms IccTA on both tests since IccTA cannot detect ICC methods called by Java reflection and encrypted strings used in intents. As shown in Table 5.1, TeICC can automatically extract-then-execute 100% of ICC flows in all apps; except for those which perform ICC involving a *Content Provider* because currently TeICC does not provide support for Content Providers. Unfortunately, we cannot evaluate Harvester [93] because it is not open source. However, we understand that it will not be successful as well because it does not support slicing across different Android components.

Our results indicate that TeICC permits to effectively extract-then-execute the target

Apps	ICC	IccTa	TeICC
DroidBench			
startActivity[1-7]	2/9	✘	✔
startActivityForResult[1-4]	0/8	✘	✔
sendBroadCast1	0/1	✘	✔
sendStickyBroadCast1	0/1	✘	✔
startService[1-2]	0/2	✘	✔
bindService[1-4]	0/4	✘	✔
ContentProvider[1-4]	4/0	✘	✘
ICC-Bench			
Explicit1	0/1	✘	✔
Implicit[1-6]	7/0	✘	✔
DynRegister[1-2]	2/0	✘	✔

Table 5.1: DroidBench/ICC-Bench apps. ICC: # of implicit/explicit transitions between components.

slices obtained from the program slicing analysis. If, for instance, the target app contains checks which could prevent the dynamic analysis (*i.e.,* emulation detection, integrity checks, etc.), they are not extracted in the slicing step (unless they hold a data dependence with the MOI).

In contrast to Harvester [93], TeICC supports the ICC mechanism which enables it to automatically extract-and-execute target slices that belong to different Android components. Similarly, R-Droid [43] lacks support for both ICC and Java reflection mechanisms. Compared to IccTa [78], TeICC, based on a hybrid approach, permits to enrich the original app after its targeted execution to resolve obfuscated parts of the app. Over different executions it permits to extract runtime values from reflection calls or dynamically loaded code and integrate them in the analysis for the next iteration.

At the moment TeICC does not support the *Content Provider* component; we leave it as future work. Moreover, it does not analyze native code. For instance, if an SMS message is sent from native code, TeICC cannot use this hidden call to *sentTextMessage()* as MOI. However, just like TeICC, both [93] and [43] also do not support native code analysis.

5.6 Related Work

In the last few years, researchers have proposed several static analysis frameworks specifically for Android. Most of these frameworks [41, 68, 69, 78, 82] employ different type of

sensitivity, *e.g.,* field sensitivity, object sensitivity, etc.

FlowDroid [41] detects sensitive information leakage with very high recall and precision. It supports context-, flow-, field- and object-sensitivity. However, it does not support the ICC mechanism. DroidSafe [69] and IccTa [78], like FlowDroid, are developed on top of SOOT framework [77] and they are able to analyze flows between different Android components. IccTa, based on FlowDroid and IC3 [87], detects flows of sensitive data with a greater context sensitivity. DroidSafe represents the current state-of-the-art for Android static analysis. It precisely models the Android runtime and its components leveraging *Object-Sensitive Points-To* analysis.

However, static analysis approaches have problems in analyzing obfuscated apps (*i.e.,* having string encryption and using Java reflection) and to capture dynamically loaded code. These issues greatly limit the results of static analysis.

Previous works have proposed a combination of static and dynamic analysis to overcome these limitations. AppAudit [107] is a program analysis framework that can dynamically analyze apps detecting data leakage using taint analysis. The most relevant works for TeICC are Harvester [93] and R-Droid [43]. They try to improve static analysis by detecting implicit intra-component data flows using program-slicing based analysis. However, neither of them supports ICC; so they are not able to automatically analyze flows between different Android components, which leaves the analysis incomplete. Moreover, R-Droid cannot properly analyze apps in the presence of Java reflection.

To the best of our knowledge, none of the existing program-slicing based hybrid approaches [43, 93] permit the analysis of ICC flows.

5.7 Chapter Summary

In this chapter, we presented a targeted triggering approach, TeICC, to stimulate ICC in Android apps. TeICC is based on a backward program slicing which in turn relies on a SDG. The SDG based backward slice extraction technique used by TeICC enables it to extract-then-execute target slices across multiple app components. Moreover, the iterative hybrid approach allows TeICC to extract runtime values (*i.e.,* reflection values, decrypted strings, etc.) to enrich the original app. These runtime values help in performing improved static analysis of obfuscated apps in the next iteration.

As a future work in this direction, we would like to provide support for content providers. Moreover, we focus on different approaches to overcome current limitations. For example, to address the extraction of slices involving native calls, we are analyzing a novel approach using the ArtDroid [56] framework to intercept sensitive Java methods called by native code.

Chapter 6

Runtime Analysis of Dynamic Code Updates

Obfuscation and complex nature of modern day feature-rich apps make stimulating an app much harder in an analysis environment. Despite the robustness of the triggering mechanisms discussed in the previous chapter, and those found in the literature, it cannot be safely stated about dynamic analysis to cover every code path in the app. Rather, the outcome of dynamic analysis will always be an under-approximation of the complete behavior of the app. Malware developers exploit this intrinsic weakness of dynamic analysis to evade the vetting process deployed at app markets.

In addition, apps - benign but having potential vulnerabilities - that pass analysis check can become victims to on-phone exploitation by adversaries. In both these contexts, the analysis check at the app market becomes pretty much useless as the real exploitation and malicious activity is only revealed once the app is installed and run on a user's device. Nevertheless, the malicious functionality has to be exposed at some stage. Therefore, an on-phone analysis mechanism - which could analyze, detect and prevent malicious behavior as it appears - could potentially solve the problems associated with de-obfuscation, triggering and runtime exploitation of vulnerable apps. Involving the user in the triggering process, in addition to the willingness of the malware to exhibit malicious functionality once it is installed on a real user's device, provides the best possible environment to trap the malware.

In this chapter, we present an API hooking based app introspection mechanism to analyze dynamic code updates as they appear in an app. The analysis mechanism is implemented and provided as a library that can be easily included inside an app without requiring the developer to modify anything in the app. We focus on detecting and preventing on-phone exploitation of benign, but vulnerable, apps that involve dynamic code updates. This solution is directed towards safeguarding apps and mobile users from

on-phone exploitation of benign apps while relying on collaborative developers. In this work, we only consider dynamic code updates, but the idea is more general and applicable to other app activities as well.

6.1 Introduction

Moving away for static analysis of apps as it suffers from some inherent limitations such as obfuscation and dynamic code updates; dynamic analysis comes to the rescue by providing solutions to these problems. However, it requires execution of the code paths that are essential to understanding an app's behavior. Interactive nature of mobile apps, environment specific triggers embedded in apps and the use of anti-debugging and anti-emulation techniques used by malware developers are some of the features that limit the outcome of dynamic analysis to an under-statement of the app's complete behavior.

In the previous chapter, we discussed targeted execution of code paths that potentially conceal malicious behavior. We presented a combination of static and dynamic analysis techniques to ensure execution of the code paths that play a vital role to understanding the app's behavior. These techniques, along with others found in the literature [93, 94, 113], shift triggering from a black-box mechanism to a rather grey-box or white-box mechanism and advance the state-of-the-art in triggering to aid dynamic analysis of Android apps.

However, performing dynamic analysis of the already enormous number of apps, and still rapidly increasing, is highly resource consuming, does not scale well and proves costly. Consequently, it is hardly sustainable for newly established app markets. Moreover, there are always chances of malware sneaking through the analysis check and infecting users' devices. Moreover, malware can delay its malicious functionality as long as possible, but it has to exhibit it at some stage once installed on a user's device. Also, keeping in mind the possibility of on-phone exploitation of benign but vulnerable apps by adversaries; it is important to shift some part of the analysis to the end users' devices which are getting more powerful by the day. A runtime analysis, detection and prevention mechanism could potentially solve the problem related to de-obfuscation, triggering and vulnerability exploitation.

Traditional solutions in this direction can be seen in most of the antivirus products which were designed keeping the resource constraint nature of mobile devices in mind. Most of these antivirus products rely on signature based detection which can be easily evaded by new variants of malware. Mobile devices, now more or less equivalent in resources to a normal PC, allow for enhanced analysis solutions. Indeed, researchers have shown the possibility of enhancement to on-phone analysis solutions [62]. However, they either require changes to the Android framework, rooting the device or largely modifying

developer's code.

In this chapter, we present an API hooking based app introspection mechanism, AppIntrospector, that can be used to analyze, detect and prevent malicious activities that involve dynamic code updates at runtime. We design and develop our runtime analysis mechanism making use of some of the existing native level and Java level hooking techniques. At the moment, we focus on analysis of dynamic code updates only, but the concept is more generic and can be extended to monitor other type of runtime activities too. The analysis mechanism is implemented in the form of a set of libraries which can be easily included in apps without requiring the developers to modify their apps.

Contributions:

- We introduce a paradigm shift by moving part of the analysis of Android apps from an artificial analysis environment to end users' devices. Careful design, implementation and deployment of this type of solutions could pave the path to solving problems like de-obfuscation, app stimulation and vulnerability exploitation at runtime.
- We investigate and provide a theoretical overview of some of the well known hooking tools in security researcher's community and techniques found in the literature.
- We design and implement an app introspection mechanism that leverages API hooking to analyze, detect and prevent malicious activities that involve dynamic code updates. Our analysis solution relies on minimal collaboration from the developers and does not require any modification to the Android framework and rooting the device.

6.2 Threat Model

This section describes the threat model, scope of this work and capabilities of an adversary. As mentioned in the previous section, we only consider adversary exploiting vulnerabilities in benign apps to remotely execute malicious code using dynamic code updates. This adversarial activity is based on remote code injection vulnerability in the target app. We consider two scenarios here.

- Adversary controls non-secure communication (app using HTTP instead of HTTPS) performed by the app to download code packages over the network. We assume that an adversary can launch a Man-In-The-Middle (MITM) attack to replace the legitimate code with malicious code, which is later on loaded by the app and executed.
- The target app loads code from a world writable location on the device, such as SD card, and the adversary has write access to that location. An adversary can replace

the code which is to be loaded by the app with malicious code using a pre-installed app that the adversary controls.

6.3 AppIntrospector

AppIntrospector is a code package that contains a set of libraries which an app developer can embed inside the app. AppIntrospector gets activated when the app is launched and analyzes, detects and prevents malicious code execution everytime the app uses dynamic code updates.

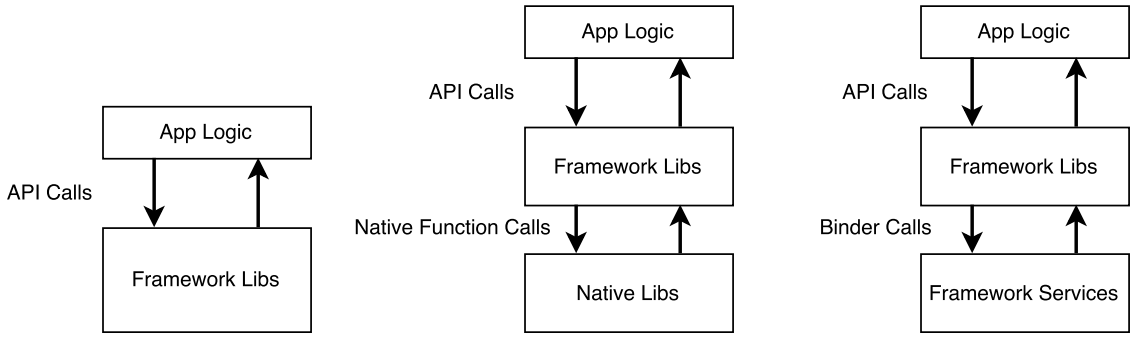
6.3.1 Design Constraints

The scenario and the problem description impose certain constraints on the design of AppIntrospector. These constraints are enlisted here and the possible solutions towards the design and implementation of AppIntrospector are discussed later in the text in the light of these constraints.

- **C1:** There should be minimal changes to the original app, if any.
- **C2:** There should be no changes to the underlying framework/kernel and should not require flashing of the device.
- **C3:** The underlying libraries used by the apps or other third party libraries should not change.
- **C4:** The solution should work on a not-rooted device.

6.3.2 AppIntrospector Overview

Android provides a set of native and Java level libraries as part of the framework. These libraries can be accessed by an app using APIs provided by the framework. Figure 6.1 draws a high level picture of the app-framework interaction to perform various activities. Features implemented as Java level framework libraries are generally accessed by calling the Java level framework APIs as shown in Figure 6.1(a). Similarly, native level libraries are usually accessed through higher level Java APIs, which in turn call the native functions. Not all, but some of these native libraries can be accessed from the app code using either native code or Java Native Interface (JNI). In addition, some of the framework features are implemented as native level services, and are not accessible directly to the app code. The app code makes use of Binder to access such services. Generally, the Android framework provides Java level stub methods which abstract Binder interaction from the developer.



(a) Java Level Libraries (b) Native Libraries (c) Framework Services

Figure 6.1: App - Framework Interaction

The key design feature of AppIntrospector is to be included in the app code as a library and use dynamic hooking to intercept API calls, such as calls to dynamic code update APIs. Figure 6.2 presents a high level view of where AppIntrospector fits in the program flow.

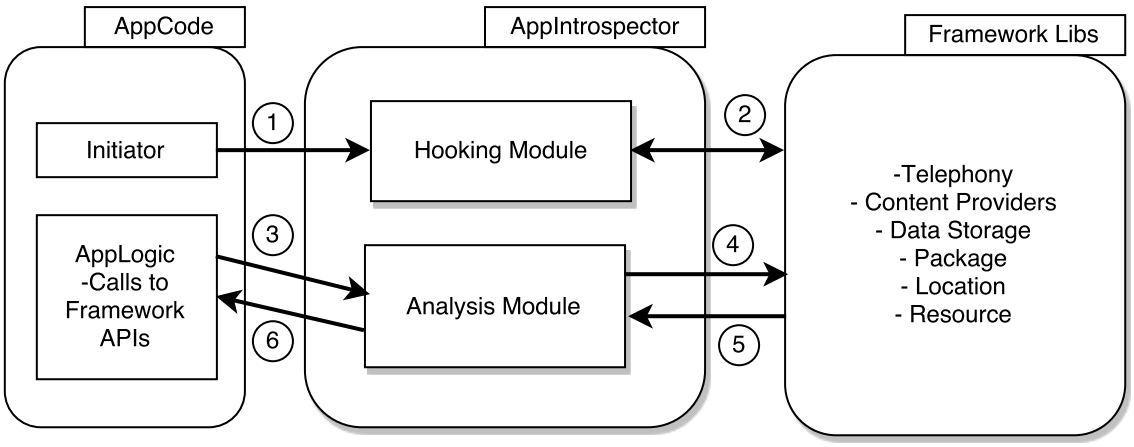


Figure 6.2: AppIntrospector

As a library, AppIntrospector is initiated when the app launches. AppIntrospector initiation can be trivially accomplished by a minor modification (Step ①) to the app's entry point. AppIntrospector then consists of two main modules, i.e., a hooking module and an analysis module. The hooking module then dynamically inserts hooks in the target framework APIs and redirects them to enhanced versions of these APIs (Step ②).

After the initial instrumentation phase, the hooking module goes into dormant mode and let the app execute according to its flow. Whenever the app makes a call to a target API, in our case any of the dynamic code update APIs, the hook placed in the target API shifts the execution flow to its enhanced version (Step ③). The enhanced version

of the API basically utilizes the analysis module of AppIntrospector to analyze the state of the API call, e.g., in the case of dynamic code loading, the analysis module checks the code to be loaded for malicious content before handing it over to the actual dynamic code loading API to load it into the memory (Step ④). AppIntrospector supports both pre- and post-call analysis. As a result, the API call returns to the analysis module before finally returning to the original app code (Step ⑤ and ⑥). Doing so, AppIntrospector analyzes the code as it is being loaded, thereby, effectively nullifying the triggering problem. AppIntrospector maintains a state of the app and raises an alarm whenever the app performs any suspicious activity.

6.4 Hooking Module

At the core of AppIntrospector, there lies the hooking module that is designed and implemented keeping in mind the constraints discussed earlier. The hooking module helps in redirecting API calls at runtime and thereby enabling the analysis module to check the code to be loaded on the fly. In this section, we provide an overview of some of the existing available hooking tools and then we discuss which among them - after necessary modifications and enhancements - serve our constraints better.

6.4.1 Review of Existing Hooking Tools

Instead of reinventing the wheel, we thoroughly analyzed the capabilities of the existing available hooking tools in order to find a suitable match and then use that as a base for our hooking module. Table 6.1 provides a summary of the tools discussed in this section.

Table 6.1: Summary of Tool Specifications. Size: Lines of code approximately.

Tool	Platform	Arch	Android	Dalvik/ART	Size
ELF-Hook	Unix/Win	x86	-	-	800
Arminject	Android	ARM32	-	-	2-3K
Adbi	Android	ARM32	4.2, 5.1, 6	Both	2-3K
SamsungAdbi	Android	ARM64	-	-	40K
ArtDroid	Android	ARM32	-	ART	3K
ArtHook	Android	ARM(32/64)	7	ART	3K
CydiaSubstrate	iOS/Android	-	Upto 4.3	Dalvik	-
Probedroid	Android	x86	5 onwards	ART	3K
Frida	All	x86,ARM(32/64),MIPS	-	Both*	250K

ELF-Hook [100]: ELF-Hook is a tool that can be used to divert function calls made

in a particular library to a modified function. It is based on patching the import tables, such as the relocation table, to divert the control flow. It reads the ELF file of the library and finds out the symbol of the function of interest in the symbol table. Based on the index of corresponding function in the symbol table, the function entry is found in relocation table and its offset is stored for later redirection. The offset for the modified function is calculated which replaces the offset of the original function.

This tool provides a simple technique to hook native functions. However, it cannot hook functions that do not have an entry in the relocation table. Therefore, a major portion of the functions cannot be hooked.

Arminject [84]: Armijenct is based on almost the same technique as ELF-Hook. However, instead of reading the library `.so` file from the disk, it reads from the process memory and modifies relocation table for redirection. Therefore, it does not require recompilation of the app. To read the memory of a process and perform modification, Arminject uses an injection module to inject a hooking library in the address space of the process. They make use of ptrace for performing injection.

It covers one aspect of our hooking module constraints, i.e., the hooking is performed at runtime. Although, it requires a rooted device to inject the library into the app address space, this constraint can be removed in our scenario as we include AppIntrospector inside the app at development time. However, Arminject suffers from the same problem as ELF-Hook.

Adbi [86]: Adbi stands for Android Dynamic Binary Instrumentaion. As the name suggests, it can be used to inject code in the memory of an Android app. Similar to Arminject, it uses a hijack module to inject a hooking library in the address space of the app. However, it uses another technique, in-line hooking, to perform redirection. It modifies the entry point of a function and makes it jump to the address of a modified function which returns the control after performing the required processing.

Similar to Arminject, Adbi also requires a rooted device for library injection in the app address space. However, this constraint can be avoided in AppIntrospector's scenario. It is a light weight tool and can be used to hook native functions on ARM32 architecture.

ArtDroid [56]: ArtDroid uses Virtual Table (vtable) tampering to divert virtual method calls used in the Android framework to patched-methods that can perform further analysis on the parameters of the original method call. The patched-methods can be coded in Java and provided in the form of a DEX file. A part of ArtDroid also makes use of Adbi (discussed above). ArtDroid uses similar concepts to Arminject and Adbi for library injection.

ArtDroid fits well within the idea of the hooking module of Introspector and can be used to hook some the dynamic code update APIs. However, the technique used in ArtDroid is

limited to hooking virtual methods only.

ArtHook [11]: ArtHook uses the the quick compile code of the framework libraries to hook Java level APIs. It accesses the code using ART representation of the methods and divert calls to these methods to patched-methods. Once the actual method is hooked and calls are diverted to a patched-method, ArtHook makes use of Java reflection to call the original methods.

ArtHook betters the Java level API hooking in comparison to ArtDroid. It also supports both ARM32 and ARM64 architectures. However, since it uses the Java reflection API `invoke` *to call the original function, it can fall into a loop if used to hook the reflection API.*

Samsung-Adbi [98]: SamsungAdbi is a more advanced version of Adbi capable of injecting code into the app's memory, perform in-line hooking for redirection and does not require recompilation of the app. It is developed and maintained by Samsung Poland R&D center. It makes use of the disassembler framework Capstone for disassembling the instructions in the apps memory [16].

Samsung-Adbi is a more sophisticated tool that can be used to hook native functions. However, keeping in mind the analysis module of AppIntrospector too, we do not consider it suitable for the hooking module which we want to be as lightweight as possible.

CydiaSubstrate [18]: It can be used to modify Android and iOS apps, but it is not open source. The authors provide their libraries in the form an SDK add-on. They provide various C, Objective-C and Java APIs for hooking native function and Java framework methods. The SDK can be used by developers and analysts to develop patched-methods that could be used for analysis purposes. CydiaSubstrate is provided in the form of an app that can be installed on a rooted device to divert API calls to the patched-methods.

It is a useful tool as it can be used to hook both native functions and Java framework methods. However, it requires a rooted device and it is not open source. Moreover, even the latest version of CydiaSubstrate only supports Dalvik runtime and can not be used with Android versions greater that 4.3. Therefore, we do not consider it in the hooking module of AppIntrospector.

Frida [23]: Frida is a multi-platform hooking tool and supports most of the hardware architectures, i.e., x86, ARM32 and ARM64, MIPS, etc. It is based on the idea of redirecting function calls to a trampoline by injecting a call to the trampoline. The trampoline can perform the desired processing (logging, modifying the parameters, etc.) and then transfer the control to the original function. This technique is also a form of inline hooking. Frida is a very sophisticated tool which basically uses a JavaScript runtime as an interface between the list-of-functions-to-hook and the underlying hooking engine written in C.

The fact that it already supports multiple platforms and architectures makes it very attractive. However, its a huge project (more than 250K LOC). Therefore, it is not trivial to modify its core engine and does not go well with the lightweightness constraint of AppIntrospector's hooking module.

ProbeDroid [99]: ProbeDroid is a dynamic Java code instrumentation tool for Android. It is based on a technique similar to Frida that is to divert a function call to a trampoline function which can perform analysis (log, modify, etc.) on the parameters and return value of the function. It provides a platform for developers and analysts to craft their own hooking tools. Based on a concept more or less similar to the tools discussed earlier, it relies on a library injected into the app's memory to hook Java method calls.

Tools like ProbeDroid can be modified a bit to make them inline with the idea of our hooking module. ProbeDroid is a tool still under development and therefore lacks some essential features, such as hooking native functions, etc. Moreover, it only targets Andorid version 5 and above. At the time of the development of AppIntrospector (and even now), ProbeDroid supported only x86 and ARM32 architectures.

6.4.2 Implementation

Keeping in mind the app-framework interaction discussed earlier in the text, the hooking module is implemented to support both native functions hooking and Java API hooking. AppIntrospector's hooking module utilizes the concepts, and code with some modifications and enhancements, used in three of the above mentioned hooking tools, namely Adbi, ArtHook and ArtDroid. Figure 6.3 provides a block diagram of the hooking module.

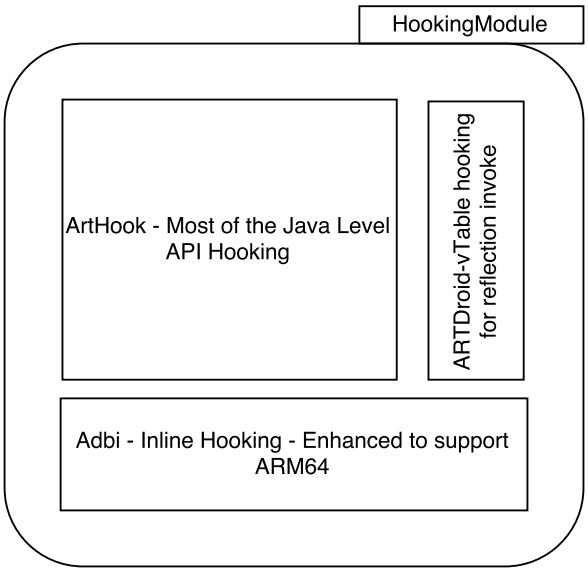


Figure 6.3: AppIntrospector - Hooking module

Native function hooking: As discussed in the previous subsection, Adbi uses inline hooking technique to insert direct jumps in target native functions at runtime and divert the flow of execution every time a target function is called. The hooking module of AppIntrospector makes use of Adbi to hook functions in the framework native libraries, such `libc.so`, `libssl.so`, etc. It is important for the hooking module to hook functions in the native libraries because in some cases apps can directly invoke these functions without calling the Java level framework APIs.

As the app launches and the hooking module of AppIntrospector initiates, Adbi looks for the base address of the library that contains the target function, e.g., the base address of `libc.so` for the `connect()` function. The base address is retrieved by traversing the `/proc/<pid>/maps` file. Once the base address of the library is retrieved, the actual address of the function in memory is computed using the ELF file of the target library. The prologue of the function is saved and replaced with a direct jump to the patched-method for redirection.

The original Adbi supports only ARM32 architecture. However, in the implementation of the hooking module, we have extended it to support ARM64 as well.

Java API hooking: Not all of the features have a native level implementation or expose native level functions to the app developers. These features are accessed using the Java level framework APIs. Also, some of the framework services implemented on the native level can only be accessed using Java level stub classes that rely on Binder. Moreover, from Android 7.0 onwards, the framework imposes restrictions and declares some native libraries as private that must not be accessed from the app code directly. These libraries include well known native libraries used for secure communication and encryption, i.e., `libssl.so` and `libcrypto.so`.

Therefore, it is important for the hooking module of AppIntrospector to hook Java level framework APIs. In this work, since we are focusing only on dynamic code updates that is accomplished mainly using Java APIs, we rely on Java level API hooking. AppIntrospector makes use of ArtHook to redirect Java level framework APIs. However, since it uses Java reflection API `invoke` to call the original method once the patched-method is called, it can fall into an indefinite loop if it is used to hook the `invoke` method itself. To overcome this limitation, AppIntrospector uses the ArtDroid virtual table tampering technique to hook the `invoke` method.

6.5 Analysis Module

The analysis module is the main part of AppIntrospector that deals with the redirected API calls at runtime. In general, it can be modeled to monitor every app activity of

dynamic nature. Malicious activities such as privacy leakage, SMS messages and calls to premium numbers, etc., can be prevented through proper deployment of the analysis module. Moreover, the concept of AppIntrospector is particularly useful in defying adversaries to exploit app vulnerabilities.

In this particular work, we focus on securing apps in the presence of app vulnerabilities that might lead to malicious code execution using dynamic code updates. An adversary exploits these vulnerabilities for execution of malicious code in the context of the victim app.

6.5.1 Dynamic Code Updates: API Selection

A first step in the implementation of the analysis module is to understand how an app performs certain activities, which APIs are used and how can they be monitored. This part of the text provides details about the APIs we selected to monitor dynamic code updates.

Dynamic code loading: Apps can load additional code into their memory space using framework provided APIs. The loaded code can be in the form `.jar`, `.apk` or `.dex` file. Apps can also load native shared libraries. This part of the analysis tries to capture the path to the code to be loaded and name of the file, etc. AppIntrospector can read and analyze the code files before loading them once it intercepts the path and name of the target file. To accomplish this task, the hooking module is presented with a list of Java level framework APIs, and calls to these APIs are redirected towards the analysis module.

Table 6.2: Code Loading APIs

Class	**Method**	**Info**
`dalvik.system.BaseDexClassLoader`	`<init>`	Dex Path
`dalvik.system.DexClassLoader`	`<init>`	Dex Path
`dalvik.system.PathClassLoader`	`<init>`	Dex Path
`dalvik.system.DexFile`	`<init>`	Dexfile Name
`dalvik.system.DexFile`	`loadClass`	Classname
`dalvik.system.DexFile`	`openDexFile`	Dexfile Name
`java.lang.Runtime`	`loadLibrary`	Native Library Path

Table 6.2 provides a list of the hooked dynamic code loading APIs. The arguments retrieved by hooking these methods generally represent paths to the `.zip`, `.jar` or `.apk` files containing a `classes.dex` file. This suites well with the idea of our previous work, discussed in Chapter 4. The analysis module can take hold of the the code to be loaded

and analyze it on the spot. Similarly, the `java.lang.Runtime.loadLibrary()` API is called when a native library is loaded by an Android app from the Java level. Hooking this API provides control over the path to the native library to be loaded. Although, our current implementation of the AppIntrospector does not analyze native libraries, it can still raise an alarm if the native code is loaded from a world writable location.

Instantiate Class and Invoke Methods: Apart from the dynamic code update APIs that load code files, there are others that can be used to create instances of classes, retrieve and invoke its methods by just providing the class names and method names. Table 6.3 provides a list of such APIs that can help an app change its behavior at runtime.

Table 6.3: Instantiate Class and Invoke Methods

Class	Method	Info
`java.lang.reflect.Constructor`	`newInstance`	Class
`java.lang.Class`	`forName`	Classname
`java.lang.ClassLoader`	`loadClass`	Classname
`java.lang.reflect.Method`	`Invoke`	Method and Params

We selected these APIs to make the work inline with our previous solution on handling dynamic code updates in an analysis environment (discussed in Chapter 4). Monitoring these APIs gives an idea about the dynamic behavior of the app which is otherwise hard to infer when statically analyzed. AppIntrospector can monitor the classes being instantiated using APIs such as `newInstance`, `loadClass` and `forName`. However, keeping track of the class objects is not sufficient for monitoring an app's behavior.

The methods called by the app, the flow of their execution and the parameters passed to the invoked methods are pivotal to app's behavior. AppIntrospector hooks and analyzes the reflection `invoke` API to detect possible malicious functionality obfuscated through reflective method calls. AppIntrospector determines the method/framework API being called using reflection and performs analysis on its sensitivity and its parameters. Similarly, it is possible to keep track of the order of methods being called during an app's execution. This helps AppIntrospector to prevent malicious activities such as privacy leakage.

6.5.2 Implementation

AppIntrospector's analysis module is basically a collection of patched-methods and analyzers. Everytime a hooked API is called, the control moves to the corresponding patched-method. Every patched-method uses one or more analyzer methods to perform the runtime monitoring.

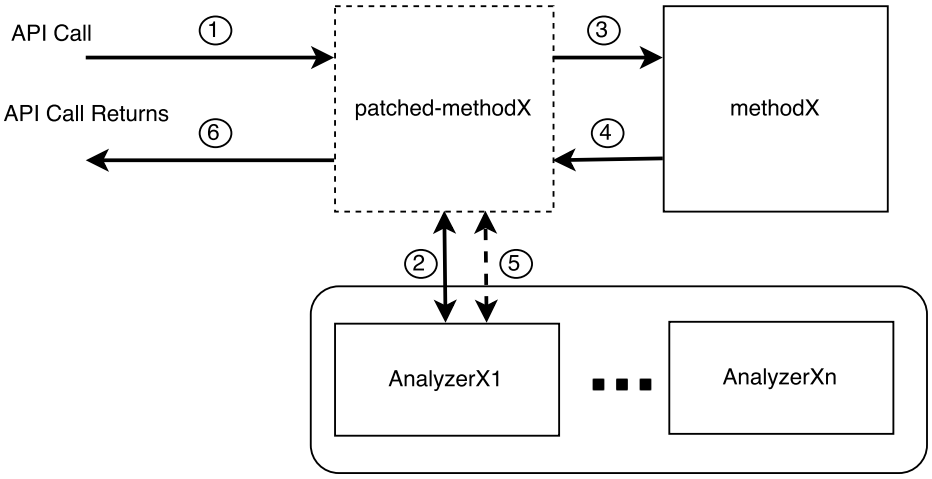


Figure 6.4: AppIntrospector - Analysis module

Figure 6.4 illustrates the flow of execution starting from the initiation of an API call by the app code. Call to the hooked methodX is intercepted by the corresponding patched-methodX. The patched-methodX then interacts with one or more analyzer methods for pre-call analysis as represented by Step ② in the figure. After the pre-call analysis, the control may or not transfer to the actual method depending upon the analysis results. Depending on the nature of the original method, the patched-methodX can call the analyzer for post-call analysis as shown by Step ⑤.

Listing 6.1 and 6.2 show example patched-methods for native functions and Java level framework APIs, respectively. The native patch method is taken for the `connect()` function of `libc.so`. Line 6 and 17 represent calls to precall and postcall analyzers, respectively, whereas the original method is called in Line 13.

Listing 6.1: Native Patched Method Example

```
1  int patched-method-connect(int sockfd, struct sockaddr* addr, socklen_t len)
2  {
3      log("-----------patched-method for connect() called-----------");
4
5    //Call pre-call analyzer
6    analyzer-precall-connect(sockfd, addr, len);
7
8    //Call original method
9      int (*orig_connect)(int sockfd, struct sockaddr* addr, socklen_t len);
10     orig_connect = (void*)eph.orig;
11
12   hook_precall(&eph);
13     int res = orig_connect(sockfd, addr, len);
14     hook_postcall(&eph);
15
16     //Call post-call analyzer
17     analyzer-postcall-connect(res);
18
```

```
19   //Return API call
20     return res;
21 }
```

Similarly, every call to the constructor of the class `dalvik.system.DexClassLoader` is intercepted by the patched method `patch-method-DexClassLoader-init()`. It calls the precall analyzer `Analyzer.precall-analyzer-DexClassLoader-init()` with the same argument as received by the patched method that include path to code to be loaded dynamically (Line 6 in Listing 6.2). The analyzer then takes care of the code file, i.e., analyzes the path as well as the code itself. Based on the verdict of the analyzer, the patched method calls the original method if nothing suspicious is found (Line 9 in Listing 6.2).

Listing 6.2: Java Patched Method Example

```
1  @Hook("dalvik.system.DexClassLoader-><init>")
2  public static void patch-method-DexClassLoader-init(DexClassLoader b, String
       dexPath, String optimizedDirectory, String
                    librarySearchPath, ClassLoader parent){
3          Log.d(TAG, "-------patched-method-DexClassLoader Called---------");
4
5          //Call pre-call analyzer
6          Analyzer.precall-analyzer-DexClassLoader-init(b, dexPath,
       optimizedDirectory, librarySearchPath, parent);
7
8          //Call original method
9          OriginalMethod.by(new $() {}).invoke(b, dexPath, optimizedDirectory,
       librarySearchPath, parent);
10     }
```

Depending upon the original method and the nature of its parameters, every analyzer perform different sort of analysis. For example, an analyzer corresponding to the `sendTextMessage()` API checks the contents of the message for potential privacy leakage. Also, it checks if the message is sent to a premium number that might belong to an adversary and cause monetary loss to the user.

The focus of this work, however, is analyzing dynamic code updates as they appear in an app. In this regard, the analysis module of AppIntrospector contains analyzers for code preliminary analysis, profiling app for possible information leakage and sensitive APIs analysis when called using Java reflection.

Part of the preliminary code analysis focuses on the paths to the DEX file, optimized directory and native library. A vulnerability in the app code loading functionality may lead to malicious code loading. Therefore, it is important to infer and report if the app loads code from world writable locations on phone. In addition, AppIntrospector follows it up with analyzing the code to be loaded. The key part of the analysis includes looking for sensitive information hard coded inside the code in plaintext. Moreover, it creates a profile of the code package based on the classes and the APIs used. The profile helps

AppIntrospector to mark the code package with a specific sensitivity level. In addition, it also helps in detecting possible privacy leakage if the order of methods called includes flow of information from source of sensitive information to corresponding leakage points.

AppIntrospector analyzes every method called using Java reflection that plays a major role in obfuscation and widely used by malware developers to evade analysis tools. Every method called using the `invoke()` API is exposed to multiple levels of analysis. In the first level of analysis, the called API is checked for the level of permission that guards the API. APIs guarded with `dangerous` permissions are subjected to further analysis specific to each API. For example, as discussed earlier in the section, the `sendTextMessage()` API is checked for its parameters to counter the SMS trojan malicious functionality. It is also checked against the profile of the app for potential privacy leakage.

6.6 Evaluation and Discussion

Although the basic objective of this work is to analyze dynamic code updates to safeguard mobile users from adversarial exploitation of possible app vulnerabilities, the concept of AppIntrospector is more generic and can be applied to monitor most sensitive activities of dynamic nature. Therefore, the current implementation of AppIntrospector is also tested for hooking functions from native libraries namely `libc.so`, `libssl.so` and `libcrypto.so`. We also successfully tested our native level hooking extension to ARM64.

Most of the dynamic code updates APIs are hooked at Java level. We tested it on multiple Android framework versions including Nougat, Marshmallow and Lollipop. The hooking and analysis functionality are tested successfully with dummy apps corresponding to each API. We deliberately injected dynamic code update vulnerabilities and malicious contents in the loaded code and AppIntrospector successfully reported it. This was the first basic level of evaluation of AppIntrospector. This ensures that the hooking and target analysis functionality of AppIntrospector works according to the plan.

However, in order to thoroughly evaluate the applicability and usability of AppIntrospector, a full scale evaluation on AppIntrospector with real world apps is required. In this regard, we plan to contact app developers who can include AppIntrospector in their apps. Some of the key points the evaluation will then focus on are: 1) malicious contents flagged by AppIntropspector, 2) performance of AppIntrospector in terms of CPU and memory usage, 3) usability, and 4) the functioning of AppIntrospector without app crashes. AppIntrospector is an ongoing work and we still await its full scale evaluation.

6.7 Related Work

Most of the traditional as well as new antivirus solutions deploy one or other type of on-phone malware detection mechanism. However, most of them rely on app package scanning either at installation time or when a user initiates it. In most cases, they utilize traditional signature based mechanisms that can be easily evaded by novel malware. Also, they require a frequent malware signature database update to detect existing malware. AppIntrospector follows a different runtime analysis based idea to circumvent exploitation of vulnerabilities inside an app.

Very similar to the goal of this work, Falsina et. al. design an overlay library to secure the `DexClassLoader` API [62]. However, our approach differs in a number of ways. We use a hooking based mechanism, rather than providing a list of overlay APIs, that does not require the developer to change any of the app's code. Moreover, we use an analysis based approach in contrast to their hash comparison based approach. Also, they only consider code loading APIs whereas our approach focuses on reflection APIs as well and can be used to target more generic problems.

Hooking or intercepting APIs/system calls has been used for various security enhancements to the existing framework in the literature. Here we discuss a few of the very relevant approahes. Traditional approaches rely on modifying the framework to intercept API calls and add analysis code inside the implementation of the target methods. Since the framework provides APIs for various functionalities, it is an intuitive solution to instrument the framework APIs with the analysis code. The analysis code is executed everytime the API is called. However, this solution requires device flashing and a rooted device.

Boxify utilizes API hooking to run an app (untrusted-app) in an isolated process inside the context of another app (monitor-app) [44]. An isolated process does not have any privileges. Any privileges required are granted to the untrusted-app by the monitor-app after certain security policy enforcement. Framework API calls by the untrusted-app are intercepted by the monitor-app using reference tampering of Binder handles in the memory of the untrusted-app. Similarly, system calls are intercepted using `libc.so` hooking using a technique similar to the one used in AppIntrospector.

A different approach, however, is used in [109] where they hijack the app startup process, change the environment variable of the app and redirect its framework API calls to a modified (security enhanced) version of the framework. The modified framework file is stored at a location readable for the app. The app startup process is hijacked by inserting code in the app that invokes an environment reset procedure. The environment reset procedure executes a modified version of the Zygote process using a native `exec()` that

replaces the current Zygote process, replaces the location pointed to by the environment variable and makes them point to the location of the modified framework files. The app is then attached to the newly started process whose calls to the framework APIs are directed towards the modified framework file. This process does not require the device to be rooted and can work with inserting minimal code at the start of the app. However, the hooking process may be different for different versions of Android depending upon how processes are started and, therefore, it may not work with newer versions of Android. Also, unlike AppIntrospector it requires a modified framework file.

Similarly, another approach for API hooking that is based on modifying the Zygote process is the Xposed framework [36]. It can be used to hook method calls without modifying the app or Android framework. Unlike the approach in AppIntrospector, i.e., injecting code into an app's virtual memory, the Xposed framework modifies the app_process which is started at the start of every process. Basically, it modifies Zygote, the center of ART that is forked to start every new process, with a library that contains native methods to hook certain methods called by the app process. Modifying Zygote, obviously, requires a rooted devices and also affects all the apps installed on the device, which is clearly not the goal of AppIntrospector and also goes against the defined constraints.

6.8 Chapter Summary

In this chapter, we presented an API hooking based runtime analysis approach to counter adversarial exploitation of dynamic code updates related vulnerabilities from within the context of the app. The idea behind this work is to engage the end user in stimulating the app and exploit the willingness of an adversary to reveal malicious behavior once the app is installed on a real user device. We presented an overview of the existing hooking techniques before providing the design and implementation details of AppIntrospector.

Chapter 7

Conclusions and Future Directions

In this dissertation, we established an argument about the widespread use of dynamic code updates in mobile apps; for extending apps' functionality in benign apps and for evading analysis and detection tools in malicious apps. We argued about the dynamic nature of these techniques preventing static analysis tools to infer the behavior of the app under analysis and we demonstrated this fact using a set of benchmark apps, reflection-bench. Moreover, we presented the case of encrypted, obfuscated and only dynamically available parameters, used in the dynamic code update APIs, being the root cause that hardens static analysis.

We proposed a hybrid approach interleaving static and dynamic analysis and demonstrated its ability to capture runtime behavior which is otherwise hidden to static analysis tools. The evaluation results on real world apps motivate towards more hybrid approaches. However, introducing dynamic analysis to the process requires efficient and effective app stimulating mechanisms. To this end, we proposed a backward slicing based mechanism for targeted execution of inter component code paths in Android apps. Moreover, to eradicate the problem associated with app stimulating, although in restricted domain, we propose an API hooking based app introspection mechanism for runtime analysis that shifts part of the analysis from an artificial analysis environment to real users' devices. Engaging real users into the app stimulating process added with the willingness of malicious apps to reveal their functionality once installed on a real user device effectively close the triggering issue and nullify most of the anti-analysis techniques used by adversaries.

In the rest of the chapter, we discuss conclusions drawn from each individual part of the work and their corresponding future directions.

7.1 App Analysis in the Presence of Dynamic Code Updates

In Chapter 3, we established an argument that its not just the dynamic code updates that lead to static analysis producing incomplete results, but a major part is played by the analysis time unavailability of the parameters used in these dynamic code update APIs, i.e., encrypted and decrypted only at runtime, read from files and received through the network, etc. Our analysis results portrayed that there are certain definite patterns used by malicious apps and profiling such patterns can help in telling apart malicious from benign apps.

To counter the problem, we proposed StaDART, a hybrid analysis approach with interleaving static and dynamic analysis, in Chapter 4. StaDART aids static analysis of Android apps by resolving dynamic code updates dynamically. We demonstrated the effectiveness of StaDART using a set of real world apps. The MCGs created using StaDART reveal much more information than their counterparts created using static analysis tools. Consequently, StaDART can be used to unfold malicious behavior of dynamic nature.

A possible future direction for this part of the work is to enrich the static analysis part of StaDART. Currently, it only supports construction of MCGs and performing basic analysis on it, such as profiling based on the protection level of the APIs used in the revealed part of the MCG. Adding other static analysis techniques to this part, such as data flow analysis, would help in revealing other kinds of malicious behavior, e.g., privacy leakage, etc.

7.2 Targeted Code Paths Execution in Android Apps

Solving issues that arise from code obfuscation and dynamic code updates, we argued about the effectiveness of a hybrid approach. However, with the introduction of dynamic analysis, there comes another challenging problem, i.e., stimulating the app to reveal concealed behavior which is a non-trivial problem for automated analysis tools. To this end, we presented the case of targeted code paths execution in Android apps, in Chapter 5. From a security analyst point of view, it is often not necessary to execute and explore all of the app's functionality. Rather, targeted execution of selected suspicious code paths can prove to be a more effective, efficient, scalable and economical solution.

We presented the design and implementation details of our backward slicing based mechanism, TeICC, for targeted execution of inter component code paths in Android apps. Moreover, we further demonstrated the effectiveness of TeICC using a set of representative obfuscated test apps. The idea behind TeICC's design is more general than just dealing with dynamic code updates and can be used to stimulate code paths leading to any target API in the app. Malware analysts can use it to trigger various suspicious APIs in the app

under analysis and understand its behavior.

Where we tested TeICC's functionality with a small set of representative test apps, a large scale analysis on real world apps would be more fruitful to determine the effectiveness and efficiency of TeICC. We plan to use it to analyze a larger dataset comprising real world benign and malicious apps. Utilizing TeICC for the analysis of apps from the official Android market to detect possible hidden malicious content is one future direction. The other is to make use of TeICC to check apps for possible vulnerabilities after identifying potential vulnerable points in apps and then stimulating the code paths leading to them.

7.3 Runtime Analysis of Dynamic Code Updates

There is an on going and never ending race between malware developers to conceal malicious functionality when analyzed in an analysis environment and security researchers to design novel approaches to uncover and detect malicious behavior. Despite some very robust security analysis approaches in practices these days, we can never rule out the possibility of malware penetrating the market and infecting user devices. Keeping in mind the security threats and the now increasing capabilities of mobile devices, it is high time to go for on-phone analysis solutions. To this end, we introduced our proposed API hooking based app introspection mechanism, AppIntrospector, that analyzes dynamic code updates on the fly. In this work, we focused on detecting and preventing exploitation of vulnerable benign apps where the vulnerability involves some dynamic code update features. We presented the design and implementation details of our runtime analysis approach in Chapter 6 and successfully tested its functionality with specially crafted apps.

Since AppIntrospector runs on users' devices, special care needs to be taken as not to overburden the device and exhaust its resources. Therefore, a large scale evaluation of AppIntrospector on real user devices would help in shaping it better. It is an on going work and we plan to perform such an evaluation for its effectiveness, efficiency, resource consumption and its working without disturbing the actual app functionality on end users' devices. Moreover, the current implementation of the analysis module of AppIntrospector focuses only on analyzing dynamic code updates. A future direction is this regard is to extend to the idea to other type of activity analysis and prevent exploitation of other forms of apps' vulnerabilities.

7.4 Closing Remarks

Most of the work discussed in this dissertation has already been published (or accepted) in international conferences or in submission to international journals. A list of the publications is provided in Appendix A. Part of the work presented in Chapter 2 and 4 is

published in [110]. Similarly, most of the work presented in Chapter 3 is published in [37]. Moreover, part of work discussed in Chapter 3 and 4 is in submission to an international journal. Also, the work presented in the Chapter 5 is accepted in an international conference (paper 1 in Appendix A.2) and going to be published in April this year.

Bibliography

[1] 13 more pieces of adware slip into the Google Play store. https://blog.lookout.com/blog/2015/03/18/adware-google-play/.

[2] AndroGuard: Reverse engineering, malware and goodware analysis of Android applications. Available Online. https://code.google.com/p/androguard/.

[3] Android - App Manifest - Permission http://developer.android.com/guide/topics/manifest/permission-element.html.

[4] Android Sandbox. http://www.androidsandbox.net/samples/.

[5] AndroidBest – Android market. http://androidbest.ru/.

[6] AndroidDrawer – Android market. http://www.androiddrawer.com/.

[7] AndroidLife – Android market. http://androidlife.ru/.

[8] Andrototal - free service to scan suspicious apks against multiple mobile antivirus. http://andrototal.org/.

[9] Anruan – Android market. http://www.anruan.com/.

[10] AppsApk – Android market. http://www.appsapk.com/.

[11] ArtHook. https://github.com/mar-v-in/ArtHook.

[12] Avast - The best free antivirus just got better. Available Online. https://www.avast.com/index.

[13] Avc undroid. http://undroid.av-comparatives.info/.

[14] AVG - Free Antivirus for everyone. Available Online. http://www.avg.com/ww-en/homepage.

[15] Bitdefender. Available Online. https://www.bitdefender.com/.

[16] Capstone - The Ultimate Disassembler. http://www.capstone-engine.org/.

[17] Contagio Mobile Malware Mini Dump. http://www.http://contagiominidump.blogspot.it/.

[18] Cydia Substrate: The powerful code modification platform behind Cydia. http://www.cydiasubstrate.com/.

[19] Dexguard - security software for Android apps. https://www.guardsquare.com/dexguard.

[20] Droidbench suite. https://blogs.uni-paderborn.de/sse/tools/droidbench/.

[21] F-Droid – Android market. https://f-droid.org/.

[22] FireEye Malware spreading in Europe. https://www.fireeye.com/blog/threat-research/2016/06/latest-android-overlay-malware-spreading-in-europe.html.

[23] Frida: Inject JavaScript to explore native apps on Windows, macOS, Linux, iOS, Android, and QNX. https://github.com/frida/frida.

[24] GDATA - Simply Secure. Available Online. https://www.gdata-software.com/.

[25] Google Play – Android official market. https://play.google.com/store/apps.

[26] Kaspersky - The power to protect what matters most. Available Online. https://www.kaspersky.com/.

[27] Lookout discovers new trojanized adware; 20K popular apps caught in the crossfire. https://blog.lookout.com/blog/2015/11/04/trojanized-adware/.

[28] Mobisec lab. http://www.mobiseclab.org/.

[29] Obfuscation in Android malware, and how to fight back. https://www.virusbulletin.com/virusbulletin/2014/07/obfuscation-android-malware-and-how-fight-back.

[30] Previously ds-andrototal - now droydseuss. http://droydseuss.necst.it/.

[31] RUMMS: The last family of Android malware attacking users in Russia via SMS phishing. https://www.fireeye.com/blog/threat-research/2016/04/rumms-android-malware.html.

[32] Sanddroid - android app analysis tool. http://sanddroid.xjtu.edu.cn/.

[33] The house always wins: Takedown of a banking trojan in Google Play. `https://blog.lookout.com/blog/2016/05/16/acecard-banking-trojan/`.

[34] UI/Application Exerciser Monkey. Available Online. `http://developer.android.com/tools/help/monkey.html`.

[35] Virustotal - free online virus, malware and url scanner. `https://www.virustotal.com`.

[36] Xposed Framework for API Hooking. `http://repo.xposed.info/`.

[37] Maqsood Ahmad, Bruno Crispo, and Teklay Gebremichael. Empirical analysis on the use of dynamic code updates in android and its security implications. In *Nordic Conference on Secure IT Systems*, pages 119–134. Springer, 2016.

[38] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

[39] AOSP. Welcome to the android open source project. `https://source.android.com/`.

[40] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[41] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 2014.

[42] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 217–228, 2012.

[43] Michael Backes, Sven Bugiel, Erik Derr, Sebastian Gerling, and Christian Hammer. R-Droid: Leveraging Android App Analysis with Static Slice Optimization. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 129–140. ACM, 2016.

[44] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock android. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 691–706, 2015.

[45] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Automatically Securing Permission-based Software by Reducing the Attack Surface: An Application to Android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 274–277, 2012.

[46] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. Brahmastra: Driving Apps to Test the Security of Third-Party Components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1021–1036, August 2014.

[47] Daniel G. Bobrow, Richard P. Gabriel, and Jon L. White. Object-oriented programming. chapter CLOS in Context: The Shape of the Design Space, pages 29–61. MIT Press, 1993.

[48] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 241–250, 2011.

[49] Jeff Bogda and Ambuj Singh. Can a Shape Analysis Work at Run-time? In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, pages 2–2, 2001.

[50] John Callaham. Google says there are now 1.4 billion active Android devices worldwide. `http://www.androidcentral.com/google-says-there-are-now-14-billion-active-android-devices-worldwide`, 2015.

[51] Gerardo Canfora, Francesco Mercaldo, Giovanni Moriano, and Corrado Aaron Visaggio. Composition-malware: building android malware at run time. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, pages 318–326. IEEE, 2015.

[52] Aske Simon Christensen, Anders Møller, and Michael I Schwartzbach. *Precise analysis of string expressions*. Springer, 2003.

[53] Fred Chung. Custom Class Loading in Dalvik. Available Online. `http://android-developers.blogspot.it/2011/07/custom-class-loading-in-dalvik.html`.

[54] "Spencer Trask Co.". Number Of Active Mobile Devices Surpasses World Population. `http://www.prnewswire.com/news-releases/`

number-of-active-mobile-devices-surpasses-world-population-278059961.html.

[55] Mauro Conti, Bruno Crispo, Earlence Fernandes, and Yury Zhauniarovich. CRêPE: A System for Enforcing Fine-Grained Context-Related Policies on Android. *IEEE Transactions on Information Forensics and Security*, 7(5):1426–1438, 2012.

[56] Valerio Costamagna and Cong Zheng. ARTDroid: A Virtual-Method Hooking Framework on Android ART Runtime. *Innovations in Mobile Privacy and Security (IMPS)*, 2016.

[57] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 73–84, 2013.

[58] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 1–6, 2010.

[59] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Conference on Security*, pages 21–21, 2011.

[60] "F-Secure". 2017 F-Secure State of Cyber Security. https://www.f-secure.com/documents/996508/1030743/cyber-security-report-2017.

[61] F-Secure. Trojan:Android/FakeNotify Gets Updated. Available Online, Dec. 2011. http://www.f-secure.com/weblog/archives/00002291.html?tduid=f57e2769518f081721ffca586e797b2a.

[62] Luca Falsina, Yanick Fratantonio, Stefano Zanero, Christopher Kruegel, Giovanni Vigna, and Federico Maggi. Grab'n run: Secure and practical dynamic code loading for android applications. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 201–210. ACM, 2015.

[63] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 627–638, 2011.

[64] Earlence Fernandes, Bruno Crispo, and Mauro Conti. FM 99.9, Radio virus: Exploiting FM radio broadcasts for malware deployment. *Information Forensics and Security, IEEE Transactions on*, 8(6):1027–1037, 2013.

[65] Lloyd D Fosdick and Leon J Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys (CSUR)*, 8(3):305–330, 1976.

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

[67] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural Detection of Android Malware Using Embedded Call Graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, pages 45–54, 2013.

[68] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, pages 291–307, 2012.

[69] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information Flow Analysis of Android Applications in DroidSafe. In *NDSS*. Citeseer, 2015.

[70] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, pages 281–294, 2012.

[71] Ben Gruver. Smali/Baksmali Tool. *https://github.com/JesusFreke/smali/wiki*, 2015.

[72] Jin Han, Qiang Yan, Debin Gao, Jianying Zhou, and Robert Deng. Comparing mobile privacy protection through cross-platform applications. In *Proc. ISOC Networks and Distributed Systems Conf*, 2013.

[73] Martin Hirzel, Daniel von Dinklage, Amer Diwan, and Michael Hind. Fast Online Pointer Analysis. *ACM Transactions on Programming Languages and Systems*, 29(2), 2007.

[74] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing Droids: Program Slicing for Smali Code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1844–1851, 2013.

[75] Cuixiong Hu and Iulian Neamtiu. Automating GUI Testing for Android Applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, pages 77–83, 2011.

[76] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. Large-scale Malware Indexing Using Function-call Graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 611–620, 2009.

[77] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[78] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 280–291. IEEE Press, 2015.

[79] Li Li, Tegawendé François D Assise Bissyande, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static Analysis of Android Apps: A Systematic Literature Review. Technical report, SnT, 2016.

[80] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection Analysis for Java. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, pages 139–160, 2005.

[81] "Hiroshi Lockheimer". Android and Security. `https://googlemobile.blogspot.be/2012/02/android-and-security.html`, 2012.

[82] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

[83] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. A whitebox approach for automated security testing of android applications on the cloud. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 22–28. IEEE, 2012.

[84] "Simone Margaritelli". An application to dynamically inject a shared object into a running process on ARM architectures. `https://github.com/evilsocket/arminject`.

[85] Nariman Mirzaei, Sam Malek, Corina S Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[86] "Collin Mulliner". Android Dynamic Binary Instrumentation Toolkit. `https://github.com/crmulliner/adbi`.

[87] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to Android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 77–88. IEEE Press, 2015.

[88] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in Android: An essential step towards holistic security analysis. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 543–558, 2013.

[89] Trend Micro Oliva Hou. `http://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer/`, Title = A Look at Google Bouncer, 2012.

[90] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of the 21st Annual Network & Distributed System Security Symposium*, 2014.

[91] Andrey Polkovnichenko and Alon Boxiner. Braintest - a new level of sophistication in mobile malware. Technical report, Check Point Technologies Ltd., September 2015.

[92] "Statista The Statistical Portal". Number of available applications in the Google Play Store from December 2009 to December 2016. `https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/`.

[93] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In

Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS), 2016.

[94] Vaibhav Rastogi, Yan Chen, and William Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, pages 209–220, 2013.

[95] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, pages 329–334, 2013.

[96] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EUROSEC, Prague, Czech Republic*, 2013.

[97] "Gartner. Press Release". Gartner Says Five of Top 10 Worldwide Mobile Phone Vendors Increased Sales in Second Quarter of 2016. `http://www.gartner.com/newsroom/id/3415117`.

[98] Ltd." "Samsung Electronics Co. Android Dynamic Binary Instrumentation tool for tracing Android native layer. `https://github.com/Samsung/ADBI`.

[99] "ZongXian Shen". A dynamic binary instrumentation kit targeting on Android(Lollipop) 5.0 and above. `https://github.com/ZSShen/ProbeDroid`.

[100] "Anthony Shoumikhin". ELF shared library import table patching for function redirection. `https://github.com/shoumikhin/ELF-Hook`.

[101] David Sounthiraraj, Justin Sahs, Garrett Greenwood, Zhiqiang Lin, and Latifur Khan. SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2014.

[102] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. 2015.

[103] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. Jekyll on iOS: When Benign Apps Become Evil. In *Proceedings of the 22nd USENIX Conference on Security*, pages 559–572, 2013.

[104] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps.

In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014.

[105] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[106] Erik Ramsgaard Wognsen, Henrik Søndberg Karlsen, Mads Chr Olesen, and René Rydhof Hansen. Formalisation and analysis of dalvik bytecode. *Science of Computer Programming*, 2013.

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

[108] Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium*, pages 29–29, 2012.

[109] Wei You, Bin Liang, Wenchang Shi, Shuyang Zhu, Peng Wang, Sikefu Xie, and Xiangyu Zhang. Reference hijacking: patching, protecting and analyzing on unmodified and non-rooted android devices. In *Proceedings of the 38th International Conference on Software Engineering*, pages 959–970. ACM, 2016.

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

[111] Yury Zhauniarovich, Olga Gadyatskaya, and Bruno Crispo. DEMO: Enabling Trusted Stores for Android. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 1345–1348, 2013.

[112] Yury Zhauniarovich, Giovanni Russello, Mauro Conti, Bruno Crispo, and Earlence Fernandes. MOSES: Supporting and Enforcing Security Profiles on Smartphones. *IEEE Transactions on Dependable and Secure Computing*, 11(3):211–223, May 2014.

[113] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 93–104, 2012.

[114] Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie. DroidAlarm: An All-sided Static Analysis Tool for Android Privilege-escalation Malware. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, pages 353–358, 2013.

[115] Yajin Zhou and Xuxian Jiang. An Analysis of the AnserverBot Trojan. Available Online, September 2011. `http://www.csc.ncsu.edu/faculty/jiang/pubs/AnserverBot_Analysis.pdf`.

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

Appendix A

Publications

A.1 Journal Publications

- [**In Submission**] **Maqsood Ahmad**, Valerio Costamagna, Bruno Crispo, Francesco Bergadano, StaDART: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications.

A.2 Conference Publications

- **Maqsood Ahmad**, Valerio Costamagna, Bruno Crispo and Francesco Bergadano, TeICC: Targeted Execution of Inter-Component Communications in Android, In proceedings of the 32nd ACM Symposium on Applied Computing April 3-6, 2017, Marrakesh, Morocco (ACM SAC 2017)
- **Maqsood Ahmad**, Bruno Crispo and Teklay Gebremicheal, Empirical Analysis on the Use of Dynamic Code Updates in Android and its Security Implications, In proceedings of 21st Nordic Conference on Secure IT Systems (NordSec 2016)
- Yury Zhauniarovich, **Maqsood Ahmad**, Olga Gadyatskaya, Bruno Crispo, Fabio Massacci, StaDynA: addressing the problem of dynamic code updates in the security analysis of android applications. InProceedings of the 5th ACM Conference on Data and Application Security and Privacy 2015 Mar 2 (pp. 37-48). ACM
- Mojtaba Eskandari, **Maqsood Ahmad**, Anderson Santana de Oliveira and Bruno Crispo, Analyzing Remote Server Locations for Personal Data Transfers in Mobile Apps, In proceedings of 17th Privacy Enhancing Technologies Symposium (PETS 2017)