Android

Analysis

Valerio Costamagna

Department of Computer Science
University of Turin

This dissertation is submitted for the degree of
Doctor of Philosophy

April 2018

I would like to dedicate this thesis to my loving parents ...

Acknowledgements

And I would like to acknowledge ...

Abstract

Table of contents

List of figures

List of tables

Chapter 1

Introduction

In recent years mobile devices have become more pervasive and ubiquitous than ever before. Mobile devices are shipped with one of the available mobile operative systems (OS), depending by the vendor. In this work we would exclusively focus on the Android platform. Since its introduction in 2008, Android has emerged as the leading operating system used for handheld devices. Mobile device provides a wide range of services via installed applications (*app* in short) that offer a variety of capabilities (i.e., voice/video recording, gps navigation) and functionalities (i.e., contacts and sms manager, calendar) aiming to enrich users experience but also to manage and share critical and sensitive user data. Android applications are distributed via a centralized official market, the Google Play Store, applications and updates are retrieved via the official system application which comes pre-installed on all Android compatible devices. To guarantee the security of its users, Android employes an automatic hybrid analysis system, named Google Bouncer, that aims to detect potentially malicious applications before they reach the official market. Most remarkable threats addressing the Android ecosystem, both enterprise and customers, can be grouped into two sets. Those applications controlling and monitoring the attacked device (i.e., rooting apps, intercepting apps) and those that do data exfiltration to third-party servers of personal and sensitive data (i.e., contact list, received sms, call history). Both class of threats may employ cleaver techniques, as social engineering, in order to get installed on the user devices, there have been several evidences of apps that masquerade themself offering an unsospicious service (i.e., contact/sms manager, in-app game) while under the hood they do sensitive data exfiltration to malicious third party servers.

As conseguence of the constant increasing amount of mobile devices running the Android platform and its expansion over many market segments during the last years (i.e., automotive, smar-tv, wearable devices), several studies have largely focused on security and privacy research fields [25, 27]. A remarkable number of investigations have been focused on malware

analysis mechanisms able to operate on real-world devices or monitoring applications operating like an antivirus countering specific well known attacks. Android privacy issues have been addressed in order to prevent sensitive data leakage which is a major threat for Android ecosystem. Despite that a running apps has its own UID, its private files space and specific set of privileges specified within its manifest file, the permission mechanism employed by Android does not allow for definition of custom fine-grained permissions. For instance, any app that has received the appropriate permission may eventually retrieve any information stored by the system contact manager, hence restricting specific contact data access to a limited set of authorized apps represents a challenging task. Also, designing a system-wise application able to monitor and prevent other application's behavior is a cumbersome task as all applications run within a sandbox enforced both at kernelspace and at userspace. Perhaps there is no concept of *assigning root privilege to a certain application* in Android. As result, practical dynamic analysis on Android stock devices requires to overcome several challenges posed by the system design itself without significant system performance degradation and without altering the least-privilege model employed by Android platform which is a solid base for security abstraction build on top of it.

Perhaps, not only spyware applications pose a risk to user privacy and security but even app's security breanch constitutes a sensible attack surface. In fact, user applications are tasty targets because a single bug in any of app's components eventually leads to complete leak of all app's data. Android permission are defined for application so they do not offer components granularity, a remote code execution bug in any app's component allows to execute any operations that has been granted to the attacked app. In addition, Android core components become attractive from the attacker point of view when it comes to achieve total control over the target device. In fact, Android core services are offering a wide range of critical and sensitive features to the end user, thus representing a juicy target for vulnerability hunters.

Programmers usually design Android apps according to functional components offered by the Android framework as Activity, Content provider, Service, Broadcast receiver. These offer several functionalities by communicating with Android system services via the Inter-procedure communication mechanism named Binder, which constitues another layer of abstraction. In fact, Android APIs exposed to programmers makes developing apps completely agnostic to how the underlying interaction is happening. Android applications rely on critical functionalities offered by the Android framework which is also in charge of dispatching system events and managing user interaction. Android app developers are not limited to employ only Java programming language, in fact Android offers the option to develop application functionalities entirely as native code via the Java Native Interface (JNI)

mechanism. JNI offers an interface to developers for instantiating objects and invoking Java methods via native code as well as calling native functions and sharing data from Java code. The ability to employ native code within an Android app makes its analysis even more cumbersome, moreover it introduces the chance that native vulnerabilities appear as memory corruption or memory violation that otherwise would not occur thanks to the memory restrinctions imposed by the JVM.

1.1 Motivation and Problem Statement

The main question we therefore want to address is: what approaches would allow to practically analyse real-world Android apps and how these would natively support analysis on stock Android devices? These approaches should not be restricted to any particular scenario nor supporting only a restricted set of Android devices. Equally important, we need a solution which offers backward compatibility and requires minimun efforts in both developing and deploying. Finally, a question of ease of development and deployment remains: can we provide these approaches without any changes to both the Android core system and the target application's code? In other words, can we offer an efficient solution that would apply on Android stock devices, hence making our approach largely system agnostic aiming to present a novel mobile application management (in short MAM) solution for Bring Your Own Device (BYOD) enviroments?
As ortogonal research topic, we want to investigate whether Android malware sandbox analysis services were designed to be resilient to evasion attacks, the main question we therefore want to address is: which artifacts and how to design them in order to prevent analysis sandbox to be detected, thus evaded by applications?

As we started exploring the state of the art regarding Android apps analysis we quickly realized that only a hybrid system as combination of static and dynamic analysis approaches would offer the requested level of inspection in order to practically analyze the app behaviour. We thus explore the Android internals, the role of security components and how different crucial components interact controlling how the user apps communicate.

The difficulty in instrumenting Android apps in order to monitor and restrict runtime behaviour is the granularity at which we can insert monitoring code and which components we require to change. A static instrumentation-based approach may offer an acceptable solution for basic and simple applications, but is definitely not suitable for enterprise scenarios where breaking the app's signature does not represent a vialable solution. On the other hand, dynamic analysis often requires consistent modifications to the underlying system or application. which is also a competitive limit in different scenarios, including enteprises

that might not be inclined to modify critical components as well as to enable rooting on employees devices.

An enterprise actor might be keen to be able to easy deploy and enforce fine-grained privacy and security policy at runtime. In particular, when the BYOD approach is in place as well as according to the recently published European Regulation GDPR. To this end, enforcing customised security policies in stock Android devices, which are those devices without any root privilege guaranteed to the end user, is a quite challenging task.

The main difficulty in analyzing Android app is its the intense events-driven behaviour and user inputs that an app relies on to change its status. How to characterize the whole behavior that an app eventually shows during its execution, how to capture its inter-procedure communications and how to monitor its dynamic behaviour via an effective and practical approach, those are the principal challenges posed by the Android system design. In fact, stock Android does not offer any runtime mechanisms for a third-party app to monitor and trace actions of other apps. Due to secure isolation offered by Android's UID-based sandboxing mechanism, apps cannot elevate their privilege to root to monitor other apps that are running under a different UID, e.g., like AV programs are allowed to do on a desktop OS.

As result, analyzing Android apps requires an unified approach which would take in account different layers, both Java and native, offering inter-procedure program analysis along with the capability of collecting app's concrete status at runtime. The main approaches offered by software testing research field employ two principal techniques, static and dynamic analysis. Static analysis relies on the availability of all the information at analysis time, hence, it suffers from dynamic features and unavailability of information that are known only at execution time. Moreover, static analysis has the limitation on analyzing apps when using techniques like code obfuscation, Java reflection and dynamic code loading to cite a few. But, dynamic analysis can help in monitoring and tampering with android app's behavior more precisely during its execution. On the other hand, dynamic analysis suffers of code coverage issue that limit the amount of code that the analysis is able to cover at runtime.

To take advatange of XXX we have been focused on employing a novel mechanism leveraging on hybrid analysis to tackle disadvantages of static analysis and take profit of YYY capabilities of dynamic analysis. Being able to design a practical and effective framework for monitoring and analyzing Android apps allows to offer a variety of services ranging from protecting user privacy as well as offering high granularity as security contromeasure to data exfiltration attacks. By static analysis of interesting hot-points in the application (i.e.,callsite to Android APIs) we are then able to guide the dynamic analysis in charge of enforcing fine-grained security and privacy policy at runtime.

In August 2015 Google has started publishing Android Security Advisories containing vulnerabilities were reported to the vendor by internal or third party researchers, few years later the Android Security Reward program were launched as a money (via bounty program) incentive to researcher whom reported vulnerabilities addressing the Android platform. As further step toward making a secure platform running secure application, Google more recently also launched the Google Play Security Reward Program which aims to further improve app security which will benefit developers, Android users, and the entire Google Play ecosystem. According to android security bulletins published so far, XXX CVE were assigned for a total of YYY vulnerabilities with an average risk score of ZZZ. The process of vulnerability mining is often composed by different steps as code audit, reverse engineering and more recently fuzzing along with symbolic execution. In different stages vulnerability mining relies on different code analysis techniques that belongs to two categories: static and dynamic analysis. Each of those analysis comes with its advantages and limitations according to the context where they are being employed, the Android platform constitutes a peculiar execution environment that is highly event-based and tightly dependent on user interaction. Moreover, the core platform code is mainly written in C and C++ programming language, instead apps are primarily developed in Java but not limited to, in fact native code is employed by leveraging on the Java Native Interface (JNI). The process of mining vulnerabilities when it comes to the Android platform might easily turn into a cumbersome challenge which requires different approaches in order to be solved.

1.2 Research Contributions

This work proposes improvements to the modern analysis of Android applications, mixing a combination of static and dynamic analysis. Perhaps, mixing static and dynamic analysis produces more accurate results and allows to monitor the app being analyzed collecting concrete runtime values. Although Android applications analysis has been extensively studied in the last years, there are still several aspects that may be significantly improved.

1.2.1 ARTDroid

We contribute a dynamic analysis framework for Android platform. It achieves dynamic instrumentations by altering app's virtual memory in order to insert dynamic hooks. Thanks to this technique, we begin able to monitor and control app's behaviour at runtime without any modifications to both Android framework and app's code. Dynamic instrumentation allows to divert the execution of the target method to a custom user code, both Android

framework's and application's methods can be instrumented by in-memory manipulation altering the virtual-table in order to divert the intended execution flow. As introduced in Android KitKat version, the Just-in-Time (JIT) compilation of Dalvik bytecode has been replaced by the Ahead-of-Time (AOH) compilation approach as employed by the new ART runtime. This means that bytecode is compiled into native code as soon as the application is installed on the device, as conseguence most of the previous dynamic techniques for the Dalvik virtual machine became ineffective.

The advantage of being able to dynamically instrument Androd applications running on ART runtime become quite relevant in several scenarios: code auditing, code protection/isolation, malware analysis to name a few. In fact, being able to instrument application's behaviour at runtime permits to monitor and trace its execution observing for anomaly behaviour or malicious patterns, moreover dynamic hooks combined along with policy specification permits to achieve more fine-grained capabilities than the ones actually offered by the platform itself.

As contribution this work propose an in-memory technique for instrumenting Android applications which could provide a benefit for different applications.

First, from the data isolation prospective, fine-grained permissions capabilities would provide a precise and effective mechanism for controlling and managing sensitive data and operations where the end user is able to define custom policy labelling a specific instance of a particular data (i.e., restrict access to private collection of pictures, business contacts/sensitive sms) without denying access to the whole set of data (i.e., camera roll, contacts list, sms list). Thus, preventing a benign applications from leaking sensitive data to third-party entities (i.e., via third-party libraries as observed in many cases).

Second, controlled execution environments (i.e., sandbox) are employed by malware guys to confine suspicious code into an artificial environment in order to limit and contain the damage caused by the malware, eventually. Preventing the malware's execution effects (i.e., deleting/altering file, data exfiltration, exploitation of known vulns) is a key issue concerning malware analysis, along with another remarkable that concerns its reproducibility. In the case of malware analysis is not always required to employ bare-metal environments, in fact most of the time malware code is executed within a VM or an emulator that offers more chances for kernel-level instrumentation or simply runs modified version of Android framework. Altought, as has been proved by several publications XXX, it is practical to execute the code on a real-world device in order to prevent evasion by anti-emulation, artifacts detection, hardware inspection techniques able to identify an emulated or virtual executon environment, thus apps will show a benign behaviour instead of the malicious one.

Finally, another scenario that takes benefits from dynamic analysis is application's code auditing. Seeking for security defeats requires a deeper knowledge about the codebase (i.e.,

which components implement what logic, how the relevant flow is carried in/out) and how its components interact, which can not be totally automated but static and dynamic analysis could reduce the human effort by identifing those code portition containing potential vulnerable code. To this end, our framework provides a solution to intercept particular Android APIs for security purposes,in fact observing runtime values and altering app's execution allows to collect precious information about its behaviour that are relevant to identify vulnerable patterns as well as allowing to reverse engineering that particular app's logic.

To summarize, this work makes the following contributions.

- We propose ARTDroid, a framework for hooking virtual-method calls without any modifications to both the Android system and the app's code.
- We discuss how ARTDroid is made fully compatible with any real devices running the ART runtime with root privilege.
- We demostrate that the hooking technique used by ARTDroid allows to intercept virtual-methods called in both Java reflection and JNI ways.
- We discuss applications of ARTDroid on malware analysis and policy enforcement in Android apps.
- We released ARTDroid as an open-source project [1].

1.2.2 TEICC

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.

To address this challenge, we apply our dynamic technique for targeting execution of interesting code portion in order to isolating some code by means of program slicing and then collect runtime values by targeted dynamic analysis, the entire process requires only the app's bytecode. The program slicing technique allows to extract a particular portion of code which is involved in creating a specific value, or partecipate in a specific callsite, that we are interested in analyzing.

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 [50] [62] [77] .

[1] https://vaioco.github.io

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 [61] and [24], are generally limited to code execution inside a signal component of the app or do not handle the dynamic code updates well.

This work contributes to achieve targeted execution of particular code in order to incremental enhance the static analysis by means of runtime concrete values. In particular, targeted execution permits to focus on analyzing only the portion code which exposes the interesting behaviour. In fact a particular portion of the code might be more relevant in terms of analysis, further reducing the amount of code to be analyzed helps also in reducing the human effort required to accomplish the entire analysis.

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, our main contribution is the automatic targeted triggering of MOI that use ICC for passing data between components. Once, we identify the interesting slices we want to execute them to capture concrete values. Thanks to our dynamic technique we do instrument the original app's entrypoint in order to load the generated slice by means of dynamic code loading capabilities and then, to jump directly to slice's entrypoint by means of reflection. By going through repetitions of this process we can harvest runtime values to improve our analysis results.

The main contributions in this regard are enlisted here.

- 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.

1.2.3 AppBox

Beside Android explosion in the consumer market, in the past decade mobile computing has also became a way for employees of organisations of all sizes to do business computing. In many cases, expensive company-owned laptops have been replaced by cheaper phones and tablets often even owned by the employees, so called Bring Your Own Device (BYOD). Business applications are quickly being rewritten to leverage the power and the ubiquitous nature of mobile devices. Mobile computing is no longer just another way to access the corporate network: it is quickly becoming the dominant computing platform for many enterprises. In this scenario, it is important for the IT security department of the enterprise to be able to configure secure policies for its employees' devices. Mobile Device Management (MDM) and Mobile App Management (MAM) services are the *de facto* solutions for IT security administrators to enforce such enterprise policies on mobile devices.

To this end, we further investigate how to apply dynamic instrumentation for achieving fine-grained enforcing capabilities on Android stock devices. We propose AppBox, a MAM solution that would enable an enterprise to select any app from the market and to be able to perform customisation with minimum collaboration from the app developer. Particularly, the developer will not have to disclose the app source code to the enterprise nor should she be involved with code customisations for satisfying the enterprise security requirements. Being able to define fine-grained policy and enforce them at runtime permits to isolate sensitive business data when shared among different user apps and to restrict runtime behaviour according to specific constraints, which is remarkable helpful especially when the BYOD approach is in place.

Our goal is twofold, ease to enable and mantain apps ready for enteprise-wise capabilites and to offer an easy to deploy MAM solution that allows an enteprise to enrich its apps park but still enforcing their internal policy on employees Android stock devices. We propose a novel approach enabling developers to make their apps enteprise ready by only changing one single line in the app manifest file, that in combination with our dynamic instrumentation technique allows an enteprise to easy customise that enterprise-ready application by defining fine-grained app specific policy.

Using AppBox , an enterprise can customise any existing app, even highly-obfuscated ones, without using any SDK or modifications to the app code. AppBox allows an enterprise to define and enforce app-specific security policies to meet its business-specific needs. More importantly, AppBox works on any Android version without requiring root privileges to control the app behaviour.

As for any other MAM solution, the basic assumption in AppBox is that the enterprise trusts the developer to deliver a benign app and uses AppBox for customisation purposes. It

is up to the enterprise to collaborate with reputable developers to deliver apps that will not include malicious logic.

To summarise, our contributions can be listed as follows:

1. We propose AppBox as an MAM solution that enables an enterprise to customise any Android app without modifying the app code. Unlike traditional enterprise mobility management solutions, AppBox is able to enforce dynamic policies without requiring integration with SDKs or other code modifications. Thus, it can work also on heavily obfuscated apps.

2. By using dynamic memory instrumentation, AppBox monitors and enforces fine-grained security policies at both Java and native levels.

3. AppBox works on stock Android devices and does not require root privileges. This is ideal especially for enterprises that support Bring Your Own Device (BYOD) policies.

4. We have implemented AppBox and performed several tests to evaluate its performance and robustness on 1000 of the most popular real-world apps using different Android versions, including Android Oreo 8.0.

5. We released AppBox as an open source project available at the following URL[2].

1.2.4 Evaluation of practical evasion of dynamic analysis systems

Antivirus companies, search engines, mobile application marketplaces, and the security research community in general largely rely on dynamic code analysis systems that load potentially malicious content in a controlled environment for analysis purposes. We evaluate the state of the art of several on-line malware analysis services by collecting artifacts exposed by those sandbox.

We implemented a probe application that collects artifacts information and transmits it to a server. The probe tool was implemented only in Java without employing reflection or other mechanism that would flag the app as sospicious by the static analyzer. The probe applications aim to collect runtime artifacts in order to give us some insights about how those sandbox artifacts were designed, whether they are randomically generated or have fixed values that does not change over different executions.

As artifacts one would define those which are most characteristic for the particular context, we identify a remarkable number of Android artifacts that strongly characterized a real-world

[2] https://vaioco.github.io/projects/

device allowing thus to detect those execution enviroment that might be artificial as malware sandbox or runtime analysis systems. Then we investigate the results discovering that most analyzed malware sandbox expose a quite predictable execution enviroment where artifacts have fixed values, hence app being analyzed can easily identify those artifical execution enviroment and evade them by exposing a benign behaviour.

To summarize, this work makes the following contributions:

- 1) **New problem.** We propose a new Android sandbox fingerprinting technique, which is based on the careless design of usage-profiles in most current sandboxes. We observe that malware developers can collect usage-profile based fingerprints from many Android sandboxes and then leverage these fingerprints to build a generic sandbox fingerprinting scheme for the sandbox analysis evasion.

- 2) **Implementation.** We conduct a measurement on collecting usage-profile based fingerprints on popular Android sandboxes. The results show that most Android sandboxes designers have not protected these fingerprints by generating the random fingerprints every time for running a different sample. Only few sandboxes generate the random fingerprints, but these random fingerprints are different from fingerprints in user's real phones.

- 3) **Mitigations.** We propose mitigations to further guide a proper design of these sandboxes against this hazard.

1.2.5 OctoDroid: Discovering vulnerabilities

Finally, we propose a methodology aiming to enhance vulnerability mining on the Android platform by means of combining static and dynamic analysis. We adopt the technique proposed by Yamaguchi et at in XXX that presents a novel static analysis approach by means of Code Property Graph (CPG). This code representation enables analysts to characterize vulnerabilities as traversals in a code property graph, a joint representation of a program's syntax, control flow, and data flow. These traversals serve as search patterns and can be expressed as queries for the graph database system. The major intuition in analyzing code via CPG is that it allows analysts building query as traversals for the underlying graph database. The graph database stores the Code Property Graph which in turn syntethises different code representetions as a multigraph. This approach permits to model potential vulnerable patterns via the query language that in turn operates on data/control flows.

In this work we employ the CPG representation to discover vulnerabilities in Android system services. Android Open Source Project (AOSP) provides the Android source code

available, Android it's an open source software, thus we can easily download the Android source code of the specific release that we want to analyze. The reasons why we focus on analyzing Android system services follow two simple but effective motivations: (I) Android system services are fundamental components of the Android framework, they are in charge of offering a varieties of core functionalities ranging from managing Activities, calls, sms and enforcing permissions at runtime. (II) system services' business logic is implemented in C++ language, which constitutes a tasty target for exploitation in comparison with the conteurpart Java code.

We employ Octopus, the analysis platform proposed by Yamaguchi in XXX, that leverages on robust code parsing to construct the CPG out of the source code. Octopus robust parser offers remarkable benefits: parsing not compilable code as well as codebase even if not entirely available (i.e., missing includes, unavailable code portion) to cite a few, but on the other hand it does not adequately handle object oriented languages. In fact, in favor of its robustness, Octopus when parsing code in tokens does not recognize any particular semantic related to a specific language, in other words parsing C or C++ code is exactly the same operation for the Octopus' parser. Octopus provides both intra- and inter- procedure analsys, as long as it is able to link call-site and destination function, on the contrary the data-flow algorithm would not be able to succesful terminate its analysis.

To address this limitation we propose a simple but effective methodologybased on clang-tidy and clang's LibTooling library, that would require none modifications to Octopus' parser but enabling it to efficiently analyze C++ code of Android system services. The main idea is to rely on code transformation driven by the class hierarchy graph in order to produce as output a semantic-equivalent code that allows Octopus to practical enhance its parsing results. The class hierarchy graph (cite XXX) provides a complete representation of base classes and the classes which are derivaded from. We apply code transformations replacing ambiguos callsites, that would make Octopus analysis ineffective, with an explicit version according to information encoded in the class hierarchy graph.

To summarize, this work makes the following contributions:

- To our knowledge, AppBox is the first automatic tool that employs automatic modeling and discovering of vulnerability in the Android codebase via CPG representation.
- Unlike previous existing works, AppBox
- An analysis plugin based on Clang/LLVM
- We have implemented and evaluated AppBox on the latest Android Orio codebase available at the time of writing. Our results show its effectiveness in discovering new vulnerabilities.

- We release AppBox as opensource to XXX

Further work on employing CPG static information for

1.3 Structure of the Thesis

This thesis consists of seven chapters, six of which remain. The first two chapters provide the BLABLA

Chapter 2

Background

During a 30-day active user monitoring in September 2015, Google confirmed that Android has 1.4 billion active users globally. 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. Google Play store surpassed 2.6 million apps in December 2016. Android operative system has been expanding across different market segments which include automotive, smart-tv and wearable devices.

Dominating the smartphone market for the last few years, it reached 86.2% of the smartphone market share in 2016. Mobile computing has gone from a niche market of gadget-driven consumers to the fastest growing way for users to perform office work, undertake banking transactions, remain active on their social networks, and much more, on the move.

The Android OS makes use of a peculiar Inter Process Communication (IPC) mechanism which has its core in the Binder component. The Binder takes care of inter-application communications as well as of intra-application ones, in fact spawning an app's internal activity involves IPC messages with the *ActivityManager* component. This mechanism plays a core role in the whole Android platform, allowing messages passing forward and backward from privileged services and user applications it allows to define system services in charge of enforcing the permissions offered by Android platform and granted by the user to that specific application. As any modern mobile operative system also Android is highly tied to user interaction offering ways to manage running apps, putting them in foreground execution, handling user input in many different means (i.e., gyroscopic, accelerometer, video camera, microphone). Moreover, Android is highly even-based which means that for instance receiving and sms or a call produces a specific event that will be sent in broadcast to all registered applications.

Chapter 3

ARTDroid: Hooking on Android ART Runtime

The analysis of Android apps becomes more and more difficult currently. Both benign and malicious developers use various protection techniques, such as Java reflection, dynamic code loading and code obfuscation [63], to prevent their apps from reverse-engineering. Java reflection and dynamic code loading techniques can dynamically launch specific behaviors, which can be only monitored in dynamic analysis environment instead of static analysis. Besides, in obfuscated apps, static analysis can only check the API-level behaviors of apps rather than the fine-grained behaviors, such as the URL in network connections and the phone number of sending SMS behavior.

Without above limitations of static analysis, the dynamic analysis approach is usually used for deeply analyzing apps [64]. Currently, dynamic analysis uses hooking techniques for monitoring behaviors of apps. The hooking techniques can be divided into two main types: 1) hooking Android framework APIs by modifying Android system [76][35], and 2) hooking APIs used in the app process by static instrumentation [26][32]. Both of them have limitations to analyze trick samples. The first hooking technique has a drawback that it cannot be used on other vendors' devices except for Google Nexus devices or emulators. This gives a chance for malicious apps, which uses the device fingerprint and anti-emulator technique to evade the dynamic detection. Even though the second hooking technique does not have this drawback, but it becomes useless when apps apply anti-repacking techniques. Apps can check the integrity of themselves in runtime and enter the frozen mode to prevent dynamic analysis if the integrity is broken. Moreover, if malicious apps implement malicious behaviors in native codes, the second hooking technique still cannot detect them.

To build a better hooking framework for dynamic analysis, we design **ARTDroid**, an framework for hooking virtual-method calls under the latest Android runtime (ART). The

idea of hooking on ART is tampering the virtual method table (vtable) for detouring virtual-methods calls. The vtable is to support the dynamic-dispatch mechanism. And, dynamic dispatch, i.e., the runtime selection of a target procedure given a method reference and the receiver type, is a central feature of object-oriented languages to provide polymorphism. Since almost all Android sensitive APIs are virtual methods, we can collect the apps behavior by using ARTDroid to hook Android APIs methods.

To summarize, this paper makes the following contributions.

- We propose ARTDroid, a framework for hooking virtual-method calls without any modifications to both the Android system and the app's code.
- We discuss how ARTDroid is made fully compatible with any real devices running the ART runtime with root privilege.
- We demostrate that the hooking technique used by ARTDroid allows to intercept virtual-methods called in both Java reflection and JNI ways.
- We discuss applications of ARTDroid on malware analysis and policy enforcement in Android apps.
- We released ARTDroid as an open-source project [1].

The rest of this paper is organized as follows. Section 3.1 introduces the background about Android and the new Android runtime ART. The ARTDrod framework is introduced in Sec. 3.2 and its implementation is discussed in Sec. 3.3. Performance evaluation is presented in Sec. 3.4, and discussion and applications are in Sec. 3.5. Section 6.4 discuss some related works, and we conclude this paper in Sec. 4.5.

3.1 Background

Android apps are usually written in Java and compiled to Dalvik bytecode (DEX). To develop an Android app, developers typically use a set of tools via Android Software Development Kit (SDK).

With Android's Native Development Kit (NDK), developers can write native code and embed them into apps. The common way of invoking native code on Android is through Java Native Interface (JNI).

[1] https://vaioco.github.io

3.1.1 ART Runtime

ART, silently introduced in October 2013 at the Android KitKat release, applies Ahead-of-Time (AoT) compilation to convert Dalvik bytecode to native code.

At the installation time, ART compiles apps using the on-device **dex2oat** tool to keep the compatibility. The `dex2oat` is used to compile Dalvik bytecode and produce an *OAT* file, which replaces Dalvik's *odex* file.

Even Android framework JARs are compiled by the `dex2oat` tool to the `boot.oat` file. To allow pre-loading of Java classes used in runtime, an image file called `boot.art` is created by dex2oat. The image file contains pre-initialized classes and objects from the Android framework JARs. Through linking to this image, OAT files can call methods in Android framework or access pre-initialized objects. We are going to briefly analyze the ART internals, using as codebase the Android version 6.0.1_r10.

The ART runtime uses specific C++ classes to mirror Java classes and methods. Java classes are internally mirrored by using **Class**[2]. In Figure **??**, virtual-methods defined in *Class* are stored in an array of `ArtMethod*` elements, called `virtual_methods_` (line 11). The `vtable_` field (line 8) is the **virtual method table**. During the linking, the vtable from the superclass is copied, and the virtual methods from that class either override or are appended inside it. Basically, the vtable is an array of `ArtMethod*` type. Direct methods are stored in the `direct_methods_` array (line 10) and the `iftable_` array (line 5) contains pointers to the interface methods. We leave interface-methods hooking for future work. The Figure **??** shows the definition of **ArtMethod** class[3]. The main functionality of `ArtMethod` class is to represent a Java method.

By definition, a method is declared within a class, pointed by the `declaring_class_` field (line 3). The method's index value is stored in the `method_index_` field (line 5). This value is the method's index in the concrete method dispatch table stored within method's class. The `access_flags_` field (line 4) stores the method's modifiers (i.e., public, private, static, protected, etc...) and the `PtrSizedFields` struct, (line 7), contains pointers to the ArtMethod's entry points. Pointers stored within this struct are assigned by the ART compiler driver at the compilation time.

3.1.2 Virtual-methods Invocation in ART

In this paragraph we describe how ART runtime invokes virtual-methods by choosing the virtual-method `android.telephony.TelephonyManager`'s `getDeviceId` as an example.

[2] art/runtime/mirror/class.h
[3] art/runtime/art_method.h

Figure **??** shows that the getDeviceId method is invoked on TelephonyManager object's class (line 4). Figure **??** shows dumped compiled codes for arm architecture by *oatdump* tool.

Before discussions on native code, in Fig. **??**, we briefly introduce the devirtualization. To speedup runtime execution, during the on-device compilation time, virtual-methods calls are devirtualized. Devirtualization process uses method's index to point to the relative element inside the vtable within `receiver` instance's class. As result, compiled code contains static memory offset used to get the called ArtMethod's memory reference.

Now, we discuss the native code generated for the method *callGetDeviceId*. The line 4 in Figure **??** is compiled in lines 11-18 in Figure **??**. The TelephonyManager instance (an Object[4] type) is stored in the register *r2*. Then, the instance's class is retrived from address in *r2* and stored in the register *r0* (line 13). The method getDeviceId (an ArtMethod type) is directly retrived (line 16) from memory using a static offset from address stored in *r0*. Finally, the getMethodId's entrypoint is called using the ARM branch instruction *blx* (line 18). The entrypoint's address is also retrived by using a static memory offset from the ArtMethod reference (line 17).

In Java, it is allowed to invoke a method dynamically specified using `Java Reflection`.

Reflection calls managed by ART runtime use the function `InvokeMethod`[5]. This function calls `FindVirtualMethodForVirtualOrInterface` which returns a pointer to the searched method by looking in the `vtable_` array of receiver's class.

A Java method can also be invoked by native code using the `Call<type>Method` family functions, exposed by JNI. For instance, function `CallObjectMethod(JNIEnv* env, jobject obj, jmethodID mid, ...)` [6] is used to call a virtual-method which returns an Object type. When a Java method is invoked from native code using a function from `Call<type>Method` family, the ART runtime will go through the `vtable_` array to find a matched method matching.

There are two different ways to get a Java virtual-method's reference. One is through the reflection APIs exposed by `java.lang.Class`. For instance, the method `getMethod` returns a reference which represents the public method with a matched method signature. All java.lang.Class' methods, which permits to get a virtual-method reference, can use the `virtual_methods_` array to lookup the requested method. The other way is offered by the JNI function `FindMethodID`. It searches for a method matching both the requested name and signature by looking in the `virtual_methods_` array within the class reference passed as argument.

[4] art/runtime/mirror/object.h
[5] art/runtime/reflection.cc
[6] art/runtime/jni_internal.cc

3.2 Framework Design

The goal of ARTDroid is to avoid both app's and Android system code modifications. So, the design of ARTDroid is oriented towards directly modify the app's virtual-memory tampering with ART internals representation of Java classes and methods. ARTDroid consists of two components. The first component is the core engine written in C and the other one is the Java side that is a bridge for calling from user-defined Java code to ARTDroid's core. The core engine aims to: find target methods' reference in virtual memory, load user-supplied DEX files, hijack the vtable and set native hooks. Moreover, it registers the native methods callable from the Java side. ARTDroid is configured by reading a user-supplied JSON formatted configuration file containing the target methods list.

Suppose that you want to intercept calls to a virtual-method. You have to define your own Java method and override the target method by using ARTDroid API. All calls to the target method will be intercepted and then go to your Java method (we call it `patch code`). ARTDroid further supports loading *patch code* from DEX file. This allows the "patch" code to be written in Java and thus simplifies interacting with the target app and the Android framework (Context, etc. . .).

ARTDroid is based on library injection and uses Android Dynamic Binary Instrumentation toolkit[ADB] released by Samsung. The ABDI tool is used by ARTDroid to insert trace points dynamically into the process address space.

ARTDroid requires the root privilege in order to inject the hooking library in the app's virtual memory, and the hooking library can be injected either in a running app or in the Zygote[45] master process.

Now, we explain the framework design in figures. Figure **??** shows the app's memory layout without ARTDroid. The class *TelephonyManager* is loaded within the boot image (boot.art). This Class contains both the *vtable_* and *virtual_methods_* arrays where the pointer to method getDeviceId is stored. Instead, Figure **??** represent the app's memory layout while ARTDroid hooking library is enabled. First, the hooking library is loaded inside the app's virtual memory (step 1), and then ARTDroid loads the user-defined patch code by DexClassLoader's methods (step 2). After this, ARTDroid uses its internal functions to retrive target methods reference. So, it can hook these methods by both `vtable_` and `virtual_methods_` hijacking (step 3).

As discussed in 3.1.2, the `vtable_` array is used by the ART runtime to invoke a virtual-method. Instead, the `virtual_methods_` array is accessed to return a virtual-method's reference from memory. ARTDroid exploits these mechanisms to hooking virtual-methods by both `vtable_` and `virtual_methods_` hijacking means.

3.3 Implementation

To get the target method's reference, ARTDroid uses the JNI function `FindMethodID`.

ARTDroid overwrites the target method's entry within both the `vtable_` and `virtual_methods_` array by writing the address of the method's patch code. The original method's reference is not modified by ARTDroid and its address is stored inside the ARTDroid's internal data structures. This address will be used to call the original method implementation.

When ARTDroid hooks a target method, all calls to that method will be intercepted and they will go to the patch code. Then, the patch code receives the *this* object and the target method's arguments as its parameters. To call the original implementation of target method, ARTDroid exports the function `callOriginalMethod` to the Java patch code. Internally, ARTDroid's core engine calls the original method implementation using the JNI `CallNonVirtual<type>Method` family of routines. These functions can invoke a Java instance method (non-static) on a Java object, according to specified class and methodID.

The original method implementation is invoked by ARTDroid using its address internally stored before the hooking phase. To guarantee a reliable hooking, ARTDroid uses ADBI features to hook the functions of `CallNonVirtual<type>Method` family. By doing this, all calls to these functions are checked by ARTDroid to block calls to an hooked virtual-method only if these calls do not come from ARTDroid's core engine.

3.4 Evaluation

3.4.1 Performance Test

To measure the effectiveness of virtual-methods hooking, we firstly need a test set of sensitive methods. SuSi[60] provides sensitive methods in Android 4.2. To verify how many of these methods are declared as virtual, we firstly test them in Android emulator in version 4.2. We use Java reflection to call these methods at runtime. The result of our experiment shows that a remarkable number of virtual-methods could be used to threaten user privacy. The following list describes our experiment results:

- 65.1% of these methods are declared as virtual
- 6.6% are non-virtual
- 28.3% methods not found

Unfortunately, the only methods list available from SuSi is from Android version 4.2. To overcome this limitation, we analyze the sensitive methods list offered by PScout[23]. The

methods of PScout are available from version 2.2 to version 5.1.1. Our another test is on Android 5.1.1 codebase and it is carried on a Nexus 6 running Android 5.1.1. After analyzing them, we know that only 1.0% of methods are non-virtual.

- 59.2% of these methods are declared as virtual
- 1.0% are non-virtual
- 39.8% methods not found

However, some methods cannot be found via Java reflection because corresponding classes or methods are not visible to normal apps. They belong to the Android system apps.

So, we can conclude that most of sensitive methods are virtual from our test results. ARTDroid can cover all sensitive methods except 1.0% methods on Android 5.1.1.

The overhead introduced by ARTDroid depends much on the behavior of the patch code.

To measure the overhead, we developed a test app, which repeatedly calls sensitive methods or APIs. In particular, this applciation attempts to perform the following operations by calling Android APIs (both via Java reflection and JNI) : initiate several network connections, access sensitive files on the SD card (such as the user's photos), send text message to premium numbers, access the user's contact list and retrive the device's IMEI.

We used the profiling facilities offered by Android calling the *android.os.Debug*'s *startMethodTracing*/*stopMethodTracing*. Then, the produced traces can be analyzed using either *traceview* or *dmtracedump*. To measure the effective overhead due to ARTDroid, we call the methods using both Java reflection and JNI in addition to the normal invocation. We ran the test 10,000 times for each method, once with ARTdroid disabled and then with ARTDroid enabled mode. The average running time for each call to an hooked method is showed in the following Table 3.1.

The most of overhead in ARTDroid is caused by the JNI call, which is internally used to invoke the original method implementation. We registered a worst case overhead of 25% for each hooked method. Therefore, the total overhead of a call to an hooked method is around 0.25 seconds. This overhead could be decreased by adding an internal cache to store methods' reference called by ARTDroid, instead of using JNI function *FindMethodID* at each call. We leave these improvements as future work.

3.4.2 Case Study

Now, we show a case study by hooking *TelephonyManager*'s *getDeviceId* in ARTDroid.

Table 3.1 Performances

ARTDroid	Invoke type		
enabled?	Normal	Reflection	JNI
Yes	1.12 s	1.39 s	1.19 s
No	0.88 s	1.14 s	0.94 s
overhead	0.24 s	0.25 s	0.25 s

Figure **??** shows the configuration file which contains the definition of methods to hook. This file is used to define the information requested by ARTDroid, which are: method's name and signature and the class' name where the patch code is defined in. The patch code called instead of method *getDeviceId* is showed inFigure **??**.

To restore the original call-flow, ARTDroid exposes to Java patch-code the native function `callOriginalMethod`. This function receivers as first argument the string key to identify the target method in the dictionary of hooked methods, internally managed by ARTDroid. Second argument represents the *this* object and the last argument is the array of method's arguments. All future calls to method getDeviceId will be redirected to the patch code, independently if these calls are made using Java reflection mechanisms or JNI.

3.5 Discussion

We note that the main goal of our work is to propose a novel technique to hook Java virtual-methods, our approach can be used to enforce fine-grained user-defined security policies either on real-world devices or emulators as well. Previous research has shown that even benign apps often contain vulnerable components that expose the users to a variety of threaths: common examples are component hijacking vulnerabilities[48], permission leaking [38],[43] and remote code execution vulnerabilities[59].

Suppose the target app is implementing the following features:

1. dynamic code loading
2. code obfuscation (Java reflection, code encryption, etc. . .)
3. integrity checks (i.e, due to copyright issue)
4. invoke of security-sensitive Java methods via JNI
5. detection/evasion of emulated environments (i.e, due to copyright issue)

An approach based only on static analysis cannot properly extract security relevant information due to the use of 1, 2 and 4. Moreover, all existing approaches based on bytecode rewriting techniques cannot analyze that app mainly for the use of integrity checks. Note that since the use of 5, in contrast to ARTDroid, all the existing approaches based on emulated environments can not properly analyze the behavior of that app. Instead, ARTDroid is still able to analyze that app. Obviously, ARTDroid has its limitations and corner cases. The main limitations is due to the running of the hooking library inside the same process space of the target app. In a scenario where an attacker want to bypass our approach, it can directly invoke a syscall through inline assembly code to gets sensitive results bypassing ARTDroid. We note that the direct system call is not a common technique used by current daily Android malware. Nevertheless, we envise that ARTDroid can be used in conjunction with existing works like [65],[76], [74] to provide an additional layer of analysis.

Even though Java direct methods are almost not used for both malicious and security-sensitive behaviors, our future work will support both interface-methods and direct-methods hooking. A possible solution is that we can statically instrument the `dex2oat` and replace the system original one once we get root privilege. The instrumented `dex2oat` can intercept all interface-methods and direct-methods.

Since ARTDroid hooking library can be injected directly either in Zygote or when the target app is going to be spaw.

Even if the app under testing can tamper with the `vtable_` , it can not get the original method's address. In fact, after ARTDroid is enabled, the original method is no more pointed by both the `vtable_` and `virtual_methods_` arrays.

In section 3.4, we have presented an evaluation about the effectiveness of virtual-methods hooking in the Android system by analyzing results obtained from both SuSi[SuS] and PScout[23] projects. Research results indicate that there is a considerable percentage of sensitive methods which are virtual. Since, ARTDroid can hook virtual-methods and tamper with their arguments, it could be used to define security policies to verifiy apps' behaviors at runtime. For instance, ARTDroid can be used to automatically identify apps which are sending SMS to premium numbers.

Since the main downside of dynamic analysis techniques is the code-coverage issue, we envise that ARTDroid can be integrated with automatic exploration system like Smartdroid[78], proposed by Cong et al.

In the following, we show some applications of ARTDroid:

- Collect apps behavior at runtime. Analysis of Android API function calls permits the extraction of information about the behavior of apps.

- Verify security policies at runtime. When users install an app, they can enforce some policies in ARTDroid, so that the new app's sensitive behaviors, such as sending SMS, can be restricted by ARTDroid.
- Android malware analysis. Some trick malware use a lot of dynamic analysis evading techniques. But in ARTDroid enforced sandbox, our hooking technique cannot be bypassed by current evading techniques. Also, we can easily build our ARTDroid sandbox either on Android emulator or on real devices.

3.6 Related Work

Several approaches have been proposed to provide methods hooking on Android. A family of approaches is based on bytecode rewriting technique. The app can be instrumented offline by modifying the app bytecode. AppGuard[26] proposed by Baches et ak, uses this approach to automatically repackage target apps to attach user-level sandboxing and policy enforcement code. Zhou et al. proposed AppCage[79], a system to confine the runtime behavior of the thid-party Android apps. Davis et al. proposed Retroskeleton[32], an Android app rewriting framework for customizing apps, which is based on their previous work, I-ARM-Droid[33].

While these approaches are valuable and each of them has its own advantages as well as disadvantages, they have different significant down sides. This approach is not feasible against apps that verify their integrity at runtime. This kind of defense (anti-tampering) is also used in benign apps as well. To be able to replace API-level calls with a secure wrapper, bytecode rewriters need to identify desidered API call-site within the target app. As mentioned in [40],[76], apps that use either Java reflection or dynamically code loading can bypass the app rewriting technique. Moreover, apps which are using JNI to call Java methods can bypass this techniques as well.

A different approach to implement methods tracing can be achieved by using a custom Android system or by using an emulated environment (e.g., a modified QEMU emulator). Enck et al. proposed TaintDroid[35], an Android modified system to detect privacy leak. StayDynA[76] a system for analyzing security of dynamic code loading in Android, uses a custom system image which can be used only on Nexus like devices. Tam et al. presented CopperDroid[65], a framework built on top of QEMU to automatically perform dynamic analysis of Android malware. These families of approaches, which are based on emulators, can be bypassed by emulation detection techniques [57] [69]. A comparison on Android sanbox has been published by Neuren et al. in [54].

Mulliner et al. proposed PatchDroid[51], a system to distribute and apply third-party securities patches for Android. This system uses the DDI[DDI] framework. DDI allows

to replace arbitrary methods in Dalvik code with native function call using JNI. In [52], Mulliner et. al. shown an automated attack against in-app billing using the DDI capabilities to control the in-app billing purchase flow. Note that the methods used to achieve in-app billing are defined as virtual.

Frida[Fri], a dynamic code instrumentation toolkit, Xposed framework [Xpo] and Cydia substrate for Android [Cyd] share similarity with the DDI intrumentation approach. These projects were created for device modding and, in contrast with DDI, require replacing of system components suck as zygote. Currently, Xposed compatibility with ART runtime is actually in beta stage[7] and the framework installation condition is to flash the device by a custom recovery image. While these approaches are very suitable under the Dalvik VM, they are totally limited for using under the ART runtime. In fact, both DDI, Frida and Cydia substrate are not able to work under the ART runtime.

Aurasium [74] builds a reference monitor into application binaries. The Dalvik code is not patched, but new classes and native code are added to ensure that the instrumentation code is run first. Clearly, such approaches are not effective if the code is obfuscated and protected against static analysis and disassembly. Also note that the package signature of the instrumented applications are broken when they are patched statically. In comparison, our approach does not need to repack the app, our modifications are in-memory only and thus we do not break code signing.

Recent works proposed novel approaches that aim to sandbox unmodified apps in non-rooted devices running stock Android. Boxify[25] presented an approach that aims to sandbox apps by means of syscall interposition (using the ptrace mechanism) and it works by loading and executing the code of the original app within the context of another, monitoring, app. A similar work, [27] uses the same approach to sandbox arbitrary Android apps. The approach presented in both of these recent works, represent one of the most promising and interesing future work direction.

3.7 Conclusion

In this paper, we present ARTDroid, a framework for hooking virtual-methods under ART runtime. ARTDroid supports the virtual-method hooking without any modifications to both Android system and app's code. ARTDroid allows to analyze apps even if they employ anti-tampering techniques or they use either Java reflection or JNI to invoke virtual-methods. Moreover, ARTDroid can be used on any real devices with ART runtime once getting the

[7] http://forum.xda-developers.com/showthread.php?t=3034811

root privilege. The applications of ARTDroid include dynamic analysis of Android malware on real devices or security policies enforcement.

Chapter 4

Targeted Execution using Effective APK Instrumentation for Dynamic Analysis of Android Apps

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 work, 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 [22, 36] based on program slicing technique [73] 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 [46]. However, to the best of our knowledge, none of the existing approaches [61, 24] 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 [41]. We modified SAAF adding more sensitivity and support for ICC using a System Dependency Graph (SDG) (cfr. §4.2). Besides that, TeICC, employs a modified version of Stadyna [76] which integrates ARTDroid [30] 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.

4.1 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 [ace, 14, 1, vbm, rum], 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 [46], the ICC mechanism offered by Android is used by both normal and malicious apps for passing data between different Android components.

Listing 4.1 MessageReceiver

```
1  public class MessageReceiver extends BroadcastReceiver {
2  public void onReceive(Context context, Intent intent) {
3      SharedPreferences v3 = ...
4      Map v0 = this.retrieveMessages(intent);
5      Iterator v6 = v0.keySet().iterator();
6      while(v6.hasNext()) {
7          Object v2 = v6.next();
8          Object v5 = v0.get(v2);
9          Intent v4 = new Intent(context, SendService.class);
10         v4.putExtra("number", ((String)v2));
11         v4.putExtra("text", ((String)v5));
12         context.startService(v4);
13     [...]
14     }
15 } }
```

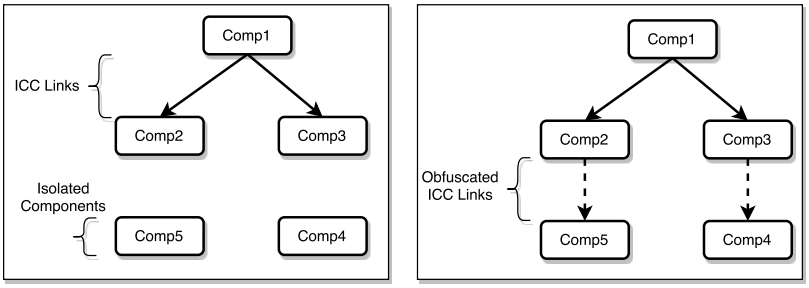
Listing 4.2 SendService

```
1  public class SendService extends IntentService {
2      protected void onHandleIntent(Intent intent) {
3          if(v1.equals("REPORT_INCOMING_MESSAGE")) {
4              Sender.request(this.httpClient, "http://37.1.204.175/?action=command",
       RequestFactory
5                      .makeIncomingMessage(v2, intent.getStringExtra("number"), intent.
       getStringExtra(
6                      "text")).toString());
7              return;
8          }
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 [fir]. Listing 4.1 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 4.2, 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 [56, 55, 46, 37], 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 [61] and [24] cannot properly analyze the sample because they lack support for ICC.

4.2 Our approach



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

Fig. 4.1 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.

4.2.1 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.

4.2.2 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 4.1 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 4.3 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 4.1(a) and 4.1(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 4.1(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 §4.1. However, TeICC allows the extraction of such data-dependent slices because it can follow the ICC flow across multiple components. Listing 4.3 shows the resulting slice extracted and refined by TeICC; it shows the corresponding aggregated Java code to ease the understanding.

4.2.3 Slice Execution

The extracted slices are put together in one or more resultant components where the irrelevant instructions are removed as shown in Listing 4.3. 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.

4.3 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.

4.3.1 Overview

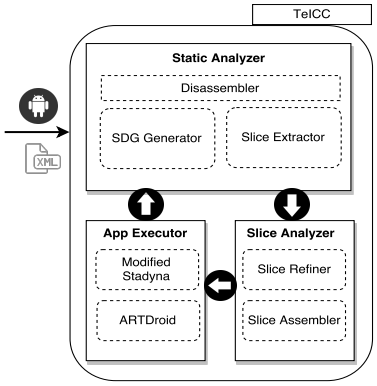


Fig. 4.2 TeICC Design

Figure 6.1 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 [39]. 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 4.3. 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 [76] 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[30], to hook sensitive APIs such as the `sendTextMessage()` API.

4.3.2 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 [41]. 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 [47]. 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 [61].

4.3.3 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[76]. 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 static 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.

4.4 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 [dro] and ICC-Bench [71], 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[dex] to evaluate TeICC.

Table 4.1 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[46] 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 4.1, 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[61] 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 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 [61], TeICC supports the ICC mechanism which enables it to automatically extract-and-execute target slices that belong to different Android components. Similarly, R-Droid [24] lacks support for both ICC and Java reflection mechanisms. Compared to IccTa[46], 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 [61] and [24] also do not support native code analysis.

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

4.5 Chapter Summary

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, 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 [30] framework to intercept sensitive Java methods called by native code.

Chapter 5

AppBox

In the past decade, mobile computing has gone from a niche market of gadget-driven consumers to the fastest growing, and often most popular, way for employees of organisations of all sizes to do business computing.

In many cases, expensive company-owned laptops have been replaced by cheaper phones and tablets often even owned by the employees (so called Bring Your Own Device). Business applications are quickly being rewritten to leverage the power and the ubiquitous nature of mobile devices. Mobile computing is no longer just another way to access the corporate network: it is quickly becoming the dominant computing platform for many enterprises.

In this scenario, it is important for the IT security department of the enterprise to be able to configure secure policies for its employees' devices. Mobile Device Management (MDM) and Mobile App Management (MAM) services are the *de facto* solutions for IT security administrators to enforce such enterprise policies on mobile devices.

MDMs enforce policies at the device level and do not cater for specific services and apps that an enterprise might want to protect. MDMs enforcement is limited by the APIs provided by the OSes. MAM solutions provide policies that are app-specific and often require the enterprise to acquire new apps. To be managed by an MAM, the code of an app needs to be customised using specific software development kits (SDKs) provided by MAM vendors.

An enterprise investing in a MAM solution has to consider not only the type of security policies that it is able to enforce but also the portfolio of apps that are already supported. To this end, MAMs provide Software Development Kits (SDKs) that enable an app developer to customise her app so that it can be managed by a specific MAM solution.

Usually, MAM providers expand their offering of supported apps over time. However, the pace at which MAMs expand the list of supported apps might not match the timing needs of an enterprise.

Alternatively, an enterprise might approach the developer of an app to ask for costly customisations to fulfill its security needs. In general, app customisations are not affordable due to the cost associated with maintenance and support. Due to the fast pace at which the app markets evolve, developers might not be too keen in engaging in such relationships. On the other hand of the spectrum, large enterprises might have the resources to implement their own customisations. However, in this case the developer should disclose to the enterprise the source code of the app.

In this paper, we propose a MAM solution that would enable an enterprise to select any app from the market and to be able to perform customisation with minimum collaboration from the app developer. Particularly, the developer will not have to disclose the app source code to the enterprise nor should she be involved with code customisations for satisfying the enterprise security requirements.

We achieve this goal by introducing **AppBox**, a novel lightweight app-level customisation solution for stock Android devices. Using *AppBox* , an enterprise can customise any existing app, even highly-obfuscated ones, without using any SDK or modifications to the app code. AppBox allows an enterprise to define and enforce app-specific security policies to meet its business-specific needs. More importantly, AppBox works on any Android version without requiring root privileges to control the app behaviour.

AppBox requires the developer to just modify two attributes in the manifest file of her app: the `android:process` and `android:sharedUserId`. AppBox provides tools for the developer to perform with minimal effort these changes in a fully automated manner.

As for any other MAM solution, the basic assumption in AppBox is that the enterprise trusts the developer to deliver a benign app and uses AppBox for customisation purposes. It is up to the enterprise to collaborate with reputable developers to deliver apps that will not include malicious logic.

To summarise, our contributions can be listed as follows:

1. We propose AppBox as an MAM solution that enables an enterprise to customise any Android app without modifying the app code. Unlike traditional enterprise mobility management solutions, AppBox is able to enforce dynamic policies without requiring integration with SDKs or other code modifications. Thus, it can work also on heavily obfuscated apps.

2. By using dynamic memory instrumentation, AppBox monitors and enforces fine-grained security policies at both Java and native levels.

3. AppBox works on stock Android devices and does not require root privileges. This is ideal especially for enterprises that support Bring Your Own Device (BYOD) policies.

4. We have implemented AppBox and performed several tests to evaluate its performance and robustness on 1000 of the most popular real-world apps using different Android versions, including Android Oreo 8.0.

5. We released AppBox as an open source project available at the following URL[1].

[1] https://vaioco.github.io/projects/

5.1 Application Scenario

In this section, we provide a motivating example to highlight the advantages of our approach. The example refers to the enterprise domain where services like MAMs and MDMs are usually deployed to manage devices and apps used by employees.

Here we stress that AppBox 's main goal is to be able to enforce security-and-privacy policies for customising the behaviour of apps without modifying their code. AppBox is not a mobile malware detector. Anti-malware solutions are complementary to AppBox . AppBox is an enterprise tool used to customise the security policy of the mobile applications employees run. Such customisations are required because the enterprise may need to monitor and possibly restrict the benign app behaviour or the use of some features of the app, due to local legislation (e.g., privacy-related regulations) and/or enterprise security policies (e.g., banning the use of WiFi in particular contexts).

The scenario involves the following parties: (i) a developer *Dev* and (ii) an enterprise *Ent*. Let us assume that *Dev* has created an app A that *Ent* wants to use. To wrap an app with a traditional MAM service, one has to have access to the source code of the app or the developer needs to apply the sandbox while developing her own app. Unfortunately, *Ent* **does not** have the source code of A, that could be heavily obfuscated, and *Dev* does not want to release the source due to IPR reasons. *Dev* is also not interested in customising A using a wrapper *sandbox* because of the extra resources needed for managing the customised version of A and the cost of supporting further updates.

AppBox . In such a scenario, our approach can be useful. We envision the developer's cooperation to offer a AppBox compatible version of A. However, *Dev* does not need to branch any new version of A or to include third-party library/code within the app. The only action needed is to run A through our *StubFactory* (more details will be provided in Section 5.3.1). This component takes A as input and returns two apps: the *StubApp* A'_{stub} and A'. The A'_{stub} will generate the sandbox where A' will be executed (details will be discussed in Section 5.3.3). Here we stress that A' is an exact copy of A except for a slightly different manifest file automatically modified by the *StubFactory* component. Moreover, it is important to note that *StubFactory* neither modifies A's bytecode nor inserts any additional code in the app. The *StubFactory* can take as input also the obfuscated version of A.

At this stage, *Dev* has to sign A'_{stub} and A' by a fresh generated self-signed certificate K. Finally, the signed new apps can be distributed so that *Ent*, which owns the right for the certificate K, is able to retrieve it.

When *Dev* releases a new version of A, the only step required is to create a new version of A'. This process is fully automated by means of *StubFactory* which produces the new

version of A'. Each time *Dev* releases a new app's update she goes through this automated procedure in order to create the AppBox compatible app, but differently from the first release there is no need to update and distribute A'_{stub} again, because the stub code does not change upon to app's update. Finally, *Dev* signs the new version of A' with the digital certificate that has been used to sign the previous version of the app.

AppBox allows *Ent* to monitor and possibly restrict the behaviour of A'. The specification of what to monitor, how and what to enforce can be done by *Ent* by writing behavioral policies for AppBox . Fine grained policy capability offered by AppBox become even more relevant in scenarios such those depicted by the recently introduced European regulation, the General Data Protection Regulation (GDPR) [gdp]. In the following, we provide two examples of such policies:

- Default enterprise security and privacy policies. This set of policies must be enforced on any app the employee installs on her phone. These policies enforce corporate-related data (i.e., contacts, calendar), copy/paste protection, corporate authentication, app-level VPN, data wipe and run-time integrity check, no HTTP connections.
- Access restrictions to selected system resources. In some locations or in some circumstances (e.g., meeting) apps may be prevented access to some system features of the phone (e.g.,, alarm, wifi connectivity, camera, mic, etc.)

We will show later in the paper how, in detail, AppBox expresses and implements such policies.

5.2 Requirements

In this section, we present the set of requirements that have driven the design of AppBox .

Our aim is to provide a lightweight app-level sandboxing solution for stock Android that does not require modifications of the apps' code. The following requirements are needed and are met by AppBox :

- **R1:** No OS modifications and no root privileges. The mechanism must not require modifications/extensions to the Android OS. Moreover, the mechanism must be able to operate without root privileges.
- **R2:** Permissions: The mechanism should not require more privileges and permissions than those originally requested by the app that will be sandboxed.
- **R3:** Universal: The mechanism can be applied to monitor and enforce policy on any app running on current and/or any previous version of Android, without requiring any app code modification or integration with SDKs.
- **R4:** Java and native: The mechanism must support the monitoring and enforcement of policies, covering the app behaviour that can be captured at both Java and at native level.
- **R5:** Lightweight: lower overhead when compared to existing solutions providing similar security features.

The first requirement, $R1$ is quite important especially when it comes to enterprise environments. Enterprises are not keen to deploy approaches that require root permissions or extensive modifications to the OS core components. In the former case, enabling root permissions might void the device's manufacturer guarantees and open the door to critical attacks (e.g., escalating privileges attacks). In the latter, modifications of the OS, especially to key components (i.e., system services, permission system) could introduce unexpected vulnerabilities in the OS code base and undermine any security guarantee.

Requirement $R2$, specifies that our solution does not alter the permissions set of the original app. This is important to avoid users rejecting existing applications on the basis of additional permissions they don't agree to grant. Furthermore, the least-privilege paradigm has a key role in Android, in fact it is a fundamental access control mechanism that the Android OS relies upon.

Requirement $R3$ is the one that makes AppBox different from existing MDM/MAM solutions. AppBox is universal in the sense that it can be used to monitor and enforce security

policies for any existing app without integration with special shared library or use of SDKs. In fact, AppBox does not require access to the app's code because it automatically wraps the app's components. All information AppBox needs to wrap the managed app are extracted from the app's Manifest file, that is always presented in clear form. Moreover, AppBox does not require any modification to the managed app's bytecode, a crucial difference with respect to the repackaging approaches that have been proposed so far.

Furthermore, the managed app's updates can be distributed by the developer to the enterprise via the usual flow, through the market used for the initial distribution (e.g., Google Play, Android for Work, Amazon market). From the user's point of view, the managed app's updates look exactly as usual app updates. In fact, no additional user interaction is requested in order to complete an app's update. Finally, the managed app can be an app developed for any version of Android, thus fully supporting backward compatibility.

In order to achieve fine-grained app-level security policies ($R4$), AppBox is able to monitor and act on the behavior of the managed app at both Java and native layers. By monitoring both levels of interaction, AppBox is able to manipulate high-level interactions made through the Android middleware (e.g., modifying the running app's context, steering android callbacks to user-defined code) as well as low-level behavior (e.g., create a socket).

Any sandboxing mechanism to be accepted for real deployment must not have significant impact in terms of performance. It must be lightweight, hence requirement $R5$.

5.3 AppBox Architecture

In this section, we provide an overview of AppBox architecture, and elaborate in more details on particular design decisions and challenges faced in its implementation. For the sake of clarity, in the rest of this discussion we will refer to the app that is being sandboxed by AppBox as the *managed app*.
AppBox creates and runs the managed app within a dedicated container that enforces enterprise policies at runtime. AppBox wraps each managed app in a sandbox that injects control hooks to intercept the app interaction with the external world. AppBox offers two levels of interception both at Java and native code representations. Being able to intercept Java methods allows AppBox to monitor and regulate all apps interactions via the Android API. However, apps might include C/C++ libraries (i.e., to boost performance). To monitor and possibly restrict the behaviour of these libraries, AppBox also offers native code hooks. Our approach requires the installation of a single *StubApp* for each managed app. The *StubApp* is an app automatically generated by the app developer using the information contained in the manifest of the managed app. The *StubApp* requests the same permissions as the managed app. The *StubApp* contains only the shim code responsible for loading the managed apps at runtime and for dynamically retrieving enterprise-defined security policies. The *StubApp* neither contains app code nor resources. For this reason, differently from repackaged apps, it can be submitted to the Google Play Store and installed on the devices as a regular app. Generating a *StubApp* is done using the *StubFactory* provided by our framework and described in Section 5.3.1. However, it is important to understand that to generate a correct *StubApp* , the manifest of the managed app has to be modified to set the values for the `android:sharedUserId` and `android:process` attributes. Finally, the developer has to sign both the *StubApp* and the managed app with the same certificate.

The use of the `android:process` and `android:sharedUserId` attributes enables the creation of our sandbox. By using these two attributes, we are able to load the code of any app in the process space of the *StubApp* . This approach has three main advantages: (i) AppBox does not require to change any part of the code in the managed app and is able to work on stock Android without the need for root privileges and modifications to the Android OS; (ii) Our solution does not rely on the emulation of Android core services which could be cumbersome in terms of deployment and updating when a new version of Android is released; (iii) AppBox does not need to re-implement several critical security checks normally performed by the Android system services which reduces the impact on performance.

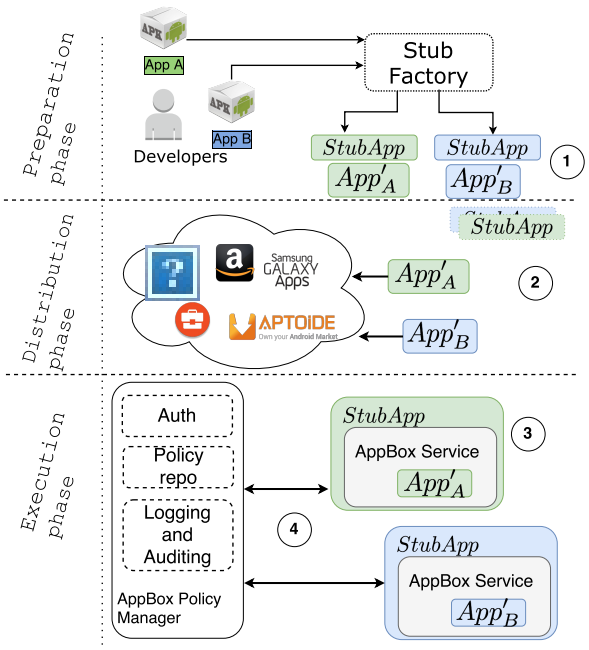


Fig. 5.1 AppBox design phases: Preparation , Distribution and Execution.

AppBox workflow consist of three main phases shown in Figure 5.1: *preparation*, *distribution* and *execution*. During the preparation, the app developer creates the managed app (App') and the *StubApp* (step 1). Then, the developer distributes the managed app via any supported android market (step 2). During the distribution phase of the managed app the requested operations are exactly those which are actually followed by developers seeking to distribute their applications, no particular additional steps are required. Finally, the user installs both apps (App' and the *StubApp*) on the target device. During the execution phase, the *StubApp* creates a sandbox to execute the managed app. The sandbox is responsible for monitoring the managed app's behaviour and enforce the policies (step 3), specified by the enterprise, offered via AppBox Policy Manager instance (step 4).

In the following, we provide a description of the components involved in each phase.

5.3.1 Preparation phase

To be able to manage an app with AppBox , the developer has to create the *StubApp* and a managed app, as shown in Figure 5.1. This steps is performed by the developer using the *StubFactory*, a set of python scripts along with a small DEX file containing the actual stub code. Another interesting aspect is that because the *StubFactory* only operates at the level of the manifest file, the managed app and *StubApp* can be created even if the original app

code is obfuscated. It is worth noting that none of the existing approaches for app behaviour customisation on stock Android devices, are able to deal with obfuscated apps.

In the following, we assume that App is an app behaviour that an enterprise wants to customise using AppBox . Using the *StubFactory*, the developer generates the managed app, indicated as App', and the stub app, indicated as $StubApp$. Finally, both the *StubApp* and the managed app App' must be digitally signed by the developer.

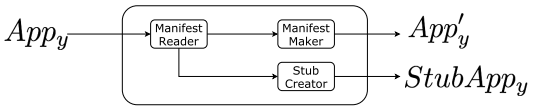


Fig. 5.2 StubFactory and its components

As shown in Figure 5.2, the *StubFactory* first extracts and decodes the Android manifest from App (obtained as APK file) using the **Manifest Reader**. This component parses the input app's manifest collecting information such as package name, main activity as well as requested permissions. Furthermore, it checks if the manifest contains components of App defined to run across multiple processes. If this is the case, then the names of these components are added to the manifest of the *StubApp* such that any additional process created by the app will be monitored by a dedicated sandbox instance. It is worth noting that the Android manifest is always in cleartext even in the case of heavily obfuscated apps.

Next, the **Manifest Maker** creates a new manifest for App' to include both the attributes `android:sharedUserId` and `android:process`. If App's manifest contains the broadcast receiver for the boot completed intent, then this will be removed from the App''s manifest. This is to prevent the situation in which App' might be launched before *StubApp* .

The last step is to create $StubApp$ through the **Stub Creator**. This last component first creates the manifest for the $StubApp$ with info previously collected by the Manifest Reader. The $StubApp$'s manifest will have the same permissions as App. By default, $StubApp$'s manifest will have the broadcast receiver component for the BOOT_COMPLETED system message. In this way, all the *StubApp* installed in a device will start as soon as the booting phase is completed. If the App's manifest also contained this broadcast receiver, then the $StubApp$ will act as a proxy and forward the boot completed intent to App'.

It is worth noting that App and App' have exactly the same bytecode. In fact, the *StubFactory* only operates on the App's manifest to output *StubApp* and App'. The developer is able to create as many *StubApp* as she may need to satisfy customers' requests. In fact, to

iterate the preparation steps the developer is asked to create a new certificate, which will be offered to each customer.

Application updates are distributed via the application market as an usual APK file, the developer needs to sign them by the same certificated used at the first place. From the developer point of view, it looks exactly the same when it comes to publish managed app updates. In fact, there is no need to manually propagate those updates to each managed app code. The only step asked to the developer is to create a new managed app by means of *StubFactory* components, as discussed before, thus updating an application is trivial as creating a managed one derived from the original application. In most cases there is no need to create a *StubApp* again, this operation is requested only if the app's manifest file has been modified by that update.

In the following, we discuss the details of the distribution phase.

5.3.2 Distribution phase

Once a developer has built an application she is interested in distributing its product to seeking consumers. In this phase the developer distributes applications as usual via any supported market.

There is no specific limit on apps distribution imposed by AppBox , in any case who owns the *StubApp* is able to control the associated managed app (App').

AppBox does not require any additional user interaction to complete an application update via the employed market and the developer does not need to accomplish any particular operation in order to distributed updates of managed apps. Whenever a new update is developed, in order to distribute it the developer uses the *StubFactory* to automatically produce an updated managed app version (same operations done by preparation phase). Thanks to our approach the developer does not need to redistribute the *StubApp* component.

5.3.3 Execution phase

After the preparation step, both $StubApp$ and App' are deployed on a device running stock Android OS.

The execution of the managed app is done by the *AppBox Service*. The AppBox Service is a process created by the *StubApp* that loads and executes the code of the managed app App'. Because the *StubApp* and the AppBox Service share the same process space, it is possible to inject hooks at runtime into managed app's virtual memory (which runs inside the AppBox Service). These are the hooks that enforce the desired policy. By sharing the same UID through the use of the `android:sharedUserId` attribute, the AppBox Service

is able to access all the private files of the managed app. In AppBox , the managed app is dynamically instrumented by means of functions interposition on both Java and native levels, this enables the enforcement of security policies related to Java APIs and native code. The AppBox Service modifies the memory of the managed app to inject hooks capable of intercepting calls to Java methods and to syscalls. The mechanism is transparent to the managed app and new versions of the target app can be easily managed without the need to change either the *StubApp* or the AppBox Service.

The biggest technical challenge at this point is to guarantee that the managed app execution will be entirely confined within our sandbox. Android offers a considerable number of features for apps to communicate with each other and share functionality. These features are accessed through callbacks such as broadcast messages, intents, and IPC. Care must be taken to avoid that these mechanisms can be exploited to let the managed app to execute outside its sandbox.

In particular, the exported components of an app, include the main activity that is always exported by default, can receive explicit intents sent by any app. When this happens, usually Android starts the exported component into a new process. If not handled properly, this could be an issue because effectively could result in a managed app starting in a process outside its sandbox. However, before starting a new process, Android searches if there is already a process where (1) the process name matches the requested component's name; (2) the process UID is the same as the one assigned to the app in which the target component has been defined. Thanks to the combination of both attributes `android:sharedUserId` and `android:process` the AppBox Service is sharing the same process name and UID, hence any intents sent to any exported component of a managed app will be captured and executed within the AppBox Service.

AppBox Policy Manager

AppBox Policy Manager is console application deployed on the enterprise infrastructure intended to be used by IT administrators. Through the AppBox Policy Manager an administration can define new policies and deploy them on the enrolled devices. Once the managed application has been started, the AppBox instance manages the authentication process with the Policy Manager.

It is worth noting that new policies can be dynamically distributed as soon as the enterprise IT department loads them into the policy repository. Furthermore, AppBox supports logging and auditing features that can be configured for each managed app, in order to monitor the status of the app while running.

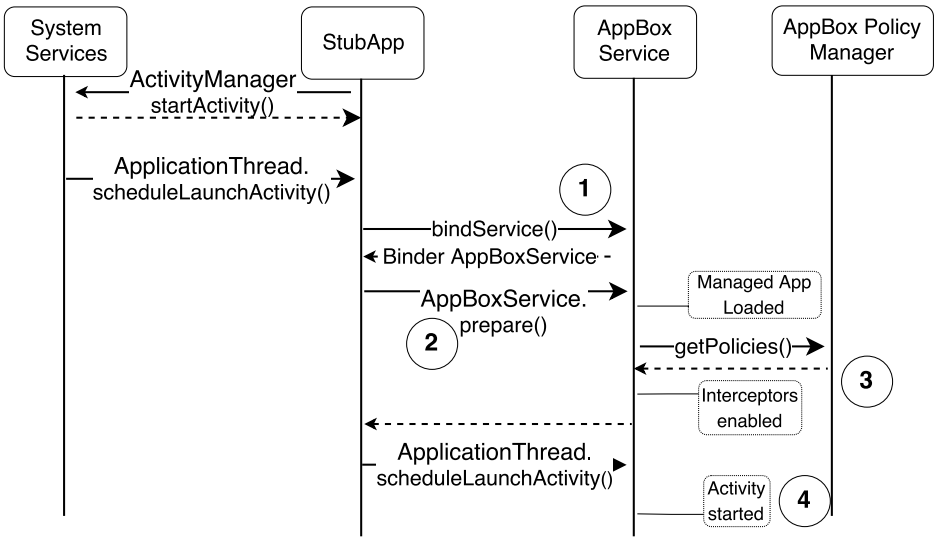


Fig. 5.3 AppBox enforcing managed apps

StubApp and AppBox Service

The *StubApp* is the key component for the realization of the AppBox Service. The *StubApp* is responsible for creating the sandbox process where the managed app will be executed as shown in Figure 5.3. When a managed app is launched, first the *StubApp* creates the AppBox Service in a separate process (step 1) and invoke the *prepare* method via the Binder to set up the interceptors (step 2). Then, the *StubApp* retrieves the set of policies and the hooking library (step 3) specific to the managed app from the *AppBox Policy Manager*. The AppBox Service loads the hooking library to instrument its virtual memory. Once the instrumentation is completed, managed app's code will be loaded by directly invoking the *bindApplication* method exposed by the *ApplicationThread* class. As a result, managed app is loaded and its main activity is executed within the AppBox Service (step 4).

As soon as the library is loaded, its virtual memory is altered to achieve functions interposition by means of different techniques, as detailed in Section 5.3.3.

Before the managed app can be executed, the *StubApp* has to create the right Android context for that app. This operation is performed by calling the Android API method *createPackageContext*[2] specifying the CONTEXT_INCLUDE_CODE flag. As an entrypoint, the *StubApp* declares in its manifest an *Application*[3] class, that is the first app component loaded by Android before any other app code. If the managed app has components that need

[2]https://developer.android.com/reference/android/content/Context.html#createPackageContext(java.lang.String,int)

[3]https://developer.android.com/reference/android/app/Application.html

to be executed in different processes then the *StubApp* will start several AppBox Services, one for each component of the app.

Java and Native Interceptors

The Java interceptors in AppBox are an extended versions of the ArtDroid hooking framework [31]. However, compared to ArtDroid, AppBox Java interceptors are able to hook static Java methods by implementing the approach proposed in [**?**]. The Java interceptors can operate at the level of Java methods defined within the managed app. We are able to intercept all calls to monitored Java methods including either calls via Java reflection, native code or dynamically loaded code. The intercepted calls are redirected to the specific Policy Enforcement Point (PEP) where the actual user-defined policy is enforced. Java interceptors achieve transparent hooking by means of memory instrumentation. It fully supports both the DVM and ART Android runtime. The interposition offered by the Java interceptors permits to monitor the access an app performs to Android APIs to interact with system services (i.e., LocationManager, TelephonyManager) and the Android environment (i.e., Context, SharedPreferences). In Android, apps can also invoke these APIs from native code via the JNI interface. In addition, in Android native code is allowed to perform direct Binder transactions without invoking any Android Java method. To address this, AppBox relies on the Native interceptor to intercept these calls that could bypass the policies defined at the Java level.

AppBox offers also native functions interposition by means of *inline hooking*, a well-known technique [**?**] that basically permits to redirect a function call to another function under the control of a monitor process. In contrast with the GOT patching techniques, AppBox can intercept calls to any native function not only the ones to global symbols (i.e., calls to functions defined in the same module will not generate entries in GOT). AppBox allows to intercept calls to functions like *open, connect, read*, access to the Binder via the *ioctl*, etc. In this way we can monitor if the managed app tries to access system services directly via native call to the *ioctl* function. Enabling native interceptors is optional. For instance, if an app does not use its own native libraries then native interceptors can be disabled. However, if the managed app dynamically loads a native library then AppBox can automatically enable the native interceptors layer.

5.4 AppBox Policy

In this section, we first present the AppBox policy language for controlling app's behavior during its execution. Afterwards, we will present how to define in AppBox policies discussed in our application scenario (see Section 5.1). Finally, we provide some details on how policies can be automatically generated.

5.4.1 Policy Language

Figure 5.4 shows the the syntax of AppBox policy. Policies are identified by a name and they define what *operation* a *Requester* application can execute on a *Resource*. In our prototype we defined two sets of *operations*: the first set contains getter methods that return data from the Resource to the Requester; the second set contains setter methods where data is being passed by the Requester to the Resource. Moreover, AppBox offers the possibility to define policies on events (i.e. boot completed, app installed, app running, etc. . .). The operations defined in the policies are then mapped to Java methods or native functions that AppBox will interpose at runtime to enforce the policies.

```
1 PolicyName: Requester can do <operation> on <Resource>
2             have to perform <action>
3             [if <condition>]
```

Fig. 5.4 AppBox Policy Language

In AppBox , a Resource identifies any sensitive data which could be retrieved via either Android middleware API (i.e., location, contact, camera) or native code (i.e., sensors, socket, microphone).

The *have to perform* clause specifies which actions have to be performed if this policy is enforced. These actions are mappped to a set of functions to control the app's behavior (i.e., filtering, anonymisation, etc.) and to change the values of the parameters of the operation being executed. An action is a callback that is registered by AppBox to dynamically forward the execution to the corresponed function and can operate on both input parameters and returned values.

A policy can have an optional clause *if* that defines a condition that must be verified before the specified action is performed. Otherwise, if the condition is not true, the action specified in the policy is not executed.

5.4.2 Fine-Grained Access Control Policies

We begin with some examples of policies for fine-grained control over apps accessing user data or using network access. Any access to a protected resource is intercepted by the hooking mechanism and diverted through a custom user-defined control code.
As discussed in Section 5.1, the enterprise *Ent* wants to enforce fine-grained app-level policies. In this scenario, *Ent* wants to protect business data (i.e., contact and calendar) against unauthorized operations according to custom corporate-level policies and protect the managed app enforcing integrity checks at runtime. Moreover, *Ent* wants that any connection made by a specific set of apps makes use of a secure channel (i.e., TLS) thus reporting any connection which makes use of insecure transport system like HTTP. The enterprise's requirements can be expressed by policies shown in Figure 5.5.

```
1  MicPolicy : AppX can do getMicrophone on Microphone
2              have to perform checkLocation();
3              if isWorkHours() == True
4
5  LocPolicy : AppX can do getLocation on Location
6              have to perform checkLocation();
7              if isWorkHours() == True
8
9  ContPolicy: AppY can do getContacts on Contacts
10             have to perform filterOut();
11             if isWorkHours() == True and isLocation() == True
12
13 HTTPPolicy: AppZ can do createConnection on Internet
14             have to perform forceHTTPS()
15             if isWorkHours() == True
16
17 CopPolicy : AppX can do runApp on Boot
18             have to perform checkApp();
```

Fig. 5.5 AppBox Policies

MicPolicy in Figure 5.5, is quite straightforward: any request for accessing to microphone capabilities made by managed apps is restricted by the policy such that AppBox intercepts the request and checks for the specified condition (if clause) , if it is validated then the access is denied. Another similar policy is *LocPolicy*. Such policy permits to avoid location information leak potentially made via apps during working hours. The artificial value returned by AppBox is totally controllable by the user, by default AppBox returns an existing location chosen at random among a user predefined set of positions.

The policy *ContPolicy*, line 9, permits to achieve a content provider isolation for an AppBox managed app. In this specific case, *Ent* wants to isolate corporate business contacts sharing them only across authorized apps that have been register throught the AppBox Enteprise Policy Manager (see Section 5.3.3). Moreover, the *Ent* wants to specify particular

criteria that must be respected to allow to the managed app to access business contacts data. In particular, the policy ContPolicy operates as following. The requested operation getContact indicates that any kind of attempt to retrieve the user contacts list must be intercepted and monitored by AppBox . Then, for each intercepted operation the specified conditions must be verified. The user-defined isWorkHours() function returns True whether the actual time is within the current working time, False otherwise. If isWorkHours() returns True then the specified action is executed. Otherwise, the execution flow will continue as if AppBox was not in place. Thanks to this policy, *Ent* is allowed to specify a customizable fine-grained access policy enforcing access to the isolated corporate contact provider exclusively to apps managed via AppBox .

The policy *HTTPPolicy* (line 13), permits to filter out any connection that is being made via HTTP protocol. Connections instantiate by the managed app can be intercepted and monitored by either hooking the appropriate Java level APIs or intercepting native layer functions if needed. The policy specifies that any operation recognized as an attempt to create a connection has to be intercepted and monitored by AppBox . The condition verifies whether the request is made during the working time. In this specific case, *Ent* wants to deny insecure connections made via HTTP protocol.

As an example of a policy defining an artifact Resource, *CopPolicy* is presented. It enables the enterprise to specify a custom integrity check to be enforced before the managed app start its execution. Given a specific app, the enterprise wants to verify that its bytecode has not been tampered with. Here we stress that AppBox does not require to modify the app's bytecode, thus its checksum value does not change. Thanks to this policy, before each execution of the managed app AppBox computes the app's bytecode checksum value to guarantee that it has not been tampered with.
In the following we present and discuss how AppBox policies can be automatically generated by the enterprise.

5.4.3 Policy Generation

In this section we present how AppBox policies can be automatically generated by extracting information from the policy specification file. The overall procedure of policy creation is presented by Figure 5.6. The policy generator takes as input the policies specification, the sets of Resource that *Ent* wants to protect (Res.) and the information about what operation is offered by what Resource (Op.). In our current prototype, we selected sensitive resources (i.e., contacts, location, internet access) and we grouped them by the relative requested permissions. Then we used the information offered by PSCout[23] to aggregate Android

APIs in terms of which permissions are needed by them. At this point, in our prototype we manually selected which API methods belong to a Getter or Setter operation. By doing this we created a mapping between Resource, Operation and the API methods which are offering capabilities to execute that specified Operation on that specified Resource. It is worth noting that *Ent* can simply extend those sets including any resource that wants to enforce, as presented for *CopPolicy*. In Table 5.1 we reported the mapping was used for the policy HTTPPolicy. For the sake of understanding we included only the Android APIs offering HTTP capabilities as they are suggested by the official Android developer guide[4]. The policy expressed by the specification shown in Figure 5.5 line 13, produces according to the requested operation on the specified Resource the interposition of the API methods listed in the third column of Table 5.1.
Finally, the policy generator produces as output the executable file encoding the corporate app-level policies that will be loaded by AppBox to enforce at runtime the policies taken as input by the generator.

Table 5.1 HTTP-Policy Generation - Intercepted APIs

Operation	API to hook (clsname, mname)
createConnection	(HttpURLConnection , $<$ *init* $>$) (URL , $<$ *init* $>$) (URL , openConnection)

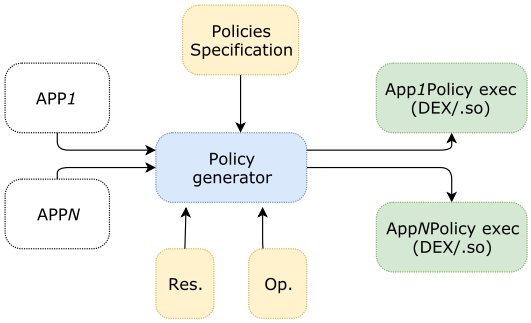


Fig. 5.6 Policy generation mechanism. Res.: sets of Resource, Op: sets of operation

[4]https://developer.android.com/training/basics/network-ops/index.htm

5.5 Evaluation

In this section, we discuss the evaluation we carried on to test performance overhead, robustness, applicability and effectiveness of AppBox . For our tests, we used a Nexus 5x (64bit) device running stock Android Oreo version 8.0.0.

5.5.1 Performance Overhead

To evaluate AppBox performance penalty on managed apps, we used benchmark apps and our custom micro-benchmarks. As benchmark apps, we used Quadrant and Vellamo[5]: the former was selected because it has been used in other related works [**?** 25, 27] so it will make easier to compare the performance of AppBox with similar approaches; the latter is a highly accurate benchmark developed by Qualcomm and contains a benchmark specifically intended for stressing the Binder communication channel. Given that the Binder is the most common means of communication for Android apps, it is important to measure the overhead AppBox introduces. Moreover, we executed the *webview* package of Vellamo that contains various benchmarks for Android's Webview API. We included this test in our experiments because a huge number of apps are either entirely developed within a Webview or specifically use its features.

As shown in Table 5.2, the impact of AppBox on the total score produced by the benchmarks is low. The test marked as *total* reports the cumulative score that the Quadrant benchmarking app produced when executed, while the I/O test were oriented to stress operations of reading/writing from/to disk. The worst score, of about 15% is low when compared to similar works, and can be attributed to the I/O test. It is worth to note that in both scores, AppBox overhead is much lower than the one introduced by solutions like Boxify and NJAS in equivalent tests. In fact employing a light-weight in-memory hooking mechanism, instead of ptrace, and avoiding critical services emulation via Broker component, AppBox reduces remarkably the performance costs requested for monitoring the target app.

As for the performance penalty introduced in the Binder communication (indicated as multi-core in Table 5.2), the score indicated by the Vellamo benchmark is really low (up to 1%) due to the fact that AppBox does not need to perform extra marshalling operations. It would have been interesting to report the same tests for NJAS and Boxify[6] but both systems were not available to us at the time of writing.

[5]https://play.google.com/store/apps/details?id=com.quicinc.vellamo

[6]At the time of writing this paper, a version of Boxify was released but it's not based on the isolated process mechanism as described in [25]

To understand the performance implication of AppBox on method invocations and function calls, we developed a synthetic app in order to perform a micro-benchmark testing performance penalties when executing Java and native method calls. The micro-benchmark tests the most significant native functions by means of AppBox native interceptors. In Table 5.3, we report the overhead introduced by AppBox when hooking functions in libc (shown in the first column). In the same table, we also compare our overhead with Boxify performance overheads as reported in [25]. As the results show, AppBox 's performance overhead is significantly less than Boxify's, mainly because of the extra rounds trip needed by Boxify to forward to the the Broker component each intercepted calls which costs in average $\approx 100\ \mu$s of delay for each call.

Table 5.5 reports the results for the overhead introduced by AppBox when interposing Java Android APIs, Table 5.4 presents hooks that were in place during our experiments. Results shows that the performance penalty introduced by AppBox 's API Hooking, while not being ideal, is acceptable. It is worth noting that micro-benchmarks test listed in Table 5.5 in some cases (i.e., openFileOutput and Create File) are responsible for triggering multiple hooks. A very fast operation such creating a new File object has a performance degradation which cost in average 10 μs of delay. The AppBox Java interceptor's performances not being ideal compared to the native interceptor's, this is an expected result because of the API hooking mechanism employed by AppBox , that relies on Java reflection for invoking the original method reference.

Table 5.2 BenchMark Apps Results For Nexus 5x (64 bit)

App	Test	AppBox	Native	Loss
Quadrant	Total	17736	17449	1.6%
	I/O	9820	8277	15.7 %
Vellamo	multi-core	1948	1930	0.9 %
	webview	2803	2688	4.1 %

5.5.2 Effectiveness

During this evaluation our goal was twofold: (i) to demonstrate that AppBox is easy-to-deploy and fully capable to wrap real-world apps and (ii) to assess AppBox robustness. To this end, we executed 1000 free apps from the Google Play Store (retrieved in November 2016). As reported in Table 5.6, the average size of those apps is around 20 MB and 66 of them (6.6%) were recognized as obfuscated. To recognize obfuscation, we employed

APKiD[7] that leverages on different heuristic in order to statically detect presence of packers, protectors and obfuscators. Being able to properly handle obfuscated apps demonstrate the code-agnostic property that AppBox has in contrast with some similar works.

Table 5.6 Percentage of Obfuscated Apps

analyzed apps	perc. obfuscated	average size
1000	6.6%	20 MB

To execute our tests, we had to modify the manifest of the collected apps adding the attributes requested by AppBox . This was required only for testing purposes. We stress that this step is not required in the operational scenario where the developer will be responsible for performing this task. We evaluated the runtime robustness of AppBox running the collected apps on a Nexus 5x with stock Android 8.0. We employed the DroidBot[**?**] tool to exercise the managed app's functionality. Droidbot first statically analyses the target app then it allows to dynamically injects event to stimulate the app under analysis. We ran each managed apps for 5 minutes similar to the Google Play Bouncer [**?**] while we were collecting log information seeking for app crashes. From the 1000 apps, only 56 (5.6%) apps reported a crash during testing. Manual investigation of the dysfunctional apps reveled that most errors were caused by bugs in those apps that were triggered during Droidbot dynamic stimulation. In particular, we noticed that most of the crashes were caused because Droidbot denied the runtime request for a permission causing the apps to crash. From this test, we can conclude that AppBox did not cause any app to crash and none of the apps were negatively affected by being executed under AppBox sandbox.

[7]https://github.com/rednaga/APKiD

Table 5.3 Native Micro-Benchmarks AppBox Performance, Compared against Boxify.

AppBox: Nexus 5x (250k runs)				Boxify: Nexus 5 (15k runs)		
Libc. Func.	Native	on AppBox	Overhead	Native	on Boxify	Overhead
open	6.49 μs	6.7 μs	3.2%	9.5 μs	122.7 μs	1191%
mkdir	92.7 μs	95.2 μs	2.7%	88.4 μs	199.4 μs	125%
rmdir	80.7 μs	85.3 μs	5.7%	71.2 μs	180.7 μs	153%

5.5.3 Applicability and Effectiveness

To further stress our system to evaluate its applicability and effectiveness, we manually executed 5 of the most popular free apps from the top categories concerning business functionality. Our goal in this experiment is threefold:

- to test the applicability of AppBox on several Android versions, we run this experiment on several versions ranging from IceCreamSandwich (ICS) to Oreo;
- to verify the correct interaction of the managed app with the Android OS, we completed the authentication process, if present, testing for correct delivery of system events (i.e., incoming SMS) and the interaction with Google apps such as Google Play Service;
- finally to stress the AppBox SandboxService we manually switched it from enabled to disable to detect possible issues when the managed app is executed outside AppBox .

Table 5.7 shows the list of apps we selected. For each app we enabled, in spirit of the scenario envisioned in Section 5.1, the following policies. First, to monitor network-related functionalities we interposed various functions to enforce policy such as deny connections to known addresses of advertisement servers taken from a public list[14] as well as monitoring of network connections that do not make use of the secure layer offered by TLS. Second, we enable file system monitoring policies to detect file operations on the SD-card. Lastly, a fake Location Provider was in place to make AppBox returning mock data to the managed app. We manually stimulated those apps for 8 minutes, performing various operations for testing functionality such as visiting web pages, acquire and share location data via GPS mechanism and user authentication through Google Play Services. It is worth of note that such authentication mechanism requires a direct communication between the apps and the Google Play Service which is a Google proprietary app, hence that communication could not be instantiate via an emulated component (i.e., the *Broker* like in Boxify). During such test, two experiments were performed addressing two different execution mode, AppBox with policies enabled and without them. Basically, in the latter experiment AppBox is in place but none policy is enforced, we enabled just logging operations to be reported in order to detect eventually byzantine behavior.
Our tests showed AppBox effectiveness because in both experiments none of the tested apps crashed and we were able to notice the enforced behavior when policies were enabled. In fact, AppBox effectively blocked any connection to the blacklisted addresses resulting in the absence of the advertisements that instead would have been regularly showed without the enforcement of the policy by AppBox , furthermore connections made without support of TLS were denied and reported correctly as well. In particular, when location policy

enforcement was in place we noticed that the actual location shared via tested apps was referring to our fixed value (i.e. the North Pole) previously set via AppBox .

It would have been interesting to test the update process of an app under AppBox . Unfortunately, since we did not have an independent developer's cooperation during our evaluation, we were unable to properly test this aspect.

Table 5.7 Popular Apps We Used for Testing AppBox Applicability and Effectiveness

App Name	Version	Category
Skype for business	6.13.06	Business
Slack	2.30.0	Business
Dropbox	38.2.4	Productivity
Intesa San Paolo Mobile Banking	2.1.0	Finance
Chrome	56.0.2924.87	Communication

Table 5.4 Android API - Micro-Benchmark Results

	Nexus 5x (15k runs)		
Android API	Native	on AppBox	Overhead
Read Contact	6.55 *ms*	9.93 *ms*	3.38*ms* (51.6%)
Socket	23.23 *ms*	26.33 *ms*	3.1*ms* (13.3%)
openFileInput	0.19 *ms*	0.22 *ms*	0.03*ms* (15,7%)
openFileOutput	0.24 *ms*	0.28 *ms*	0.04*ms* (16.6%)
Create File	0.02 *ms*	0.03 *ms*	0.01*ms* (50%)
Open Camera	227.09 *ms*	237.02 *ms*	9,93 *ms* (4.4%)

Table 5.5 Android APIs Monitored During Java Micro-Benchmarks

Alias	Class package
Read Contact	android.content.ContentResolver
Networking	java.net.Socket
File	android.app.ContextImpl
	android.app.ContextImpl
	java.io.File
Camera	android.hardware.camera2.CameraManager

5.6 Chapter Summary

This paper presents AppBox , a novel black-box app sandboxing solution for stock Android particularly suitable for enterprise domains, where apps running on employees' smartphones are often managed by specialised services such as MAMs and MDMs. AppBox allows to monitor and if needed, also to enforce fine-grained security policies regulating the behaviour of a mobile app covering both Java components and native libraries, including those belonging to third parties. AppBox relies on its ability of running the managed app confined within an instrumented process space, while avoiding any modifications to its bytecode. This instrumentation is achieved by runtime memory modifications. The instrumentation allows full monitoring of Android Java APIs as well as native functions. AppBox reduces quite a lot the maintaining costs in comparison with those requested by customizable approaches. In such cases, the developer is asked to port app updates across all the managed apps, requiring time and financial efforts. Furthermore, a preliminary evaluation showed that AppBox produces a limited performance overhead. Further tests have shown that the mechanism is quite robust and of general applicability to real apps and not only to toy examples. To allow other researchers to use and work with AppBox , we have made available its implementation.

Chapter 6

Identifying and Evading Android Sandbox through Usage-Profile based Fingerprints

Among the massive volume of Android apps used by Android users, there exists a lot of Android malware, which become the main threat for Android users currently. To mitigate the threat of the Android malware, static and dynamic analysis techniques are the main solutions to detect Android malware. Static analysis has the limitation on detecting the malware when using the code obfuscation, native code, Java reflection and packer. But, dynamic analysis can help detect such Android malware more precisely in its dynamic sandboxes. The traditional dynamic analysis sandbox is built either on the Android emulator or the real device to enable fast and effective malware detection.

To evade dynamic analysis, some anti-emulator techniques [58, 70, 44] were proposed, and they are commonly used by Android malware. In general, these techniques were designed to obtain fingerprints of the runtime environment of the Android emulator. Recently, BareDroid [53] was proposed to use real devices to build the Android sandbox. This method can mitigate the anti-emulator techniques , but we believe the arms race between dynamic analysis techniques and evasion techniques is endless.

No matter whether the Android sandbox is built on the Android emulator or the real device, the Android malware can still evade the detection of the sandbox through identifying the difference between the emulated phone and the real user phone. In this paper, we conduct a measurement on collecting fingerprints from public Android sandboxes, including AV sandboxes, online detection sandboxes and even sandboxes used by app markets. Through analyzing collected fingerprints, we propose an evading technique based on a new type of fingerprints of Android sandboxes: *usage-profile based fingerprints* (e.g., the contact list

information, SMS, and installed apps). The measurement result shows that these Android sandboxes have no usage-profile based fingerprints, or only have a fixed usage-profile based fingerprint, or have a random artifact fingerprint. So, most current Android sandboxes have not protected their usage-profile fingerprints, and are potentially bypassed by malware samples.

Therefore, by analyzing the usage-profile based fingerprints of the Android sandboxes and real Android user's device, the Android malware can still potentially evade the dynamic sandbox because of two reasons: 1) extracting some fingerprints, such as installed apps, are not very sensitive towards the verdict making of dynamic analysis, since those behaviors are commonly existed in benign apps. For example, Android Ads SDK extracts the installed app list for accurately distributing ads; 2) even though extracting some fingerprints (e.g. the contact list and SMS) may be regarded by dynamic analysis as malicious, the Android malware can still evade the detection by mimicking or repackaging [75] as a contact app or SMS app. So, the advanced Android malware could firstly inspect these fingerprints and then launch other more powerful behaviors (e.g., rooting and sending SMS) if it does not identify the current environment as a sandbox environment.

To summarize, this paper makes the following contributions:

- 1) **New problem.** We propose a new Android sandbox fingerprinting technique, which is based on the careless design of usage-profiles in most current sandboxes. We observe that malware developers can collect usage-profile based fingerprints from many Android sandboxes and then leverage these fingerprints to build a generic sandbox fingerprinting scheme for the sandbox analysis evasion.

- 2) **Implementation.** We conduct a measurement on collecting usage-profile based fingerprints on popular Android sandboxes. The results show that most Android sandboxes designers have not protected these fingerprints by generating the random fingerprints every time for running a different sample. Only few sandboxes generate the random fingerprints, but these random fingerprints are different from fingerprints in user's real phones.

- 3) **Mitigations.** We propose mitigations to further guide a proper design of these sandboxes against this hazard.

The remainder of the paper is structured as follows: in Section **??** we introduce background and motivations underlying our research, then in Section 6.1 we discuss our system design and in Section 6.2 we present collected results which effectively shows the effectiveness of our techniques. Proposed mitigations and related work are discussed in Section 6.3 and Section 6.4 respectively. Finally, Section **??** concludes the paper.

6.1 System Implementation

Fig. 6.1 The design of fingerprint collector

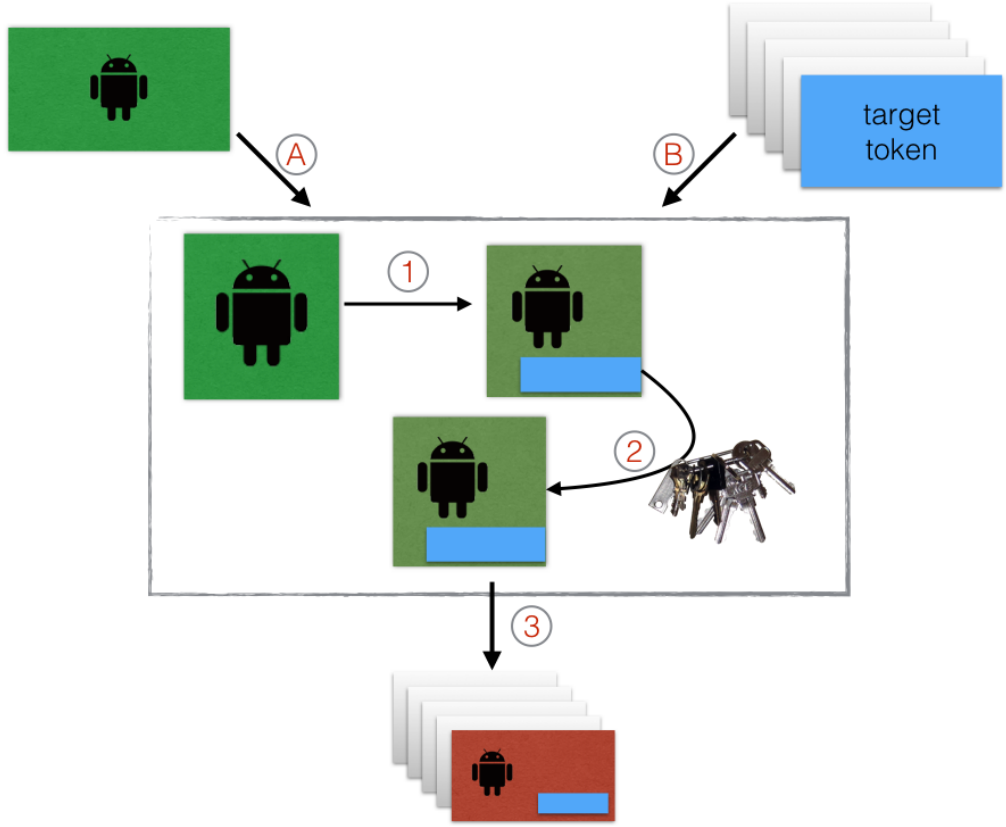


Fig. 6.2 Scouting apps generator

In this section, we present the design of the fingerprint collector. As depicted in Figure 6.1, it consists of two components: (I) scouting-app and (II) fingerprint extractor. The scouting-app is used to collect usage-profile fingerprints. It uses only public Android APIs to get information about the execution environment. The app does not use native code, neither Java reflection nor code obfuscation techniques to hide its sensitive behaviors. The reason is that the prior static analysis in most analyzers would highly regard the scouting app as a

Table 6.1 usage-profile Data

Type	Data
Contact	Name, numbers, email
Location	GPS, network, latitude, longitude
SMS	inbox, send, draft
WiFi	SSID, signal-strength
App	package-name, version, hash
Battery	battery-level, stat, chargePlug
Call	recent calls

benign app and filter it out if none suspicious evidence is found. To this end, and to make the fingerprint process even more stealthy as well we implemented several scouting-app each testing for a subset of usage-profile we were interested in. By following this approach we avoid to create a single application that requires a lot of sensitive privileges which could be marked as suspicious and then manually analyzed later on, instead we use several scouting-app such that none of those would require a suspicious combination of sensitive privileges. The scouting app uses different threads to execute the scouting logic. Each analysis sends its results back to the remote server in the JSON format. All types of collected fingerprints are shown in Table 6.1.

To track which sandbox analyzes the scouting app, we need to build a scouting app generator. In the scouting app generator, we use a testing app as the base app to generate different apps for each target sandbox. Each generated app is signed by a different certificate. We also make the signature of each app different to avoid the trivial caching mechanism used by the sandbox. Moreover, since we need to classify the results coming back from the testing app, we repackaged the testing app for inserting a *target token*. Then, at run-time the app sends back that token within its fingerprint data. Besides that, the app also sends back its certificate signature, so that we can check if the app has been repackaged by the sandbox for using the static instrumentation hooking framework. Advanced sandbox might employ mechanisms in order to fool the certificate signature check, to countermeasure those we included Java code which is responsible for calculating the DEX file hash value at runtime. With these checks in place, we can detect sandbox which eventually have modified the application being analyzed in order to disable signature checks.

Figure 6.2 shows the generator design: the system receives the testing app (A) and a list of *target tokens* (B) as inputs. Then, for each target token the generator performs repackaging

of the testing app by inserting the target token (1) in app's assets directory. Then, a new digital certificate is created and the repackaged app is signed with. This procedure repeats for each target token. Finally, the output is a set of apps which contain a token for each different target sandboxes.

The fingerprint extractor component first does normalization on the results collected by our scouting applications. It then stores the unique data in a database and uses the token mechanism to create the mapping between the scouting app and the target sandbox. Finally, the collected data is analyzed to determine whether the produced data from the sandbox is dynamically generated or not.

6.2 Results

In this section, we describe the fingerprints collected by the scouting app. Because of the difference in the Android app distribution channel, we choose different types of mobile sandboxes accordingly as the target of this work, which are shown in the Table 6.2. In fact, it is composed by both official and third-party stores, i.e. Google Play, aptoide, F-Droid, etc... and also by stock applications installed on real-world devices. First column in Table 6.2 shows the type of sandbox being analyzed, second and third columns represent sandbox's name and its availability at the time of writing respectively. We choose to target mobile anti-virus vendors because they use either their own customized sandboxes or an online sandbox service to dynamically analyze collected samples. Moreover, we collect the environment-related data from third-party stores, because it is one of the most popular malware spreading channel. Considering that online malware analysis services are used by both mobile anti-virus and third-party stores, we also collect environment-related data from available online sandboxes. Unfortunately, compared to other previous works [69, 57, 54, 64], we find that just few of these online services are available at the time of writing, as evicted by third column of Table 6.2.

We use the system described in Section 6.1 to generate 10 applications for each target sandbox. Then, depending of the target type, we manually upload each generated app by using the web interface provided by the online sandbox service, or install on a real device or send it to the third-party store. In the latter case, we immediately remove the app as soon as the scouting app has been analyzed to make sure nobody has ever downloaded it. Moreover, we do not disclose the mapping between the sandbox name and its representing label to allow to sandbox maintainers to fix this problem.

We receive the environment-related data from 80% of available sandbox in the Table 6.2. Table 6.3 and Table 6.4 show a summary of the most relevant environment-related data collected by our testing app. The former contains results of Contact data while the latter contains results of SMS data. Both Tables have the last two columns in common which report whether the specific sandbox does dynamically generates the environment-related data (Random data column) and count the number of collected results (num. of results column) respectively. As for Table 6.3 we reported name, number and email were collected by the analysis, instead in Table 6.4 we included sending number and body if any. We have not included the phone call data since all sandboxes return the empty result. Thus, the phone call data could be one of the best user-profile fingerprints.

As describe in Section 6.1, our system allows us to detect if a target sandbox returns a fixed data for a specific *environment artifact*. Unfortunately, as shown by "Random Data"

Table 6.2 Mobile Sandboxes employed for evaluation. (✗ means not available at the time of writing)

Type	Sandbox	Available
Online	Andrubis [72]	✗
	SandDroid [66]	✓
	TraceDroid[68]	✗
	CopperDroid[65]	✗
	HackApp [app]	✗
	NVISO ApkScan [nvi]	✓
	Koodous [koo]	✓
	VirusTotal [67]	✓
	Joe Sandbox mobile [21]	✓
	ForeSafe	✗
Antivirus	Bit Defender	✓
	360 mobile	✓
	TrendMicro	✓
	Kaspersky	✓
	Tencent mobile	✓
App store	Amazon [ama]	✓

column in Table 6.3 and 6.4, the results indicate that most Android sandboxes do not use dynamically generated environment data.

One interesting finding is that two mobile anti-virus sandboxes ran the application on a real-world device. In fact, they returned concrete
WiFi artifact data (i.e. "wifiscan":"DIRECTCnFireTV_8048",
"wifiscan":"AppStore","wifiscan":"AVcontrol").
Moreover, the data collected from the *battery artifact*, presented in Table 6.5 is exactly the same for all the sandbox, except for the data returned by these two anti-virus sandboxes. In fact, an unmodified Android emulator returns a fixed battery stats, which consists of the following string:
"chargePlug":"1","batteryStat":"2","batteryLevel":"50.0".
Instead, results collected from a real-world device look like different:
"chargePlug":"2", "batteryStat":"3", "batteryLevel":"100.0".
Regarding *application artifact* data, we found that about 77% of targets the sandboxes contain the identical set of applications found on stock Android emulator. As Tables 6.6 shown, some anti-virus and all online sandboxes present only the default installed app list from the Android emulator, and they never return random data for the installed app list.

In the following list, we include some interesting apps' package name:

- com.amazon.geo.contextcards
- com.amazon.rialto.cordova.webapp.
  webapp50f95ccb054443059066310aefdf969b
- IAPV2AndroidSampleAPK
- com.amazon.otaverifier
- com.tencent.token

Table 6.3 Contacts usage-profile Results

Sandbox	Name	Phone	Email	Random data
store_1	Mary Edwards Harry Grace	867-5309 867-5319	✗ ✗	✗
online_x	✗	✗	✗	✗
online_y	Firstname1 Firstname2 Firstname3	1 301-234-5678 1 381-234-5678 1 381-234-5678	✗ ✗ ✗	✗
av_1	Ion Gheo Txet4321	074-354-3219 072-345-6789 074-212-3456	✗ ✗ ✗	✗
av_2	Cynthia Alexander Alexandra Dolores	✗ ✗ ✗ ✗	✗ ✗ ✗ ✗	✗
av_3	MARS	1 566-666-6666	chengkai_tao@****.com.cn	✗
av_4	Boulder Hypnotherapy Ctr St. John Ambulance Maidstone Golf Centre	(303) 776 8100 061 412480 01622 863163	z**y@gmail.com l**k@stjohn.ie nick@totalgolfcoaching.co.uk	✓
av_5	Jian Li Jian Li Xuri Jin	1 3743888229 13606500401 13250324837	lijxev@admin.cn b0***54@admin.cn 55**43@yuepao.cn	✓

Table 6.4 SMS usage-profile Results

Sandbox	Num	Body	Random data	num. of results
store_1	2020845845 +18454119384 5618675309	Hey Who Important	✗	3
online	✗	✗	✗	1
av_1	12345 1234	smsmomealain smsmomealain	✗	2
av_2	1354-587-2365 1857-667-8565	Mum Jefferson	✗	2
av_3	1301-234-5678 1301-234-5678 1381-234-5678 1581-234-5678	Ggggggggg Testzzzzzz 123456789 Fffffffffmmmmmm	✗	12
av_4	10668820 10086 13770837893	guchulaichonghuafei nihao,laikaitong4g-songliuliango nihao,nishi4staryonghu,keyihuantingji	✓	15
av_5	13540877911 15874984303 18660928896	u0oydyemvub4lu86kfcbwad46pvhmh6o 2fqjfkr629blowmxso4jh6dzqtk3f4j2 lhb0j8hxi48wdjiua1q0qvsleeffgt6g	✓	22

Table 6.5 Battery usage-profile Results

Sandbox	Battery stats	Random data
av_1	batteryStat'='3, 'chargePlug'='2', batteryLevel = '100'	✓
av_2	batteryStat'='2, 'chargePlug'='1', batteryLevel = '98'	✓
online, store	batteryStat'='2','chargePlug'='1','batteryLevel'= '50.0'	✗

Table 6.6 Installed Apps usage-profile Results

Sandbox	Emulator apps	uncommon apps	Random data
av_x	✓	✓	✗
av_y	✓	✗	✗
store	✓	✓	✗
online	✓	✗	✗

6.3 Defense

As we discussed in previous sections, the environment-related data represents an interesting source of information, which could be exploited to identify the mobile sandbox. Building a database of fingerprints from all sandboxes, an attacker could take the advantage to easily detect an existing mobile sandbox by checking the presence of matching data in the running environment.

To avoid this trivial detection mechanism, it is important to generate environment-related data dynamically, so each application under analysis would see a different environment.

Note that the presence of each previous discussed *environment artifact* is also an important indicator of the goodness of the running environment. As discussed in Section 6.2, sandboxes have not included the Call artifact. Even though such type of environment data i.e. WiFi data, could not be generated, one can artificially inject it by using hooking frameworks introduced in previous works [Costamagna and Zheng, Xpo, Cyd, And].

In addition to set up the sandbox environment as close as a real user phone, the sandbox developer could take a hybrid approach to detect such fingerprint collection behavior. For example, some sandboxes are equipped with the same set of environmental data, and other sandboxes are equipped with complete different data. When performing the dynamic analysis, each suspicious sample should be run in these two different sandboxes with similar triggering events. If two types of sandboxes yield different malicious behaviors, it indicates that the malware performs the evasion attack by checking the usage-profile based fingerprints.

6.4 Related Works

A couple of previous research works focus on evading the Android sandbox or Android anti-virus scanners. Huang, et al. [42] discovered two generic evasions that can completely evade the signature based on-device Android AV scanners, while we are focusing on the Android sandboxes used offline. Sand-Finger [49] collects fingerprints from 10 Android sandboxes and AV scanners to bypass current AV engines and all of fingerprints used by Sand-Finger are hardware-related or system-related, which are different from our user profile based fingerprints. Timothy [70] presented several sandbox evasions by analyzing the differences in behavior, performance, hardware and software components. He also revealed that dynamic analysis platforms for malware that purely rely on emulation or virtualization face fundamental limitations that may make evasion possible. The above approaches are mainly related to detect the Android emulator environment. Wenrui, et al. [34] proposed to evade the Android runtime analysis through by identifying automated UI explorations. Similarly, our work is to distinguish the difference between the sandbox environment and the real user device environment through usage-profile based fingerprints.

In [28] Blacktorne et al. proposed *AVLeak* a blackbox technique to extract emulator fingerprints. Although it address the similar problem about how to efficiently extract emulator fingerprints the implemented methodology is not suitable on Android. Moreover *AVLeak* was not focused on usage-profile based fingerprints which are quite relevant ones for the mobile ecosystem.

6.5 Chapter Summary

In this paper, we present a novel mobile sandbox fingerprinting for the sandbox evasion by checking the usage profiles. As demonstrated by the evidence of collected results, usage profiles data could be used by a malware to fingerprint current sandboxes. We demonstrate that most of our analyzed sandboxes are built with fixed usage profiles and they completely overlook this potential hazard. Mitigations are provided by us to prevent such evasion hazards, e.g., different app runtime should present dynamically generated usage profiles, or hybrid usage profiles. Our research raises the alert for the usage-profile based fingerprinting hazard when developing mobile sandboxes and sheds lights on how to mitigate similar hazards.

Chapter 7

OctoDroid

Chapter 8

Conclusions and Future Work

References

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

[ama] Amazon application store. https://www.amazon.com/mobile-apps/b?ie=UTF8&node=2350149011. Accessed: 2016-02-27.

[ADB] Android dynamic binary instrumentation. https://github.com/Samsung/ADBI. Accessed: 2016-02-27.

[And] Android hooker. https://github.com/AndroidHooker/hooker. Accessed: 2016-02-27.

[app] Apphack mobile scanner. https://apphack.com. Accessed: 2016-02-27.

[Cyd] Cydia substrate for android. http://www.cydiasubstrate.com. Accessed: 2016-02-27.

[DDI] Dalvik dynamic instrumentation framework. https://github.com/crmulliner/ddi. Accessed: 2016-02-27.

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

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

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

[Fri] Frida.re. https://frida.re. Accessed: 2016-02-27.

[gdp] General data protection regulation. https://www.eugdpr.org/. Accessed: 2017-01-30.

[koo] Koodus collaborative platform. https://koodous.com. Accessed: 2016-02-27.

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

[nvi] Nviso apk-scan. https://apkscan.nviso.be/. Accessed: 2016-02-27.

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

[rum] 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.

[SuS] Susi. https://github.com/secure-software-engineering/SuSi. Accessed: 2016-02-27.

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

[Xpo] Xposed framework. https://repo.xposed.info. Accessed: 2016-02-27.

[21] (2016). Joe mobile sandbox. http://www.joesecurity.org/joe-sandbox-mobile.

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

[23] Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. (2012). Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM.

[24] Backes, M., Bugiel, S., Derr, E., Gerling, S., and Hammer, C. (2016). 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.

[25] Backes, M., Bugiel, S., Hammer, C., Schranz, O., and von Styp-Rekowsky, P. (2015). Boxify: Full-fledged app sandboxing for stock android. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 691–706.

[26] Backes, M., Gerling, S., Hammer, C., Maffei, M., and von Styp-Rekowsky, P. (2013). Appguard–enforcing user requirements on android apps. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 543–548. Springer.

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

[28] Blackthorne, J., Bulazel, A., Fasano, A., Biernat, P., and Yener, B. (2016). Avleak: Fingerprinting antivirus emulators through black-box testing. In *Proceedings of the 10th USENIX Conference on Offensive Technologies*, pages 91–105. USENIX Association.

[Costamagna and Zheng] Costamagna, V. and Zheng, C. Artdroid: A virtual-method hooking framework on android art runtime. *Innovation in Mobile Privacy & Security IMPS'16*.

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

[31] Costamagna, V. and Zheng, C. (2016b). Artdroid: A virtual-method hooking framework on android art runtime. *Proceedings of the 2016 Innovations in Mobile Privacy and Security (IMPS)*, pages 24–32.

[32] Davis, B. and Chen, H. (2013). Retroskeleton: retrofitting android apps. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 181–192. ACM.

[33] Davis, B., Sanders, B., Khodaverdian, A., and Chen, H. (2012). I-arm-droid: A rewriting framework for in-app reference monitors for android applications. *Mobile Security Technologies*, 2012.

[34] Diao, W., Liu, X., Li, Z., and Zhang, K. (2016). Evading android runtime analysis through detecting programmed interactions. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec '16, pages 159–164, New York, NY, USA. ACM.

[35] Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2014). Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5.

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

[37] Gordon, M. I., Kim, D., Perkins, J. H., Gilham, L., Nguyen, N., and Rinard, M. C. (2015). Information Flow Analysis of Android Applications in DroidSafe. In *NDSS*. Citeseer.

[38] Grace, M. C., Zhou, Y., Wang, Z., and Jiang, X. (2012). Systematic detection of capability leaks in stock android smartphones. In *NDSS*.

[39] Gruver, B. (2015). Smali/Baksmali Tool. *https://github.com/JesusFreke/smali/wiki*.

[40] Hao, H., Singh, V., and Du, W. (2013). On the effectiveness of api-level access control using bytecode rewriting in android. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 25–36. ACM.

[41] Hoffmann, J., Ussath, M., Holz, T., and Spreitzenbarth, M. (2013). Slicing droids: Program slicing for Smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1844–1851. ACM.

[42] Huang, H., Chen, K., Ren, C., Liu, P., Zhu, S., and Wu, D. (2015). Towards discovering and understanding unexpected hazards in tailoring antivirus software for android. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*.

[43] Jiang, Y. Z. X. (2013). Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*.

[44] Jing, Y., Zhao, Z., Ahn, G.-J., and Hu, H. (2014). Morpheus: Automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 216–225, New York, NY, USA. ACM.

[45] Lee, B., Lu, L., Wang, T., Kim, T., and Lee, W. (2014). From zygote to morula: Fortifying weakened aslr on android. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 424–439. IEEE.

[46] Li, L., Bartel, A., Bissyandé, T. F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., and McDaniel, P. (2015). 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.

[47] Li, L., Bissyande, T. F. D. A., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., and Le Traon, Y. (2016). Static Analysis of Android Apps: A Systematic Literature Review. Technical report, SnT.

[48] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. (2012). 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.

[49] Maier, D., Protsenko, M., and Müller, T. (2015). A game of droid and mouse. *Comput. Secur.*, 54(C):2–15.

[50] Mirzaei, N., Malek, S., Păsăreanu, C. S., Esfahani, N., and Mahmood, R. (2012). Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5.

[51] Mulliner, C., Oberheide, J., Robertson, W., and Kirda, E. (2013). Patchdroid: scalable third-party security patches for android devices. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 259–268. ACM.

[52] Mulliner, C., Robertson, W., and Kirda, E. (2014). Virtualswindle: An automated attack against in-app billing on android. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 459–470. ACM.

[53] Mutti, S., Fratantonio, Y., Bianchi, A., Invernizzi, L., Corbetta, J., Kirat, D., Kruegel, C., and Vigna, G. (2015). Baredroid: Large-scale analysis of android apps on real devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 71–80, New York, NY, USA. ACM.

[54] Neuner, S., Van der Veen, V., Lindorfer, M., Huber, M., Merzdovnik, G., Mulazzani, M., and Weippl, E. (2014). Enter sandbox: Android sandbox comparison. *arXiv preprint arXiv:1410.7749*.

[55] Octeau, D., Luchaup, D., Dering, M., Jha, S., and McDaniel, P. (2015). 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.

[56] Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., and Le Traon, Y. (2013). 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.

[57] Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., and Ioannidis, S. (2014a). Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, page 5. ACM.

[58] Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., and Ioannidis, S. (2014b). Rage against the virtual machine: Hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, EuroSec '14, pages 5:1–5:6, New York, NY, USA. ACM.

[59] Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., and Vigna, G. (2014). Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS*, volume 14, pages 23–26.

[60] Rasthofer, S., Arzt, S., and Bodden, E. (2014). A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*.

[61] Rasthofer, S., Arzt, S., Miltenberger, M., and Bodden, E. (2016). Harvesting runtime values in android applications that feature anti-analysis techniques. In *NDSS*.

[62] Rastogi, V., Chen, Y., and Enck, W. (2013a). Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 209–220. ACM.

[63] Rastogi, V., Chen, Y., and Jiang, X. (2013b). 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. ACM.

[64] Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., and Hoffmann, J. (2013). Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1808–1815. ACM.

[65] Tam, K., Khan, S. J., Fattori, A., and Cavallaro, L. (2015). Copperdroid: Automatic reconstruction of android malware behaviors.

[66] Team, B. R. et al. (2014). Sanddroid: An apk analysis sandbox. xi'an jiaotong university.

[67] Total, V. (2012). Virustotal-free online virus, malware and url scanner.

[68] Van Der Veen, V., Bos, H., and Rossow, C. (2013). Dynamic analysis of android malware. *Internet & Web Technology Master thesis, VU University Amsterdam*.

[69] Vidas, T. and Christin, N. (2014a). Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458. ACM.

[70] Vidas, T. and Christin, N. (2014b). Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 447–458, New York, NY, USA. ACM.

[71] Wei, F., Roy, S., Ou, X., et al. (2014). 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.

[72] Weichselbaum, L., Neugschwandtner, M., Lindorfer, M., Fratantonio, Y., van der Veen, V., and Platzer, C. (2014). Andrubis: Android malware under the magnifying glass. *Vienna University of Technology, Tech. Rep. TRISECLAB-0414*, 1:5.

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

[74] Xu, R., Saïdi, H., and Anderson, R. J. (2012). Aurasium: practical policy enforcement for android applications. In *USENIX Security Symposium*, volume 2012.

[75] Zhang, F., Huang, H., Zhu, S., Wu, D., and Liu, P. (2014). Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*.

[76] Zhauniarovich, Y., Ahmad, M., Gadyatskaya, O., Crispo, B., and Massacci, F. (2015). 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.

[77] Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., and Zou, W. (2012a). 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. ACM.

[78] Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., and Zou, W. (2012b). 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*, SPSM '12, pages 93–104. ACM.

[79] Zhou, Y., Patel, K., Wu, L., Wang, Z., and Jiang, X. (2015). Hybrid user-level sandboxing of third-party android apps. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 19–30. ACM.