Advances In Security Analysis Techniques To Increase Privacy And Security Of The Android Ecosystem

Valerio Costamagna

Supervisor: Prof. F. Bergadano, Università degli studi di Torino, Italia

Examiners: Prof. Cosimo Anglano, Università Piemonte Orientale, Italia
Prof. Antonio Lioy, Politecnico di Torino, Italia
Prof. Dario Catalano, Università degli studi di Catania, Italia

Department of Computer Science
University of Torino

This dissertation is submitted to the department of Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
September 2018

Acknowledgements

I would like to take the opportunity to thank anyone who has supported me during the last couple of years. The resolute presence of all of you made my journey as a PhD student an unforgettable experience.

First, I would like to thank Prof. Dr. Francesco Bergadano for his time in assisting me as advisor. In addition, thanks for your precious time spent in debating around several computer security research topics.

I would also like to thank Prof. Dr. Bruno Crispo for the opportunity to join his research group in Trento, an experience I would not want to miss. I would also like to express my gratitude for sharing your invaluable time and experience in applied computer security. Any place or time was the right one for discussing about arising challenges, most of the ideas I am happy of have seen the light during lunch breaks.

In addition, I would like to thank Prof. Dr. Giovanni Russello for his contribution as co-author and his countless advice that consolidated my research.

Furthermore, I would also like to express my gratitude for having had the opportunity to work with highly skilled and motivated researchers from other institutions, including Dr. Cong Zheng from Palo Alto Network and his colleagues as well as the colleagues from University of Trento. Also, I would like to extend my thankful to the examiners for the precious time dedicated to review and help with this dissertation.

Finally, last but not least, I am greatly in debt with my parents, you made all this possible in the first place. Yet, I would like to express my delighted gratitude to all of you, priceless friends close and far away that constantly make my life a better experience. ROPtors never die.

Abstract

The proliferation of Android's user base, unfortunately, has also made the devices become prominent targets for a variety of attacks ranging from ex-filtration of users' privacy-sensitive data or data encryption via ransomware as well as malicious apps attempting to achieve privilege escalation attacks. Furthermore, identifying and eliminating vulnerabilities is gaining in importance as a single missed flaw within an application's component can suffice for an attacker to fully achieve any privileges granted to the attacked application.

This dissertation details our research on approaches and solutions aiming to improve security and privacy of Android ecosystem at large by a combination of static and dynamic analysis techniques. *ARTDroid*, a dynamic instrumentation framework based on memory modifications has been proposed to ease intercepting of any calls at runtime. ARTDroid relies on dynamic instrumentation and fully supports the latest ART runtime running on any versions above to Android KitKat. Consequently, we investigate how to combine static analysis results to drive application's execution. Typically dynamic analysis, and hybrid analysis too for that matter, brings the problem of stimulating the application's behavior which is a non-trivial challenge especially for Android applications. To this end, we propose a backward slicing based targeted inter component code paths execution technique, *TeICC*. TeICC leverages a backward slicing mechanism to extract code paths starting from a target point in the application. The extracted code paths are then instrumented and executed inside the application context to capture sensitive dynamic behavior as well as to resolve dynamic code updates and obfuscation. As dynamic instrumentation provides a variety of capabilities to monitor and alter application's behavior at runtime, we combine those in couple with static analysis techniques and propose *StadART* an hybrid approach which combines static and analysis to cover the inherent shortcomings of static analysis techniques to analyze applications in the presence of dynamic code updates. To address the growing demand for Mobile Application Management (MAM) capabilities, especially when Bring Your Own Device (BYOD) mechanism is in place, we proceed to investigate solutions that would allow enterprises to configure fine-grained security and privacy policies on employees' devices. Existing MAM solutions provide policies that are app-specific and often require the enterprise to acquire new applications. To be managed by an MAM, the code of an application needs to be customised using specific software development kits (SDKs) provided by MAM vendors, in general, application customisations are not affordable due to the cost associated with maintenance and support. Moreover developers might not be too keen to provide app source code to the enterprise. To this end, we propose *AppBox* a novel app-level Mobile Application Management (MAM) solution that enables enterprises to enforce custom security and privacy policy at the minimum effort for the developers. Using *AppBox* , an enterprise can customise any existing application, even highly-obfuscated ones, without using any SDK or modifications to the application bytecode. The main idea of AppBox is to provide a complete and accurate MAM app-level mechanism by altering a single line in the application's manifest file. Thanks to its novel approach, AppBox would enable an enterprise to select any application from the market and to be able to perform fine-grained customisations with minimum collaboration from the app developer. As orthogonal research, we aim to improve the actual state-of-the-art of sandboxing analysis services for Android applications. To this end we examine those artifacts that provide information which may be exploited by malicious applications to detect the analysis environment, thus concealing their malicious behavior. Our results raise the alert for the usage-profile based fingerprinting hazard when developing mobile sandboxes and sheds lights on how to mitigate similar hazards. The capability of detecting malware apps by means of sandboxing analysis services provides a valuable barrier against the proliferation of malware. Although, it is also relevant to identify code vulnerabilities that may be exploited by malware apps as well as other malicious actors to achieve device remote control. To this end, we investigate how to improve code auditing of C++ codebase, in particular concerning the Android system services C++ implementation. We propose *OctoDroid* a Clang based plugin for the Octopus platform, a prominent static analysis tool based on the Code Property Graph (CPG) representation. *OctoDroid* employs the Clang's LibTooling library capabilities to produce an augmented code that enable Octopus's analysis of C++ code. *OctoDroid* employs a two stages approach. First, *OctoDroid* performs the Class Hierarchy Analysis (CHA) offered by Clang to build a class hierarchy graph including all those classes that have been encountered by the Clang's AST parser. Second, *OctoDroid* exploits the Clang's LibTooling library to apply code transformation. To help the Octopus' analysis we aim to transform virtual function calls into direct calls, applying a super set of the possible targets according to the information provided by the class hierarchy graph.

We evaluate our approach on the latest Android version available at the time of writing (Android Oreo), our results show its effectiveness in discovering new vulnerabilities as well as identifying well known ones published over the last years on the Android bulletin.

Table of contents

List of figures

List of tables

List of Publications

International Journal

1. Costamagna, V., Bergadano, F.: **HOOKDROID: DALVIK DYNAMIC INSTRUMENTATION FOR SECURITY ANALYTICS**. – *International Journal on Information Technologies Security, 8(3).* 2016.

2. Ahmad M., Bergadano F., Costamagna V., Crispo B., Zhauniarovich Y.: **StaDART: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications** – *(to appear in 2018 - IEEE Transactions on Information Forensics Security.)*

3. Bergadano F., Costamagna V., Crispo B., Russello G.: **AppBox: Black-Box Mobile App Management Solution (MAM) For Stock Android** – *(in submission, 2018 - IEEE Transaction on Dependable and Secure Computing)*

International Conference and Workshop

1. Costamagna, V., Zheng, C.: **ARTDroid: A Virtual-Method Hooking Framework on Android ART Runtime**. – *In IMPS@ ESSoS (pp. 20-28)*, 2016, April

2. Costamagna, V., Zheng, C., Huang, H.: **Identifying and Evading Android Sandbox Through Usage-Profile Based Fingerprints**. – *In Proceedings of the First Workshop on Radical and Experiential Security (pp. 17-23).* ACM. 2018, May.

3. Ahmad, M., Costamagna, V., Crispo, B., Bergadano, F.: **TeICC: targeted execution of inter-component communications in Android**. – *In Proceedings of the Symposium on Applied Computing*, (pp. 1747-1752), ACM. 2017,

April

4. Costamagna, V., Crispo, B., Bergadano, F.: **OctoDroid: Discovering Vulnerabilities in Android System Services via Code Property Graphs**. – *(in submission, 2018), Reversing and Offensive-oriented Trends Symposium 2018 (ROOTS), ACM*

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 operating systems (OS), depending by the vendor. In this work we exclusively focus on the Android platform. Since its introduction in 2008, Android has emerged as the leading operating system used for handheld devices. Mobile devices provide 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 functionality (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 (other not-official third-party markets are employed but nor recommended), 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 employs 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 aim to achieve data ex-filtration to third-party servers of personal and sensitive data (i.e., contact list, received sms, call history). Both class of threats may employ clever techniques in order to get installed on the user devices, there have been several evidences of apps that masquerade them-self offering an unsuspicious service (i.e., contact/sms manager, in-app game) while under the hood they do sensitive data ex-filtration to malicious third party servers. As consequence of the constant increasing amount of mobile devices running the Android platform and its expansion over many market segments during the last few years (i.e., automotive, smart TV, wearable devices), several studies have largely focused on security and privacy research fields. 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 anti-virus countering specific well known attacks. Yet, Android privacy issues have been addressed in order to prevent sensitive data leakage which is a major threat for Android ecosystem. Despite that Android apps have its own UID, private files space and specific set of privileges specified within the Android 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 kernel-space and at user-space. 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.

As Android applications are mainly events-driven, the entire Android OS follows the same paradigm, they require a different approach in order to achieve a complete and precise analysis. In fact, there are a variety of either system and user events that provide inputs that applications consume. For instance, applications can register several callbacks to react to those events (i.e., incoming SMS/call), furthermore messages are widely used as mean to share data and provide features for the app running on the Android system. For these reasons, embracing all the layers that provide the communications features represents a challenging task that requires an unified analysis that takes in consideration both layers that provide communications and access to system features (i.e, Java and native layers). Moreover, analyzing Android applications without relying on system code modification, which include altering Android core components code or introducing new component within the core to provide enforcing capabilities, represent a quite challenging task due to the limited OS features that an application could rely on in order to monitor and inspect third-party applications. Applications' analysis become even more cumbersome when they employ mechanisms to enrich functionalities Yet, Android apps rely on user interactions as well as on input from hardware sensors that are embedded within the devices. Moreover, Android offers to developers different logic components in order to interface with the variety of functionalities provide by the OS. Those components allow to easy the development of Android applications and offer different functionalities by communicating with Android system services via the Inter Procedure Communication (IPC) mechanism named Binder, which constitutes 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.

Different threats jeopardize the Android ecosystem, those that aim to target the Android core components (i.e., components that allow to achieve a privilege escalation, including the kernel-land as well) and those that address user applications. As a privilege escalation attack allows the attacker to execute arbitrary code as privileged user they represent a main threat in terms of breakage of the entire security mechanisms, those flaws allowing privilege escalation attacks must be present in Android core services that are maintained by Google itself. 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. In contrast, threats addressing the user sensitive data (i.e., SMS, contact list) as well as data managed by user applications (i.e., instant messaging communications, private emails) have as target third-party applications which run in user-space and were installed by the user. In terms of privacy and security violations, not only spyware applications constitute a risk to the users but even app's security breach constitutes a sensible attack surface. In fact, an application receives a variety of untrusted inputs either via network layer (i.e, multi-media contents, emails, SMS, etc.) or exposing services to other applications running on the user devices as well as offering interfaces to the outside world (i.e., bluetooth, NFC, etc). As consequence, 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. For instance, breaking the security mechanism in place within an user application which is able to retrieve user contacts allows to collect and exfiltrate those sensitive data via malicious applications that were not even allowed to access to such data in the first place.

In addition, Android application developers are not limited to employ only Java programming language, in fact Android offers the option to develop application functionalities entirely as native code (C/C++) 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 restrictions imposed by the JVM.

1.1 Motivation and Problem Statement

The main question we therefore want to address is: given the intricate and complex interaction happening across OS components and applications, what approaches would allow to practically analyze 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 minimum 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) environments?
As orthogonal 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 [40, Api, 140, 53] 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 viable solution. On the other hand, dynamic analysis often requires consistent modifications to the underlying system and/or application [59, 117] which is also a competitive limit in different scenarios, including enterprises that might not be inclined to modify critical components as well as to enable rooting on employees devices [123, 135].

An enterprise actor might be keen to be able to easily 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 on 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 an Android app is 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. Furthermore, 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.

In this dissertation we provide an hybrid analysis to tackle disadvantages of static analysis and take profit of dynamic analysis capabilities. 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 countermeasure to data ex-filtration 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.

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. Smartphones also accept a wide set of touch commands, such as swipe and tap, which is unlike the traditional mouse and keyboard input. This added complexity can complicate analysis, as it is hard to automatically traverse all possible execution 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. 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 [97] [117] [150] . 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 [116] and [37], are generally limited to code execution inside a signal component of the app or do not handle the dynamic code updates well.

In August 2015 Google has started publishing Android Security Advisories containing vulnerabilities that were reported to the vendor by internal or third party researchers, few years later the Android Security Reward program were launched as a monetary (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. Mobile devices are also accessible, and vulnerable, through multiple (sometimes simultaneous) “connections” to the outside world, such as email, WiFi, GPRS, HSCSD, 3G, LTE, Bluetooth, SMS, MMS, and web browsers. They also utilize a complex plethora of technologies such as camera, compass, and accelerometers, which may also be vulnerable, e.g., via drivers. 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 Virtual-Method Hooking Framework on Android ART Runtime

We contribute a dynamic analysis framework for Android platform. It achieves dynamic instrumentation by altering app’s virtual memory in order to insert dynamic hooks. Thanks to this technique, we provide capabilities for monitoring and control ling 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 consequence most of the previous dynamic techniques for the Dalvik virtual machine became ineffective.

The advantage of being able to dynamically instrument Android 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 labeling 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 analysts to confine suspicious code into an artificial environment in order to limit and contain the damage caused by the malware, eventually. Preventing malware execution effects (i.e., deleting/altering file, data ex-filtration, 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. Although, as has been proved by several publications [112, 79, 102, 89], 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 executing 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 defects 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 identifying those code portion 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 demonstrate 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 Ensuring Execution of Targeted Code Paths during Dynamic Analysis

To address the triggering problem, peculiar for dynamic analysis, 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 participate in a specific call-site, that we are interested in analyzing.

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

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 allows 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 entry-point in order to load the generated slice by means of dynamic code loading capabilities and then, to jump directly to slice's entry-point 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 Handling Dynamic Code Updates Using a Combination of Static and Dynamic Analysis

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

- We propose, design and implement StaDART, a system that interleaves static and dynamic analysis in order to reveal the hidden/updated behavior. By utilizing

ArtDroid, we avoid modifications to the Android framework and make it largely frame- work independent. StaDART downloads and makes available for analysis the code loaded dynamically, and is able to resolve the targets of reflective calls complementing app's method call graph with the obtained information. Therefore, StaDART can be used in conjunction with other static analyzers to make their analysis more precise.

- We integrate StaDART with DroidBot to make it fully automated and to ease the evaluation. Moreover, we analyze a dataset of 2,000 real world apps (1,000 benign and 1,000 malicious). Our analysis results show the effectiveness of StaDART in revealing behavior which is otherwise hidden to static analysis tools.
- We plan to release our tool as open-source to drive the research on app analysis in the presence of dynamic code updates.

1.2.4 Black-Box Mobile App Management Solution (MAM) For Stock Android

Beside Android explosion in the consumer market, in the past decade mobile computing has also became a way for employees of organizations 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 maintain apps ready for enteprise-wise capabilities and to offer an easy to deploy MAM solution that allows an enterprise 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 enterprise ready by only changing one single line in the app manifest file, that in combination with our dynamic instrumentation technique allows an enterprise to 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 for customisation purposes.

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 bytecode 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. 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].

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

1.2.5 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 sandboxes.

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 suspicious 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 randomly 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 device allowing thus to detect those execution environment 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 environment where artifacts have fixed values, hence app being analyzed can easily identify those artificial execution environment 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.

1.2.6 Discovering Vulnerabilities in Android C++ code via Code Property Graph Analysis

In this work we propose OctoDroid , a practical analysis tool combining the precise parsing offered by Clang along with the graph-based analysis of Octopus platform. OctoDroid aims to aid the uncovering of vulnerabilities in the Android system service C++ implementation. System services employ as communication mechanism the Binder IPC, which in turn add an opaque layer when it comes to discovering vulnerability. In fact, the Binder IPC mechanism permits to invoke remote procedures as they were local, basically it is a client/server communication via Binder messages, so called *Parcel*. In [141, 143, 142] Yamaguchi et. al. proposed an innovative approach for modeling and discovering vulnerabilities employing Code Property Graphs (CPGs), the implementation named *Octopus* has been released. Octopus aids the analyst discovering vulnerability by query traversal on the CPG stored in a graph database. The difficulty of parsing C++ is well-studied [**?** **?**]. Octopus employs a fuzzy parser based on island grammars [98] which performs analysis on selected portion of the code rather than performing a detailed analysis of a complete source code. The fuzzy parser allows to continue the analysis even in case of parsing issues or in case of missing code portions. As natural consequence we noticed that Octopus presents very low detection rate when it comes to analyzing C++ code base, mainly as consequence of its parser's design which sacrifices level of details in favor of tolerance.

OctoDroid compensates the Octopus' fuzzy parser inherent incompleteness in analyzing C++ code. OctoDroid allows to automatic modeling and discovering vulnerability in Android system services C++ codebase. Differently from existing approaches based on fuzzing, we focus on static exploration via CPGs which allows to traverse the produced graphs in order to query for specific pattern that may lead to vulnerabilities. We propose an uncouple design which via Clang's LibTooling Class Hierarchy Analysis (CHA) first builds the graph, then exploit collected information to enhance the static analysis employed by Octopus platform analysis. We build the Class Hierarchy graph by means of Clang's LibTooling library. Then, we use LibTooling again to enhance the code with the information collected in the previous stage. As result, the augmented code would contain virtual function calls replaced with their corrispective explicit form(s), whenever it is possible. The augmented code can now be processed by the Octopus's fuzzy parser and then analyzed leveraging on the type information which has been written explicitly in the augmented code. It is worth noting that OctoDroid on one hand benefits from the clang's Libtooling capabilities and on the other hand exploits the Octopus's fuzzy parser and analysis platform to perform query on the augmented code.

We concretely show how OctoDroid can easily detect various already known security bugs in Android system services and we demonstrate how OctoDroid minimize the manual effort requested . We further show OctoDroid effectiveness by considering recent uncovered vulnerabilities in native code which lead to memory corruption in system services process.

To summarize, this work makes the following contributions:

- To our knowledge, OctoDroid is the first CPG-based tool that aims to enhance Octopus analysis capabilities specifically for automatic modeling and discovering of vulnerability in C++ code, in particular we targeted the Android system services codebase.

- Unlike previous existing works, OctoDroid provides effective and practical analysis of Android system services codebase by means of Code Property Graph representation. OctoDroid takes the advantage of Clang's full parsing approach to enhance the CPG by Class Hierarchy Analysis (CHA).

- We have implemented and evaluated OctoDroid on the latest Android Orio codebase available at the time of writing. Our results show its effectiveness in discovering new vulnerabilities.

- OctoDroid is opensource, available at the following URL[3]

Chapter 2

Background

This chapter explores the evolution of Android and introduces its architecture. Although Android is built on top of Linux kernel, it has become an operating system in a class by itself. Android introduces a vast collection of frameworks, as well as a runtime (Dalvik/ART) to support them. We then turn to examine the Android architecture, each layer is described in detail to set the foundation for the deeper exploration carried out in the next chapters of this work. Then, we consider and discuss static and dynamic analysis approaches and how they adapt for the Android platform. In addition we consider a practical use case showing main static analysis limitations when it comes to Android applications.

2.1 Android

Android is a modern operating system with a layered software stack, the following Figure 2.1 illustrates its layers. The Android's software stack can run on many different hardware configurations.

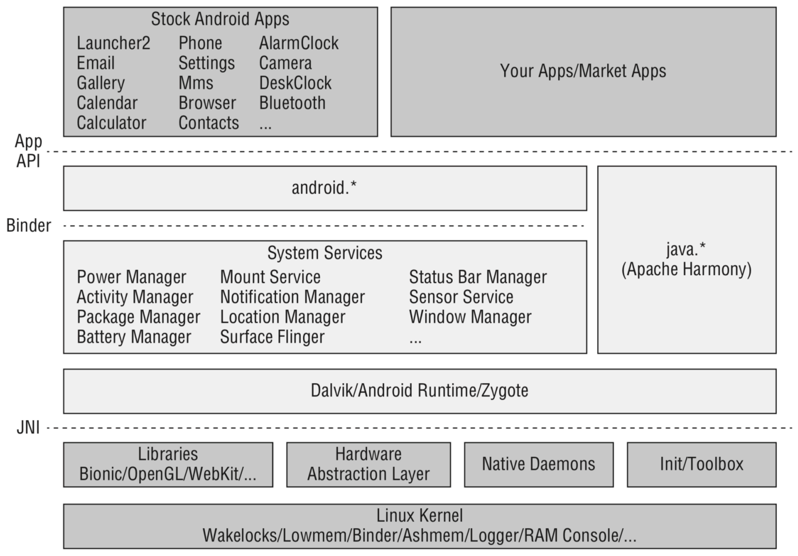

Stock Android Apps
Launcher2 Phone AlarmClock
Email Settings Camera
Gallery Mms DeskClock
Calendar Browser Bluetooth
Calculator Contacts ...
Your Apps/Market Apps
App API
android.*
Binder
System Services
Power Manager Mount Service Status Bar Manager
Activity Manager Notification Manager Sensor Service
Package Manager Location Manager Window Manager
Battery Manager Surface Flinger ...
java.* (Apache Harmony)
Dalvik/Android Runtime/Zygote
JNI
Libraries Bionic/OpenGL/WebKit/...
Hardware Abstraction Layer
Native Daemons
Init/Toolbox
Linux Kernel
Wakelocks/Lowmem/Binder/Ashmem/Logger/RAM Console/...

Fig. 2.1 Android System Architecture

Most of the user-facing features and enhancements in between versions have to do with additional frameworks and APIs being added, with only a relatively small portion of them at the system level. At the time of writing the latest Android version is Oreo 8.1, API number 27.

Android applications allow developers to extend and improve the functionality of a device without having to alter lower levels. In turn, the Android Framework provides developers with a rich API that has access to all of the various facilities an Android device has to offer (see 2.1.1). This includes building blocks to enable developers to perform common tasks such as managing user interface (UI) elements, accessing shared data stores, and passing messages between application components.

The Android OS is based on a Linux kernel but offers a different application abstraction than found in traditional Linux distributions. Android apps are mostly written in Java and compiled into Dalvik bytecode to be executed by the Dalvik Virtual Machine (DVM). Apps may optionally contain native code components. Newer versions of Android [1] employ the Android Run Time (ART) that converts bytecode to native code at install time (Ahead Of Time compilation). Android apps are distributed as an APK file that basically is a ZIP file containing the app's bytecode (in *classes.dex*) and its resources.

[1]https://source.android.com/devices/tech/dalvik/

The Android operating system utilizes two separate, but cooperating, permissions models. At the low level, the Linux kernel enforces permissions using users and groups. This permissions model is inherited from Linux and enforces access to le system entries, as well as other Android specific resources. This is commonly referred to as Android's sandbox (see 2.1.4).

Although each app executes within a dedicated sandbox, Android allows apps to communicate with each other through a well-defined Inter-Process Communication (IPC) mechanism, referred to as Binder (see 2.1.2). It provides message passing (called *parcels*) taking care of migrating the execution of a request from the requester to the target process transparently to the apps. The Binder system includes a kernel module, accessed through the `/dev/binder` file. Communications between different components in the same app are handled by the Binder.

2.1.1 Framework

The glue between apps and the runtime, the Android Framework provides the pieces for developers to perform common tasks and building Android applications. The common framework packages are those within the *android.** namespace, such as *android.content* or *android.telephony*. Android also provides many standard Java classes (in the *java.** and *javax.** namespaces), as well as additional third-party packages. The Android Framework also includes the services used to manage and facilitate much of the functionality provided by the classes within.

Even though almost all Android OS functionality above the kernel level is implemented as system services, it is not exposed directly in the framework but is accessed via facade classes called managers. Typically, each manager is backed by a corresponding system service; for example, the `BluetoothManager` is a facade for the `BluetoothManagerService`. System services implement most of the fundamental Android features, including display and touch screen support, telephony, and network connectivity and they are written in native code (C/C++).

With a few exceptions, each system service defines a remote interface that can be called from other services and applications. Coupled with the service discovery, mediation, and IPC provided by Binder, system services effectively implement an object-oriented OS on top of Linux.

2.1.2 Binder

While Android's Binder is a new implementation, it's based on the architecture and ideas of OpenBinder. The Binder driver is the central object of the framework, and all IPC calls go through it. Inter-process communication is implemented with a single `ioctl()` call that both sends and receives data through the *binder_write_read* structure.

Binder acts as a mediation point for all IPC. Access to system resources (e.g., GPS receivers, text messaging, phone services, and the Internet), data (e.g., address books, email) and IPC is governed by permissions assigned at install time. The permissions requested by the application and the permissions required to access the application's interfaces/data are defined in its manifest file.

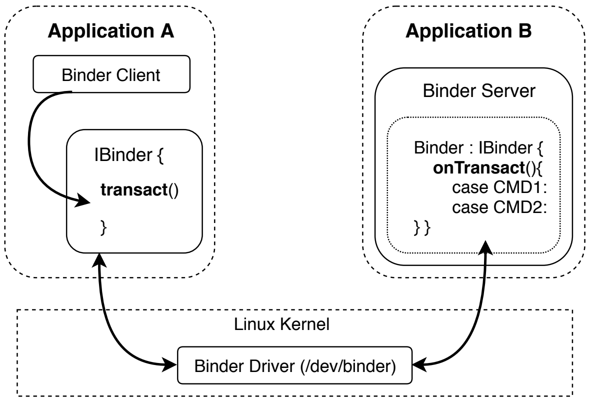


Fig. 2.2 Android Binder Mechanism

As shown in Figure 2.3, when a process sends a message to another process, the kernel allocates some space in the destination process's memory, and copies the message data directly from the sending process. It then queues a short message to the receiving process telling it where the received message is. The recipient can then access that message directly (because it is in its own memory space). When a process is finished with the message, it notifies the Binder driver to mark the memory as free.

Higher-level IPC abstractions in Android such as `Intents` (commands with associated data that are delivered to components across processes), `Messengers` (objects that enable message-based communication across processes), and `ContentProviders` (components that expose a cross-process data management interface) are built on top of Binder. Additionally, service interfaces that need to be exposed to other processes can be de ned using the `Android Interface Definition Language` (AIDL), which enables clients to call remote services as if they were local Java objects. The associated aidl tool automatically generates stubs (client-side representations of the remote object) and proxies that map interface methods to the lower-level transact() Binder method and take care of converting parameters to a format that Binder can transmit (this is called parameter marshalling/unmarshalling). Because Binder is inherently typeless, AIDL-generated stubs and proxies also provide type safety by including the target interface name in each Binder transaction (in the proxy) and validating it in the stub.

2.1.3 Application

While all apps have the same structure and are built on top of the Android framework, we distinguish between system apps and user-installed apps. System apps are included in the OS image, which is read-only on production devices (typically mounted as /system), and cannot be uninstalled or changed by users. Therefore, these apps are considered secure and are given many more privileges than user-installed apps. System apps can be part of the core Android OS or can simply be pre-installed user applications, such as email clients or browsers. User-installed apps are installed on a dedicated read-write partition (typically mounted as /data) that hosts user data and can be uninstalled at will. Each application lives in a dedicated security sandbox and typically cannot affect other applications or access their data. Additionally, apps can only access resources that they have explicitly been granted a permission to use. Privilege separation and the principle of least privilege are central to Android's security model, and we will explore how they are implemented in 2.1.4.

Android apps are organised in components[2] that permit to achieve different functionalities. Android offers four types of components: Activity, Content Provider, Service and Broadcast Receiver. The app's user interface is composed of a sets of activity components. Content provider components offer per-application data servers that are queried by other installed apps. Service components are intended for background processing. Broadcast receiver components handle asynchronous messages across apps as well as Android system. An Android Intent[3] is used as a messaging object to request an action from another app

[2]https://developer.android.com/guide/components/index.html
[3]https://developer.android.com/guide/components/intents-filters.html

component. Although intents facilitate communication between components in several ways, there are three fundamental use cases: starting an activity, a service or delivering a broadcast message.

Android apps require to define a special file called `AndroidManifest.xml`, known as manifest[man], that contains specific meaningful information about the related Android app. Every app must have a manifest file in its root directory because the Android system needs to access its content before it can run any of the app's code. As a consequence, the manifest file cannot be obfuscated. The information declared in the manifest cannot be changed at runtime: even dynamically loaded code must comply with the permissions and the components defined in the app's manifest.

Android `sharedUserId` attribute. The manifest attribute `android:sharedUserId`, introduced since the first Android version, is a feature that allows to execute different apps under the same UID if and only if they are signed with the same certificate. Once installed, apps that share the same UID will have access to each other private data because they share the same Linux permissions set. This feature is extensively used by Android for core framework services and system apps. For instance, the Play Service and the Google location service use the `android:sharedUserId` to request to run in the same process of the login service to be able to sync data in the background, without user interaction. This feature is available to and widely used also by third-party developers to update their apps and shared libraries. Removing such features will have an important impact on backward compatibility.

Android `process` attribute. The attribute `android:process` allows to execute two different apps within the same process space. It can be specified for any components in an app. Whenever the execution of a component is requested, Android first looks for a running process matching the name specified in `android:process`. If a process is found, then that process will be used to execute the requested component. This avoids spawning a new process for a component if there is already a running instance of that component. For instance, this is used to reuse the background activity's process when it is called in foreground again.

Java Native Interface. Android Application allows the inclusion of native libraries (ELF shared objects) in application code, through the Java Native Interface (JNI). From the Linux perspective, all executables are ELF binaries. It is therefore not at all uncommon to see JNI used in applications optimizing for performance, or seeking resistance to reverse engineering. Google therefore provides the Native Development Kit (NDK) (Android Developer), which developers can use to build native libraries (and binaries). Android's critical system component are implemented in C/C++, and are compiled into native binaries. User applications are compiled into Dalvik bytecode, but the bytecode runs (or, in ART, is compiled ahead-of- time) in the context of a Dalvik Virtual machine, which is, in and of itself, an ELF binary. Thus, while most developers remain oblivious to binaries, they nonetheless play an important role in Android.

2.1.4 Security Model

Android provides a sandbox for each installed app, as shown in Figure 2.3. To enforce this isolation at the Linux kernel, Android assigns at install time a unique User ID (UID) to each app. Moreover, since Android version 4.3, SELinux was adopted with its Mandatory Access Control (MAC) model in order to enforce a more fine-grained UID-based isolation and to harden the OS components mitigating the risk of flawed and malicious code. In addition, Android combines the traditional Linux permissions with a Mandatory Access Control (MAC) mechanism at framework level. During install time, apps are assigned permission labels representing the resources they can access during runtime. The developer of an app must declare the permissions the app requires in its manifest file. SELinux can operate in one of two global modes: permissive mode, in which permission denials are logged but not enforced, and enforcing mode, in which denials are both logged and enforced. SELinux also supports a per-domain permissive mode in which specific domains (processes) can be made permissive while placing the rest of the system in global enforc- ing mode. In the Android 5.0 L release, Android moves to full enforcement of SELinux. This builds upon the permissive release of 4.3 and the partial enforcement of 4.4. In short, Android is shifting from enforcement on a limited set of crucial domains (installd, netd, vold and zygote) to everything.

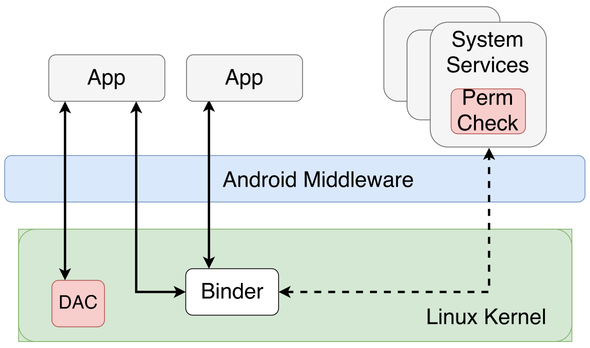


Fig. 2.3 Android Security model

Android requires all APKs to be digitally signed with a certificate. A public-key certificate contains the public key of a public/private key pair, as well as some other metadata identifying the owner of the key. The owner of the certificate holds the corresponding private key. When a developer signs an APK, the signing tool attaches the public-key certificate to the produced APK. The public-key certificate serves as a fingerprint that uniquely associates the APK to its developer and his corresponding private key. This helps Android to ensure that any future updates to that APK come from the original developer. In fact, developers must use the same certificate throughout the lifespan of their apps to push new versions of their apps to the users' devices. In Android, a certificate authority is not mandatory: typically the app certificates do not need to be signed by a certificate authority and most developers use self-signed certificates.

2.1.5 Overview of Dynamic Code Update

Dynamic code updates techniques, such as reflection and dynamic class loading, are used to extend apps' functionality at runtime. Inherited from Java into the Dalivk Virtual Machine (DVM), these features are equally supported by Dalvik's successor Android Runtime (ART). Android uses ART to run apps and system services which uses ahead of time (AOT) compilation using a dex2oat tool to convert DEX files into `.oat` binaries. ART is backward compatible with Dalvik runtime and can execute apps compiled for the DVM.

A growing number of malware samples found in the Android ecosystem reveals that malware developers bypass such vetting processes using various evasion techniques. Previous research shows that the use of dynamic code update techniques along with various forms of obfuscation makes it extremely hard for the state-of-the-art analysis tools to understand the behavior of an app [113, 32].

At the same time, previous approaches that enhanced static analyzers of Java code in the presence of dynamic code update techniques (e.g., [45]) cannot be directly applied to Android due to the differences in the platforms (in Android, load-time instrumentation of classes is not available). Moreover, offline instrumentation also cannot solve the problem because this approach breaks the application signature, while some apps check it at runtime. If the signature does not correspond to some hardcoded value they may refuse to work. In case of malicious apps this check may be used to conceal illicit behavior. Follows an overview on each technique for code updating.

Dynamic Class Loading

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

Android provides a set of class loaders, arranged in a hierarchical manner, which are used to load classes to memory from internal storage. Every child class loader holds a reference to its parent class loader where the root of the tree is the BootStrap ClassLoader, which has a null reference to its parent. A common interface required by all the class loaders is implemented by an `abstract` class named ClassLoader whereas other specific class loaders are then derived from ClassLoader, such as *DexClassLoader*, *PathClassLoader*, etc. ClassLoader provides methods such as *loadClass()*, *findLoadedCalss()* and *defineClass()*, which allows a developer to load a class, search for loaded classes and define a class from a byte sequence at runtime, respectively. Android also provides a class *DexFile* whose methods can be used to load classes directly. However, these methods require a reference to a class loader as an argument.

DCL is usually used for the following purposes:

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

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

Common Frameworks: Depending upon functionality, apps might use certain common frameworks, e.g., an advertisement framework, which shows advertisements to the user. Common frameworks are installed as separate apps whose code can be loaded dynamically by the reliant apps when needed. In the absence of DCL, the functionality provided by the framework must have been implemented in each of the reliant apps. Similarly, in the case of updating that common functionality provided by the framework, only the framework needs to updated rather than updating all the reliant apps.

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

Java Reflection

Reflection is the ability of a program to treat its own code as data and manipulate it during execution [44]. Using reflection, an app can reason about and modify its execution state during runtime. The dynamically loaded code is usually accessed using reflection. Android uses the same reflection APIs as used in Java.

In the following, the functionality provided by reflection APIs is outlined:

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

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

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

In the following, we provide an overview of what reflection offers to a developer[139]:

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

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

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

In general, we can conclude that dynamic code loading and reflection are both highly useful and essential for apps, specifically Android apps. Thus, the use of these evasion techniques in newly found malware is not surprising [Polkovnichenko and Boxiner].

2.2 Static Analysis

In this section we introduce static analysis approaches for analyzing Android apps, we discuss in brief main advantages and limitations that belong to each approach. A complete and accurate evaluation of static analysis techniques is out of the scope of this dissertation.

Static analysis examines a program without executing any code. Two Android apps essential components are (I) the *Android-Manifest.xml* which describes permissions, defines external libraries and app components (e.g., activity, service) and (II) the *classes.dex* file that contains the app bytecode as DEX format. It is worth of note that the app's manifest must be present in clear text, thus even highly obfuscated apps show this file in plain text as the Android system leverage on this file in order to read and parse app's components information.
Static analysis of Android apps mainly involve different representations when it comes to bytecode analysis. The DEX files are often decompiled first into a more comprehensible format. There are many level of formats, from low level bytecode, to assembly code, to human-readable source code. Both PScout [36] and AppSealer [147] use Soot directly on the dex to acquire Java bytecode, while Enck et. al. in [58] uses ded/DARE. Alternatively, [61] decompiles dex into an assembly-like code with dedexer, while others choose to study Dalvik bytecode [82, 68, 147], smali [74, 151], or the source code [65, 55]. Android apps can also be decompiled and reassembled either inserting or altering the bytecode [40], operation that breaks the app's signature.

To be able to effectively predict the app's control flow, static analyses must not only model app's basic interaction with the Android framework (e.g., activity lifecyle), but must also integrate further callbacks for system-event handling (e.g., for phone sensors like GPS), UI interaction, and others. Several approaches have been published to address these challenges with a diversity range of sensitivity and precision [35, 92, 108] and a variety of investigations show how those approaches in turn fit properly according to the context. As proposed in [136], an approach to conduct static analysis for security vetting of Android apps as well as ScanDal [82] and others [84, 65] addressing privacy leak detection. Yet, ScanDroid [63] proposed an automated security certification of Android applications which relies on WALA for implementing data flow analysis on Java code and Chex [92] presented a static analysis method to automatically vet Android apps for component hijacking vulnerabilities, that allow to gain unauthorized access to protected or private resources through exported components in vulnerable apps.

In [86] Li et. al. have provided a systematic literature review on static analysis of Android apps, they analyzed in deep several approaches and tools enumerating keys aspect of each of them. In their comprehensive study they identified six fundamental categories including taint analysis [35, SuS, 147], abstract interpretation [110, 82], symbolic execution [145], program slicing [74, 32], code instrumentation [84, 134] and model checking [91, 60].

Particularly for Android, analyzing ICC/IPC is essential for understanding and detecting stealth behaviors [77] and leaked information [84] as its IPC Binder protocol is unique, a key part of the Android system, and much more powerful and complex than most other IPC protocols. In one static study, Epicc [108] created and analyzed a control-flow super graph to detect ICC information leaks. While Epicc relied on Soot for majority of its needs, Amandroid used a modified version of dexdump (i.e., dex2IR) to study inter-component data flows [136]. Furthermore, while Epicc built control flow graphs, Amandroid built data dependence graphs from each app's ICC data flow graph.

Although static analysis could potentially reveal all possible paths of execution, there are several limitations. Both benign and malicious developers use various protection techniques, such as Java reflection, dynamic code loading and code obfuscation [118], to prevent their apps from both reverse-engineering and repackaging. [124, 81, 154, 119]. All static methods are vulnerable to obfuscations (e.g., encryption) that remove, or limit, access to the code. Similarly, 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. While some static techniques are resilient to obfuscations, each technique is vulnerable to a specific obfuscation method. For example, feature based analysis is generally vulnerable against data obfuscation and, depending on its construction, structural analysis is vulnerable to layout, data, and control obfuscation [127].These limitations have long been the downfall of static analysis frameworks for traditional analyses [99] and mobile malware analysis [118, 74].

The analysis of Android apps becomes more and more difficult currently, the dynamic analysis approach is usually coupled with the static approach for deeply analyzing apps. Unless also a hybrid solution, no static framework can fully analyze Android applications using full bytecode encryption or dynamic code update.

2.3 Bypassing Static Analysis via Dynamic Code Update

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

2.3.1 Reflection-Bench

The usefulness of reflection in Android apps development is undoubted. However, reflection's inherent property to hinder static analysis of apps makes it attractive for malware developers. Although, researchers have worked on app analysis in the presence of reflection in Android apps, literature and the research community still lacks a benchmark of apps which could be used as a test suite to determine the effectiveness of app analysis tools in the presence of reflection. We present *reflection-bench*, a set of Android apps, which use reflection to conceal information leakage to make detection harder for static analyzers. Reflection-bench is designed so that it can be used to test tools which perform taint analysis as well as those that only generate call graphs for other forms of static analysis.

Overview: Reflection-bench consists of 14 apps which use reflection in various forms to conceal information leakage and make the flow of the program ambiguous. The hardness of resolving the targets of reflection depends upon the nature of the arguments used in the reflection APIs. We divide them into two classes, i.e., statically available arguments (those string arguments which are provided as part of the app package, e.g., strings defined inside the program, read from a file which is part of the app, etc.) and statically unavailable arguments (those received over the network, read from files on disk, received from other apps, etc.).

Statically unavailable arguments can make it impossible for static analysis tools to resolve reflection. In reflection-bench, we only consider the case of statically available arguments. However, with each case the complexity is gradually increased. In the first few cases, the arguments of reflection APIs are constant strings assigned to program variables. In the latter cases, we consider reading the arguments from a properties file (part of the APK file) and from a hashtable defined inside the program. Moreover, we also consider the cases where the string arguments are formed from the concatenation of multiple strings or decrypted from encrypted strings using crypto APIs. In addition, we consider two levels of complexity where in level one, reflection is used to call only the methods defined inside the app and in level two, both the methods defined inside the program as well as the sensitive APIs, which are responsible for leaking sensitive information, are called through reflection.

Implementation: There are two major classes in each app, i.e, BaseClass and MainActivity. BaseClass has two methods, where `Method1` gets the device ID using the `getDeviceID` API and stores it in a local field `Str`. `Method2` gets a string and sends it out using the `sendTextMessage` API. MainActivity calls `Method1` of BaseClass, gets its field `Str` and sends it to the `Method2` of BaseClass which leaks it out. In the following, we describe how different combinations of reflection APIs are used in each case.

① MainActivity retrieves the field `Str` of BaseClass using `getField` reflection API.

② MainActivity retrieves an instance of BaseClass using the reflection API `forName`, creates its object using the `newInstance` API and gets its field `Str` using the `getField` reflection API.

③ MainActivity retrieves an instance of BaseClass using the reflection API `forName`, gets its Constructor using the `getConstructor` API, creates its object using the `newInstance` API and gets its field `Str` using the `getField` reflection API.

④ MainActivity retrieves an instance of BaseClass using the reflection API `forName`, creates its object using the `newInstance` API and gets its field `Str` using the `getField` reflection API. It also retrieves the methods of

BaseClass using the `getMethod` reflection API and calls them using the `invoke` reflection API.

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

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

(6) Reads the names of BaseClass, its methods and its field from a file.

(7) Reads the names of BaseClass, its methods and its field from a Hashtable.

(8) Constructs the names of BaseClass, its methods and its field from multiple strings in the program.

(9) Decrypts the encrypted names of BaseClass, its methods and its field using Crypto APIs.

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

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

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

(11) Reads the names of TelephonyManager class, methods `getDeviceId` and `sendTextMessage` from a file.

(12) Reads the names of TelephonyManager class, methods `getDeviceId` and `sendTextMessage` from a Hashtable.

Table 2.1 Analysis with State-of-the-art tools

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

(13) Constructs the names of TelephonyManager class, methods `getDeviceId` and `sendTextMessage` from multiple strings inside the app.

(14) Decrypts the encrypted names of TelephonyManager class, methods `getDeviceId` and `sendTextMessage` using Crypto APIs.

Tools analysis results: We report the results of analysis on recent state-of-the-art tools, e.g., Flowdroid [35], Androguard [3], Amandroid [136], SAAF [74], SCandroid [63] and IccTa [84]. A summary of the results is provided in Table 2.1. Those tools which perform taint analysis, such as Amandroid, etc., are analyzed by performing taint analysis of the apps in reflection-bench. However, for those tools which do not perform taint analysis, such as Androguard, etc., we analyze them by generating call graphs of the apps using these tools. In Table 2.1, a ✓ in column X, indicates that the app is successfully analyzed by tool **X**, whereas, a ✗ indicates otherwise.

These analysis results show that reflection makes static analysis of apps harder. Specially, when the parameters of reflection APIs are not readily available in the code, static analysis tools find it extremely hard to properly analyze apps.

2.3.2 InboxArchiver: Test Malware using DCL

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

Overview: InboxArchiver is a simple app that reads the SMS inbox and sends some statistics to a number provided by the user. These statistics include the number of SMS messages sent to and received from certain numbers. A user can configure InboxArchiver to receive a daily, weekly or monthly SMS message containing these statistics. The malicious part of the app, however, downloads some additional code from the Internet which contains other numbers potentially owned by an adversary, loads this code using the DCL APIs and leaks these SMS inbox statistics.

Implementation: The main features of InboxArchiver are the use of DCL and reflection having encrypted strings representing the code paths, class names and method names. This helps InboxArchiver to evade static analysis tools. In order to evade dynamic analysis, it makes use of a simple delay technique where again the APIs are called using reflection with encrypted parameters. It waits for 10 minutes before downloading the malicious code from the Internet and loading it using DCL. Although there are other more sophisticated anti-analysis techniques available, such as emulator detection, root detection, etc., the use of just a delay technique in InboxArchiver highlights the role of DCL/reflection in evading analysis tools.

InboxArchiver consists of three main classes, i.e., a `MainActivity` class, a `MessageSender` class and a `Loader` class. The `MainActivity` class presents an interface to the user as shown in Figure 2.4. The `MessageSender` class, which is a Service and runs in the background, is responsible for retrieving the

Table 2.2 InboxArchiver: Analysis Results

Analysis System	**Analyzed**	**Obfuscation**	**DCL**	**Malware**
VirusTotal [130]	✓	✗	✗	✗
UnDroid [und]	✓	✓	✗	✗
AndroTotal [and]	✓	✗	✗	✗
ds-andrototal [ds-]	✓	✗	✗	✗
MobiSec Lab [mob]	✓	✗	✗	✗
CopperDroid [128]	Queued	-	-	-
SandDroid [129]	✓	✓	✓	✗

inbox statistics and sending it periodically to a pre-configured number. After a certain delay, the `MessageSender` class instantiates an object of the `Loader` class which handles the downloading of additional code from the Internet and dynamically loading it using DCL APIs. It makes use of encrypted parameters and encryption/decryption functionality provided by other auxiliary classes.

Analysis results: We uploaded InboxArchiver to a number of online Android app analysis systems. Table 2.2 shows a summary of the obtained results. Column *Analyzed* shows whether the app is properly analyzed or not. The next two columns, *Obfuscation* and *DCL*, show if the analysis systems detect obfuscation and the use of dynamic code loading, respectively. The last column in the table represents the final remarks about the app.

Among the online analysis tools shown in Table 2.2, we did not receive any results from CopperDroid and the app is still in the queue for more than a year now. All other tools were unable to detect that the submitted app is malicious. VirusTotal scanned the app with 54 antivirus tools, including BitDefender, GData, AVG, Avast and Kaspersky, etc., and none of them labeled it suspicious. UnDroid and SandDroid termed the app as obfuscated, while SandDroid could also detect dynamic code loading in the app. However, it could not detect the loaded file and analyze it.

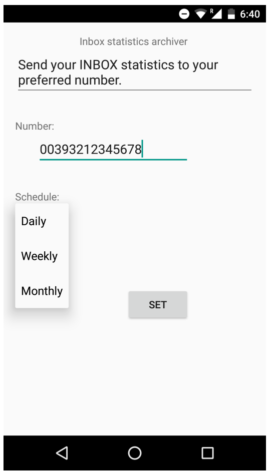

6:40
Inbox statistics archiver
Send your INBOX statistics to your preferred number.
Number:
00393212345678
Schedule:
Daily
Weekly
Monthly
SET

Fig. 2.4 InboxArchiver

2.4 Dynamic analysis

In this section we turn to describe and discuss dynamic analysis approaches for analyzing Android applications. We succinctly present how dynamic analysis provides to overcome some limitations of static analysis and discuss its different applications for the Android platform.

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, e.g., the parameters used in the dynamic code update APIs. In contrast to static analysis, dynamic analysis executes a program and observes the results. Various dynamic analysis approaches have been proposed for monitoring apps behavior at runtime, they can be organized in two main categories: those that modify the Android platform [123, 59, 128, 149, 121, 72, 122, 48, 49] and those that operate at the application layer [40, 116, 56, 140, 38, 42]. Among the former belongs approaches altering the Android Framework, instrumenting the APIs or other core components, as well as approaches that operate at the kernel level. Instead, the latter category includes any approaches that operate at user-space by means of app's code instrumentation both statically or dynamically as well as approaches that offer app virtualization [38] or dynamic sandboxing via ptrace capabilities [42] to cite a few.

Applying considerable modifications to the Android platform, TaintDroid [59] was among the first dynamic analysis tools allowing to track propagation of information. Sources of sensitive information are typically the device sensors or private user information, and sinks are network interfaces; thus the main scope of TaintDroid is detection of privacy leaks. This approach is followed by DroidScope [144]. DroidScope allows to emulate app execution and trace the context at different levels of the Android software stack: at the native code level, at the Dalvik bytecode level, at the system API level, and at the combination of both native and Dalvik levels. Other approaches operate by requiring only little modifications to the Android platform in order to acquire the root privilege [123] or altering the ART compiler [39, 126] for monitoring target apps. Yet, emulator and the ptrace capabilities are exploited to trace runtime execution [152, 128]. Those approaches offer remarkable security guarantees as strong boundaries are defined by the monitoring system, but on the other hand they might not be suitable for several scenarios where strong modifications to the Android platform or the root privilege would be avoided as they are hard to maintain and might participate to increase the attack surface as overall.

A different line of investigations have proposed to alter the app's code itself in order to provide dynamic analysis capabilities. The technique for inserting extra code into an application to observe its behavior is called *instrumentation* and is becoming the common way for dynamic analysis of android applications. Aurasium [140] presented the first app-based static instrumentation approach allowing to monitor and alter app's behavior at runtime without any modifications to the Android platform. This approach based on altering app's files has been applied in several successive works as to insert Inline Reference Monitor (IRM) within the app's bytecode [53, 40], or modifying the bytecode to extract runtime values as in [116] as well as sandboxing the application at both Java and native layers [155]. One of the main and remarkable disadvantage of these approaches is that altering the target app's bytecode, or any of its files, changes also its signature. Moreover, any static instrumentation based approach present the same limitations as for static analysis, in fact those approaches need first to locate the interesting code (e.g., call-site to Android APIs) in order to then apply the instrumentation (i.e., IRM), thus they suffer from dynamic code update as well as code obfuscation. To overcome this restrictions, dynamic instrumentation approaches allow to place hooks dynamically. Dynamic instrumentation is a very well known approach [93, 104, 47] for tracing and altering code behavior. Instead of rewriting interesting call-site locations within the app's bytecode, as app's bytecode static instrumentation does, the dynamic instrumentation approach on the Android platform shows two main techniques: (I) leveraging on the root privilege to interpose hooks via direct manipulation of target app's process space and (II) inserting a lightweight stub code within the target app that would dynamically instrument its virtual memory setting dynamic hooks in place. It is worth noting that the latter technique does not alter any bit of app's bytecode, but still it shares a remarkable limitation common to static instrumentation approaches for the reason that it breaks the app's signature. A completely different approach based on app virtualization has been presented by Backes et. al. in [38], it relies on the *isolated_process* Android feature and provides a Broker core component that manages instances of virtualized apps bridging communications back and forward to the Android platform. As enterprise-wise solution, Wang et. al. proposed a dynamic security policy enforcement scheme [135] via dynamic memory instrumentation.

Dynamic analyses leads to under-approximates, as it is challenging to cover all code, and thus tend to produce false negatives. Code coverage is a well-known limitation of dynamic analysis approaches. However, it can be improved with stimulation. As Android apps are highly interactive, many behaviors need to be triggered via the interface, received intents, or with smart, automatic event injectors [66, 94, 95, 88, 117, 150]. Furthermore, 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.

Chapter 3

ARTDroid: A Virtual-Method Hooking Framework on Android ART Runtime

Various static and dynamic analysis techniques are developed to detect and analyze Android malware. Some advanced Android malware can use Java reflection and JNI mechanisms to conceal their malicious behaviors for static analysis. Furthermore, for dynamic analysis, emulator detection and integrity self-checking are used by Android malware to bypass all recent Android sandboxes. In this chapter we present **ARTDroid**, an framework for hooking virtual-method calls under the latest Android runtime (ART). A virtual-method is called by the ART runtime using a dispatch table (vtable). ARTDroid can tamper the vtable without any modifications to both Android framework and app's code. The ARTDroid hooking framework can be used to build an efficient sandbox on real devices and monitor sensitive methods called in both Java reflection and JNI ways.

3.1 Introduction

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 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 demonstrate 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 chapter is organized as follows. Section 3.2 introduces the background about Android and the new Android runtime ART. The ARTDrod framework is introduced in Sec. 3.3 and its implementation is discussed in Sec. 3.4. Performance evaluation is presented in Sec. 3.5, and discussion and applications are in Sec. 3.6. Section 3.7 discuss some related works, and we conclude this chapter in Sec. 3.8.

3.2 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

3.2.1 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.

```
 1 // C++ mirror of java.lang.Class
 2 class MANAGED Class FINAL : public Object {
 3
 4   [...]
 5   HeapReference<IfTable> iftable_;
 6   HeapReference<String> name_;
 7   HeapReference<Class> super_class_;
 8   HeapReference<PointerArray> vtable_;
 9   uint32_t access_flags_;
10   uint64_t direct_methods_;
11   uint64_t virtual_methods_;
12   uint32_t num_virtual_methods_;
13   [...]
14 }
```

Fig. 3.1 ART Class type

The ART runtime uses specific C++ classes to mirror Java classes and methods. Java classes are internally mirrored by using **Class**[2]. In Figure 3.1, 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

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

work. The Figure 3.2 shows the definition of **ArtMethod** class[3]. The main functionality of `ArtMethod` class is to represent a Java method.

```
1  class ArtMethod FINAL {
2    [...]
3    GcRoot<mirror::Class> declaring_class_;
4    uint32_t access_flags_;
5    uint32_t method_index_;
6    [...]
7    struct PACKED(4) PtrSizedFields {
8        void* entry_point_from_interpreter_;
9        void* entry_point_from_jni_;
10       void* entry_point_from_quick_compiled_code_;
11     } ptr_sized_fields_;
12 }
```

Fig. 3.2 ART ArtMethod type

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.

3.2.2 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. Figure 3.3 shows that the getDeviceId method is invoked on TelephonyManager object's class (line 4). Figure 3.4 shows dumped compiled codes for arm architecture by *oatdump* tool.

```
1 package org.sid.example;
2 public class MyClass {
3   public String callGetDeviceId(TelephonyManager tm){
4     String imei = tm.getDeviceId();
5     return imei;
6   }
7 }
```

Fig. 3.3 Call to method getDeviceId

[3]art/runtime/art_method.h

```
1  CODE: (code_offset=0x002d93b5 size_offset=0x002d93b0 size=60)...
2        0x002d93b4: f5bd5c00  subs    r12, sp, #8192
3        0x002d93b8: f8dcc000  ldr.w   r12, [r12, #0]
4        suspend point dex PC: 0x0000
5        GC map objects:  v1 ([sp + #36]), v2 (r6)
6        0x002d93bc: e92d40e0  push    {r5, r6, r7, lr}
7        0x002d93c0: b084      sub     sp, sp, #16
8        0x002d93c2: 1c07      mov     r7, r0
9        0x002d93c4: 9000      str     r0, [sp, #0]
10       0x002d93c6: 9109      str     r1, [sp, #36]
11       0x002d93c8: 1c16      mov     r6, r2
12       0x002d93ca: 1c31      mov     r1, r6
13       0x002d93cc: 6808      ldr     r0, [r1, #0]
14       suspend point dex PC: 0x0000
15       GC map objects:  v1 ([sp + #36]), v2 (r6)
16       0x002d93ce: f8d00234  ldr.w   r0, [r0, #564]
17       0x002d93d2: f8d0e02c  ldr.w   lr, [r0, #44]
18       0x002d93d6: 47f0      blx     lr
```

Fig. 3.4 Compiled native code of callGetDeviceId

Before discussions on native code, in Fig. 3.4, 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 3.3 is compiled in lines 11-18 in Figure 3.4. 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.

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

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.

3.3 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

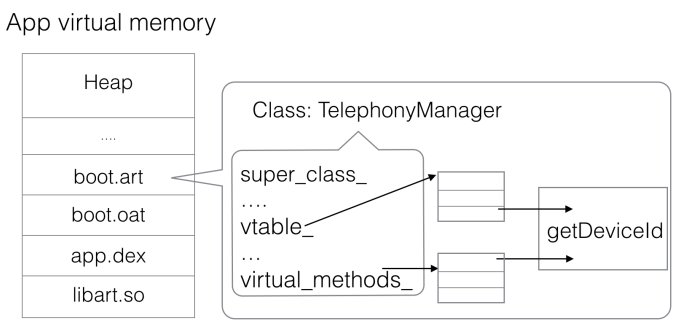
[6]art/runtime/jni_internal.cc

simplifies interacting with the target app and the Android framework (Context, etc. . .).

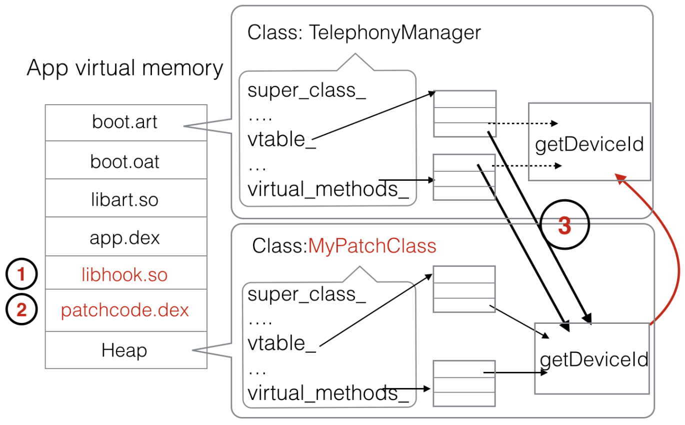
ARTDroid is based on library injection and uses Android Dynamic Binary Instrumentation toolkit[ADB] released by Samsung. The ABDI tool is used by ARTDroid to insert trace points dynamically into the process address space.

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[83] master process.

Now, we explain the framework design in figures. Figure 3.5(a) 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 3.5(b) 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).



(a) ARTDroid not enabled



(b) ARTDroid enabled

Fig. 3.5 App virtual memory layout

As discussed in 3.2.2, 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.

3.4 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.

3.5 Evaluation

3.5.1 Performance Test

To measure the effectiveness of virtual-methods hooking, we firstly need a test set of sensitive methods. SuSi[115] 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[36]. 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 application 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.

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

3.5.2 Case Study

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

```
{"config": {
    "debug": 1,
    "dex": [{"path": "/data/local/tmp/dex/target.dex"}]
    "hooks": [
    {
        "class-name": "android/telephony/TelephonyManager",
        "method-name": "getDeviceId",
        "method-sig": "()Ljava/lang/String;",
        "hook-cls-name": "org/sid/example/HookCls"
    }]
}}
```

Fig. 3.6 ARTDroid configuration file

Figure 3.6 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 3.7.

To restore the original call-flow, ARTDroid exposes to Java patch-code the native function `callOriginalMethod`. This function receivers as first argument the

```
public String getDeviceId() {
    String key = "android/telephony/TelephonyManagergetDeviceId()Ljava/lang/String;";
    Object[] args = {};
    return (String) callOriginalMethod(key, this, args) + " IMPS2016!!";
}
```

Fig. 3.7 Patch code for method getDeviceId

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.

3.6 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[92], permission leaking [69],[78] and remote code execution vulnerabilities[113].

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 [128],[149], [140] 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 3.5, 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[36] 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[150], 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.

3.7 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[40] 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[155], a system to confine the runtime behavior of the thid-party Android apps. Davis et al. proposed Retroskeleton[52], an Android app rewriting framework for customizing apps, which is based on their previous work, I-ARM-Droid[53].

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 [71],[149], 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[59], an Android modified system to detect privacy leak. StayDynA[149] 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[128], 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 [112] [132]. A comparison on Android sanbox has been published by Neuren et al. in [105].

Mulliner et al. proposed PatchDroid[100], 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 [101], 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 [140] 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[38] 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, [42] uses the same approach to sandbox arbitrary Android apps. The approach presented in

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

both of these recent works, represent one of the most promising and interesing future work direction.

3.8 Chapter Summary

In this chapter, we presented 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 root privilege. The applications of ARTDroid include dynamic analysis of Android malware on real devices or security policies enforcement.

Chapter 4

TeICC: Targeted Execution of Inter-Component Communications in Android

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

4.1 Introduction

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. For the purpose of security analysis rather than testing, it is required to stimulate/reach only specific points of interest (POI) in the code rather than stimulating all the code paths in an app. In literature, researchers have focused mainly on providing inputs to make an app follow a specific path. Providing the exact inputs and environment becomes very hard as different apps may require different execution environments. Moreover, not all inputs can be predicted statically, because of obfuscation or other hiding techniques. In this chapter, we propose a fully automated hybrid system which uses a slicing based approach for target triggering of a given MOI. It performs static data-flow analysis [33, 62] based on program slicing technique [138] 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 [84]. However, to the best of our knowledge, none of the existing approaches [116, 37] for targeted triggering support the extraction of interesting paths across different Android components.

In our proof of concept, TeICC, we leverage an enhanced version of SAAF to achieve program slicing [74]. We modified SAAF adding more sensitivity and support for ICC using a System Dependency Graph (SDG) (cfr. §4.3). Besides that, TeICC, employs a modified version of Stadyna [149] which integrates ARTDroid [50] 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.

4.2 Motivating example

The rising use of techniques such as obfuscation and ICC for information leakage by newly found malware motivates this work. Existing analysis approaches generally do not support information-flow analysis across multiple app components in obfuscated apps. As a result, malware use these features for evading these analysis tools. As reported by different antivirus companies [ace, 21, 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 [84], 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

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

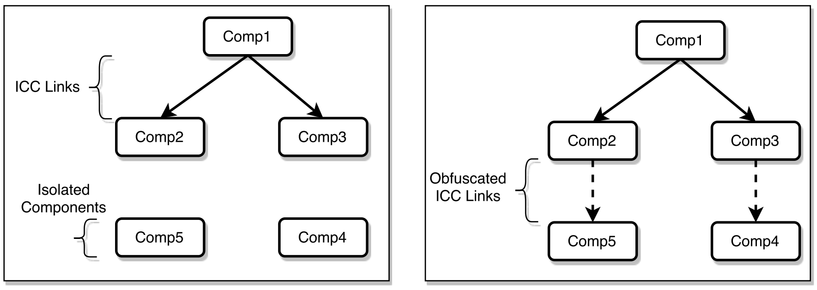
Listing 4.2 SendService

```
public class SendService extends IntentService {
    protected void onHandleIntent(Intent intent) {
        if(v1.equals("REPORT_INCOMING_MESSAGE")) {
            Sender.request(this.httpClient, "http://37.1.204.175/?action=command
    ", RequestFactory
                    .makeIncomingMessage(v2, intent.getStringExtra("number"),
    intent.getStringExtra(
                    "text")).toString());
            return;
        }
}}
public class Sender {
    public static JSONObject request(DefaultHttpClient hc, String serverURL,
    String data) throws Exception {
        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 [108, 107, 84, 67], 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 [116] and [37] cannot properly analyze the sample because they lack support for ICC.

4.3 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.

4.3.1 Slice Extraction

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

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

4.3.2 Inter-Component Communication

Our approach extends backward slicing across multiple app components. We build a System Dependency Graph (SDG) before starting slice extraction. A SDG is a representation of the program highlighting the inter-connectivity and program flow among various components. Figure 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

```
public class MessageReceiver_fake extends BroadcastReceiver {
    public void onReceive(Context context, Intent intent) {
        Map v0 = MessageReceiver_fake.retrieveMessages(intent);
        Iterator v6 = v0.keySet().iterator();
        while(v6.hasNext()) {
            Object v2 = v6.next();
            Object v5 = v0.get(v2);
            Intent v4 = new Intent(context, SendService_fake.class);
            v4.setAction("REPORT_INCOMING_MESSAGE");
            v4.putExtra("number", ((String)v2));
            v4.putExtra("text", ((String)v5));
            context.startService(v4);
        }
    }
}
public class SendService_fake extends IntentService {
    public void onCreate() {
        [...]
        this.httpClient = new DefaultHttpClient();
    }
    protected void onHandleIntent(Intent intent) {
        String v2 = SendService.settings.getString("APP_ID", "-1");
        Sender.request(this.httpClient, "http://37.1.204.175/?action=command",
    RequestFactory
                .makeIncomingMessage(v2, intent.getStringExtra("number"), intent
    .getStringExtra(
                "text")).toString());
}}
```

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 §4.2. 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.

4.3.3 Slice Execution

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

4.4 Design and Implementation

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

4.4.1 Overview

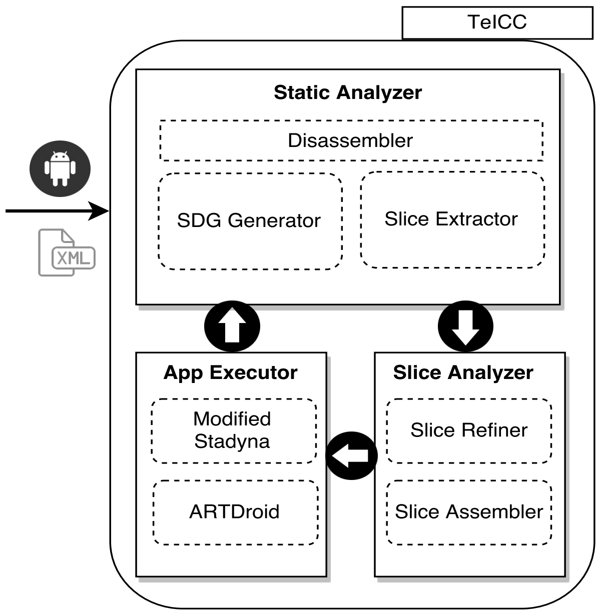


Fig. 4.2 TeICC Design

Figure 7.1 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 [70]. 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 [149] 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[50], to hook sensitive APIs such as the `sendTextMessage()` API.

4.4.2 Enhancement to Backward Slicer

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

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

Apart from extending backward slicing to cover ICC, we added other features which are important for the soundness of static analysis. The most important features we added are path-, context- and object-sensitivity [87]. 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 [116].

4.4.3 Capturing Dynamic Behavior

The idea behind a multiphase iterative model is to overcome the shortcomings of both static and dynamic analysis. TeICC relies on a modified version of Stadyna to handle reflection and dynamic code loading[149]. 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.

4.5 Evaluation and Discussion

This section presents experimental results that characterize the effectiveness of TeICC to analyze apps that conceal sensitive information flow using obfuscated ICC. We evaluate TeICC on two benchmark test suites, DroidBench [dro] and ICC-Bench [136], 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[84] 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[116] 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 [116], TeICC supports the ICC mechanism which enables it to automatically extract-and-execute target slices that belong to different Android components. Similarly, R-Droid [37] lacks support for both ICC and Java reflection mechanisms. Compared to IccTa[84], 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

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.

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 [116] and [37] also do not support native code analysis.

4.6 Chapter Summary

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

As a future work, 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 [50] framework to intercept sensitive Java methods called by native code.

Chapter 5

StaDART: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications

Static analysis of Android applications is inherently susceptible to be evaded by applications using dynamic code update techniques, i.e., dynamic class loading and reflection. These techniques, now heavily used in modern real-world malware, thwart even the latest of static analysis tools. We demonstrated this fact in section 2.3 by testing some of the state-of-the-art static analysis tools with Reflection-Bench and using InboxArchiver to evade online analysis systems. In this chapter, we present StaDART, an extented version of Stadyna[149], which combines static and dynamic analysis of Android applications to reveal the concealed behavior of malware. Unlike Stadyna, StaDART utilizes ARTDroid [50] to avoid modifications to the Android framework. Furthermore, we integrate it with a triggering solution, DroidBot, to make it more scalable and evaluate it with more Android applications. We present our evaluation results with a dataset of 2,000 real world applications; containing 1,000 legitimate applications and 1,000 malware samples

5.1 Introduction

Ensuring smartphone users' privacy and security is a major concern and requires adequate measures from app developers, framework providers, and app stores, etc. Google's open source operating system, Android, being the most popular platform for mobile devices, uses Google Bouncer as an app vetting process at its official Google Play store. Vetting processes generally use some form of static/dynamic analysis to scrutinize apps for malicious content and Google Bouncer is no different. In addition, starting from Android 7.0, Android introduced Verify Apps, a new security feature to analyze apps downloaded from sources other than the Google Play store.

However, a growing number of malware samples found in the Android ecosystem reveals that malware developers bypass such vetting processes using various evasion techniques. Previous research shows that the use of dynamic code update techniques along with various forms of obfuscation makes it extremely hard for the state-of-the-art analysis tools to understand the behavior of an app[113, 32]. Thus, the use of these evasion techniques in newly found malware is not surprising [Polkovnichenko and Boxiner]. This work provides an empirical demonstration of the lack of effectiveness of the state-of-the-art tools when it comes to analysing apps that hide suspicious behavior using reflection and dynamic code loading. We develop a set of benchmark apps that use reflection in different ways to conceal information leakage. Our analysis of reflection-bench using some of the state-of-the-art static analysis tools shows their ineffectiveness to handle apps that use reflection. Furthermore, we develop InboxArchiver, a seemingly benign app that uses dynamic code loading to hide its suspicious functionality, and use it to test some of the most well known online analysis services. The analysis show that InboxArchiver easily bypasses these security analysis services.

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, e.g., the parameters used in the dynamic code update APIs. Therefore, reflection that is a programming technique widely used by mobile app developers can be only partially investigated by current static analysis tools. As a result, reflection is usually used by malware developers to hide malicious code. The inherent limitation of all static analyzers (e.g., [35, 74]) is the operational assumption that the code base does not change dynamically and the targets of reflection calls can be discovered in advance. This is a clear simplification of what happens in the real world, where many apps rely on code base updates instantiated only at runtime.

There exist approaches that enhanced static analyzers of Java code to deal with the presence of dynamic code update techniques (e.g., [45]). However, they cannot be applied directly to Android due to the differences in the Java and Android platforms. The alternative of instrumenting the app offline has the major drawback of breaking the app signature, that some apps check at runtime. As the app starts, it checks the integrity of the signature against a value hardcoded in the app and terminates if the check fails. In case of malicious apps this check may be used to conceal illicit behavior.

In this chapter, we present StaDART, a mobile app security analysis tool that combines static and dynamic analysis to address the presence of dynamic code updates. Instead of relying on modifications to the Android framework, StaDART utilizes a vtable tampering technique for API hooking to perform dynamic instrumentation [50]. Furthermore, we integrate StaDART with DroidBot, a triggering tool for Android apps, to make the analysis fully automated. StaDART is evaluated using a dataset of 2,000 real world apps (both malicious and benign) and the results of our evaluation reveal that it is more common in malicious apps to use dynamic code updates to conceal malicious behavior which is only exhibited once the app is installed on a real device. Moreover, 33% of malware samples that use DCL introduce APIs guarded with new (not used in the initial code base) dangerous permissions in the newly loaded code, whereas the analysed benign apps do not exhibit such behavior.

Contributions:

– We present the design and implementation of StaDART, a system that interleaves static and dynamic analysis in order to reveal the hidden/updated behavior of Android apps. By utilizing vtable tampering for API hooking, we avoid modifications to the Android framework and make it largely framework independent. StaDART analyzes the code loaded dynamically, and is able to resolve the targets of reflective calls complementing app's method call graph with the obtained information. Therefore, StaDART can be used in conjunction with other static analyzers to make their analysis more accurate.

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

- We release our tool as open-source to drive the research on app analysis in the presence of dynamic code updates.
- We design and develop reflection-bench, a set of benchmark apps that use reflection to conceal information leakage, and use it to test some of the state-of-the-art static analysis tools. We publish reflection-bench so that researchers can test the effectiveness of their analysis tools in the presence of dynamic features (i.e., reflection).

Chapter Organization:

§2.3.1 discusses the design and implementation details of reflection-bench and InboxArchiver. It also provides the analysis results highlighting the shortcomings of state-of-the-art Android app analysis tools. §5.2 gives a high-level description of StaDART, while §5.4 covers the implementation details. §5.3 presents our approach to build method call graphs and visualise them. §5.5 reports the evaluation results of StaDART on real world apps. §5.6 discusses the limitations of the current implementation, and envisages the future work. §5.7 overviews the related work, and §5.8 concludes the chapter.

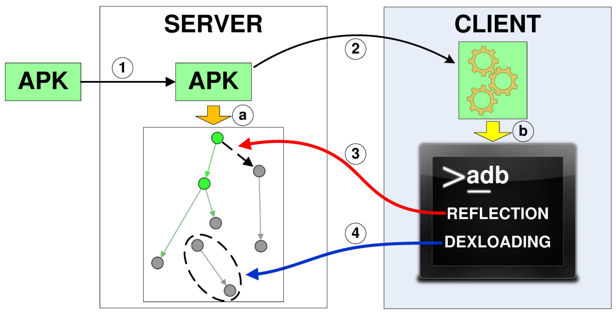


Fig. 5.1 System Overview

5.2 An Overview of StaDART

The architecture of StaDART presented in Figure 5.1 comprises two logical components: a server and a client.

The static analysis of an app is performed on the server. StaDART allows an analyst to easily plug in and use any static analyzer in its architecture. The static analyzer on the server builds the initial *method call graph* (MCG) of the app, integrates the results of the dynamic analysis coming from the client, and stores the results of that analysis. The client part of StaDART is based on an API hooking technique that intercepts calls to dynamic code update APIs and captures the dynamic behavior. The client part can be hosted either on a real device or an emulator. The client runs the app whenever dynamic analysis is required. StaDART interleaves the execution of the static and dynamic analysis phases and an app can have several of these phases. However, for simplicity of the presentation without loss of generality, we describe them sequentially.

Preliminary analysis The server statically analyzes an app package and builds a MCG of the application (see Step *a* in Figure 5.1; solid arcs denote edges resolved statically). Dynamically loaded code cannot be analyzed during this phase and, thus, the corresponding nodes and edges are not present in the MCG. Further, the names of methods called through reflection may also not be inferred if they are represented as encrypted strings or generated dynamically. Still, a static analyzer can effectively detect the points in the MCG where the functionality of an app may be extended at runtime. Indeed, the usage of reflection and DCL requires to use specific API calls provided by the Android platform. The server detects these calls during the static analysis phase by searching for methods where

DCL and reflection API calls are performed. We call these methods *methods of interest (MOI).*

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

Analysis consolidation The server performs an analysis of the obtained information. In case of a reflection call, the server complements the MCG of the app with a new edge (in Figure 5.1, it is represented by a dashed arc). This edge connects the node of the method that initiated the call through reflection (the node at the beginning) with the one corresponding to the called function (the node at the end). When DCL is triggered, the client captures the location of the code file. Using this evidence, the server downloads the file (Step *4*) containing the code, and analyze it statically. The MCG of the app is then updated with the obtained information (see part of the MCG in dashed oval in Figure 5.1). Additionally, for each downloaded file the server analyzes whether it contains other MOIs. If it does, the list of the MOIs for the app is updated. This allows StaDART to unroll nested MOIs. The stack trace data for both the reflection and DCL cases is used to detect which MOI initiated the call.

Marking suspicious behavior In Android, some API calls are guarded by permissions. Since APIs protected by permissions could potentially harm the system or compromise a user's data, the permissions must be requested in the AndroidManifest.xml file. However, there is no actual check on the permissions required to execute the written code and sometimes developers request more permissions than they actually use. In this case, those apps are called overprivileged. Many researchers, e.g., Bartel et al. [41], identified that malware, adware and spyware exploit additional permissions to get access to security sensitive resources at runtime.

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

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

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

StaDART automatically detects such suspicious patterns and raises a warning if such patterns occur during the analysis. Section 5.5 shows that indeed malware samples do expose such suspicious patterns.

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

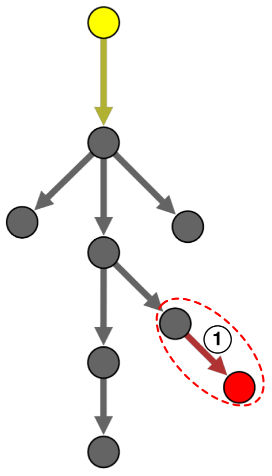
5.3 Method Call Graph

Method call graphs (or function call graphs) identify the caller-callee relationships for program methods. These structural representations of programs are widely used for different purposes. In the scope of Android, method call graphs are used to detect malware, identify potential privacy leaks in apps, find vulnerabilities and execution paths for automatic testing, etc. StaDART extends the initial MCG generated with a traditional static analyzer with the information detected at runtime. Thus, if an app exposes dynamic behavior, all mentioned approaches can benefit from the expanded MCG obtained with StaDART.

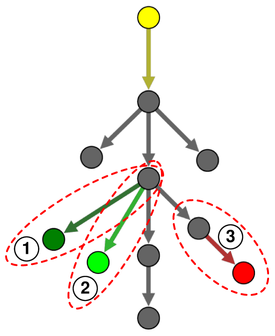
Example To visualize the capabilities of StaDART and the process of method call graph expansion, we show the evolution using an example of a *demo_app*. Figure 5.2(a) shows the MCG of the app obtained with the AndroGuard static analyzer [54]. Figure 5.2(b) shows the one gained with StaDART before dynamic execution phase, and Figure 5.2(c) presents it with dynamic execution phase. The *demo_app* dynamically loads some code from an external `.jar` file at runtime and calls the loaded methods through reflection.

Figure 5.2(a) illustrates that AndroGuard identifies only the presence of ordinary methods and DCL calls (Ellipse 1) but no further analysis is done about those. Yet, Figure 5.2(b) shows that after preliminary analysis StaDART selects 3 paths, which are surrounded by dashed ellipses. Ellipse 1 shows that a MOI (the dark grey node) invokes a constructor (the dark green node) through reflection. Similarly, Ellipse 2 displays a method invocation through reflection. Ellipse 3 depicts that a DCL call (the red node) is performed in a MOI (the dark grey node).

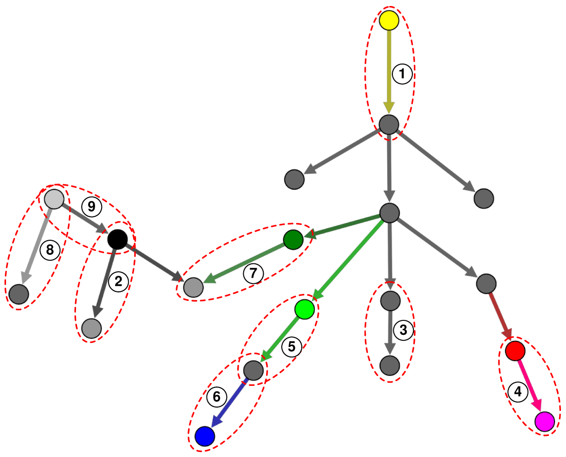
During the dynamic analysis, StaDART adds the edges that are outlined by Ellipses 4-7 (see Figure 5.2(c)). These ellipses show the cases when the MOIs are resolved and corresponding nodes and edges are added to the MCG. Ellipse 4 shows that as a result of a DCL call (the red node) a new code file has been loaded (the pink node). Ellipse 7 shows that a class constructor (the grey node) is called through reflection. Ellipse 5 shows a method invoked through reflection. This method contains an API call protected by the Android permission indicated by the blue node in Ellipse 6. There are also nodes and edges that appear as a result of the analysis of the code file (the pink node) loaded dynamically. These



(a) b



(b) b



(c) b

Fig. 5.2 MCG of *demo_app* Obtained with a) AndroGuard b) StaDART after Preliminary Analysis c) StaDART after Dynamic Analysis Phase

nodes and edges are connected with the rest of the graph through the reflection *new instance* call (see Ellipse 7).

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

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

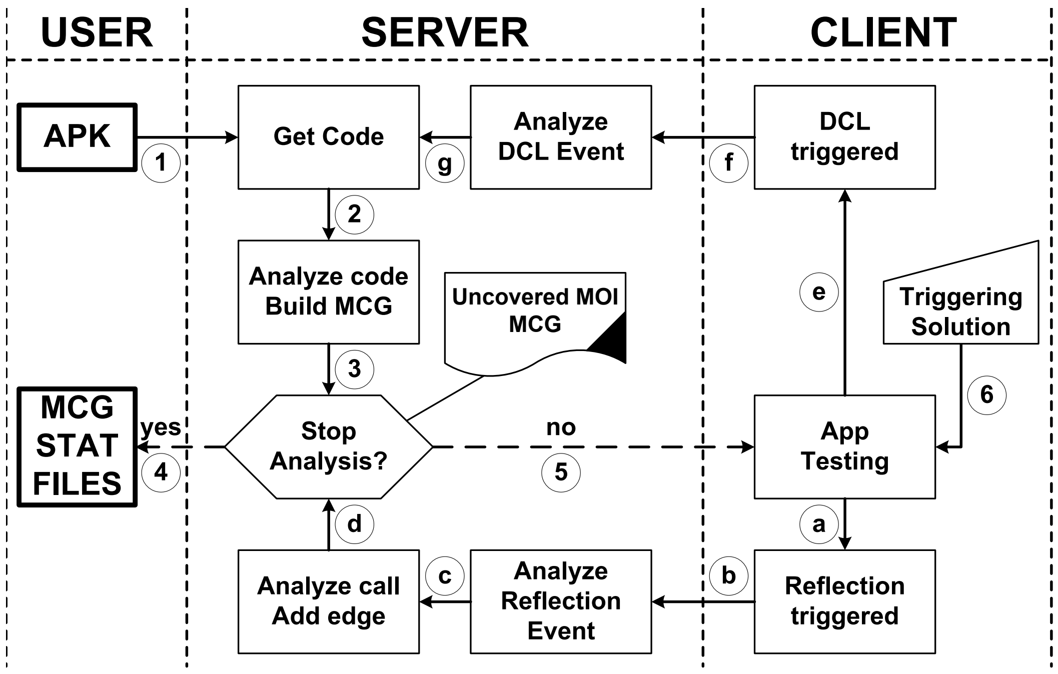


Fig. 5.3 StaDART Workflow

5.4 Implementation

This section provides the implementation details of some key aspects of StaDART. The workflow of StaDART's operation is shown in Figure 5.3. App analysis starts at the server side. All occurrences of reflection and DCL methods are identified in the code of the application under analysis. In case neither of them is found, StaDART builds a MCG of the app and exits. Otherwise, the app is analyzed using StaDART client on a device.

5.4.1 The server

The server side of StaDART is a Python program that interacts with a static analysis tool. Currently, StaDART uses AndroGuard [3] as a static analyzer. AndroGuard represents compiled Android code as a set of Python objects that can be manipulated and analyzed. However, StaDART can work with any static analysis tool that is able to analyze `apk` and `dex` files. To improve suspicious behavior detection we substituted the permission map embedded in AndroGuard (built for Android 2.2 in [61]) with the one generated by PScout [36] for Android

5.1.1, which is the latest API-permission mapping available in the research community.

Algorithm 1 App Analysis Main Function Algorithm

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

The pseudo-code of the main server function is presented in Algorithm 1. The server starts the analysis of the provided app by extracting the `classes.dex` file (see Step *1*, *2* and *3* in Figure 5.3; Line 2 in Algorithm 1), and then dissects the extracted code. During this step StaDART searches for all the occurrences of reflection and DCL calls in the code. The list of searched patterns for these API calls is presented in Table 5.1.

If MOIs are found, StaDART selects a device (a real phone or an emulator) to perform the dynamic analysis on (Line 8) and installs the app under analysis (Line 10) onto the client (Step *5* in Fig. 5.3). After that the server obtains the UID of the installed package (Line 11) and starts a loop (Lines 13-25) that analyzes, one by one, the messages (Line 12) obtained using the *logcat* utility from the *main* log file of the Android system. Basically, each obtained message is represented in the JSON format and contains values for the following fields: *UID* (required), *operation* (required), *stack* (required), *class* (optional), *method* (optional), *proto* (optional), *source* (optional), *output* (optional). The value of the `UID` field is used to select the messages produced by the analyzed app. If the user stops the analysis, StaDART saves the results and finishes its execution.

Table 5.1 The List of Searched Patterns

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

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

The routines for reflection messages analysis are similar, so we consider them on the example when operation corresponds to *reflection invoke*. The algorithm for analysis of the *reflection invoke* messages is shown in Algorithm 2 (algorithm for analysis of *reflection newInstance* messages is very similar so we do not show it). Lines 2 - 4 extracts the method name along with its class name and the prototype, which has been called through reflection. Line 5 gets the stack from the message. Line 7 searches for the first *reflection invoke* occurrence in the stack. The next stack entry corresponds to the method that has performed the reflection call `invSrcFrStack` (Line 9). Then in the loop StaDART compares this method with the list of MOIs extracted from the app executable (Lines 10 - 20). If the method is found StaDART complements the MCG with the obtained information (Line 15), and deletes it from the list of uncovered invoke MOIs (Line 17). Otherwise, it adds this method to the list of vague methods (Line 21). This information is later analyzed to see why the method calling reflection was not found in the app executable during the static analysis phase.

The processing function for the DCL messages is slightly different (see Algorithm 3). From the message received from the client the server extracts the source path of the file containing the code loaded dynamically (Line 2). Using this information, StaDART downloads the file locally (Line 4), and processes it (Line 5). This process includes computation of the file hash and copying the

Algorithm 2 Analysis of the Reflection Invoke Message

```
1: function processReflInvokeMsg($message$)
2:     $cls \leftarrow message.get(JSON\_CLASS)$
3:     $method \leftarrow message.get(JSON\_METHOD)$
4:     $prototype \leftarrow message.get(JSON\_PROTO)$
5:     $stack \leftarrow message.get(JSON\_STACK)$
6:     $invDstFrCl \leftarrow (class, method, prototype)$
7:     $invPosInStack \leftarrow findFirstInvokePos(stack)$
8:     $thrMtd \leftarrow stack[invPosInStack]$
9:     $invSrcFrStack \leftarrow stack[invPosInStack + 1]$
10:    for all $invPathFrSrcs \in sources\_invoke$ do
11:        $invSrcFrSrcs \leftarrow invPathFrSrcs[0]$
12:        if $invSrcFrSrcs \neq invSrcFrStack$ then
13:            $continue$
14:        end if
15:        $addInvPathToMCG(invSrcFrSrcs, thrMtd, invDstFrCl)$
16:        if $invPathFrSrcs \in uncovered\_invoke$ then
17:            $uncovered\_invoke.remove(invPathFrSrcs)$
18:        end if
19:        return
20:    end for
21:    $addVagueInvoke(thrMtd, invDstFrCl, stack)$
22: end function
```

Algorithm 3 Analysis of the DCL Message

```
1: function processDexLoadMsg($message$)
2:     $source \leftarrow message.get(JSON\_DEX\_SOURCE)$
3:     $stack \leftarrow message.get(JSON\_STACK)$
4:     $newFile \leftarrow dev.get\_file(source)$
5:     $newFilePath \leftarrow processNewFile(newFile)$
6:     $dlPathFrStack = getDLPathFrStack(stack)$
7:     if $dlPathFrStack$ then
8:         $srcFrStack \leftarrow dlPathFrStack[0]$
9:         $thrMtd \leftarrow dlPathFrStack[1]$
10:        if $dlPathFrStack \in uncovered\_dexload$ then
11:            $uncovered\_dexload.remove(dlPathFrStack)$
12:        end if
13:        $addDLPathToMCG(srcFrStack, thrMtd, newFilePath)$
14:        if $!fileAnalyzed(newFilePath)$ then
15:            $makeAnalysis(newFilePath)$
16:        end if
17:        return
18:    end if
19:    $addVagueDL(newFilePath, stack)$
20: end function
```

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

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

5.4.2 The client

The client side can run either on a real device or on an emulator. Using the emulator is more convenient because one can run the client and server on the same machine. The main drawback is that currently the Android emulator is quite slow. Moreover, mobile apps may suppress some functionality if they detect they are running in an emulated environment. With these limitations in mind, we implemented and tested our client on a real device. However, the code is device-independent and easily portable to any other device/emulator.

To capture the dynamic behavior offered by reflection and DCL, we intercept a number of Android API methods that provide an interface to DCL and reflection capabilities. A brief overview of these APIs is provided earlier in §5.4. Some of them have been modified across different Android versions moving their implementation to the native side (e.g., `java.lang.Class.newInstance` has only a native implementation in Android 6). To achieve dynamic instrumentation of Java-level APIs we used the approach proposed in ArtDroid [50] to intercept Java virtual methods. It intercepts all calls to monitored Java virtual methods including calls via Java reflection, native code or dynamically loaded code without any modification to both Android OS and the target app. In addition, we integrated native function hooking capabilities in StaDART by means of *inline hooking* technique. The client side employed by StaDART is completely Android version-agnostic and it is able to interpose custom code on both Java methods and native functions. Therefore, it can be used to analyze Android apps on any Android version intercepting DCL and reflection calls irrespective of the actual code representation (i.e., Java or native). To support all available Android versions, we included in StaDART the capability of intercepting DCL and reflections calls according to the running Android version. In the following we describe methods intercepted by StaDART on both Dalvik and ART runtime. The code added by StaDART to perform requested analysis is not influenced by the underlying Android version.

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

To get the information about method invocation through reflection, a hook was placed into the `invoke` method of the `Method` class. As of the release of Android version 6, this method is defined as `public native`, thus the client will hook the appropriate function by means of the proper hooking engine, according to the running Android version. Each `Method` object has `declaringClass`, `name` and `parameterTypes` member fields, which represent class name, method name and prototype of the invoked method, respectively. Moreover, `invoke` gets an array of Object type as input which represents the arguments intended for the target method. This information along with the stack trace is put into the StaDART message. Similarly, to log the information about new class creation through reflection, we put our hooks into the `newInstance` method of the `Class` and `Constructor` classes. As for the `invoke`, different hooks were added targeting `newInstace` code representation for both DVM and ART runtime.

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

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

5.5 Evaluation

Experiment Setup and Test Suite: This section describes our app test suite and reports on the results of our experiments. We evaluated StaDART with a dataset of real world benign and malicious apps. The server runs on a machine with 3.2 GHz Intel Core i7 processor and 8 GB DDR3 memory. The client is a Google Nexus 6 smartphone running stock Android OS version 7.1.1 connected to the server using a standard USB cable. The evaluation test suite consists of a set of 1,000 benign and 1,000 malicious apps. The benign apps were selected based on their popularity. The malware samples were selected from Drebin[34] dataset populated by 5,560 apps from 179 different malware families.

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

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

Analysis Results: Figure 5.4 illustrates the prevalence of dynamic code update APIs in the analyzed dataset and the effectiveness of StaDART in expanding the MCGs. It shows the percentage of apps with invoke, newInstance and DCL among both benign and malicious app dataset. The right most bar represents the percentage of apps where StaDART expanded the MCG. In the dataset, close to 90% of the apps use `invoke` and/or `newInstance` APIs. Similarly, 48% of the apps use DCL feature which is considerably higher to previous analysis

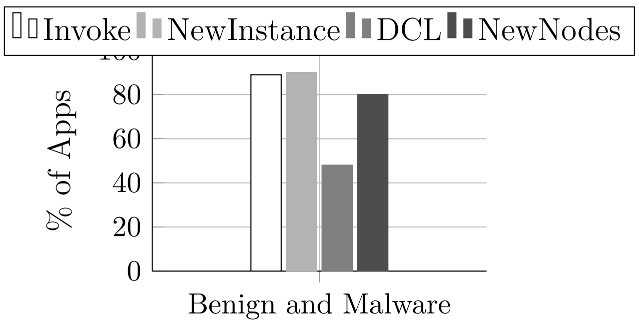


Fig. 5.4 Prevalence of Reflection/DCL and StaDART effectiveness in expanding MCG

results [149] (first part of RQ1). Increase in the number of apps using DCL could largely be related to the increasing complexity of the Android apps. StaDART was able to expand the MCG by at least one node in 80% of the analyzed apps (second part of RQ1).

Figure 5.5 shows MCG expansion using StaDART for the apps in the analyzed dataset using reflection only, both benign and malicious. It shows the average percentage increase in the number of nodes, edges, nodes with normal permission and nodes with dangerous permissions. Clearly, the lower percentage increase is attributed to apps that use only reflection as dynamic code update feature. The MCG expansion in these apps, which do not use DCL, is minimal and more or less similar in benign and malicious apps (RQ2).

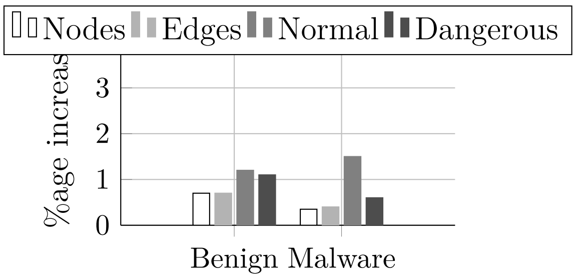


Fig. 5.5 MCG Expansion

To clarify the role of DCL in MCG expansion and dynamic behavior, we extracted the results from apps that use DCL. Figure 5.6 shows the effectiveness of StaDART when the apps use DCL. It shows the average percentage increase in the number of nodes, edges, nodes with normal permissions and nodes with dangerous permissions. It clearly shows a considerably higher increase in the number of nodes, edges and nodes guarded with permissions (both normal and dangerous). In addition, it can be seen that the malicious apps hugely increase their code base when they use DCL (RQ4). Similarly, the number of nodes guarded with permissions for malicious apps doubled or in some cases quadrupled (RQ3). This clearly indicate that malicious apps make use of sensitive APIs in the loaded code. We also check the added nodes for Signature level permission and SignatureOrSystem level permission. However, we did not observe a noticeable increase in the number of nodes guarded with these permissions.

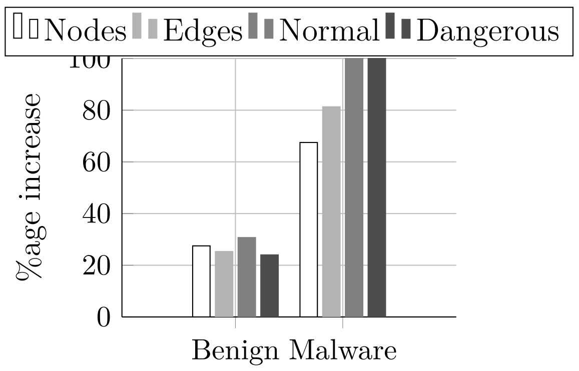


Fig. 5.6 MCG Expansion when apps use DCL

Although, the high increase in the number of nodes guarded with dangerous permissions is indeed suspicious, we investigate the analysis results further for a more suspicious malware behavior. In practice, malicious payloads are packaged inside legitimate apps and their manifest files are modified to cover for the extra permissions needed by the payload. In this scenario, the final MCG of the app contains nodes guarded with new permissions, i.e., those not found in the initial MCG. Figure 5.7 shows the distribution of apps based on increase in the number of nodes guarded with permissions in the form of pie-charts, in benign apps and malware. Here we discuss only those apps which use DCL. The white part shows the percentage of apps with no increase in the number of nodes guarded with permissions, whereas the grey part represents the percentage of apps with increase in the number of nodes guarded with permissions. The darker grey part shows the percentage of apps where new permissions are used in the dynamically added part using StaDART.

The pie-chart for the benign apps shows that a very small fraction of the apps observe an increase in the number of nodes guarded with dangerous permissions. In contrast, a considerably higher number of malicious apps reveal such behavior. Also, in none of the benign apps in the dataset, the loaded coded contained nodes guarded with new dangerous permission. However, all the malicious apps

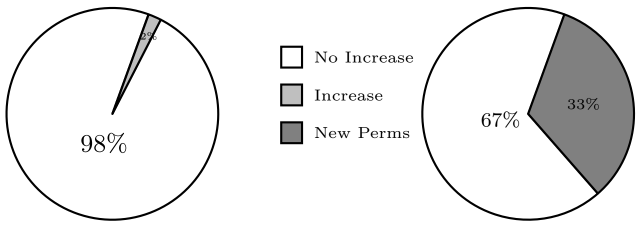


Fig. 5.7 Increase in permission nodes. (L) Benign (R) Malicious apps

in the dataset that loaded code dynamically contained nodes guarded with at least one new dangerous permission (RQ5). Moreover, a further analysis of the loaded code in malicious apps reveals a pattern of dangerous permissions, e.g., `READ_PHONE_STATE` and `INTERNET`, that could be associated with malicious functionality, such as privacy leakage, etc.

Also, noteworthy here is the fact that the revealed behavior is only due to triggering of a small fraction of the total MOIs. Albeit the most advance automated triggering tool in the research community, DroidBot does not serve well for app exploration from a security point of view. Taking into account the low exploration that DroidBot achieved in most of the apps and the suspicious results that we observed, the actual hidden suspicious/malicious behavior could be alarming.

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

5.6 Discussion

For any dynamic (or hybrid for that matter too) analysis tool, coverage is the main limiting factor and StaDART is no different in that regard. For StaDART the coverage of MOIs (the ratio between the number of executed MOIs at least once and total number of discovered MOIs) is especially important. In order to achive higher MOI coverage, we explored if the tools like *monkey* can be handy. However, in our experiments we found out that pseudo-random events generated by the tool do not produce tolerable coverage values for MOIs. Therefore, we opted for a more advance automated triggering tool, DroidBot, to trigger MOIs. However, as discussed in the previous section, even DroidBot did not achieve reasonable coverage of MOIs. Possible enhancement can be achieved using techniques such as the one used in SmartDroid [150]. SmartDroid allows an expert to specify sensitive API methods required to be triggered. In case of StaDART the sensitive API methods correspond to reflection and DCL calls.

Another possible direction to reduce the dependence on the triggering tool is to resolve as many targets of reflection calls as possible statically, at least those which are represented by constant strings [74]. The analysis performed in [61] has shown that it was possible to resolve automatically the targets of reflection calls in 59% of applications that used reflection. At the same time, the analysis was performed for the “closed world” scenario, which is not realistic, given that dynamic class loading is a popular technique for modern apps. Consequently, we can minimize the more expensive dynamic part of the analysis.

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

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

5.7 Related Work

Apps are analyzed for malicious contents before being published to the app markets. Many static and dynamic analysis techniques have been proposed for Android. The *ded* system re-targets Dalvik bytecode into Java class files that can be analyzed by the variety of tools developed for Java. DroidAlarm [153] performs static detection of privilege-escalation vulnerabilities in apps by constructing paths in inter-procedural call graphs from a sensitive permission to a public interface accessible to other apps. Gascon et al. employ comparison of functional call graphs (FCG) mined using AndroGuard to detect malicious Android apps [64]. StaDART can complement these techniques by providing more precise graphs required for analysis.

TaintDroid was among the first dynamic analysis tools for Android apps [59]; it tracks propagation of information via the TaintDroid infrastructure-equipped smartphone software stack. It detects leakage of user private information to network interfaces. This approach is followed by DroidScope [144]. DroidScope allows to emulate app execution and trace the context at different levels of the Android software stack: at the native code level, at the Dalvik bytecode level, at the system API level, and at the combination of both native and Dalvik levels. While executing an app in DroidScope a security analyst can track events at different levels and instrument parameters of invoked methods to discover a malicious activity.

Dynamic analysis techniques are especially difficult to automate due to the need of emulating a comprehensive interactions of apps with the system and a user (UI interactions). Several approaches are proposed to automate the triggering of UI events, from random event generation [75] to more advanced approaches like AppsPlayground [117] and SmartDroid [150]. However, all of them still have many limitations on the type of events they can handle and the coverage.

Poeplau et al. [113] selected possible vulnerable patterns of dynamic code loading and built a tool that can analyze Android apps for the found patterns. Moreover, they propose to use whitelists to prevent dynamic code loading that can potentially expose dangerous behavior. Whitelisting prevents unauthorized code from running. To get authorization the code must either be signed and its signature has to be included into a special list distributed by trusted authorities. However, as mentioned in the article [113], extraction of the dangerous behavior is a difficult problem by itself, especially when the protected API is called through reflection. In contrast, StaDART aims not at preventing this loading (because a lot of legitimate apps use it and extra complications will not be welcomed by the developers) but at its analysis.

Comparing to Stadyna [149], StaDART differs in various aspects. The client side of StaDART is based on API hooking using a vTable tampering technique used in ArtDroid, rather than modification to the Android framework, and therefore, can be easily ported to different versions of Android. Also, StaDART analyzes the arguments passed to the APIs/methods called using reflection API `invoke`. On the client side, unlike Stadyna which requires a human user to interact with the app during analysis, StaDART relies on a triggering tool, DroidBot, to make the analysis fully automated. StaDART is evaluated on a much larger set of applications, 2,000 apps (1,000 benign and 1,000 malicious) in comparison to Stadyna's 10 apps (5 benign and 5 malicious).

Gaps in the static analysis techniques in the presence of dynamic class loading, reflection and native code were previously studied for Java. For example, similarly to our approach, in [73] a pointer analysis (based on program call graphs) technique for the full Java language is extended by addressing dynamic class loading and reflection via an "online" analysis, when a call graph is built dynamically based on the program execution, and dynamic class loading, reflection and native code are treated in real time by modifying the pointer analysis constraints accordingly.

A run-time shape analysis for Java is investigated in [46]. Traditionally a shape analysis operates on the call graph of a program and determines how heap objects are linked to each other (e.g., if a variable can be accessed from several threads). AS call graph produced from java program can be incomplete, [46] suggests how to execute an incremental shape analysis when the call graph evolves dynamically. Our proposal does not involve a shape analysis, yet the ideas behind our proposal and [46] are similar. Livshits et. al. [90] proposed a refinement of the static algorithms to infer more precise information on approximate targets of reflective calls, as well as to discover program points where user needs to provide a specification in order to resolve reflective targets.

Relevant to StaDART is TamiFlex [45] that complements static analysis of Java programs in the presence of reflection and custom class loaders. Using the load-time Java instrumentation API, TamiFlex modifies the original program to perform logging of class loading and reflection call events. This information is used to seed a tool that performs static analysis of the program having the information obtained during the dynamic analysis phase. This work differs from StaDART in several aspects. First, TamiFlex uses a special Java API that is not available in Android. Second, although in Android it is possible to instrument an app before loading it on a device (offline instrumentation), some Android apps check the app signature in its code that is changed during the patching. Thus, for these apps the TamiFlex approach will not work in Android. Third, TamiFlex requires some debug information (the line number of the function call) to be present. In Android during the obfuscation phase this kind of information may be deleted from the final package. Therefore, the TamiFlex approach will not work, while StaDART is able to process correctly this case due to dynamic API hooking.

More recently, reflection aware analysis of Android apps has been the focus of some research publications. For example, DroidRA uses string inference analysis to resolve reflective calls and replaces them with regular Java calls by instrumenting apps for further analysis [85]. However, DroidRA cannot resolve the targets of reflection when the arguments to reflection APIs are not readily availabe in the app. StaDART's dynamic element could prove fruitful in this regard. Ripple uses a combination of formal analysis and pointer analysis to ensure reflection aware static analysis in incomplete information environment (IIE) [148]. Although Ripple resolves most targets of reflection, various cases of IEE lead to high false positives. In fact, Ripple in conjunction with StaDART could prove beneficial for both where Ripple reducing the dynamic analysis part of StaDART and StaDART reducing the falst positive rate of Ripple.

5.8 Chapter Summary

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

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

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

Chapter 6

AppBox: Black-Box Mobile App Management Solution For Stock Android

A fast growing number of organizations allow their employees to use smartphones to do business computing. In this scenario, it is important for the IT security department of the enterprise to be able to configure fine-grained security policies for the employees' devices. To address this increasing demand of easy and secure management, several Mobile Device Management (MDM) and Mobile Application Management (MAM) services have been launched on the market. However, these services impose critical limitations by the lack of fine-grained capabilities of these services and by the customisation required to the applications to be included in the list of supported apps. In this chapter, we present AppBox , a novel black-box app-sandboxing solution for app customisation for stock Android devices. AppBox enables enterprises to select any app, even highly-obfuscated ones, from any market and perform a set of target customisations by means of fine-grained security-and-privacy policies. We have implemented and tested AppBox on various smartphones and Android versions, including Oreo. The evaluation shows that AppBox can effectively enforce fine-grained policies on a wide set of existing apps, with an acceptable overhead.

6.1 Introduction

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 work, we propose an app-level 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 MAM 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 bytecode. 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 bytecode 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/

6.2 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 is not a mobile malware detector neither a security sandbox for suspicious applications. Anti-malware solutions are complementary to AppBox .

AppBox is a MAM solution for enterprises willing 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).

As we discuss in details in Section 7.6, most of the proposed MAM solutions for enterprise achieve those customisations via either altering the app's bytecode or requiring modifications to Android core components (i.e., Linux Kernel, Android framework). Unfortunately, both these two approaches have shown their limitations, especially bytecode rewriting which requires strong modifications to the app's bytecode that are difficult to maintain other than restricting its applicability to very specific scenarios (i.e., the rewritten application can not be delivered via Google Play market). On the other hand, traditional MAM solutions (see Section 7.6) requires a strong collaboration with the developer. In fact, developers employ specific API offered by the MAM. To this end, we propose a novel approach that leverages on two main features offered by the Android system. Thanks to AppBox , developers can provide different customised versions of the same application without any modification to the app's bytecode itself. In Section **??** we have introduced the main distinction between the app's bytecode and the app's manifest, which is essential to understand our approach.

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 (i.e., under appropriate monetary compensation) 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 6.4.1). 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 6.4.3). 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. Since both the app A' and the stub A'_{stub} are now signed with the same K the Android manifest attribute *android::sharedUserId* allows to execute both apps under the same UID. 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 (policy can be written and extended at will):

- 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 chapter how, in detail, AppBox expresses and implements such policies.

6.3 Requirements

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

Our aim is to provide a black-box app-level mobile app management solution for stock Android that does not require modifications of the apps' bytecode and developers are in charge of the small effort as possible (none SDKs integration nor app's modifications). The following requirements are needed and are met by AppBox :

- **R1:** 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 neither modifications/extensions to the Android OS nor the root privileges. The mechanism allows developers to provide customised app versions without suppling any intellectual property (i.e., app's source code) to any third-party entities. In addition, developers are not required neither to insert any supplementary code within the application nor provide any integration with SDKs. Finally, it can operate without requiring any app's bytecode modification.
- **R2:** Permissions: The mechanism should not require more privileges and permissions than those originally requested by the app that will be sandboxed.
- **R3:** 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.

The first requirement, $R1$, is the crucial one in order to provide a practical and portable app-level MAM solution. It states that the solution must be universal in the sense that it can operate without any modification to both Android OS components (i.e., Linux Kernel, Android framework, etc.) as well as lacking the root privileges. This allows to achieve a portable solution that is able to operate on any Android running devices, despite it can ben rooted or not, and is totally suitable for BYOD environments where OS level modifications do not apply. In addition, the solution must be able to operate without integration with any SDKs or special shared library as well as without rewriting the app's bytecode. Rewriting app's bytecode has shown its strong limitations: in primis it does not operate well under heavily obfuscated applications (note that obfuscation is a crucial barrier against attacks to PI) or those implementing anti-tampering mechanism, repacking an application (then breaking its signature) and distributing may be against the copyright, also and more important the developer is in charge to insert and maintains new code according to the MAM APIs in place (which basically requires developers to branch their app for each enterprise asking for customisations). We designed this requirement in order to define properties that must apply to offer a practical and agnostic MAM app-level solution.

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.

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 ($R3$), 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).

6.4 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 6.4.1. 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 bytecode 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.

AppBox workflow consist of three main phases shown in Figure 6.1: *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).

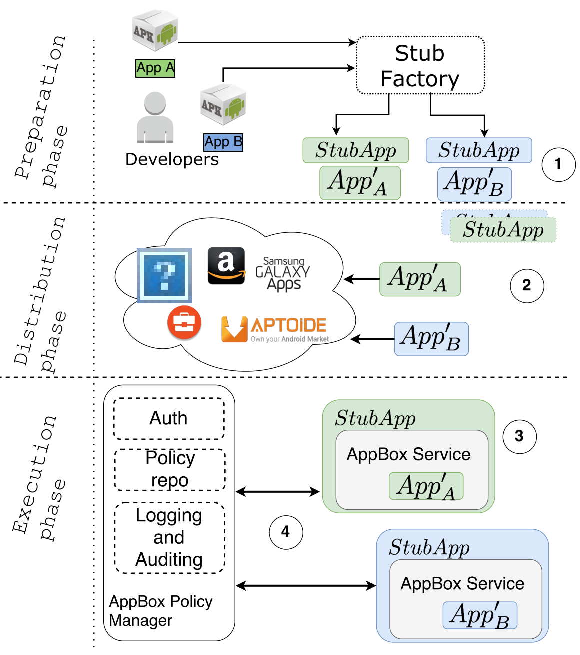


Fig. 6.1 AppBox design phases: Preparation , Distribution and Execution.

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

6.4.1 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 6.1. This step 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.

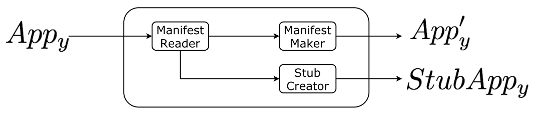


Fig. 6.2 StubFactory and its components

As shown in Figure 6.2, 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 identify 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.

6.4.2 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.

6.4.3 Execution phase

After the preparation step, both *StubApp* and *App′* are deployed on a device running stock Android OS. It is worth mentioning that the *StubApp* is designed to provide a user-friendly mechanism, neither additional icons shown nor management cost in charge of the end user.

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.

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 6.3. 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.

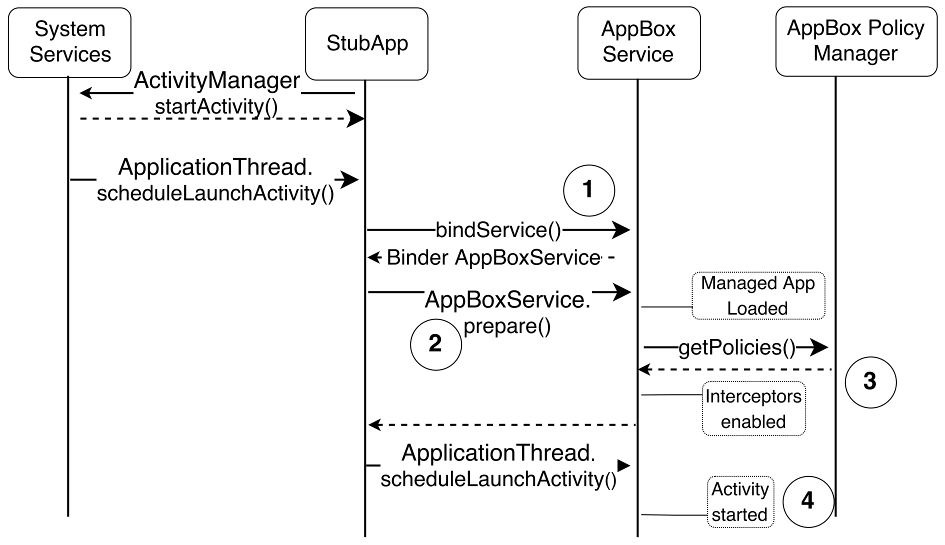


Fig. 6.3 AppBox enforcing managed apps

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 6.4.3.

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 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 [50]. 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

[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

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 [inl] 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.

6.5 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 6.2). Finally, we provide some details on how policies can be automatically generated.

6.5.1 Policy Language

Figure 6.4 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.

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

Fig. 6.4 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 correspond 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.

6.5.2 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 6.2, 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 6.5.

```
MicPolicy : AppX can do getMicrophone on Microphone
            have to perform checkLocation();
            if isWorkHours() == True

LocPolicy : AppX can do getLocation on Location
            have to perform checkLocation();
            if isWorkHours() == True

ContPolicy: AppY can do getContacts on Contacts
            have to perform filterOut();
            if isWorkHours() == True and isLocation() == True

HTTPPolicy: AppZ can do createConnection on Internet
            have to perform forceHTTPS()
            if isWorkHours() == True

CopPolicy : AppX can do runApp on Boot
            have to perform checkApp();
```

Fig. 6.5 AppBox Policies

MicPolicy in Figure 6.5, 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 6.4.3). 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.

6.5.3 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 6.6. 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[36] 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 6.1 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 6.5 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 6.1.
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 6.1 HTTP-Policy Generation - Intercepted APIs

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

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

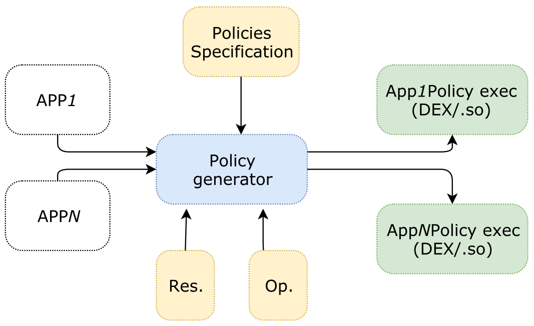


Fig. 6.6 Policy generation mechanism. Res.: sets of Resource, Op: sets of operation

6.6 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.

6.6.1 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 [123, 38, 42] 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 6.2, 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 6.2), 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

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

tests for NJAS and Boxify[6] but both systems were not available to us at the time of writing.

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 6.3, 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 [38]. 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 6.5 reports the results for the overhead introduced by AppBox when interposing Java Android APIs, Table 6.4 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 6.5 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 6.2 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 %

[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 [38]

6.6.2 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 6.6, 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 6.6 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

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

Table 6.3 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%

managed apps for 5 minutes similar to the Google Play Bouncer [106] 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.

6.6.3 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 6.7 shows the list of apps we selected. For each app we enabled, in spirit of the scenario envisioned in Section 6.2, 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[21] 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 6.7 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 6.4 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 6.5 Android APIs Monitored During Java Micro-Benchmarks

Alias	Class package	Method name
Read Contact	android.content.ContentResolver	query
Networking	java.net.Socket	<init>
File	android.app.ContextImpl	openFileInput
	android.app.ContextImpl	openFileOutput
	java.io.File	<init>
Camera	android.hardware.camera2.CameraManager	openCamera

6.7 Chapter Summary

In this chapter we have presented AppBox , a novel app-level mobile app management (MAM) solution for stock Android particularly designed 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 7

Identifying and Evading Android Sandbox Through Usage-Profile Based Fingerprints

Android sandbox is built either on the Android emulator or the real device with a hooking framework, thus fingerprints of the Android sandbox could be used to evade the dynamic detection. In this chapter we present our investigation to address the question of which artifacts and how they are actually implemented by online analysis services. We first conduct a measurement on eight Android sandboxes and find that their customized usage profile (e.g., contact, SMS) can be fingerprinted by attackers for evading the sandbox. From our measurement results, most Android sandboxes have empty usage profile fingerprints, or fixed fingerprints, or random artifact fingerprints. So, without protections on such user profiles, Android malware can identify these fingerprints that associate with different sandboxes and hide its malicious behaviors. At last, we propose several mitigation solutions trivial to implement, including generating and feeding random real usage profiles to the malware sample every time, as well as a hybrid approach, which combines both random and fixed usage profiles.

7.1 Introduction

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 [112, 133, 80] 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 [103] 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 chapter, 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 [146] 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 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.

The remainder of the chapter is structured as follows: in Section 7.2 we introduce background and motivations underlying our research, then in Section 7.3 we discuss our system design and in Section 7.4 we present collected results which effectively shows the effectiveness of our techniques. Proposed mitigations and related work are discussed in Section 7.5 and Section 7.6 respectively. Finally, Section 7.7 concludes.

7.2 Background and Motivation

In this section, we first describe the current status of the Android sandbox for detecting Android malware, and then present the existing evading techniques commonly used in Android malware. In this paper, we explore a new direction: distinguish the environment where the malware is running and focus on bypassing the sandbox though a new type of fingerprint: the usage-profile based fingerprint which basically exploits the content generated by real users. Though the measurement in section 7.4, we show that most sandboxes can be bypassed by using this kind of fingerprint.

7.2.1 Android sandbox

The Android sandbox is a running environment with a hooking framework, either on the real device or on the emulator. With the hooking framework, the Android sandbox can check the suspicious API calls or system calls, which are used to make a verdict for the app. Many hooking frameworks and approaches have been proposed in past years, including the static instrumentation, [Xpo, ADB**?**]. Different hooking frameworks can hook different levels of behaviors. Xposed can only hook the Dalvik instruction call by replacing the Zygote process. Adbi can hook system calls by the inline hooking. ArtDroid permits to intercept calls to Java virtual-methods. These hooking frameworks can also be implemented in the real device for detecting malware, even if malware applies the anti-emulator technique.

The automated UI interaction is the most important technique used in the Android sandbox to improve its coverage. One approach commonly used in most Android sandboxes is using the MonkeyRunner tool to generate random user interactions and system events to the sandbox. But, the random events are very limited on triggering all malicious behaviors of apps. To address this problem, SmartDroid [**?**], AppsPlayground [**?**], CuriousDroid [109] propose different approaches to improve the automatic capability of UI interaction.

7.2.2 Evading technique

To evade the dynamic analysis in the Android sandbox, Android malware adopt many evading techniques, which can be categorized as two aspects: 1) anti-emulator. 2) anti-interaction.

Anti-emulator techniques [**?** 112] are used widely in most Android malware samples. The anti-emulator technique comes from identifying the difference of system and device information between the emulator and the real device. As the emulator is a default customized Android system, some system information is assigned with default values, e.g. the IMEI in the emulator is a string of zeros by default. The device information includes the information of sensors and sim card. Because of emulating these device information, attackers can find the difference from the emulated data. For anti-anti-emulator, the Android sandbox can send some artifacts to malware. However, there is an arm race when building the Android sandbox on the emulator, namely, the anti-emulator and anti-anti-emulator arm race. A countermeasure for anti-emulator techniques is building the Android sandbox on real devices.

Anti-interaction [57] technique in essence, tries to tell the identity of the current app controller (human user or automated exploration tool), by finding intrinsic differences between human user and machine tester in interaction patterns.For efficiency, exploration tool injects simulated user events and avoids accessing the underlying devices. Such simulated events and hardware generated ones are inconsistent in most cases. Also, to achieve high coverage of execution paths, exploration tool tends to trigger all valid controls, among which some are not supposed to be triggered by human.

7.2.3 Motivation

Even though the Android sandbox can be built on the real device, the Android malware can still evade its dynamic analysis through identifying the usage-profile based fingerprint, which reflects the real user information or the fake information, such as the contact list, SMS and the installed app list. Such usage-profile based fingerprints could be potentially leveraged to evade the Android sandbox by attackers. If the fingerprint contains fixed deterministic data or is empty or an artifact data which is not obviously created by users (e.g., "abcdefg" for the name in the contact list), Android malware can directly identify the sandbox environment and hide its following malicious behaviors. Even if all fingerprints look exact the same as for the fingerprints in a user's real device, there still be a way to identify the Android sandbox if the fingerprint is fixed or not random enough in every running time. The attackers can send a lot of probing apps to the sandbox of AVs, extract all pesudo-random fingerprints and build a general pattern for this sandbox. Later, other malware can use this pattern to evade the Android sandbox.

Therefore, in Section 7.4 we first conduct a measurement on fingerprints of some popular sandboxes, and then we conduct a study on whether these sandboxes protect their fingerprints for preventing themselves from evading.

7.3 System Implementation

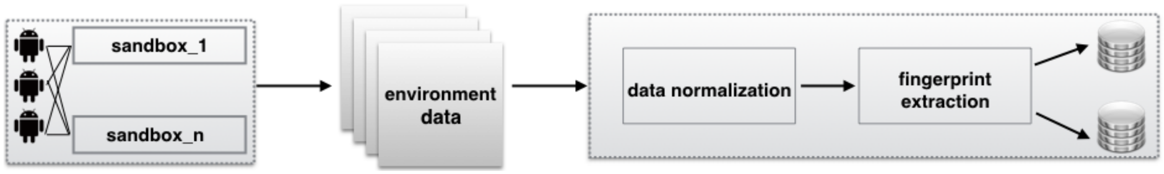


Fig. 7.1 The design of fingerprint collector

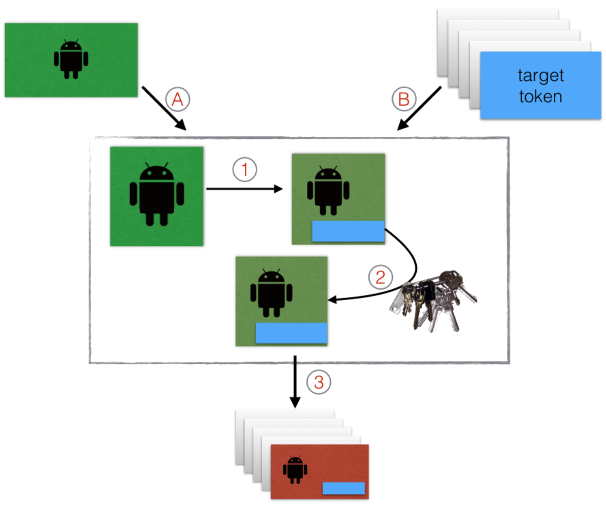


Fig. 7.2 Scouting apps generator

In this section, we present the design of the fingerprint collector. As depicted in Figure 7.1, 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 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.

Table 7.1 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

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 7.1.

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 7.2 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.

7.4 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 7.2. 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 7.2 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 [132, 112, 105, 125], we find that just few of these online services are available at the time of writing, as evicted by third column of Table 7.2.

We use the system described in Section 7.3 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 7.2. Table 7.3 and Table 7.4 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 7.3 we reported name, number and email were collected by the analysis, instead in Table 7.4 we included sending number and body if any. We

Table 7.2 Mobile Sandboxes employed for evaluation. (✗ means not available at the time of writing)

Type	Sandbox	Available
Online	Andrubis [137]	✗
	SandDroid [129]	✓
	TraceDroid[131]	✗
	CopperDroid[128]	✗
	HackApp [app]	✗
	NVISO ApkScan [nvi]	✓
	Koodous [koo]	✓
	VirusTotal [130]	✓
	Joe Sandbox mobile [30]	✓
	ForeSafe	✗
Antivirus	Bit Defender	✓
	360 mobile	✓
	TrendMicro	✓
	Kaspersky	✓
	Tencent mobile	✓
App store	Amazon [ama]	✓

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 7.3, 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" column in Table 7.3 and 7.4, 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 7.5 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 7.6 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 7.3 Contacts usage-profile Results

Sandbox	Name	Phone	Email	Random data	num. of results
store_1	Mary Edwards	867-5309	✗	✗	2
	Harry Grace	867-5319	✗		
online_x	✗	✗	✗	✗	1
online_y	Firstname1	1 301-234-5678	✗	✗	3
	Firstname2	1 381-234-5678	✗		
	Firstname3	1 381-234-5678	✗		
av_1	Ion	074-354-3219	✗	✗	4
	Gheo	072-345-6789	✗		
	Txet4321	074-212-3456	✗		
av_2	Cynthia	✗	✗	✗	2
	Alexander	✗	✗		
	Alexandra	✗	✗		
	Dolores	✗	✗		
av_3	MARS	1 566-666-6666	chengkai_tao@****.com.cn	✗	10
av_4	Boulder Hypnotherapy Ctr	(303) 776 8100	z**y@gmail.com	✓	36
	St. John Ambulance	061 412480	l**k@stjohn.ie		
	Maidstone Golf Centre	01622 863163	nick@totalgolfcoaching.co.uk		
av_5	Jian Li	1 3743888229	lijxev@admin.cn	✓	20
	Jian Li	13606500401	b0***54@admin.cn		
	Xuri Jin	13250324837	55**43@yuepao.cn		

Table 7.4 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 Ffffffffmmmmmm	✗	12
av_4	10668820 10086 13770837893	guchulaichonghuafei nihao,laikaitong4g-songliuliango nihao,nishi4staryonghu,keyihuantingji	✓	15
av_5	13540877911 15874984303 18660928896	u0oydyemvub4lu86kfcbwad46pvhmh6o 2fqjfkr629blowmxso4jh6dzqtk3f4j2 lhb0j8hxi48wdjiua1q0qvsleeffgt6g	✓	22

Table 7.5 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 7.6 Installed Apps usage-profile Results

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

7.5 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 7.4, 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 [50, Xpo, Cyd, 5].

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.

7.6 Related Works

A couple of previous research works focus on evading the Android sandbox or Android anti-virus scanners. Huang, et al. [76] 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 [96] 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 [133] 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. [57] 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 [43] 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.

7.7 Chapter Summary

In this chapter, we presented 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 8

OctoDroid: Discovering Vulnerabilities in Android System Services via Code Property Graphs

The Android Framework is a fundamental component of the Android system. The entire Android OS could not operate without it. The Framework is responsible for critical system services (i.e., ActivityManager, Wifi Service, etc.) and it manages app's components (i.e., ContentProvider). Any installed app can communicate with the system services either via Java middleware or directly via Binder IPC. The implementation of the Android Framework is cross-layers, it resides both at Java and native level (mainly C++ code). This characteristic makes the framework a potential exploitable channels for application to reach the inner parts of the system, this a natural target for memory corruption vulnerabilities. Moreover, system services run in privileged process which may lead to a privilege escalation attack.

This chapter presents *OctoDroid*, a plugin for code analysis that combines the benefits from the powerful Clang's C++ AST parser with the Octopus analysis platform powered by Code Property Graph (CPG) representation. Analyzing object-oriented programming language, such as C++, presents complex challenges due to the features such as polymorphism and inheritance. To allow efficient exploitation of the CPG employed by Octopus, we designed and implemented a simple yet effective analysis algorithm that leverages on class hierarchy analysis (CHA). The augmented code produced by OctoDroid allows Octopus to perform precise analysis of C++ Android system services codebase. We prototyped *OctoDroid* using the Clang LibTooling library and integrated it in Octopus as plugin. We evaluated *OctoDroid* on latest Android available at the time of writing[1]. Using our approach we have uncovered three previously unknown bugs, one of them a memory corruption reported and patched by the Android security team. Furthermore, we evaluated our approach on previously known vulnerabilities. Our work demonstrates that exploiting the Clang AST C++ parser we improved the Octopus analysis of C++ code enabling it to discover previously missed bugs.

8.1 Introduction

The Android OS has been dominating the mobile market since a few years, as of 2016 there were more than 1.4 billion active Android devices and over 65 billion Android apps installed so far. Android applications make use of the Android API offered by Android Application Framework in order to properly operate and consume services as offered by the Android system. The role of system services is quite crucial in terms of app's functionality, in fact those services offer fundamental capabilities such the WifiManagerService (in charge of managing Wifi operations), LocationManagerService (managing GPS communication). Nevertheless, all the user interface of Android apps (i.e., Activity) is controlled via the ActivityManagerService (AMS) which operates as a system service. Thus, the Android Framework represents a fundamental component participating in operative functionalities offered by the OS. On the other hand, it represents an interesting attack surface in order to conduct a privilege escalation attack over running system services. In this work we focus especially on native code (mainly C++) implementation of those services in order to spot buggy code, eventually.

System services employ as communication mechanism the Binder IPC, which in turn add an opaque layer when it comes to discovering vulnerability. In fact, the Binder IPC mechanism permits to invoke remote procedure as they were local, basically it is a client/server communication via Binder messages, so called *Parcel*. Serialized objects and methods are specified using Android Interface Definition Language (AIDL). To locate a particular service an Android app queries the *servicemanager* for the handle which is responsible to maintain the

[1]Android Oreo 8.1.0

service directory and to map an interface name to a Binder handle. Most of the Android system services just run as a thread hosted by the *system_server* process which executes with *system* privilege. Any *Parcel* being sent via Binder has been serialized by the Proxy (on the client side) and would be deserialized by the Stub (on the server side), this operations requires that both sides agree on the semantic of the object which is being serialized/deserialized, hence the adoption of the AIDL.

Several vulnerabilities affecting Android system services have been reported by Android security bulletins[2] over the past years proving that finding and mitigating vulnerabilities is a crucial task that requires continuous incremental analysis over the time (i.e., new code and features might introduce new bugs).

Yamaguchi et. al. proposed in [141] an innovative approach for modeling and discovering vulnerabilities employing Code Property Graphs (CPGs), the implementation named *Octopus* for C/C++ languages has also been released. Octopus aids the analyst discovering vulnerability by query traversal on the CPG stored in a graph database. The difficulty of parsing C++ is well-studied [**? ?**]. Octopus employs a fuzzy parser based on island grammars [98] which performs analysis on selected portion of the code rather than performing a detailed analysis of a complete source code. The fuzzy parser allows to continue the analysis even in case of parsing issues or in case of missing code portions. As natural consequence we noticed that Octopus presents very low detection rate when it comes to analyzing C++ code base, mainly as consequence of its parser's design which scarifies level of details in favor of tolerance. This has been shown to be a totally reasonable compromise. Although, our goal is to employ Octopus and its CPG based analysis for discovering vulnerabilities within Android system service codebase which consist of mostly C++ code.

In this work we propose OctoDroid , a practical analysis tool combining the precise parsing offered by Clang along with the graph-based analysis of Octopus platform. OctoDroid compensates the Octopus' fuzzy parser inherent incompleteness in analyzing C++ code. OctoDroid allows to automatic modeling and discovering vulnerability in Android system services codebase. Differently from existing approaches based on fuzzing, we focus on static exploration via CPGs which allows to traverse the produced graphs in order to query for specific pattern that

[2]https://source.android.com/security/bulletin/

may lead to vulnerabilities. We propose an uncouple design which builds the full inheritance graph analyzing the codebase via Clang, then exploit collected information to enhance the static analysis employed by Octopus platform analysis.

We concretely show how OctoDroid can easily detect various already known security bugs in Android system services and we demonstrate how OctoDroid minimize the manual effort requested . We further show OctoDroid effectiveness by considering recent uncovered vulnerabilities in native code which lead to memory corruption in system services process.

This work makes the following contributions:

- To our knowledge, OctoDroid is the first CPG-based tool that aims to enhance Octopus analysis capabilities specifically for automatic modeling and discovering of vulnerability in C++ code, in particular we targeted the Android system services codebase.
- Unlike previous existing works, OctoDroid provides effective and practical analysis of Android system services codebase by means of Code Property Graph representation. OctoDroid takes the advantage of Clang's full parsing approach to enhance the CPG by class hierarchy information.
- We have implemented and evaluated OctoDroid on the latest Android Orio codebase available at the time of writing. Our results show its effectiveness in discovering new vulnerabilities.
- We release OctoDroid as opensource to

.

The remainder of this chapter is organized as follows: Section 8.2 briefly introduces the background of our approach. Next, we presents a motivating example highlighting the problems we want to address and the current limitation of Octopus' analysis platform when it comes to C++ language in Section 8.3. We proceed describing OctoDroid methodology and design in Section 8.4 and Section 8.5 respectively. We show and discuss evaluation results in Section 8.6 and compare related works in Section 8.7. Finally, we conclude our work in Section 8.8.

8.2 Background

In this section we introduce few basic concepts that help to understand our scenario and contributions. First, we introduce the Clang parsing library we have used to extract the AST. Second we give a brief introduction on object-oriented languages, C++ in particular, and why they are challenging to parse by the Octopus fuzzy parser. Then, we provide few fundamental concepts about Binder and its cross-layers (Java and native) functionality. Finally, we introduce the CPG based analysis approach as presented by Yamaguchi et. al. in [141].

8.2.1 Clang and LibTooling

To build a precise AST we exploit *Clang*, an open source compiler front-end for the C,C++,Objective-C and Objective-C++ programming languages []. Clang uses LLVM as backend. In particular, we employed LibTooling, a Clang C++ library which allows to construct tools that leverage on the Clang parsing front-end. Libtooling is able to extract both syntactic and semantic information about a program by accessing to the Clang abstract syntax tree (AST). Several papers have been published showing the benefits of Clang AST in parsing C++ code [].

8.2.2 Object-oriented language

As most of the Android system services codebase is written in C++, we give a brief description of those challenges faced by Octopus fuzzy parser when it comes to object oriented languages as C++. A major advantage of object-oriented languages is abstraction. One of the most important feature is that it allows dynamic dispatching of methods based on the runtime type of an object. In C++ language programmers must explicitly request dynamic dispatching by declaring a method to be *virtual*. The difficulty of parsing C++ code is well-studied[**? ?**], a comprehensive description of those is beyond the scope of this work. What is more important for the research presented in this work is that for a program without virtual function calls (or function pointer) a complete graph can be produced, instead when virtual functions are in place, each virtual call site has multiple potential targets.

Android system services implementation consist of several classes which extend and provide different capabilities, resulting in a massive usage of virtual function calls. This represents the main limitation of Octopus's fuzzy parser when analyzing a C++ codebase.

8.2.3 Android System Services

Android framework plays a key role in Android OS, it provides a collection of system services which provide crucial features that are accessed by the app developers via Java-layer middleware. The transport for service (and all inter-app) communications in Android is the *Binder* mechanism, accessed via /dev/binder, which is in charge with not only dispatching messages, but also with passing around descriptors, objects, and more, as well as providing reliability and security. Objects passed thought Binder are actually handles to the remote object, hence they can not be just recorded as raw data because they become meaningless once out of the transaction context. All framework services are invoked following the same invocation pattern and the Android Interface Description Language (AIDL) is employed to provide the interface exported by those services.

The Binder high-level Java implementation resides in *Android.os.Binder*[3] which represent the base class for a remotable object and implements the IBinder[4] interface. Most developers will not implement this class directly, instead using the aidl tool to describe the desired interface, having it generate the appropriate Binder subclass. The key IBinder API is transact() matched by Binder.onTransact(). These methods allow to send a call to an IBinder object and receive a call coming in to a Binder object, respectively. The data sent through transact() is a Parcel, a generic buffer of data that also maintains some meta-data about its contents. The meta data is used to manage IBinder object references in the buffer, so that those references can be maintained as the buffer moves across processes.

The Binder native implementation offers several interfaces and classes: IBinder, BpBinder, BBinder, IInterface, BpInterface and BnInterface. Respectively, IBinder[5] defines common interfaces that will be shared among all subclasses,

[3]http://androidxref.com/8.0.0_r4/xref/frameworks/base/core/java/android/os/Binder.java
[4]http://androidxref.com/8.0.0_r4/xref/frameworks/base/core/java/android/os/IBinder.java
[5]http://androidxref.com/8.0.0_r4/xref/frameworks/native/libs/binder/include/binder/IBinder.h

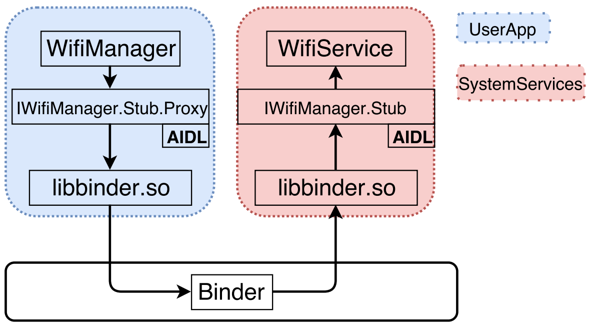


Fig. 8.1 Android Binder & System Services

BBinder[6] is the base class for all server implementation, normally on the server side it implements the onTransact() function in BnInterface subclasses, BpBinder[7] holds the remote server handle at client side, it is the base class for all client implementations. The latter interfaces (IInterface, BpInterface, BnInterface) contain code to establish communication between IBinder and business class implementation code, acting as some sort of language glue. The coding pattern followed by those interface result in using *Bp* prefix for client side proxy classes and *Bn* for those implementing server proxy object class. This pattern is quite common but not mandatory, in fact there are services' class which does not follow this pattern (i.e., WifiService). Despite the reasonable intuition that interesting vulnerabilities may reside only in the server side, it really depends by in which process the client side implementation is running. In fact, in services like mediaserver it is quite common that the client implementation resides in privileged process.

The high-level interaction between system services and user apps is pictured in Figure 8.1 whereas the green and the red dashed boxes represents respectively the user app process and the WifiService process space. The user app invokes the Android framework API in order to consume Wi-Fi functionalities via WifiManager. The WifiManager holds the handle for the Binder interface IWifiManager, controlling and quering Wi-Fi connectivity operations are forwarded via IWifiManager.Stub.Proxy (Stub and Proxy classes are usually generated via aidl tool). Finally, data serialization occurs in libbinder which is also in charge to instantiate the Binder communication. The Binder component extracts information attached

[6]http://androidxref.com/8.0.0_r4/xref/frameworks/native/libs/binder/include/binder/Binder.h#27
[7]http://androidxref.com/8.0.0_r4/xref/frameworks/native/include/binder/BpBinder.h#27

in the transaction sent by the client in order to identify the requested service. The WifiService which is responsible to handle control and query operations made by the clients would reverse the serialization and pass the data to the WifiService implementation via the IWifiManager.Stub. The WifiService is the component which implements the actual business logic.

8.2.4 Code Property Graphs

In what follows we briefly review some basic concepts related to CPG based approach. Octopus employs a fuzzy parser to analyze the source code and build the Code Property Graphs (CPG) which is then stored by a graph database. The CPG contains different code representations, the abstract syntax tree (AST), the control flow (CFG) and program dependence graph (PDG) of source code. All those representation (AST, CFG and PDG) compose a CPG which formally is an edge-labeled, attributed multigraph [**?**]. Octopus employs a fuzzy parser based on island grammars [98] which performs analysis on selected portion of the code rather than performing a detailed analysis of a complete source code. The Octopus's parser goal is to parse as much as code as possible with graceful fails whenever a portion is missing but without stopping the process before complete the parsing on all the input code. To leverage this representation, Octopus expresses search pattern as graph traversals by means of the Gremlin [120] programming language.

The CPG provides a inter-procedural code property graph, as detailed in [143] tuned to be processed using graph database queries and augmented with syntax and dominance information.

The CPG along with the Gremlin language for graph database provides an efficient solution to model taint style vulnerability as graph query as the one shown in Figure 8.2. Gremlin allows to achieve a very powerful query mechanism by means of what is called traversal, basically it allows to define how to navigate within the graph according to specific properties. Figure 8.2 illustrates three different traversals which are employed to model a typical taint-style vulnerability where the traversal *getCallsTo* discovers all call-site of *sink*, then analyse each sink call site separately using the traversals *taintedArgs* and *unchecked* respectively. All definitions that match the descriptions are passed to the traversal *unchecked* which is in charge for analysis of security checks along the way to the sink. As

```
getCallsTo(sink)

.taintedArgs(
        [arg1Source,..., argnSource]
)

.unchecked(
        [arg1Sanitizer, ... argnSanitizer]
)
```

Fig. 8.2 Template for taint-style vulnerability as a graph traversal in the query language Gremlin

discussed in [143], this approach permits to define graph traversal that can be reused on different codebase in order to mine code for bugs.

8.3 Motivating Example

In this section we present a motivating example to describe the overall problem in analyzing C++ by the Octopus analysis platform. In particular, by this example we want to describe the overall Octopus fuzzy parser limitation especially when virtual function calls are in place (polymorphic call site).

Figure 8.3 shows a trivial program which helps in illustrating the object-oriented features such dynamic-binding. This motivating example aims to show Octopus difficulty in detecting the inter-procedure data flow from the *source* function to the *sink* function. In particular, both the functions *source* and *sink* are present in both classes *A* and *B* as well as the function *parser*. Note that only the *sink* implementation offered by B is vulnerable. As for classic taint style vulnerabilities, the source produces an attacker controllable value that is then consumed by the sink, in our example we simulated a memory corruption vulnerability (line 28) in the *B::sink* function.

Octopus's fuzzy parse tokenizes code in 8.3 for building the AST, as well as the other representations, without collecting type-wise information, in fact its main goal is parsing without failing. When performing a query via the Octopus's analysis platform, the analyst must specify both the interesting source and sink functions. As disadvantage of the imprecise parsing performed by Octopus, the data flow analysis is missing the type information. Thus, will return any call-site containing the string *sink* as the correct one for the function *B::sink*. This is obvious an outcome that does not provide any useful information to the analyst.

In what follows, we provide an example where the analysist provides the proper name of the desired sink function (*B::sink* in our example) but the Octopus analysis platform is not able to detect the proper data flow going from the source to the sink in Figure 8.3 (line 37 to line 33). Hovewer, for this trivial example the analyst may provide *sink* as sink function's name but in this case the analysis would retrieve as sink function any function name containing the string *sink* and as its call-site any one matching the same string, regardless the type of the object involved in the call site. This obvious does not scale for real-world C++ codebase, in fact it increases the false positive rate significantly as well as causing performance degradation if the selected function's name is a common one within the whole codebase. (i.e., *copy*).

```
class A {
  public:
      virtual int source();
      virtual void sink(int x);
      virtual void parser(int k);
};
class B : public A {
  public:
      virtual int source() override;
      virtual void sink(int x) override;
      virtual void parser(int k) override;
};
int A::source(){
  return 0xdeadbeef;
}
void A::sink(int x ){
  // NOT VULNERABLE
  printf("sink A %d\n", x);
}
void A::parser(int k){
    sink(k);
}
int B::source(){
  return 0xdeadbeef;
}
void B::sink(int x ){
  // VULNERABLE
  char buf[10];
  printf("sink B %d\n", buf[x]);
}
void B::parser(int k){
    int x = k + 1;
    sink(x);
}
int main(int argc, char** argv){
  B* p = new B;
  int res = p->source();
  p->parser(res);
  return 0;
}
```

Fig. 8.3 Trivial code illustrating object-oriented features

Figure 8.4 presents an Octopus query for an inter-procedure traversal which performs backward analysis starting from the sink to the source. Unfortunately, even if the query is modeling the exact vulnerability presented in Figure 8.3 it does not return any data-flow. The *getCallsTo* function (line 2) is in charge to traverse the graph looking for any call-site matching the regular expression string, $B :: sink.*$ provided as argument. Unfortunately, there is none explicit call matching that pattern, then the analysis is stopped at this stage because for the given sink none call-site was found. If the analyst specify as sink function name the string $sink.*$, then the Octopus platform returns very imprecise results. In fact, any call-site containing the string *sink* would match for both the functions named *A::sink* as well as *B::sink*.Moreover, the string $sink.*$ would match also the call-site at line 21 producing a false positive because that function is never called in our example. We observed that this outcome is not suitable for huge codebase, as Android AOSP, especially when polymorphic call sites are in place.

```
asink=".*B::sink.*"

getCallsTo(asink)
.taintedArgs([sourceMatches("B::source.*")])
.unchecked([ANY_OR_NONE])
```

Fig. 8.4 Octpus query for 8.3

8.4 Methodology

OctoDroid aims to discover vulnerabilities in Android system services by means of CPG analysis employed via Octopus analysis platform. In this work we will focus on a specific kind of vulnerability so called *taint style*. In the class of taint-style vulnerabilities, attacker-controlled data is passed un-sanitized from an input source to a sensitive sink. Hence, this procedure requires to accurately identify the callee and its call-site, in order to propagate the data flow to the correct destination. As taint-style vulnerability class fits many common security defects well, including buffer overflow and other memory corruption flaws. In this work we will focus on undercovering vulnerabilities which belong to this class. For more details about how Octopus deals with this class of vulnerability refer to [141, 143, 111].

Our analysis is composed by two stages. First, we build the Class Hierarchy graph [**?**] by means of Clang's LibTooling library [**?**]. Then, we use LibTooling again to enhance the code with the information collected in the previous stage. As result, the augmented code would contain virtual function calls replaced with their corrispective explicit form(s), whenever it is possible. The augmented code can now be processed by the Octopus's fuzzy parser and then analyzed leveraging on the type information which has been written esplicitely in the augmented code. It is worth noting that OctoDroid on one hand benefits from the clang's Libtooling capabilities and on the other hand exploits the Octopus's fuzzy parser and analysis platform to perform query on the augmented code. The augmented code might be not compilable anymore, as we will explain in the following, but the Octopus' fuzzy parser will be still able to complete its parsing processing the code enhanced with type information.

OctoDroid 's methodology enables to identify links among business code implementation (Bn) and its counterpart in the client side (Bp), those calls most of the times appear as virtual function call. OctoDroid builds a dependency and inheritance graphs according to the information extracted via Clang. We begin describing how we leverage on the Clang's LibTooling library. Then we continue introducing the challenges we needed to overcome in order to enhance the analysis carried out via Octopus platform.

Clang's LibTooling C++ library provides access to the AST representation and supports the construction of tools that leverage the Clang parsing front-end. The Class Hierarchy Analysis (CHA) [**?**] allows to use the combination of the statically declared type of an object with the class hierarchy of the program in order to determine the set of possible targets of a virtual function call. In Figure 8.3, *p* is a pointer whose static type is *B* *, which basically means that *p* can point to objects whose type is *B* or any of *B*'s derived classes. Analyzing the Class Hierarchy graph and combining its information we can infer that there are no derived classes of *B* so that the only possible target of the first call (line 37) is *int B::source()*, for the call at line 38 we can infer the only target is *B::sink* as well. In order to conduct a proper CHA the complete program source code must be available, any missing piece of code could produce imprecise results.

Figure 8.5 shows the augmented code obtained by OctoDroid , due to space issue only the relevant code modifications were reported. Performing the analysis via Clang OctoDroid collected the type information used for transforming the virtual calls into an explicit form. It is worth noting that we do not aim to optimize the source code or any similar tasks in charge of a compiler. In fact our goal is to make the Octopus data-flow analysis able to recognize those flows involving a virtual function call which OctoDroid has now turned into direct calls.

First we describe how OctoDroid improves the Octopus analysis capability as for the motivating example shown in Section 8.3. Second, we continue providing a more complex example where the object dynamic type cannot be resolve statically and we show how OctoDroid performs its analysis in those cases.

As shown by Figure 8.5, the virtual function calls at lines 37 and 38 Figure 8.3 has been transformed into an explicit call to *B::source* and *B::parse*, respectively. Also the virtual function call at line 21 Figure 8.3 was turned into the direct form *A::sink*. The Octopus platform provided with this augmented code version is then able to detect the correct data-flow as requested by the analyst, from the source function named *B::source* to the sink *B::sink*. In fact, executing the the query in Figure 8.4 the precise result provided by platform contains only the correct vulnerable flow, none false positive result is returned.

Challenges. As shown by Figure 8.6 we modified the previous example employed in Section such to reflects a real-world scenario where code complexity is much more than that shown in the previous trivial code from Figure 8.3. We added the function named *bridge* that performs dynamic-binding calling a virtual function defined by the object received as argument. This new example in Figure 8.6 will drive the following discussion about the challenges when the static type is a superclass of the dynamic type observed at runtime.

```
void A::parser(int k){
    A::sink(k);
}
int B::source(){
  return 0xdeadbeef;
}
void B::sink(int x ){
  // VULNERABLE
  char buf[10];
  printf("sink B %d\n", buf[x]);
}
void B::parser(int k){
    int x = k + 1;
    B::sink(x);
}
int main(int argc, char** argv){
  B* p = new B;
  int res = B::source();
  B::parser(res);
  return 0;
}
```

Fig. 8.5 Trivial code in 8.3 transformed by OctoDroid

```
static void bridge(A* p){
    int res = p->source();
    p->parser(res);
}
```

Fig. 8.6 strong dynamic dispatching

The dynamic type information for the object received as argument by the function *bridge* could be either A or any classes which is extending it. We can not infer its type statically, then in this case we can provide a superset of its possible targets. According to the class hierarchy graph we can derive those classes that extend from A, any of those is a candidate target. We further analyze all found classes inspecting whether they define a virtual method named as the target one, this simple but practical heuristic allows to reduce the number of candidates without loosing any precision. The set of candidates collected by OctoDroid may include object types which would not be observed at runtime, this is a natural disadvantage of our static analysis approach. Although, on the other hand it provides Octopus the ability to perform a query traversal on C++ code as precisely as the analyst required for. In the following we details the augmented code for 8.3. The augmented code produced by OctoDroid is shown by Figure 8.7. The virtual function calls in function *bridge* have been replaced with all the good candidates found by OctoDroid , as we mentioned before this is a super set of the possible targets. Being a super set does not impact on the overall analysis platform system, when the analyst executes the query in 8.4 she will get the intended flow from *B::source* to *B::sink* regardless that the function *bridge* contains call sites that might not appear at runtime (lines 2 and 3 Figure 8.7).

```
1 static void bridge(A* p){
2     int res = A::source();
3     A::parser(res);
4     int res = B::source();
5     B::parser(res);
6 }
```

Fig. 8.7 Trivial code in 8.6 transformed by OctoDroid

Once we have explained how we combine clang powerful parsing capabilities along with graph-based Octopus analysis platform, we proceed detailing the mechanics of OctoDroid .

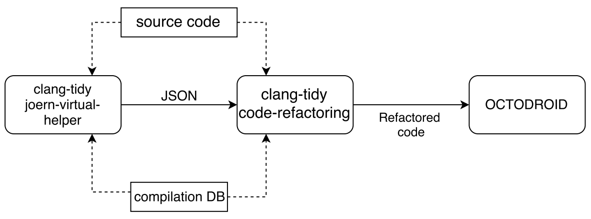


Fig. 8.8 Octodroid design

8.5 OctoDroid

In this section we describe how we designed OctoDroid and integrated it in couple with Octopus analysis platform. Figure 8.8 depicts the two Clang modules and how they participate in producing the augmented code. The first module named *joern-virtual-helper* is implemented as clang-tidy module [] it is in charge of building the class hierarchy graph. It requires the compilation database and the source code as inputs. The former is a json file containing all the files processed by the clang compiler, to create this file we compiled Android AOSP proving the appropriate flags for creating the clang compilation database. This file is requested to successfully complete the Libtooling AST parsing, any include or definition must be resolved. In fact, because of the clang's full parsing approach, the analyzed codebase must be compilable, in contrast with Octopus fuzzy parser that is able to parse even not compilable ones. However, this is a requirement satisfied by Android AOSP. The second module in Figure 8.8, *joern-code-refactoring* takes the output of the first module and the source code again, producing as output the augmented version. Finally, the augmented code is provided to the Octopus analysis platform.

8.6 Evaluation

We proceed to evaluate OctoDroid by means of two concrete vulnerabilities reported to Android security team and discovered by member of KeenTeam by Tencent. The first one, CVE-2015-6612[8] is a heap overflow found in ICrypto system service implementation. The second one has been found by *OctoDroid* powered analysis, CVE-2017-0712 and is a stack overflow in *startScan* function exposed by the WiFiManager system service. These vulnerabilities highlights the importance of properly identify the callee function as well as some difficulties involved.

The Figures 8.9, 8.10 and 8.11 show the taint-style vulnerability, the data flow starts in *BnCrypto::OnTransact*, goes to *Crypto::decrypt* and finally reaches the vulnerable function *CryptoPlugin::decrypt*. The *BnCrypto::OnTransact* function shown by Figure 8.9 is defined in ICrypto.cpp and it is in charge to process an incoming Binder transaction. This function can be called by any user apps by means of the Binder mechanism, as we discussed in Section **??**. If the being processed transaction type is *DECRYPT*, the attacker controllable data is read by *data.read* at line 13 into a heap allocated object of type *CryptoPlugin::SubSample*. Then, at line 18 the *decrypt* function is called. This is a virtual function call, any class that extends the *BnCrypto* and implements the virtual *decrypt* function is a candidate target. The vulnerable flow continues to *Crypto::decrypt* function (*Crypto* extends *BnCrypto*), Figure 8.10, at line 10 a virtual function call to *mPlugin->decrypt* invoke the final vulnerable *CryptoPlugin::decrypt* function. The *CryptoPlugin* function contains a for loop over the *subSamples* array and within the loop it calls the memcpy function, line 8. The attacker controllable value *subSample.mNumsBytesOfClearData* is passed as length argument to memcpy, which leads to heap memory corruption.

To discover this vulnerability which is a taint-style one, Octopus lacks the information needed to follow the data-flow across the different virtual function call sites, as we detailed in Section 8.3. *OctoDroid* exploits the class hierarchy graph to produce the augmented code shown in Figures 8.12 and 8.13. The class hierarchy graph enables to extract all those classes extending *BnCrypto* and for those matching the target virtual method, *decrypt*, *OctoDroid* rewrites the virtual call into a direct call as shown in Figure 8.12 line 8. Then *OctoDroid* exploits type-based reference analysis to retrieve the type information for the

8

```
 1 status_t BnCrypto::onTransact(
 2     uint32_t code, const Parcel &data, Parcel *reply, uint32_t flags) {
 3     switch (code) {
 4     [...]
 5         case DECRYPT:
 6         {
 7             CryptoPlugin::SubSample *subSamples =
 8                 new CryptoPlugin::SubSample[numSubSamples];
 9             data.read(
10                     subSamples,
11                     sizeof(CryptoPlugin::SubSample) * numSubSamples);
12             [...]
13             AString errorDetailMsg;
14             ssize_t result = decrypt (
15                     secure,
16                     key,
17                     iv,
18                     mode,
19                     srcData,
20                     subSamples, numSubSamples,
21                     secure ? secureBufferId : dstPtr,
22                     &errorDetailMsg);
```

Fig. 8.9 ICrypto.cpp. Binder related *BnCrypto::decrypt*

```
 1 ssize_t Crypto::decrypt(
 2         bool secure,
 3         const uint8_t key[16],
 4         const uint8_t iv[16],
 5         CryptoPlugin::Mode mode,
 6         const void *srcPtr,
 7         const CryptoPlugin::SubSample *subSamples, size_t numSubSamples,
 8         void *dstPtr,
 9         AString *errorDetailMsg) {
10     return mPlugin->decrypt(
11             secure, key, iv, mode, srcPtr, subSamples, numSubSamples, dstPtr,
12             errorDetailMsg);
13 }
```

Fig. 8.10 Crypto.cpp. Crypto implementation of the *decrypt* virtual function

virtual function call at line 10 Figure 8.10 and transforms it into the corrispective direct call to *CryptoPlugin::decrypt*, as shown by Figure 8.13.

To discover this taint-style vulnerability via a query traversal we specify as sink the third argument to **??** and as source the first parameter of *data.read*. The analysis begins at the sink and the producer of the tainted variable is retrieved , repeating the process until either the function entrypoint or the target source is reached. In 8.11 the analysis continues up to the function entrypoint, because *subSamples* is an argument, then the analysis continues in function's callees. The callee is retrieved by looking for any call site statement matching the target function name, which is *CryptoPlugin::decrypt*. The augmented code provided by *OctoDroid* allows Octopus analysis platform to identify the proper

```
ssize_t CryptoPlugin::decrypt(bool secure, const KeyId keyId, const Iv iv,Mode mode,
    const void* srcPtr,const SubSample* subSamples, size_t numSubSamples,void* dstPtr
    , AString* errorDetailMsg) {
    if (mode == kMode_Unencrypted) {
        size_t offset = 0;
        for (size_t i = 0; i < numSubSamples; ++i) {
            const SubSample& subSample = subSamples[i];
      [...]
            if (subSample.mNumBytesOfClearData != 0) {
                memcpy
                  (reinterpret_cast<uint8_t*>(dstPtr) + offset,
                  reinterpret_cast<const uint8_t*>(srcPtr) + offset,
                  subSample.mNumBytesOfClearData);
                offset += subSample.mNumBytesOfClearData;
            }
        }
        return static_cast<ssize_t>(offset);
        [...]
```

Fig. 8.11 CryptoPlugin.cpp. CryptoPlugin implementation of the *decrypt* virtual function

data-flow involving the two functions in Figure 8.12 and 8.13. The call site for *CryptoPlugin::decrypt* is found at line 9 Figure 8.10, then the respective callee of *Crypto::decrypt* function is identified at line 8 Figure 8.12, reaching then the selected source. The traversal query that allows to discover the vulnerability is presented in Figure 8.14, without the augmented code provided by OctoDroid this query would return and empty set of findings.

```
status_t BnCrypto::onTransact(                                                  1
    uint32_t code, const Parcel &data, Parcel *reply, uint32_t flags) {         2
    switch (code) {                                                             3
    [...]                                                                       4
        case DECRYPT:                                                           5
        {                                                                       6
      [...]                                                                     7
            ssize_t result = Crypto::decrypt(                                   8
                    secure,                                                     9
                    key,                                                        10
                    iv,                                                         11
                    mode,                                                       12
                    srcData,                                                    13
                    subSamples, numSubSamples,                                  14
                    secure ? secureBufferId : dstPtr,                           15
                    &errorDetailMsg);                                           16
```

Fig. 8.12 *BnCrypto::onTransact* transformed by *OctoDroid*

```
ssize_t Crypto::decrypt(
        bool secure,
        const uint8_t key[16],
        const uint8_t iv[16],
        CryptoPlugin::Mode mode,
        const void *srcPtr,
        const CryptoPlugin::SubSample *subSamples, size_t numSubSamples,
        void *dstPtr,
        AString *errorDetailMsg) {
    return CryptoPlugin::decrypt(
            secure, key, iv, mode, srcPtr, subSamples, numSubSamples, dstPtr,
            errorDetailMsg);
}
```

Fig. 8.13 *Crypto::decrypt* trasformed by *OctoDroid*

```
arg3Sanitizer = {it, symbol ->
    conditionMatches(".*%s (<|<=).*", symbol)
}

getCallsTo("memcpy")
.taintedArgs([ANY, ANY, sourceMatches("data.read.*")])
.unchecked([ANY_OR_NONE,ANY_OR_NONE, arg3Sanitizer])
```

Fig. 8.14 Traversal query to discover the taint-style vulnerability in 8.13

8.7 Related Work

Android is an opensource project (AOSP)[9], consequently has received a significant attention from the security research community. In the past few years several works have been proposed to address privacy-related issue due to information leakage[59], various approaches were presented addressing the privilege escalation attack[? ?], offering security oriented sandbox for Android apps[38, 42, 40], other were proposed to enhance security in enterprise scenarios[123, 72]. A part of those approaches were employing static instrumentation as well as other were employing dynamic instrumentation techniques [50].

Discovering vulnerabilities is a typical computer security topic, consequently several works have been published over the past few years. A significant fraction of them were focused on discovering vulnerabilities in the application level, especially Java level code[92? ?]. Existing works addressing Android codebase (AOSP) employ a fuzzing strategic for discovering new vulnerabilities[? ?]. Feng et. al. presented in [?] BindCracker, an automating testing framework which supports parameter-aware fuzzing in order to assess Android system services robustness. As Binder transactions can transfer primitive type but also serialized objects, it is essential to achieve semantic comprehension on the data being sent via Binder. Without semantic comprehension, the fuzzing engine would not be able to properly apply mutations on the serialized objects were included in the transaction. To this end, BinderCracker first needs to record as many different valid Binder transactions as possible, then it applies mutation guided by dependency graph built by the recording transactions step. In [?] Guang Gong, security researcher from security firm Qihoo 369, presented a fuzzing system services strategy similar to BindCracker but without leveraging on previously recorded Binder transactions. A different approach is presented by Luo et al. in [?], which enables symbolic execution on Android Framework. Despite those approaches are quite valuable, they do no employ static code analysis as offered by CPGs representation.

The efficient of static analysis in assisting discovering vulnerabilities has been discussed in several previous works and has also shown its capabilities in aiding the fuzzing strategy in exploring deeper the code paths. In [?] Shastry. et al. proposed a staged approached,for discovering vulnerability by means of enhanced analysis via LLVM framework capabilities, but unfortunately they did

[9]https://source.android.com/

not published the code. Aiding fuzzing analysis via static code exploration has been proposed by Shastry et al. in [**?**]. It allows for managing, conducting, and assessing dictionary-based fuzzing testing and it supports Clang/LLVM instrumentation and the AFL ecosystem.

8.8 Chapter Summary

In this chapter we have presented OctoDroid to combine the benefits of Clang precise parsing with the Octopus graph-based analysis platform. OctoDroid allows to enhance the Octopus platform capability of analyzing C++ codebase, especially when virtual function calls are in place. By leveraging on class hierarchy analysis (CHA) offered by Clang's LibTooling library and its code refactoring features, OctoDroid provides a practical plugin for Octopus platform able to produce an augmented code that achieves a more precise analysis by the Octopus platform.

We have show that OctoDroid provides a relevant performance improvement for the detection rate in comparison with the one offered by Octopus during an execution without our plugin in place. We have performed a real-world evaluation by means of concrete known vulnerabilities reported by the Android bulletin as well as some unknown vulnerabilities we have found.

Chapter 9

Conclusions and Outlook

This dissertation has presented our research on countering privacy and security threats on the Android platform. We first investigated on improvements for dynamic instrumentation on the latest Android runtime to provide a user-space approach for tampering with applications at runtime without requiring any modifications to both the Android platform and the application's bytecode. To analyze Android application practically and effectively, only the combination of static and dynamic analysis allows to overcome the number of challenges that the Android system design presents to the analyst. As one of the peculiar challenge for dynamic analysis, the code coverage. Indeed, exploring all the code paths dynamically is quite impractical but not always the main goal. In fact, from the security prospective it is relevant to cover all the critical paths that lead to a security violation, eventually. To this end, static analysis helps in identifying those Point Of Interest (POI) then dynamic instrumentation completes the task by triggering and monitoring previously identified POI. In particular, we have shown as program slicing helps for extracting interesting code slices and collect runtime values by dynamic execution of those. Further, we addressed the growing enterprise demands for fine-grained security and privacy policies that allow to control and monitor employees' devices. To this end, we proposed a novel approach that fits enterprise needs of fine-grained capabilities and enables developers to accomplish to this demands by simply modifying a single line in app's manifest file, without any other modifications to app's bytecode. We have shown how our approach is able to practically enforce fine-grained enterprise-level policies at runtime maintaining a negligible overhead. Embracing a comprehensive analysis of both Java and native layers, our approach offers runtime instrumentation capabilities for interposing custom code to automatically enforce policies at any layers, still remaining completely transparent to the application being monitored. To further investigate how malware analysis online services perform the dynamic analysis of suspicious apps, we proposed a novel approach and developed a tool for assessing the stealth capability of those sandboxing online service. Here, the term stealth means how the sandbox is able to hide its presence to the malware being analyzed. Android sandbox is built either on the Android emulator or the real device with a hooking framework, thus fingerprints of the Android sandbox could be used to evade the dynamic detection. In this chapter we present our investigation to address the question of which artifacts and how they are actually implemented by online analysis services. We first conduct a measurement on eight Android sandboxes and find that their customized usage profile (e.g., contact, SMS) can be fingerprinted by attackers for evading the sandbox. From our measurement results, most Android sandboxes have empty usage profile fingerprints, or fixed fingerprints, or random artifact fingerprints. So, without protections on such user profiles, Android malware can identify these fingerprints that associate with different sandboxes and hide its malicious behaviors. Even though detecting malicious apps before they reach the public markets is a great barrier against malware proliferation, another important barrier is to detect and fix security vulnerability afflicting a codebase, eventually. To this end, we proposed OctoDroid , a plugin for the Octopus code analysis platform that combines the benefits from the powerful Clang's C++ AST parser with the Octopus analysis platform powered by Code Property Graph (CPG) representation. We designed and implemented a simple yet effective analysis algorithm that exploit the Class Hierarchy Analysis (CHA) offered via Clang's LibTooling library. The augmented code produced by OctoDroid allows Octopus to perform precise analysis of C++ Android system services codebase.

Most of the work discussed in this dissertation has already been published (or accepted) in international conferences or in submission to international journals. A list of the publications is provided in Appendix A. The work presented in Chapter 3 and 4 is published in [50] and [31]. Similarly, most of the work presented in Chapter 7 is published in [51]. Moreover, the work discussed in 5 is in submission to an international journal and going to be published in December this year. Also, the work presented in the Chapter 6 is under submission to an international journal. Moreover, out latest work presented in 8 is under submission to an international workshop.

9.0.1 Future Directions

There are few ideas that are not mature enough hence have not been developed yet, even though they were planned to be included in this dissertation.

Regarding app-level Mobile Application Management solutions, we are designing an app-level firewall-like system implemented as policy module for AppBox . It would be able to monitor and report suspicious usage of app's endpoints (i.e., Content Provider, Broadcast Receiver, etc.) back to the IT monitoring console where they can be collected and further analyzed. This app-level detection mechanism in couple with AppBox allows to enforce the runtime protection of enterprise apps being able to detect known attacks and assist the analysis of suspicious collected anomalies.

Regarding the static code analysis of C++ codebase, we are currently investigating the possibility of extending OctoDroid with the ability to identify a wrong C++ code patch given the CPG representation of the vulnerable code and the patch itself. If such approach is viable, we can couple this ability with a continuous system for patch monitoring that, in couple with a symbolic engine, allows to observe patch code and detect potentially mistakes as soon as they are published.

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.

[3] AndroGuard. https://code.google.com/p/androguard/.

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

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

[man] Android manifest file. https://developer.android.com/guide/topics/manifest/manifest-intro.

[and] Andrototal - free service to scan suspicious apks against multiple mobile antivirus. http://andrototal.org/. Accessed: 2016-02-27.

[Api] Apimonitor sandbox. https://code.google.com/p/droidbox/wiki/APIMonitor. Accessed: 2016-02-27.

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

[und] Avc undroid. http://undroid.av-comparatives.info/. Accessed: 2016-02-27.

[dcl] Custom class loading in dalvik. http://android-developers.blogspot.it/2011/07/custom-class-loading-in-dalvik.html. Accessed: 2017-01-30.

[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.

[inl] J.breamer: inline hooking demystified. http://jbremer.org/x86-api-hooking-demystified/. Accessed: 2017-01-30.

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

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

[mob] Mobisec lab. http://www.mobiseclab.org/. Accessed: 2016-02-27.

[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.

[ds-] Previously ds-andrototal - now droydseuss. http://droydseuss.necst.it/. Accessed: 2016-02-27.

[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.

[30] (2016). Joe mobile sandbox. http://www.joesecurity.org/joe-sandbox-mobile.

[31] Ahmad, M., Costamagna, V., Crispo, B., and Bergadano, F. (2017). Teicc: targeted execution of inter-component communications in android. In *Proceedings of the Symposium on Applied Computing*, pages 1747–1752. ACM.

[32] Ahmad, M., Crispo, B., and Gebremichael, T. (2016). Empirical analysis on the use of dynamic code updates in android and its security implications. In *Nordic Conference on Secure IT Systems*, pages 119–134. Springer.

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

[34] Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., and Rieck, K. (2014). Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*.

[35] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. (2014). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *ACM SIGPLAN Notices*, 49(6):259–269.

[36] 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.

[37] 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.

[38] 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.

[39] Backes, M., Bugiel, S., Schranz, O., von Styp-Rekowsky, P., and Weisgerber, S. (2017). Artist: The android runtime instrumentation and security toolkit. In *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*, pages 481–495. IEEE.

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

[41] Bartel, A., Klein, J., Le Traon, Y., and Monperrus, M. (2012). Automatically securing permission-based software by reducing the attack surface: An application to android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 274–277. ACM.

[42] 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.

[43] 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.

[44] Bobrow, D. G., Gabriel, R. P., and White, J. L. (1993). Clos in context: the shape of the design space. *Object Oriented Programming: The CLOS Perspective*, pages 29–61.

[45] Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., and Mezini, M. (2011). Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 241–250. ACM.

[46] Bogda, J. and Singh, A. (2001). Can a shape analysis work at run-time? In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*, pages 2–2. USENIX Association.

[47] Bruening, D. and Amarasinghe, S. (2004). *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science.

[48] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., and Sadeghi, A.-R. (2011). Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*.

[49] Conti, M., Nguyen, V. T. N., and Crispo, B. (2010). Crepe: Context-related policy enforcement for android. In *International Conference on Information Security*, pages 331–345. Springer.

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

[51] Costamagna, V., Zheng, C., and Huang, H. (2018). Identifying and evading android sandbox through usage-profile based fingerprints. In *Proceedings of the First Workshop on Radical and Experiential Security*, pages 17–23. ACM.

[52] 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.

[53] 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(2):17.

[54] Desnos, A. et al. (2011). Androguard. *URL: https://github. com/androguard/androguard*.

[55] Desnos, A. and Gueguen, G. (2011). Android: From reversing to decompilation. *Proc. of Black Hat Abu Dhabi*, pages 77–101.

[56] Desnos, A. and Lantz, P. (2011). Droidbox: An android application sandbox for dynamic analysis.

[57] 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 &#38; Privacy in Wireless and Mobile Networks*, WiSec '16, pages 159–164, New York, NY, USA. ACM.

[58] Enck, W. (2011). Defending users against smartphone apps: Techniques and future directions. In *International Conference on Information Systems Security*, pages 49–70. Springer.

[59] 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.

[60] Ernst, M. D., Just, R., Millstein, S., Dietl, W., Pernsteiner, S., Roesner, F., Koscher, K., Barros, P. B., Bhoraskar, R., Han, S., et al. (2014). Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1092–1104. ACM.

[61] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. (2011). Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM.

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

[63] Fuchs, A. P., Chaudhuri, A., and Foster, J. S. (2009). Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/˜ avik/projects/scandroidascaa.*

[64] Gascon, H., Yamaguchi, F., Arp, D., and Rieck, K. (2013). Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM.

[65] Gibler, C., Crussell, J., Erickson, J., and Chen, H. (2012). AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *International Conference on Trust and Trustworthy Computing*, pages 291–307. Springer.

[66] Gomez, L., Neamtiu, I., Azim, T., and Millstein, T. (2013). Reran: Timing- and touch-sensitive record and replay for android. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 72–81. IEEE.

[67] 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.

[68] Grace, M., Zhou, Y., Zhang, Q., Zou, S., and Jiang, X. (2012a). Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 281–294. ACM.

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

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

[71] 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.

[72] Heuser, S., Nadkarni, A., Enck, W., and Sadeghi, A.-R. (2014). Asm: A programmable interface for extending android security. In *USENIX Security Symposium*, pages 1005–1019.

[73] Hirzel, M., Dincklage, D. V., Diwan, A., and Hind, M. (2007). Fast online pointer analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(2):11.

[74] 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.

[75] Hu, C. and Neamtiu, I. (2011). Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, pages 77–83. ACM.

[76] 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*.

[77] Huang, J., Zhang, X., Tan, L., Wang, P., and Liang, B. (2014). Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1036–1046. ACM.

[78] 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)*.

[79] Jing, Y., Zhao, Z., Ahn, G.-J., and Hu, H. (2014a). Morpheus: automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 216–225. ACM.

[80] Jing, Y., Zhao, Z., Ahn, G.-J., and Hu, H. (2014b). 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.

[81] Jung, J.-H., Kim, J. Y., Lee, H.-C., and Yi, J. H. (2013). Repackaging attack on android banking applications and its countermeasures. *Wireless Personal Communications*, 73(4):1421–1437.

[82] Kim, J., Yoon, Y., Yi, K., Shin, J., and Center, S. (2012). Scandal: Static analyzer for detecting privacy leaks in android applications. *MoST*, 12.

[83] 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.

[84] 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.

[85] Li, L., Bissyandé, T. F., Octeau, D., and Klein, J. (2016a). Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 318–329. ACM.

[86] Li, L., Bissyandé, T. F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., and Traon, L. (2017a). Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88:67–95.

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

[88] Li, Y., Yang, Z., Guo, Y., and Chen, X. (2017b). Droidbot: a lightweight ui-guided test input generator for android. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, pages 23–26. IEEE.

[89] Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., Van Der Veen, V., and Platzer, C. (2014). Andrubis–1,000,000 apps later: A view on current android malware behaviors. In *Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), 2014 Third International Workshop on*, pages 3–17. IEEE.

[90] Livshits, B., Whaley, J., and Lam, M. S. (2005). Reflection analysis for java. In *Asian Symposium on Programming Languages and Systems*, pages 139–160. Springer.

[91] Lortz, S., Mantel, H., Starostin, A., Bähr, T., Schneider, D., and Weber, A. (2014). Cassandra: Towards a certifying app store for android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 93–104. ACM.

[92] 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.

[93] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM.

[94] MacHiry, A., Tahiliani, R., and Naik, M. (2013). Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM.

[95] Mahmood, R., Mirzaei, N., and Malek, S. (2014). Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 599–609. ACM.

[96] Maier, D., Müller, T., and Protsenko, M. (2014). Divide-and-conquer: Why android malware cannot be stopped. In *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*, pages 30–39. IEEE.

[97] 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.

[98] Moonen, L. (2001). Generating robust parsers using island grammars. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 13–22. IEEE.

[99] Moser, A., Kruegel, C., and Kirda, E. (2007). Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE.

[100] 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.

[101] 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.

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

[103] Mutti, S., Fratantonio, Y., Bianchi, A., Invernizzi, L., Corbetta, J., Kirat, D., Kruegel, C., and Vigna, G. (2015b). 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.

[104] Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM.

[105] 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*.

[106] Oberheide, J. and Miller, C. (2012). Dissecting the android bouncer. *SummerCon2012, New York*.

[107] 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.

[108] 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.

[109] P. Carter, C. Mulliner, M. L. W. R. and Kirda, E. (2016). Curiousdroid: Automated user interface interaction for android application analysis sandboxes. In *In Financial Cryptography and Data Security - 20th Internationa Conference (FC)*.

[110] Payet, É. and Spoto, F. (2012). Static analysis of android programs. *Information and Software Technology*, 54(11):1192–1201.

[111] Perl, H., Dechand, S., Smith, M., Arp, D., Yamaguchi, F., Rieck, K., Fahl, S., and Acar, Y. (2015). Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 426–437. ACM.

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

[113] 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.

[Polkovnichenko and Boxiner] Polkovnichenko, A. and Boxiner, A. Braintest - a new level of sophistication in mobile malware. Technical report, Check Point Technologies Ltd. Accessed: 2017-01-30.

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

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

[117] 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.

[118] 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.

[119] Ren, C., Chen, K., and Liu, P. (2014). Droidmarking: resilient software watermarking for impeding android application repackaging. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 635–646. ACM.

[120] Rodriguez, M. A. (2015). The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, pages 1–10. ACM.

[121] Russello, G., Conti, M., Crispo, B., and Fernandes, E. (2012). Moses: supporting operation modes on smartphones. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pages 3–12. ACM.

[122] Russello, G., Crispo, B., Fernandes, E., and Zhauniarovich, Y. (2011). Yaase: Yet another android security extension. In *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on*, pages 1033–1040. IEEE.

[123] Russello, G., Jimenez, A. B., Naderi, H., and van der Mark, W. (1023). Firedroid: Hardening security in almost-stock android. In *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM.

[124] Schulz, P. (2012). Code protection in android. *Insititute of Computer Science, Rheinische Friedrich-Wilhelms-Universitgt Bonn, Germany*, 110.

[125] 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.

[126] Sun, M., Wei, T., and Lui, J. (2016). Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 331–342. ACM.

[127] Tam, K., Feizollah, A., Anuar, N. B., Salleh, R., and Cavallaro, L. (2017). The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, 49(4):76.

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

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

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

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

[132] 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.

[133] 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.

[134] Wala, T. (2014). Watson libraries for analysis.

[135] Wang, X., Sun, K., Wang, Y., and Jing, J. (2015). Deepdroid: Dynamically enforcing enterprise policy on android devices. In *NDSS*.

[136] 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.

[137] 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.

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

[139] Wognsen, E. R., Karlsen, H. S., Olesen, M. C., and Hansen, R. R. (2014). Formalisation and analysis of dalvik bytecode. *Science of Computer Programming*, 92:25–55.

[140] Xu, R., Saïdi, H., and Anderson, R. (2012). Aurasium: Practical policy enforcement for android applications. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 539–552.

[141] Yamaguchi, F., Golde, N., Arp, D., and Rieck, K. (2014). Modeling and discovering vulnerabilities with code property graphs. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 590–604. IEEE.

[142] Yamaguchi, F., Lindner, F., and Rieck, K. (2011). Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX conference on Offensive technologies*, pages 13–13. USENIX Association.

[143] Yamaguchi, F., Maier, A., Gascon, H., and Rieck, K. (2015). Automatic inference of search patterns for taint-style vulnerabilities. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 797–812. IEEE.

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

[145] Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., and Wang, X. S. (2013). Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM.

[146] 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*.

[147] Zhang, M. and Yin, H. (2014). Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *NDSS*.

[148] Zhang, Y., Tan, T., Li, Y., and Xue, J. (2017). Ripple: Reflection analysis for android apps in incomplete information environments. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 281–288. ACM.

[149] 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.

[150] 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.

[151] Zheng, M., Lee, P. P., and Lui, J. C. (2012b). Adam: an automatic and extensible platform to stress test android anti-virus systems. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 82–101. Springer.

[152] Zheng, M., Sun, M., and Lui, J. C. (2014). Droidtrace: A ptrace based android dynamic analysis system with forward execution capability. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2014 International*, pages 128–133. IEEE.

[153] Zhongyang, Y., Xin, Z., Mao, B., and Xie, L. (2013). Droidalarm: an all-sided static analysis tool for android privilege-escalation malware. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 353–358. ACM.

[154] Zhou, W., Zhang, X., and Jiang, X. (2013). Appink: watermarking android apps for repackaging deterrence. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 1–12. ACM.

[155] 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.