Challenges of native android applications : obfuscation and vulnerabilities

Pierre Graux

THÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par
Pierre Graux

Challenges of Native Android Applications: Obfuscation and Vulnerabilities

Thèse présentée et soutenue à Lieu, le 10 décembre 2020
Unité de recherche : IRISA
Thèse N° :

Rapporteurs avant soutenance :

Guillaume Bonfante	Maître de Conférences HDR, Université de Lorraine
Christian Rossow	Professor, Saarland University

Composition du Jury :

Examinateurs :	Marie-Laure Potet	Professeur des Universités, Grenoble INP
	Erven Rohou	Directeur de Recherche, Inria
	Sarah Zennou	Research engineer, Airbus
Dir. de thèse :	Valérie Viet Triem Tong	Professeur, CentraleSupelec
Co-dir. de thèse :	Jean-François Lalande	Professeur des Universités, CentraleSupélec
Encadrant :	Pierre Wilke	Enseignant Chercheur, CentraleSupélec

Abstract

Android is the most used operating system and thus, ensuring security for its applications is an essential task. Securing an application consists in preventing potential attackers to divert the normal behavior of the targeted application. In particular, the attacker may take advantage of vulnerabilities left by the developer in the code and also tries to steal intellectual property of existing applications. To slow down the work of attackers who try to reverse the logic of a released application, developers are incited to track potential vulnerabilities and to introduce countermeasures in the code. Among the possible countermeasures, the obfuscation of the code is a technique that hides the real intent of the developer by making the code unavailable to an adversary using a reverse engineering tool. Mobile applications are complex entities that can be made of both bytecode and assembly code. This creates new opportunities to enhance obfuscation techniques, and also makes deobfuscation a more difficult challenge.

Obfuscating and deobfuscating programs have already been widely studied by the research community, especially for desktop architecture. For mobile devices, ten years after the first release of Android, researchers have mainly worked on the deobfuscation of the intermediate language, named Dalvik bytecode, executed by the embedded virtual machine. Nevertheless, with the growing amount of malware and applications carrying sensitive information, attackers want to hide their intents and developers want to protect their intellectual property and the integrity of their application. Thus, a new generation of obfuscation methods based on native code has appeared. Studying the consequences of mobile native code has not – so far – received the same amount of attention as desktop programs even though more than one third of the available applications embed assembly code.

This thesis presents the impact of native code on both reverse-engineering and vulnerability finding applied to Android applications. First, by listing the possible interferences between assembly and bytecode, we highlight new obfuscation techniques and software vulnerabilities. Then, we propose new analysis techniques combining static and dynamic analysis blocks, such as taint tracking or system monitoring, to observe the code behaviors that have been obfuscated or to reveal new vulnerabilities. These two objectives have led us to develop two new tools. The first one spots a specific vulnerability that comes from inconsistently mixing native and Java data. The second one extracts the object level behavior of an application, regardless of whether this application contains native code, embedded for obfuscation purposes. Finally, we implemented these new methods and conducted experimental evaluations. In particular, we automatically found a vulnerability in the Android SSL library and we analyzed several Android firmware to detect usage of a specific class of obfuscation.

Publications

[1] **Pierre Graux**, Jean-François Lalande, and Valérie Viet Triem Tong. 'Etat de l'Art des Techniques d'Unpacking pour les Applications Android'. In: *Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information*. La Bresse, France, May 2018.

[2] Jean-François Lalande, Valérie Viet Triem Tong, Mourad Leslous, and **Pierre Graux**. 'Challenges for reliable and large scale evaluation of android malware analysis'. In: *2018 International Conference on High Performance Computing & Simulation*. Orléans, France: IEEE, July 2018, pp. 1068–1070.

[3] **Pierre Graux**, Jean-François Lalande, and Valérie Viet Triem Tong. 'Obfuscated Android Application Development'. In: *3rd Central European Cybersecurity Conference*. Munich, Germany: ACM, Nov. 2019, pp. 1–6.

[4] Valérie Viet Triem Tong, Cédric Herzog, Tomàs Concepción Miranda, **Pierre Graux**, Jean-François Lalande, and Pierre Wilke. 'Isolating Malicious Code in Android Malware in the Wild'. In: *14th International Conference on Malicious and Unwanted Software*. Nantucket, MA, USA: IEEE, Oct. 2019.

[5] Jean-François Lalande, Valérie Viet Triem Tong, **Pierre Graux**, Guillaume Hiet, Wojciech Mazurczyk, Habiba Chaoui, and Pascal Berthomé. 'Teaching android mobile security'. In: *50th ACM Technical Symposium on Computer Science Education*. Minneapolis MN, USA: ACM, Feb. 2019, pp. 232–238.

[6] Jean-François Lalande, **Pierre Graux**, and Tomás Concepción Miranda. 'Orchestrating Android Malware Experiments'. In: *27th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. Rennes, France: IEEE, Oct. 2019, pp. 1–2.

[7] **Pierre Graux**, Jean-François Lalande, Pierre Wilke, and Valérie Viet Triem Tong. 'Abusing Android Runtime for Application Obfuscation'. In: *Workshop on Software Attacks and Defenses*. Genova, Italy: IEEE, Sept. 2020, pp. 616–624.

[8] **Pierre Graux**, Jean-François Lalande, Valérie Viet Triem Tong, and Pierre Wilke. 'Preventing Serialization Vulnerabilities through Transient Field Detection'. In: *36th ACM/SIGAPP Symposium On Applied Computing*. Gwangju, Korea: ACM, Mar. 2021, pp. 1598–1606.

Acknowledgments - Remerciements

First and foremost, I would like to thank a lot Valérie Viet Triem Tong, Jean-François Lalande and Pierre Wilke, who have advised this thesis. They were always available to give great advises. I can measure the luck to have them as advisors. They taught me to look at problems as a researcher and no longer as an engineer.

I would like to give a special thank to Marie-Laure Potet, who helped me when looking for a thesis and drove me to find the CIDRE team. I made my first step in the research world with her, I wouldn't have carried this thesis without her help. I am glad she accepted to examine it and I hope it will fulfill her expectations.

I would like also to thank Christian Rossow and Guillaume Bonfante, who have accepted to review my thesis and helped me to improve it, and Erven Rohou and Sarah Zennou, who have followed my work annually during each *Comité de Suivi Individuel (CSI)* where they gave me useful and relevant advises. By giving me part of their time they have allowed me to expose my ideas to benevolent external point of views.

Last but not least, I would like to thank all my relatives, my friends and my colleagues. Despite giving technical advises, they gave me moral support, necessary to conduct research work which can, sometimes, be frustrating.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Glossary

Allocator
Mechanism in charge of reserving and freeing memory areas. 4.0.0, 7.2.1

AOSP
Stands for Android Open Source Project, open source code of Android. 2.3.4, 3.1.0, 4.2.3, 5.2.0, 5.2.2, 7.4.0, 7.6.0, 8.1.0, 8.3.0, 9.2.0, 0.0

Bytecode Free OAT
Android obfuscation technique which consists in deleting bytecode from OAT files. 3.2.1, 3.2.2, 3.3.0, 4.2.0, 5.0.0, 5.2.0, 5.2.1, 5.2.3, 5.3.0, 6.4.0, 7.1.0, 7.2.0, 7.2.5, 7.3.0, 7.3.3, 9.1.0, B.0.0, B.2.0, 0.0

Common Vulnerabilities and Exposures
System that list publicly known compter security vulnerabilties and exposures. By extension, a publicly known compter security vulnerabiltie or exposure. 1.1.1, 4.1.1, 6.2.0, 0.0

Concolic Analysis
Symbolic analysis that relies on values obtained during a classical execution called concrete execution. 7.2.0, C.0.0

Concrete (value/execution)
Execution, or values obtained during this execution, that is used during a concolic analysis. 2.3.3, 7.2.4, C.0.0

Control Flow Graph
Graph that represents how the different instructions of code can be chained. Nodes are instructions and two nodes are connected by a directed edge if the destination node can be executed after the source node. 1.3.0, 2.1.0, 7.2.0, 7.2.2, 7.2.4, 7.2.5, 8.2.0, 9.1.0, B.2.0

Dataflow
Dependency between the variables of a code. 2.3.3, 6.0.0, 6.2.0

DEX
Stands for Dalvik EXecutable, file format used to store Dalvik bytecode. 2.3.3, 3.1.0–3.1.4, 3.2.1, 3.2.2, 4.3.0, 5.0.0, 5.1.0, 5.2.0, 5.2.3, 5.3.0, 7.1.0, 7.2.1, 7.2.5, 7.4.0, B.0.0, B.1.0, B.2.0, 2.1, 0.0

Direct Heap Access
Android obfuscation technique which consists in modifying Java values using native code without the help of Java Native Interface (JNI). 4.2.0–4.2.3, 4.3.0, 6.0.0, 6.3.0, 6.4.0, 7.1.0, 7.2.1, 7.2.5, 7.3.0, 7.3.1, 7.3.3, 9.1.0, B.0.0, B.2.0, 0.0

Dynamic Analysis
Analysis that do run the analysed code. 3.2.0, 6.0.0, 7.2.0, 7.2.4

Firmware
Set of files installed on a smartphone by the smartphone constructor. It includes, among other, the Android system and the defaults application. 2.2.0, 3.2.1, 5.2.0–5.2.3, 5.3.0, 9.1.0, A.0.0, 0.0

Heap
Memory area where allocated entities are stored including objects. 4.1.1, 4.2.0–4.2.3, 6.1.0, 6.3.0, 6.4.0, 7.1.0, 7.2.1, 7.3.2, 7.4.0, 8.1.3, 8.1.4, 8.1.6, 8.1.7, B.1.0, B.2.0

Manifest
File that describes an application. It contains, among other, requiered permissions or activities and services classes name. 2.1.0, 2.3.2, 5.1.0

Nopping
Replacing instrucions by No-OPerations, that is by instructions that have no special effects. 3.2.2, 5.2.2, 5.3.0, 9.1.0

OAT
Unknown acronym, file format used to store compiled Dalvik bytecode. 2.4.0, 3.2.1, 3.2.2, 4.3.0, 5.0.0, 5.2.0, 5.2.3, 5.3.0, 7.2.1, 7.2.5, 7.3.1, 7.4.0, B.2.0, 2.1, 0.0

Packer
Tool that ciphers all or part of a program code without modifying its overall behavior. 3.1.0–3.1.2, 3.1.4, 5.0.0, 5.1.0, 6.4.0, 0.0

Runtime
Android library in charge of runtime the applications. 1.1.1, 2.1.0, 2.3.3, 2.3.4, 2.4.0, 3.2.0, 4.0.0, 4.2.0, 4.2.3, 5.0.0, 6.0.0, 6.3.0, 7.0.0, 7.1.0, 7.2.0, 7.2.1, 7.2.5, 7.4.0, 8.0.0, 8.1.0–8.1.2, 8.1.4, 8.3.0, 0.0

Serialization
Process of converting an object into a stream of bytes. 4.1.0, 4.1.2, 6.2.3, 0.0

Stack
Memory area where variables and parameters may be stored. 7.1.0, 7.2.2, 7.2.5, 7.3.3, 8.1.3, B.1.0

Static Analysis
Analysis that does not run the analysed code. 2.1.0, 2.3.0, 2.3.3, 2.4.0, 6.2.1, 7.0.0, 7.1.0, 7.2.5, 9.1.0

Symbolic Analysis
Analysis that consists in observing the execution of a code by replacing value by abstract values called symbols and evaluating the code instructions in the abstract value space. 1.2.0, 7.0.0, 7.1.0, 7.2.4, 7.3.2, 9.1.0, C.0.0, 0.0

Taint Analysis
Analysis that consists in checking if some values, generated by expressions called sources, can reach expressions called sinks. 2.3.3, 2.3.4, 2.4.0, 4.1.2, 6.1.0, 6.2.1, 6.2.2

Transient
Java field keyword that indicates that the qualified field is not pat of the serialization process. 4.1.0–4.1.2, 6.0.0, 6.2.0, 6.2.1, 6.2.3

Unpacker
Tool that reverses the obfuscation made by a packer. 3.1.0–3.1.4

Acronyms

ABI Application Binary Interface.
AIDL Android Interface Definition Language.
AOSP Android Open Source Project.
AOTC Ahead Of Time Compilation.
ASLR Address Space Layout Randomization.
AST Abstract Syntax Tree.
BFO Bytecode Free OAT.
CFG Control Flow Graph.
CVE Common Vulnerabilities and Exposures.
DEX Dalvik EXecutable.
DHA Direct Heap Access.
GC Garbage Collector.
IPC Inter-Process Communication.
JGRE Java Global Reference Exhaustion.
JIT Just In Time.
JNI Java Native Interface.
MMU Memory Management Unit.
NDK Native Development Kit.
PIE Position Independent Executable.
SLOC Single Line Of Code.
VM Virtual Machine.

Prologue

Introduction 1

1.1 Problem statement

1.1.1 Android core security features

Android is the prevalent operating system for modern smartphones. Due to the tremendous number of users, Android has attracted lots of malicious activities [9]. As shown in Table 1.1, since the release of the first version of Android, vulnerabilities are searched and found in this system. More than six thousand CVEs[1] contain the keyword "*Android*". Very critical vulnerabilities, such as Full Chain with Persistence (FCP) zero click, can be sold for more than $2,500,000[2].

[9]: Mohamed and Patel (2015), 'Android vs iOS security: A comparative study'

1: Common Vulnerabilities and Exposures, publicly known vulnerabilities

2: https://zerodium.com/program.html

< 2010	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020
18	23	89	169	123	1686	422	872	1191	457	771	528

Source: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Android

Table 1.1: Number of CVE containing the keyword Android

Because of the increasing number of Android users, and the increasing number of applications handling sensitive information, Google has brought a lot of attention to securing the Android platform. Indeed, they have adopted a security-oriented architecture for running Android applications that mainly relies on two core features: **application sandboxing** and **permission management**. Each application is run by a dedicated Unix user, allowing application isolation using tried-and-tested kernel mechanisms. Additionally, applications cannot access device features without owning specific capabilities called permissions. These permissions are reviewed and granted by the user himself. For example, an application will be able to send SMSs only if it has been granted the `SEND_SMS` permission. Furthermore, each new version of Android comes with specific security features. Table 1.2 shows an excerpt of security features added in each new Android version. In this table, we can see that features target all aspects of security. For example, hardening techniques such as ASLR[3] or PIE[4] have been added to the system. The operating system constrains the accesses to resources using SELinux mandatory access control. Network communications such as DNS queries are ciphered.

3: Address Space Layout Randomization

4: Position-Independent Executable

However, securing the whole Android system is not enough. Indeed, applications installed by the user are potentially malicious or vulnerable. An application is considered vulnerable if it can be diverted into performing malicious operations. Since Android is a system built for mobile platforms, these operations can differ from desktop ones [10, 11]. We can cite as notable malicious operations:

- Premium SMS services: an application can send SMS to premium services, *i.e.* services that include fees.
- Privilege escalation: an application can exploit system vulnerabilities to perform actions while not being granted the corresponding permission.

[10]: Faruki, Bharmal, Laxmi, Ganmoor, Gaur, Conti, and Rajarajan (2014), 'Android security: a survey of issues, malware penetration, and defenses'
[11]: Sadeghi, Bagheri, Garcia, and Malek (2016), 'A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software'

Table 1.2: Extract of security additions in Android system

Android version	Release date	Added feature
4.0	Oct. 2011	Address Space Layout Randomization (ASLR) support
4.1	Jul. 2012	Position Independent Executable (PIE) support
4.1	Jul 2012	Read only relocation (RelRO) binaries
4.2	Nov. 2012	SELinux support
4.3	Jul. 2013	SELinux enabled by default
4.4	Oct. 2013	SELinux set in enforcing mode
5.0	Nov. 2014	Support of non-PIE executable dropped
6.0	Oct. 2015	App permissions granted at runtime
9	Aug. 2018	DNS over TLS

Source: https://en.wikipedia.org/wiki/Android_version_history

- Permission leakage, colluding applications: an application can perform privileged operations when requested by other applications and omit to check requester permissions. If the omission is intentional, the application is colluding.
- Privacy leakage: an application can steal users' private data such as SMS contents or contact list, or spy on the user by, for example, recording the microphone.
- Ransomware: an application can make smartphone data, such as pictures or contacts, unavailable by ciphering them and ask money from user in exchange for the stolen data.
- Application cloning: an application can copy the code of another and replace the Google Ads ID of the real owner of the application by its own in order to steal its wages.
- Aggressive advertisement: an application can display numerous advertisements by, for example, modifying the smartphone background or spawning pop-up windows.
- Botnet: an application can participate to massive network attacks.
- Denial of Service: an application can stress resources such as the CPU or the battery of the smartphone to make it unusable.

Unfortunately, relying on application isolation and permission restriction to keep the user safe is not enough. Indeed, the permission system is misunderstood and harmful permissions may be granted to malicious applications [12, 13]. For example, in 2014, a fake copy of the eagerly awaited video game *Pokémon Go* has been created and distributed to countries where the official game was not released yet. The fake version, which contained a malware called *Droidjack*, was installed by users impatient to play the game and willing to grant any permission asked by the application.

[12]: Felt, Ha, Egelman, Haney, Chin, and Wagner (2012), 'Android permissions: User attention, comprehension, and behavior'
[13]: Benton, Camp, and Garg (2013), 'Studying the effectiveness of android application permissions requests'

The security mechanisms provided by Android cannot prevent this type of attack. Malicious or vulnerable application will eventually be installed on some users' smartphone. To minimize the impact of this phenomenon, this thesis tackles the two following problems:

- Detecting malicious or vulnerable applications in order to remove them from the Google Play store.

- Understanding the behavior of such applications in order to evaluate the damage after a compromision.

Of course, Google already started, since 2012, to set up an automatic service, called *bouncer*,[5] to address the malicious application detection problem. Since the `bouncer` architecture is not public, we have no clue on how and if the vulnerability detection problem is addressed.

5: `http://googlemobile.blogspot.com/2012/02/android-and-security.html`

This service scans applications available on the Google Play store[6], the Android application official repository, in order to find malicious and unsafe applications. When detected, applications are removed from the store, therefore preventing users from installing them. Improving on the *bouncer* service, Google released Google Play Protect[7] in 2017. In addition to the features provided by *bouncer*, this new service offers the possibility to scan applications offline, *i.e.* observe the behavior of applications while they are running directly on the users' smartphone.

6: Formerly Android market

7: `https://www.blog.google/products/android/google-play-protect/`

Despite all these efforts, some malicious applications still find their way to the Google Play store,[8] and vulnerabilities are still found in Android applications, as shown in Table 1.1.

8: https://research.checkpoint.com/2020/google-play-store-played-again-tekya-clicker-hides-in-24-childrens-games-and-32-utility-apps/

We believe that one of the reasons that malicious and vulnerable applications can still bypass analysis systems is the usage of native code inside applications. This thesis focuses on this specific problem. The following section explains why the presence of native code makes code analysis more difficult for malicious or vulnerable application detection.

1.1.2 Challenges in analyzing native applications

Problems involved in malware and vulnerability detection, such as determining if a given program is equivalent to an other, are undecidable in the general case [14]. Thus, countermeasures such as the bouncer or Google Play Protect can only partially solve these problems. This keeps the door open for malicious applications to hide from program analysis. Similarly, perfect obfuscation techniques do not exist [15], *i.e.* obfuscators always leave information about the behavior of the original program.

[14]: Selçuk, Orhan, and Batur (2017), 'Undecidable problems in malware analysis'

[15]: Beaucamps and Filiol (2007), 'On the possibility of practically obfuscating programs towards a unified perspective of code protection'

Consequently, the race between malicious applications and analysis services takes the form of a cat and mouse game: malicious applications hide their intent using new techniques, analysis services adapt their detection, and so on. Unfortunately, as shown in Figure 1.1, this race is in favor of malicious applications since malware can take advantage of the time that separate the usage of a new technique and its detection ($t_2 - t_1$). Thus, computer security researchers should focus on reducing this time by:

- developing more general analysis (increasing t_1): this makes the creation of new obfuscation techniques harder.
- predicting future obfuscation techniques (decreasing t_2): this allows to faster adapt detection technique.

It is worth noting that this assessment also applies to new vulnerabilities exploitation and detection.

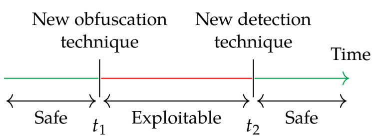


Figure 1.1: Exploitable window

In this context, Android security researchers have shown that native applications are more and more present in Google Play store and that state-of-the-art tools should improve their analysis on this kind of applications [16–18].

[16]: Afonso, Geus, Bianchi, Fratantonio, Kruegel, Vigna, Doupé, and Polino (2016), 'Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy'
[17]: Tam, Feizollah, Anuar, Salleh, and Cavallaro (2017), 'The evolution of android malware and android analysis techniques'
[18]: Sadeghi, Bagheri, Garcia, and Malek (2017), 'A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software'

Applications are traditionally written in Java or Kotlin, compiled into bytecode and run by a Virtual Machine. This machine enforces the correct execution of this bytecode as expected by the developer and is the privileged interface for observing an execution. A native application is an application that contains both Dalvik bytecode and assembly code. Due to optimization purposes, Android supports applications that embed assembly code obtained from, for example, C or C++ source code.

The usage of native code opens two new challenges:

- Native code usage allows to highly obfuscate applications. Indeed, the cat and mouse game for obfuscating and desobfuscating assembly code is a well studied area since the seventies, that is way older than Android. Thus, the attacker can easily adapt advanced assembly obfuscation techniques and bypass analysis tools.
- Native code usage may introduce vulnerabilities in applications. The languages in which native code is typically written (C or C++) are known to be error-prone. That is to say, it is easy for developers using these languages to leave security vulnerabilities in their programs. Indeed, contrary to Java/Kotlin, these languages do not implement security mechanisms such as strong type verification or security context execution. Then, allowing native code inside Android applications drastically increases the attack surface for malicious intents. Additionally, tips and best practices given by Google for native Android application development[9], are not enforced when the applications are running. Native code and bytecode run in the same context and the same address space[19, 20], which allows native code to interfere with bytecode.

9: `https://developer.android.com/training/articles/perf-jni`

[19]: Sun and Tan (2014), 'Nativeguard: Protecting android applications from third-party native libraries'
[20]: Athanasopoulos, Kemerlis, Portokalidis, and Keromytis (2016), 'NaClDroid: Native Code Isolation for Android Applications'

In this thesis, we mimic the cat and mouse game by building obfuscation techniques and exploiting vulnerable applications and in a second time, proposing associated detection techniques and analysis tools. We limit our study to the challenges linked to the usage of native code inside Android applications.

1.2 Contributions

The contributions of this thesis are the following:

1. We propose two new obfuscation methods of the java bytecode, one targeting the code and the other targeting the data [1, 3, 7].
2. We conducted two experimental studies of the usage of these obfuscation methods in the wild [2, 7].
3. We developed an analysis framework, named *OATs'inside*, which combines dynamic and symbolic analysis to retrieve the behavior of obfuscated Android applications.
4. We designed and implemented a new detection method of application vulnerabilities due to forgotten `transient` keyword [8].

[1]: **Graux**, Lalande, and Viet Triem Tong (2018), 'Etat de l'Art des Techniques d'Unpacking pour les Applications Android'
[2]: Lalande, Viet Triem Tong, Leslous, and **Graux** (2018), 'Challenges for reliable and large scale evaluation of android malware analysis'
[3]: **Graux**, Lalande, and Viet Triem Tong (2019), 'Obfuscated Android Application Development'
[7]: **Graux**, Lalande, Wilke, and Viet Triem Tong (2020), 'Abusing Android Runtime for Application Obfuscation'
[8]: **Graux**, Lalande, Tong, and Wilke (2021), 'Preventing Serialization Vulnerabilities through Transient Field Detection'

1.3 Outline

This dissertation is divided in five parts. The first part contains this introduction and Chapter 2, that gives the necessary background about Android native and non-native application analysis techniques.

In order to describe the contributions of this thesis we reflect the cat and mouse game by dividing the manuscript in two supplementary parts. Chapters 3 and 4 explore attackers' possibilities. They describe how native code can lead to security issues, *i.e.* code obfuscation or vulnerable code. These security issues are split on whether they impact Java code (Chapter 3), or the Java data (Chapter 4). In addition to already known issues, we introduce new obfuscation techniques.

The next two chapters, Chapters 5 and 6, tackle these security issues by proposing detection methods and measuring their presence in the wild. These two chapters are also divided into code and data issues.

The last two chapters before concluding, Chapters 7 and 8, present *OATs'inside*, a new Android analysis tool and the technical challenges involved in its implementation. *OATs'inside* is a stealth analysis framework that recovers object-level CFGs of Android applications despite all known obfuscation techniques.

Finally, Chapter 9 summarizes the contributions of this thesis and gives perspectives for future work.

2 Analyzing native Android applications: state of the art

This chapter reviews the contributions related to the security analysis of Android applications. We will focus on approaches that output qualitative and detailed information about the analyzed application. During this review, we will recall the technical notions about the Android architecture.

At the end of the chapter, we focus on the impact of native code on the challenges introduced in Chapter 2.1, *i.e.* obfuscation of applications and vulnerabilities in applications.

We will make a review of the articles of the state-of-the-art that tries to solve the aforementioned challenges. More thorough comparisons with our work will be given later in the appropriate chapters of this manuscript.

Section 2.1 presents goals that researchers follow when analyzing Android applications. Then, Section 2.2 details the datasets available for evaluating analysis methods. Section 2.3 reviews the techniques that are used to achieve the previously described goals. Finally, Section 2.4 highlights the challenges that native code raises for using these techniques.

2.1 Research goals

Android applications analyses take an APK file as input. An APK file is an archive that contains three types of files:

- metadata: a `Manifest` file that declares the permissions, the services and the activities of the application.
- code: files that contain Dalvik bytecode, usually obtained from the compilation of Java or Kotlin source code.
- resources: additional files such as pictures, fonts, or sounds.

APK files can be processed in various ways: static approaches that only look at the file itself, or dynamic ones that observe its execution.

Independently of the method used, security researchers have different common goals in mind:

- Detecting malicious applications: decide whether a given application is malicious or benign.
- Studying code protection: find new obfuscation techniques and associated countermeasures.
- Exposing vulnerable applications: spot security vulnerabilities inside applications.

Detecting malicious applications Malicious application detection can be declined in different annex problems. While some researchers focus on deciding the maliciousness of an application [21–26], others try to classify malicious applications into families [27–33]. The definition of what a family is depends on the context. For example, family can designate the malicious operation performed such as ransomware, Remote Access Tool (RAT), adware. It can also designate different versions of the same malware. Some also try to identify clones and repackaged applications. Here the goal is to detect when an attacker has introduced his code inside an other application. These goals are often treated using artificial intelligence and machine learning algorithms that uses APK characteristics and artifacts obtained at execution time.

The problem of detecting malicious applications is outside of the scope of this thesis: as stated in Chapter , we focus on studying code protection and exposing vulnerable applications. Thus, we will not describe the entirety of works related to this problem. Nevertheless, we highlight DroidClone [30] which focuses on a problem close to this thesis: detection of native Android malware specifically. DroidClone provides a mechanism to build malware signatures. It operates on assembly code but handles both native code and bytecode by compiling the bytecode using the compiler provided by the ART runtime[1]. This idea is an elegant way to handle bytecode and assembly code simultaneously. We used a similar approach to propose a new obfuscation method in Chapter 3.

1: This runtime is described in Section 2.4

Studying code protection Studying code protection consists in two opposite goals that both need to be explored. One may want to make the analysis of an application more difficult. This process is called *obfuscation*. At first sight, it could be surprising that some security researchers try to invent new obfuscation techniques or improve existing ones since they are used by malicious applications to circumvent analysis tools. However, benign applications can legitimately use obfuscation, for example, to protect their intellectual property or to avoid being repackaged. Additionally, as mentioned in Chapter 2.1, determining what kinds of obfuscations malware will potentially use in the future allows to develop countermeasures and tackle malicious application faster.

On the contrary, some researchers try to break obfuscation. Breaking an obfuscation technique can itself be divided in different goal variations discussed hereinafter. It can consist in detecting the usage of the obfuscation, retrieving the original code, or getting information about the real application behavior while being agnostic about the targeted obfuscation.

Detection techniques are useful to determine if a specific obfuscation technique is used by applications in the wild. It can be used as a first step, to determine if a deobfuscation technique, potentially resource-consuming, should be launched. Most of the time detection techniques try to spot artifacts that reveal traces of the usage of a known obfuscation technique. Consequently, unknown obfuscation methods are not detected since their artifacts are also unknown.

When a tool tries to analyze an obfuscated application, it may fail, for example if an obfuscation technique ciphers the code, therefore making it unavailable for static analysis. Facing this problem, the first solution is to retrieve the original code of the application in order to work as if the application had never been obfuscated. This process is called *deobfuscation*.

Since deobfuscation is the principal threat of obfuscation, sophisticated techniques aim at preventing it. In some cases, such as the usage of packers or reflection methods, the obfuscation cannot be fully reverted. Then, analysis tools need, in order to work properly, to determine information that is relevant for their goal but invariant with the obfuscation. For example, malicious application detection can be conducted over network communications [34] or system calls patterns [35].

Discussions about code obfuscation in the specific context of native applications are developed in Sections 3.1 and 5.1.

Exposing vulnerable applications Exposing vulnerable applications consists in determining if a given application is vulnerable to security attacks. This goal seems to be inherently malicious. But, this is also legitimately used by developers or companies that want to check, before using it, that an external library or an application is safe. It is also used by developers to check their own application and, if a vulnerability is found, patch their application.

First, security researchers can manually look for vulnerabilities and highlight new problematic issues. For example, Peles and Hay [36] showed that a missing `transient` keyword in a Java field can lead to severe exploits if such a field contains a native address. This approach is precisely described in Section 4.1 because a solution of this problem is one contribution of this thesis.

Then, for particularly widespread vulnerabilities, researchers design methods targeting them. This can be done by statically analyzing the bytecode. Lu et al. [37] and Zhang and Yin [38] looked for component hijacking vulnerabilities and Sounthiraraj et al. [39] for SSL man-in-the-middle vulnerabilities. Gu et al. [40] found JNI Global References exhaustion (JGRE) by statically analyzing native code and bytecode. These solutions have a high accuracy for detecting the considered vulnerability but keep bounded to this specific vulnerability.

Some researchers adopt a more generic approach. They do not search for a specific vulnerability but have designed methods that can work for different ones. For example, Qian et al. [41] transform the bytecode of an application into an annotated CFG and translate vulnerabilities into graph-traversal properties. Dhaya and Poongodi [42] built a machine learning system that translates application code into N-grams and automatically learns to recognize vulnerable applications. However, these approaches [41, 42] do not evaluate their detection ratio but only report vulnerabilities found in application datasets. While these approaches are useful in the wild, they cannot prove that a given application is safe.

Some researchers focus on generic dynamic approaches. Sounthiraraj et al. [39] run the application and redirect external SSL connections to a crafted server that attempts to perform man-in-the-middle attacks. Yang et al. [43] and Sasnauskas and Regehr [44] developed an Intent fuzzer. A fuzzer is a tool that consists in generating invalid and faulty inputs and pass them to a system under attack. This approach is well suited for Android. Indeed, in Android, applications communicate with each other using Intents. Intents are Java objects that are sent trough the binder, the Android IPC mechanism. In order to work properly, the binder requires applications to declare, in an AIDL[2] file, the types of Intents they are willing to receive. This file is stored in the APK archive and fuzzers exploit this AIDL file to generate inputs that are not rejected by the application, revealing vulnerabilities. Similarly to the preceding generic approach, the accuracy detection is difficult to evaluate.

2: Android Interface Definition Language

Finally, researchers develop solutions to improve and facilitate the correction of vulnerabilities. For example, Zhang and Yin [38] automatically propose a patch for vulnerable bytecode and [45] developed a system to patch the Android system when manufacturers do not update properly the system.

In this thesis, we have studied a specific vulnerability involving native code introduced by Peles and Hay [36] for which we have built a dedicated method.

The implementation of some of the solutions presented above introduces additional challenges. First, dynamic systems that emulate Android are not transparent. This means that malicious applications can detect that they are being analyzed, and then choose to behave differently. This is called emulation system evasion [46, 47]. Another challenge, extensively studied in Android, is the problem of code coverage [48–52]. The Android application architecture is very modular and event-driven. Applications are composed of multiple activities and services, that can be triggered using various ways such as user interaction and IPCs[3]. These services and activities constitute multiple entry-points of the Android application, in contrast with classic desktop executables that only have one single entry-point. This is problematic when a dynamic system wants to stimulate the execution to cover as much code as possible. For example, Abraham et al. [49] propose to explore exhaustively the graphical interface of applications under analysis.

3: Inter-Process Communication

These challenges have not been faced during this thesis and thus, these solutions are not further described.

2.2 Android application datasets

As in any research field, the Android security community needs datasets to evaluate their methods and produce easily reproducible experiments. Thus, various datasets exist [53]:

- Google Play store: official Android application repository. It contains more than two million applications. However, this dataset is not suitable for scientific experiments since it highly dynamic: applications are added, removed or updated frequently. It not easily retrievable: Google does not provide an API to download applications. Finally, applications are not labeled as goodware or malware since malware are removed from the store.

- Genome [54]: ground-truth dataset of more than 1,200 malware samples. This dataset is qualified as ground-truth, meaning that every application has been manually verified to be malicious. Unfortunately, the service is no longer maintained by the authors. Copies of this dataset are still available but it is no longer representative of malware in the wild [55].
- Drebin [56]: a dataset of 123,453 applications and 5,560 malware. Malware have been detected using VirusTotal [4], an online malware detection service. This dataset is largely used as a detection benchmark in the research community, such that at the time of writing, it has been cited more than two thousands times. However, similarly to the Genome dataset, it is getting old and not representative. For example, it contains only very few native applications.
- AndroZoo [57]: a dataset of more than 13 million applications. The authors are continuously downloading new samples from various stores, including the Google Play store. Samples come with metadata such as size, checksum and retrieve date. Additionally, they also tag if the application is malicious using VirusTotal.
- AMD [58]: a ground-truth dataset of 24,553 malware classified among different families.
- GM19 [4]: contains two balanced sets of 5,000 goodware (GOOD) and 5,000 malware (MAL) with an homogeneous distribution of dates (2015-2018) and APK size to avoid statistical biases.
- Contagio mobile[5]: web repository containing 252 malicious applications.
- Koodous[6]: web repository containing 19 million malware out of 66 million applications.

4: https://www.virustotal.com

5: https://contagiominidump.blogspot.com/

6: https://koodous.com/

It is worth noting that Drebin, AndroZoo and GM19 datasets use VirusTotal, an online detection service which aggregates the results of around 50 antivirus, to classify applications between goodware and malware. However, we believe that this approach is not reliable. In [2], we collected 2,000 malware samples by downloading each day 20 recent samples from the Koodous repository and 30 random samples from the AndroZoo repository. As shown in Figure 2.1, 48% of samples are not recognized by any antivirus used by VirusTotal. Figure 2.1 shows that there is no obvious threshold to decide that a sample has been recognized by enough antiviruses to classify it as a malware. We were expecting a drop of detection for a certain number of antiviruses[7], as represented by the light blue curve. Additionally, these results may change with time, as the pool of antiviruses used by VirusTotal frequently updates their signature database. From this experiment, we conclude that using VirusTotal as an oracle for confirming that a sample is malicious is not reliable, especially for recent samples.

7: Arbitrarily set to 27 in the graph.

All the aforementioned datasets are used to test malicious detection techniques. For obfuscation studies of this thesis, as we do not attempt to detect malicious applications, we only use datasets to detect the presence of obfuscation techniques in the wild and thus, focus on recent datasets: AndroZoo, AMD and GM19.

Also, we have not found any dataset of firmware applications, that is applications pre-compiled and installed on the smartphone by the manufacturer or the firmware vendor. Since we have developed an obfuscation technique specifically for this kind of applications[8], we have

8: See Chapter 3.2.2.1

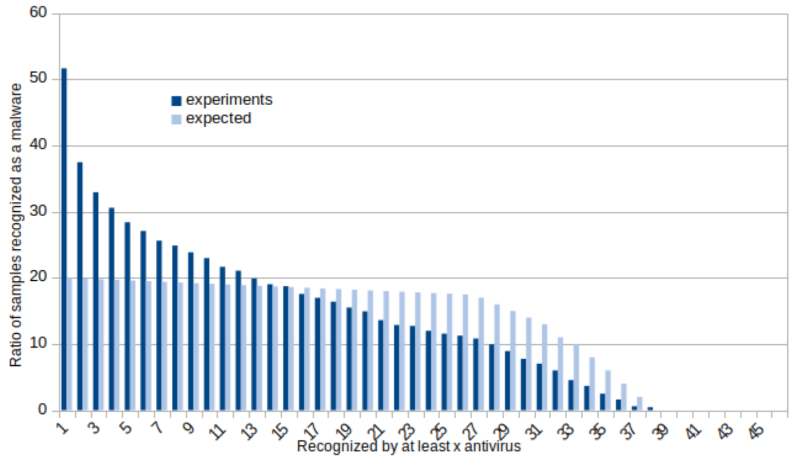


Figure 2.1: Ratio of recognized malware by at least x antivirus

constructed one for our experiments. We have downloaded 17 firmwares from six different brands[9]. All the firmwares run Android 7.0 or 7.1. For each firmware, all compiled applications have been extracted. The complete list of firmware is available in Appendix A.

9: Alcatel, Archos, Huawei, Samsung, Sony, Wiko

For the detection of vulnerable applications, we found only one dataset named Ghera [59]. It is an open source repository of vulnerable and safe applications. For each vulnerable application, details about the vulnerabilities present in the application are provided. Unfortunately, this dataset does not contain samples for the vulnerability we have studied in this thesis: missing `transient` keyword.

2.3 Analysis techniques

To achieve their goals (detecting malicious applications, studying code protection and exposing vulnerable applications), researchers rely on several techniques, used as building blocks that can be tuned and combined together to tackle specific problems. This section reviews these different techniques. Classically, techniques are separated between static, that study the data and the code of the applications, and dynamic ones, that observe executions of the applications.

However, these two sets of techniques are not disjoint. For example, symbolic execution is a static technique since it does not execute the application. Nevertheless it attempts to mimic a possible set of executions. On the other side, some dynamic techniques, such as fuzzing, rely on a preliminary static analysis phase used to configure the subsequent dynamic phase.

In this section, we have chosen to present techniques from high-level to low-level. Indeed, this thesis deals with applications composed of Java and assembly code, that are languages of completely different levels. Such a classification is relevant in this context.

2.3.1 System side-effects

We consider as high level techniques, the ones that work with artifacts left by the execution of the analyzed application rather that the application itself. For example, Shao et al. [60] observe the network communications to detect unsafe applications that accept remote commands without any preliminary authentication phase. Bhatia et al. [61] analyze memory snapshots and reconstruct a timeline composed of, for example, activities and services that have been launched. These approaches are too high-level for handling specifically native code.

2.3.2 Application metadata

Looking at techniques getting closer to the application and the system, researchers can work on the application metadata. Metadata about Android applications is stored in the APK archive inside a file called `Manifest`. This file contains:

- The list of permissions required by the application.
- The list of activities: classes that represent an interface window.
- The list of services: classes that are launched in the background.
- The list of receivers: classes that are able to receive messages sent by other application or by the system.

Metadata-based techniques can work on permissions and API calls [62] or permissions and application description[10] [63]. Actually, all the information stored in the`Manifest` can be used to achieve malicious detection [64]. Again, such approaches cannot be of interest for native code.

10: Description shown in the Google Play store and written by the application developer.

2.3.3 Bytecode level

Techniques can look at the application bytecode. This bytecode is stored inside the APK archive as DEX[11] files. New DEX files can be loaded during the execution by the bytecode itself. That means static analysis cannot, in the general case, cover all the code. This bytecode is named Dalvik bytecode, after the name of the virtual machine that interprets or JIT-compiles[12] it: the Dalvik VM. This bytecode is a register-based version of the Java bytecode. Bytecode level analysis techniques are widely used by researchers because the bytecode is clearly the place where the behavior of the application is described.

11: Dalvik EXecutables

12: Just-In-Time

In order to analyze application bytecode, solutions can use bytecode simplification techniques before conducting their analysis. This allows to reduce the amount of resources needed to process an application. For example, SAAF [65] proposes to compute slices of the bytecode according the dataflow of this instruction. An instruction is part of a slice if, given a value (variable, object field, ...), this instruction participates to the computation of this value. Similarly, Harvester [66] slices the program according to the dataflow of a given value. Then, it executes the obtained slice and logs, at runtime, the value.

To simplify their analysis, tools can also use Intermediate Representation (IR) such as Jimple [67]. An Intermediate Representation (IR) is a language that abstract lower-level languages. It is used to make writing optimization

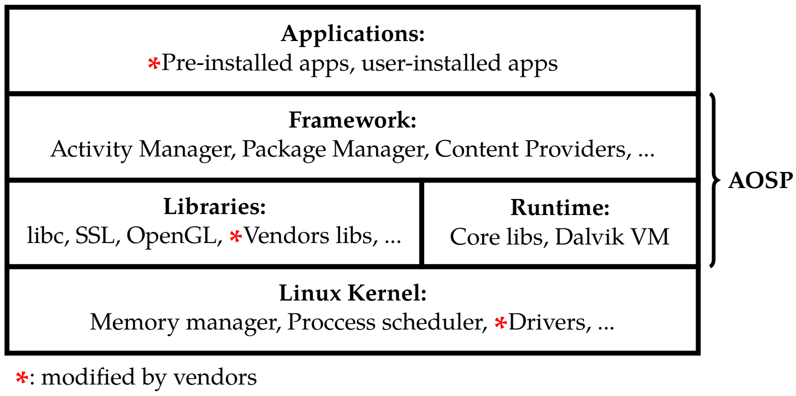


Figure 2.2: Classical representation of Android system architecture

rules easier. For example, AppSealer [38] uses program slicing according to dataflow performed over Jimple, rather than on the bytecode, to search for component hijacking vulnerabilities.

Taint analysis is a common analysis conducted on application bytecode. In particular, we have used the taint analyzer provided by FlowDroid [68] to perform the analysis conducted in Section 6.2. A taint analysis consists in identifying, for a given list of sources, all the sinks that can receive a taint. Usually, in bytecode analysis, sources and sinks are calls to framework methods and taints are the return values of the sources. It allows to represent, for example, the leakage of the IMEI[13] using the `getImei` method as a source and methods such as `Socket.writeUTF` or `File.write` as sinks. In this case, the taint is the IMEI number.

13: International Mobile Equipment Identity, number that identify uniquely a mobile device

An other common analysis technique is symbolic execution [69–71]. It consists in following the program instructions and recording, for each value, the constraints that are applied. This is midway between the static and the dynamic execution: the code is run "symbolically" using abstract values instead of concrete ones. In particular, it is used to compute all the possible values that a variable can contain during an execution or to determine if a given instruction could be reach during an execution.

2.3.4 Framework, runtime and system level

Analyzing only the bytecode of an application does not allow to easily manipulate the application behavior. Indeed, for introducing or modifying a specific behavior into an Android application, one has to translate this behavior into bytecode and inject this bytecode in the application itself.

To overcome this limitation, some techniques propose to modify the framework, the runtime or the system. As shown in Figure 2.2, applications rely on these three elements to be executed. Thus, by modifying these low-level architecture elements, solutions can manipulate and instrument applications freely. These elements can be modified by researchers since they are open-source: the system is a Linux kernel and AOSP[14] provides the source code of the framework, the libraries and the runtime. However, Android systems installed on smartphones are customized by the smartphone provider (also called vendor) in order to

14: Android Open Source Project: `https://source.android.com/`

provide a functional device. This limits the portability of the techniques' implementations.

Thus, numerous framework, runtime or system level techniques have been proposed. For example, AndroBlare [72, 73] enhances the system using Linux modules to hook system calls and perform taint analysis. TaintDroid [74] modifies the Dalvik Virtual Machine (DVM) interpreter to manage taints. CopperDroid [75] runs Android inside the QEMU [76] emulator and introspects this emulator to reconstruct the behavior of the application such as activity launching and SMS sending.

These kinds of contributions are very tuned and propose a complete overview. Thus, most of them handle native code, which is the subject of the following section.

2.4 Challenges implied by native applications

The Android runtime allows applications to embed native libraries using the classical shared object file format[15] and to call native code from Java code. Such applications are called native applications. The communication between Java and native code is realized through a dedicated interface called the Java Native Interface (JNI). This interface allows not only to call native functions from the Java world, but also gives native code the opportunity to access Java objects and fields.

15: ".so" extension

Since 2014, the usage of native Android applications is rising. Indeed, researchers have started reporting usage of obfuscation techniques designed for assembly code [77]. More recently, Afonso et al. [16] performed a large-scale analysis to evaluate the usage of native code in a dataset of 1.2 million Android applications, and showed that more than one third of these applications potentially used native code.

Additionally, Android released a new runtime called ART in 2014[16]. This runtime no longer interprets or JIT[17] compiles the bytecode of applications but instead compiles the bytecode into assembly before the execution. This is called AOTC for Ahead Of Time Compilation. Two years later[18], Android has reintegrated the interpreter and the JIT compiler on top of the AOTC-compilation. Since then, Android applications are not fully compiled. A method is compiled only when it is frequently executed. The resulting assembly is stored using a new file format called OAT[19]. For the sake of clarity, when the differentiation between assembly codes stored in shared objects and OATs file is needed, the assembly code from OAT is named `quick_code`[20].

16: Android 5.0 "Lollipop"

17: Just In Time

18: Android 7.0 "Nougat"

19: We could not find an official definition for this acronym.

20: This denomination comes from the Android source code.

The usage of native obfuscation techniques and the compilation of application bytecode have increased the needs for adapting solutions to the assembly world. As discussing in details all the contributions of the literature would be technically difficult at this stage of the manuscript, we propose to briefly categorize the different objectives of authors. Then, later in the manuscript, we point out the limitations of each approach for the specific problem we solve. Globally, the research community has:

- **Studied new obfuscations**: Researchers have developed solutions to tackle the new rising obfuscation technique called packing [78–

80]. This technique consists in ciphering the bytecode to make it unavailable for static analysis. The various deobfuscation techniques are detailed in Section 3.1 and the associated detection methods are presented in Section 5.1. In Section 3.2, we proposed new native obfuscation techniques and associated detection techniques in Section 5.2.
- **Exposed new vulnerabilities**: New vulnerabilities targeting Android applications have been discovered [36, 40]. In Section 4.1, we precisely describe the vulnerability proposed by Peles and Hay [36]: a field that stores a native address can be exploited if it is not declared `transient`. We propose, in Section 6.2, a solution to the unresolved problem of detecting such vulnerable fields.
- **Ported taint analysis across the Java Native Interface:** Many works [81–84] aimed at tracking the information flow during the execution of a native Android application. For example, NDroid [81] propagates taints generated by TaintDroid [74] by hooking the Android framework methods that call native code and all JNI entry points. All proposed solutions rely on JNI to perform their analysis which, as stated in Section 4.2, it is possible to bypass. Also, as discussed in Section 6.1, they cannot achieve taint analysis that requires the type of assembly values.
- **Improved instrumentation systems:** Several works have presented generic framework solutions [85–88] where the analyst can insert some hooking code to audit native code actions. These frameworks can be used to observe, for example, virtual method calls [85] (vtable hooking) and library calls [86] (PLT hooking). While these solutions bridge the gap between native code and bytecode analysis, we show in Section 7.1 that they are not resilient to all native obfuscations. We propose a new approach in Section 7.

While numerous articles focus on solving specific challenges implied by native applications, no systematic studies about the impact of native code on the security of Android applications in its entirety has been conducted. This is one of the contributions of this thesis.

2.5 Conclusion

We discussed the different global goals of researchers when dealing with Android security. We focused our research efforts on code protection ensured by obfuscation techniques and the research of vulnerabilities. The chapter summarized the state of the art for these two problems and we identified that the introduction of native code in applications brings new challenges. The precise discussion of the articles that are close to our contributions are discussed in the relevant chapter. We also summarized the analysis building blocks that are classically used when analyzing statically and dynamically applications. Some of them will be reused in our contributions.

As new challenges that the thesis address are brought by the interaction between the native and bytecode codes, we conduce a systematic review of all interferences between native and bytecode worlds, both at code and data level, respectively in Chapter 3 and 4. We demonstrate that these interferences help the developer to create new obfuscation techniques, but in the meantime, can introduce vulnerabilities. Then, the next part of the manuscript gives solutions to detect these interferences in Chapter 5 and 6. Finally, in the last part, we describe in Chapter 7 a generic analysis solution for obfuscated applications and gives insight on its implementation in Chapter 8.

Security issues introduced by native interferences in Java Android applications

3 Security issues introduced by interferences in Java code

Android applications, when developed in Java or Kotlin, are composed of Dalvik bytecode[1]. This bytecode works at the object level, that is, bytecode instructions use the abstractions defined in object-oriented programming. Here are a few examples of Dalvik bytecode instructions that realize object-level actions:

- `move-object vA, vB`: moves an object from the register `vB` to the register `vA`;
- `check-cast vAA, type@BBBB`: throws a `ClassCastException` if the reference in the given register `vAA` cannot be cast to the type indicated by `type@BBBB` ;
- `invoke-virtual vC, vD, vE, vF, vG, meth@BBBB`: invokes the virtual method `meth@BBBB` using `vC-G` as arguments.

1: https://source.android.com/devices/tech/dalvik/dalvik-bytecode

On the other side, functions in C/C++ can be compiled into assembly code and inserted into an Android application. This assembly code changes depending on the underlying smartphone processor[2]: assembly instructions are architecture-dependent. Thus, assembly code is low-level, compared to the Dalvik bytecode. Object-oriented programming abstractions such as methods, types, memory management are completely absent from assembly code. Because assembly code has access to the processor architecture, it can perform operations that Dalvik bytecode cannot. For example, it directly accesses the memory and so can manage it entirely. It also has access to kernel system calls and can, if given the appropriate system permissions, tune the kernel.

2: x86/x86_64 for Intel processors, ARMv7/v8 for ARM processors, MIPS32/MIPS64 for MIPS processors.

Consequently, assembly code is harder to understand than bytecode. Therefore, application developers may want to leave native code instead of Dalvik bytecode in their applications in order to prevent them from being analyzed. However, developers are not likely to develop applications in C/C++ rather than in Java. Indeed, the lack of high-level abstractions complicates the development of C/C++ only applications. Developing an application in C/C++ for the sole purpose of obfuscation is therefore not conceivable. Even if native code cannot totally replace bytecode, it can still be used to hide or to alter Dalvik bytecode behavior and, thereby, fool analysis tools.

This chapter describes two obfuscation techniques that use native code to hide Dalvik bytecode. The first technique, presented in Section 3.1, is called packing and consists in ciphering the bytecode using native code. The second technique, presented in Section 3.2, is called AOTC-based bytecode hiding scheme and consists in replacing all the bytecode by native code.

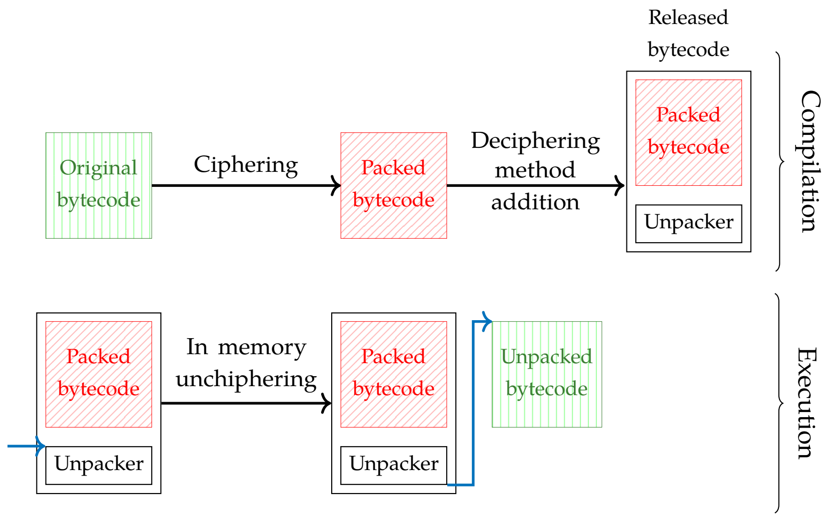


Figure 3.1: Packing technique

3.1 Modification of the Java bytecode by the native code

A packer is a tool that aims to make reverse-engineering of a program more complex, while retaining its original behavior, by ciphering all or part of the program code. Packers are not specific to Android and therefore have already been widely studied, especially for the x86 environment [89]. These works study the effects of packers at the operating system level. However, they are not applicable to Android [78–80]. Indeed, AOSP introduces an additional level between the kernel and the application which is not present in classical operating systems, and which must be taken into account.

[89]: Ugarte-Pedrero, Balzarotti, Santos, and Bringas (2015), 'SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers'

[78]: Zhang, Luo, and Yin (2015), 'Dexhunter: toward extracting hidden code from packed android applications'
[79]: Yang, Zhang, Li, Shu, Li, Hu, and Gu (2015), 'Appspear: Bytecode decrypting and dex reassembling for packed android malware'
[80]: Xue, Luo, Yu, Wang, and Wu (2017), 'Adaptive unpacking of Android apps'

A packer creates a new application, called a packed application, from the original one. This process can be split into two main phases which happens at compilation time and are depicted in Figure 3.1. First, the packer ciphers the original code which is contained in the DEX[3] of the original application and therefore creates a new DEX called packed DEX. Second, it adds a decryption routine, called unpacker, to this packed DEX. This routine is in charge of deciphering the DEX file and loading it dynamically during the execution. This mechanism is generally implemented in a native library.

3: The file that contains the bytecode of the application. See Section 2.3.3.

Thus, the DEX file of a packed application does not contain the original bytecode but a stub that calls the native decryption routine. Statically analyzing this DEX file would lead to analyze the decryption routine which would be expensive and difficult. Manual techniques, that is to say, understanding the complete functioning of the decryption to apply the reverse function to all packed DEX files, are not processed as they vary for each packer and therefore pose a scalability problem.

The evolution of packers has followed the evolution of unpackers, that is the tools that aims to retrieve the original application from the packed one. The remaining of this section traces this cat-and-mouse game.

3.1.1 Full unpacking

The decryption routines of the first packers fully decrypt the packed DEX file before using the Android framework to dynamically load the whole unpacked DEX file [77, 90]. The original DEX file is therefore present in memory when the application is executed. Thus, the first unpacking techniques consist in running the packed application. Once it is launched, the unpacker searches for the signature of the DEX file inside the memory. For example, it is possible to search for its magic number, *i.e.* the characteristic bytes of the start of a DEX file[4]. When the original DEX file is found, traditional analyses can be launched on the recovered DEX file.

[77]: Yu (2014), 'Android packers: facing the challenges, building solutions'
[90]: Strazzere and Sawyer (2014), 'Android hacker protection level 0'

4: All DEX files begin by the following bytes: `64,65,78,0A,30,33,35,00`

3.1.2 Unpacked bytecode hiding

When the Android framework loads a DEX file, some parts, for example the magic number, are not used. It is therefore possible to modify them without changing the behavior of the application. By altering these specific points, some packers manage to prevent the localization of the unpacked DEX file in memory. Thus, previously described unpackers become ineffective. In any case, the Android framework needs to know the location of the unpacked DEX file and so, this address is supplied by the packer during the dynamic loading of the DEX file. The unpackers have therefore chosen to overload the functions of the Android framework responsible for loading the DEX files [91–93]. Thanks to this method, the memory location of the unpacked file is available to them which allows them to recover the original bytecode of the application even if some characteristics of the DEX are altered.

[91]: Kim, Kwak, and Ryou (2015), 'Dwroiddump: Executable code extraction from android applications for malware analysis'
[92]: Park (2015), 'We can still crack you! general unpacking method for android packer (no root)'
[93]: Bashan and Makkaveev (2017), 'Unboxing Android: Everything You Wanted To Know About Android Packers'

3.1.3 Partial unpacking

In order to prevent the original DEX file from being unpacked entirely, the behavior of the decryption routines was subsequently changed to no longer leave the DEX file completely unpacked in memory. DEX is deciphered by parts. So a deciphering routine deciphers only one function or one class, right before using it, and then re-ciphers it. Unpackers have therefore also evolved in order to be able to recover the different parts of the DEX file before assembling them again [78, 79, 94]. They overload the Android framework functions that are responsible for loading classes, methods, and opening a DEX file. When overloaded functions are called, the unpacker retrieves each corresponding part. At the end of the execution of the application, all parts are assembled in one final DEX. These techniques, although automatic, suffer from not fully retrieving the original DEX. Indeed, only the parts that are loaded during a specific execution are retrieved. Some unpackers [94] fix this problem by, for example, simulating class loading by directly calling the function of the Android framework that loads classes. It assumes that the method decryption is performed when loading the class.

[78]: Zhang, Luo, and Yin (2015), 'Dexhunter: toward extracting hidden code from packed android applications'
[79]: Yang, Zhang, Li, Shu, Li, Hu, and Gu (2015), 'Appspear: Bytecode decrypting and dex reassembling for packed android malware'
[94]: Jiang, Zhou, Liu, Jia, Liu, and Zuo (2017), 'CrackDex: Universal and automatic DEX extraction method'

3.1.4 Android framework bypassing

The last type of packer that have appeared are the packers which embed their own copy of the Android framework [80, 95]. They use their own functions to load the different elements of the DEX file. Thus, the function overloading made by unpackers are ineffective since the Android framework is never called. Unpackers that manage this type of packers are not fully automatic [80, 95]. They propose to trace the execution of the packed application while monitoring modifications of the DEX files present in memory. This trace is realized by hooking numerous Android framework functions, system calls and all store instructions[5]. By analyzing such traces, it is then possible to determine the moment when a part of the DEX is unpacked. This is down manually by Xue et al. [80] and automatically by Wong and Lie [95]. These points are used as collecting points during a new run of the packed application. During this new run, a new trace is also made and new collecting points can be defined. The process is repeated until no new collecting points are defined.

[80]: Xue, Luo, Yu, Wang, and Wu (2017), 'Adaptive unpacking of Android apps'
[95]: Wong and Lie (2018), 'Tackling runtime-based obfuscation in Android with TIRO'

5: Using tools such as Valgrind [96]
[96]: Nethercote and Seward (2007), 'Valgrind: a framework for heavyweight dynamic binary instrumentation'

3.2 Replacement of the Java bytecode by native code

Assembly code is harder to understand than Dalvik bytecode. Thus, writing application fully in C/C++ is better for obfuscation purposes than in Java. However, the low level of abstraction proposed by C and C++ makes the development of such applications very difficult and error prone. If the manual creation of full native applications is not conceivable, the bytecode can be translated, compiled, into assembly. This is not uncommon: the Android runtime performs this compilation, for optimization purposes, when installing applications[6]. Such compilation, which happens before executing the application, is called "ahead-of-time", in opposition with "just-in-time", which corresponds to a compilation happening during the execution of the application.

6: Since Android 7.0 (Nougat, 2016), the Android runtime compiles methods of the application when they are frequently run.

This section presents an obfuscation technique named AOTC-based[7] bytecode hiding scheme which consists in compiling the Dalvik bytecode into assembly code and then modifying the bytecode in order to make the bytecode unavailable for analysis. Thus, the original bytecode, which no longer exists, is protected against both static and dynamic analysiss. This technique can be characterized by the type of compiler used and the type of modifications made to the bytecode.

7: AOTC stands for Ahead Of Time Compilation

3.2.1 Bytecode compiler used

The bytecode of obfuscated methods can be compiled using a custom compiler [97] or the one given by Android system. If a custom compiler is used, the resulting assembly is put into a shared library that is added to the application and the DEX file is modified to remove bytecode and set the method as `native`. This step is mandatory because Android does not support that a method tagged as `native`, has bytecode. Finally, calls to obfuscated methods are converted into Java Native Interface (JNI) calls.

[97]: Bao, He, and Wen (2018), 'DroidPro: An AOTC-Based Bytecode-Hiding Scheme for Packing the Android Applications'

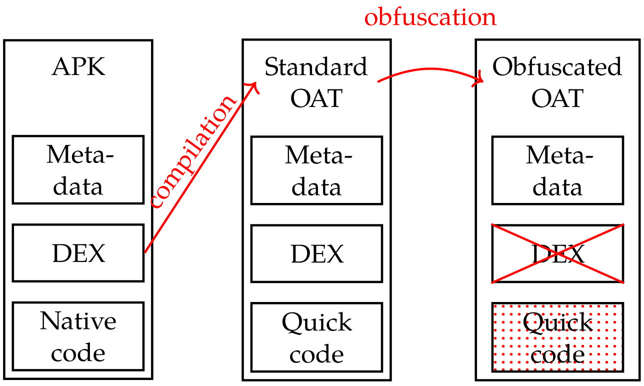


Figure 3.2: Bytecode Free OAT technique

If the Android compiler is used, the resulting assembly cannot be put into a shared library file. Indeed, the Android compiler outputs code which is supposed to be stored inside an OAT file. This assembly, named quick code, differs from classical native code. The Android compiler is customized to optimize the code for the smartphone on which the compilation is made[8]. Additionally, native and quick code have different calling conventions and thus can not be used interchangeably.

8: Compilers can hardcode offsets of system libraries functions for example.

Since OAT files cannot be distributed through classical application markets, using the Android compiler to obfuscate the application would be particularly well-suited to firmware vendors: these companies provide their applications already pre-compiled for a specific phone model.

We have called this specific technique, represented in Figure 3.2, Bytecode Free OAT (BFO). After the compilation, the OAT file is directly modified to change the bytecode inside the DEX. Contrary to shared libraries, OAT files support interleaved bytecode and assembly. When Android executes an application, the quick code is always executed, if it is available, regardless of whether bytecode is present or not. Thus, an attacker could tamper with the bytecode without modifying the executed quick code. Thereby, the application behavior is not changed but the analysis of the bytecode would be erroneous since it is not performed on the actual code. Possible bytecode modifications are discussed in the following section.

3.2.2 Types of bytecode modifications

Depending on how the bytecode is tampered with, we propose three different BFO sub-techniques in the next three following sections: removing, replacing or modifying the bytecode. These techniques are classified according to three criteria: their robustness, their stealthiness and the possibility to automate them. Since assembly code works at a lower abstraction level than the bytecode, we consider that analyzing bytecode is simpler than analyzing assembly. Consequently, we consider that an obfuscation technique is more robust than another if it requires to analyze more assembly code. On the other hand, we consider that an obfuscation technique is stealthier than another if the difference between the behavior described by the bytecode and the one observed is smaller: analysts only look at the assembly code if the result of the bytecode analysis seems incorrect with respect to the behavior of the application.

3.2.2.1 Removing the bytecode

The first variant of this BFO technique consists in removing or "nopping" the bytecode. This means replacing the bytecode by nop[9] instructions or, by extension, by instructions that do not have any special effect. Removing the bytecode is allowed by the OAT file format in order to represent `abstract` methods. The native part, which is always executed regardless of whether bytecode is present, is not modified, in order to preserve the application behavior.

9: No-OPeration, assembly instruction that does not have any effect except increasing the instruction pointer

This technique perfectly fools bytecode analysis tools since the information on which they perform their analyzes is deleted. Instead, the reverse engineering of the application needs to be done directly on the assembly code. Additionally, this technique is easily automatable since the modifications applied to the bytecode are the same for all applications and do not depend on the removed bytecode. However using this technique is not stealthy since bytecode analysis cannot give any result if there is no bytecode provided at all.

3.2.2.2 Replacing the bytecode

As previously stated, removing or nopping the bytecode is not stealthy. A stealthier approach is to replace the bytecode: the bytecode of a sensitive method can be replaced by a benign method[10]. Thus, the robustness of the obfuscation technique is kept while improving its stealthiness: the bytecode still does not give any information about the behavior of the application and analysis tools generate wrong results since they do not have the right bytecode to work on.

10: For example, a method that checks the PIN code can be replaced by a hello world

The automation of this technique is still possible but is not trivial. The bytecode cannot be replaced by repetitive patterns of bytecode because this would be easily detectable. The automation can neither generate random patterns because this would result in incorrect bytecode instructions, which is also easily detectable using a bytecode verifier. Thus, automating this technique requires to generate random valid bytecode, that is, bytecode which respects, among other, the signature of the replaced methods. While it is still doable, it requires some engineering work, and is left as future work.

3.2.2.3 Modifying the bytecode

Finally, if the stealthiness of the obfuscation is considered more important than the robustness, a third BFO technique can be used. This technique consists in slightly modifying the bytecode. Instead of completely modifying the bytecode behavior, only a few instructions that are chosen very carefully are minutely touched.

For example, if someone wants to protect a code that contains a CRC check of incoming network packets, obfuscating the creation of the CRC table would be a typical goal. The bytecode corresponding to such a method is presented in Listing 3.3a. This bytecode has been obtained by compiling an application containing CRC computations and inspecting the resulting DEX file. At line 8, the bytecode initializes the polynomial that is used to compute the CRC table[11]. If only this line is modified,

11: `0xedb88320` in this case

```
1200           | const/4 v0, #int 0
1301 0800      | const/16 v1, #int 8
3510 1400      | if-ge v0, v1, 14
dd01 0401      | and-int/lit8 v1, v4, #int 1
1212           | const/4 v2, #int 1
3321 0a00      | if-ne v1, v2, 11
e201 0401      | ushr-int/lit8 v1, v4, #int 1
1402 2083 b8ed | const v2, #edb88320
9704 0102      | xor-int v4, v1, v2
2803           | goto 12
e204 0401      | ushr-int/lit8 v4, v4, #int 1
d800 0001      | add-int/lit8 v0, v0, #int 1
28eb           | goto 02
1500 00ff      | const/high16 v0, #int -16777216
b740           | xor-int/2addr v0, v4
0f00           | return v0
```

(a) Original CRC32 bytecode

```
1200           | const/4 v0, #int 0
1301 0800      | const/16 v1, #int 8
3510 1400      | if-ge v0, v1, 14
dd01 0401      | and-int/lit8 v1, v4, #int 1
1212           | const/4 v2, #int 1
3321 0a00      | if-ne v1, v2, 11
e201 0401      | ushr-int/lit8 v1, v4, #int 1
1402 2ed8 31eb | const v2, #eb31d82e
9704 0102      | xor-int v4, v1, v2
2803           | goto 12
e204 0401      | ushr-int/lit8 v4, v4, #int 1
d800 0001      | add-int/lit8 v0, v0, #int 1
28eb           | goto 02
1500 00ff      | const/high16 v0, #int -16777216
b740           | xor-int/2addr v0, v4
0f00           | return v0
```

(b) Modified CRC32 bytecode

Figure 3.3: Bytecode modification example

as shown in Listing 3.3b, the bytecode analysis of the application does not raise any alarm: the result is consistent with the behavior of the application. Nevertheless, using the wrong polynomial[12] would not allow the analyst to generate correct CRCs.

12: `0xeb31d82e` in the modified bytecode

However, this technique presents two main drawbacks. First, it is less robust. Even if the bytecode differs from the assembly, it still gives a lot of insight about what is the behavior of the application. Second, it is not automatable. Indeed, modifications that are made to the bytecode require a very precise knowledge about the behavior of the bytecode to be obfuscated.

3.2.2.4 Comparison of the three bytecode modification sub-techniques

The three BFO sub-techniques previously described are classified in Figure 3.4 according to their robustness, their stealthiness and the possibility to automate them. Removing the bytecode is the one that is the easiest to automate. However it is the least stealthy. Replacing the bytecode can be viewed as an improvement of simply removing it, since it improves the stealthiness while not reducing the robustness. Nevertheless, its process is harder to automate. Finally, modifying the bytecode is the stealthiest sub-technique but reduces robustness of the obfuscation and requires manual editing.

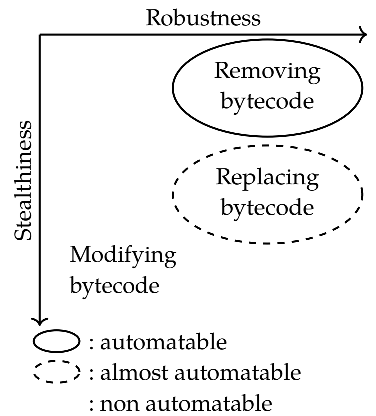


Figure 3.4: Classification of BFO sub-techniques

3.3 Conclusion

This chapter has presented the possible interferences occurring on the Java code from the native code. These interferences are particularly useful for a developer who wants to obfuscate the bytecode of an application. Different techniques of unpacking have been presented and we introduced a new hiding technique, called BFO, where the native code replaces the bytecode of a compiled pre-installed application. Chapter 5 will present the detection methods for these obfuscation techniques.

In the next chapter, we continue to review the possible interferences of the native code over the bytecode world by moving our attention from the code to the data: more precisely, we will study the possible impact of the native code on Java data.

4 Security issues introduced by interferences in Java data

Java is an object-oriented programming language. That means data is represented as small functional entities named objects. These objects contain values named fields. These fields, in Java, can be either an other object or a primitive type[1].

1: Java defines eight primitive types: boolean, byte, char, short, int, long, float, and double

The memory management of the objects is not handled by the programmer but rather by a library, called runtime[2], in charge of running the Java program. This library contains, among other, an allocator and a Garbage Collector (GC). When an object is created, its corresponding memory is reserved by the allocator. In Java, there is no standard way to delete an object. Instead, the GC is in charge of detecting unused objects by tracking references to objects and deleting them when they are no longer referred to[3]. The implementation of the runtime library is not specified.

2: Runtime library is stored in the file `libart.so`.

3: This task is not trivial and numerous GC algorithms exist.

During the compilation, the types used in Java code are checked. In a word, it checks that when a value is assigned to another, types of both values are compatible[4]. When such checks are not doable at compilation time, the compiler adds type checking instructions inside the code. If the check fails, then a runtime exception is raised. This exception can be caught and the problematic case can be handled at runtime. Additionally, checks that do not focus on type checking but rather on performed operations are also inserted at runtime. For example, before accessing an array, the index is compared with the array size in order the raise an exception if a buffer overflow or underflow occurs. When trying to use an object that has not been initialized, a `NullPointer` exception is raised.

4: In the sense of class hierarchy

Whereas C++ is also an object-oriented programming language and represents data as objects, the proposed data abstraction differs from the Java language. Multiple primitive types[5] exists depending on the size used to store them in memory. For example, an integer typed `int` is stored on 32 bits and can contain values ranging from $-(2^{31})$ to $2^{31}-1$, while an integer typed `unsigned short` is stored on 16 bits and can represent non negative values from 0 to $2^{16}-1$. Additionally, C/C++ propose types used to represent the address of entities. This types are called pointers and are recognized using the $*$ character. For example, `unsigned int*` is the type of a pointer which stores the address of an unsigned integer.

5: Integer, floating-point value, boolean, character

C/C++ offers a data abstraction that is closer to the processor behavior than that of Java. No garbage collector is provided, that is, programmers have to manually free allocated memory when using these languages. Moreover, no runtime-checks are added: if the compiled code tries to perform operations forbidden by the operating system, the program crashes and no standard way exists to recover from such errors.

When developing Android application using both Java and C/C++ languages, a developer may want to manipulate and transfer data from one language to another. The conversion between data abstraction is made by a dedicated interface called Java Native Interface (JNI). This interface allows the developer to receive and modify Java data in C/C++ functions and so, C/C++ values can be spread in Java ones.

As mentioned above, the data abstractions offered by Java and by C/C++ gives different guarantees: C/C++ data are more permissive and expressive than Java ones. Thus, interferences between Java and C/C++ data can lead to security issues. In this chapter, we describe precisely two types of security issues introduced by the duality between Java and C/C++ code. Each issue belongs to a different type of security research fields.

The first issue belongs to the field of vulnerability detection: the injection of native data in Java data can lead to security vulnerabilities since untrusted data may be used by Java as "trusted" data. This first point is described in Section 4.1.

The second issue belongs to the field of obfuscation techniques: the developer can intentionally bypass the JNI in order to directly modify Java data using C/C++ code. This can be used as an obfuscation technique. This second point is described in Section 4.2.

4.1 Injection of native data in Java data

Java and C/C++ data do not guarantee the same properties. Because C/C++ languages are lower-level programming languages than Java, some abstractions and data specifications, that are available in Java, lack a C/C++ equivalent. A good example of such a property is the `transient` keyword that can qualifies Java fields. An Android developer uses this keyword to customize the serialization process of classes. Its functioning is described in the following.

Android applications are composed of multiple components that are running concurrently. Components communicate together by sending intents. This messaging mechanism is also used to communicate between different applications. At the implementation level, an intent is composed of bytecode objects that are serialized. Serialized data is deserialized by the called component. To tune the serialization process, developers can declare fields as `transient`. A transient field is a field that is not part of the persistent state of an object, and thus, should not be serialized. Transient fields can, for example, be used to accelerate the serialization process by not transmitting fields that can be recomputed using other fields. Additionally, the `transient` keyword is used to avoid serializing fields that have a meaning only in the current process state. For example, a field storing a memory address should not be sent to another process because the memory layout is randomized for each process. That is why, each object reference[6] should be declared transient since its value is the address of the referred object. To avoid such a time-consuming and error-prone task, the serialization process is able to handle object references automatically and reconstructs the references in the destination process.

6: Object instance inside an other object

However, serialization is not part of C/C++ language standard. Serialization should be implemented either manually or using external libraries[7]. Thus there is no equivalent to the `transient` keyword in C/C++ and no test can be made, either statically or dynamically, to check if the transient property is kept when data is transferred between C/C++ and Java. For

7: For example cereal (`http://uscilab.github.io/cereal/`) or boost (`https://www.boost.org/`)

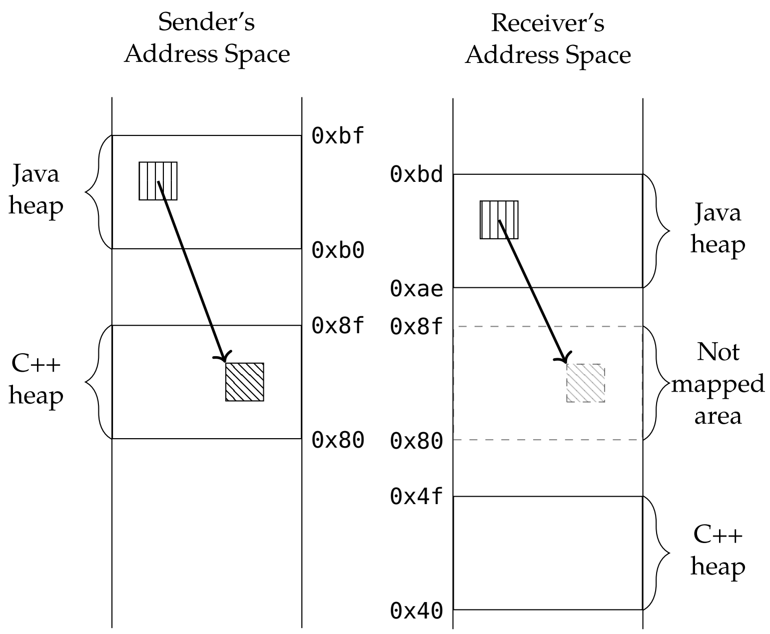


Figure 4.1: Address space of processes serializing X509 Certificate

example, if an integer[8] field is set to an address by some native code, the serialization process cannot determine whether it refers to a memory address or not. Then, it is processed as a number value and sent to the other process.

8: `long` or `int`

Peles & Hay [36] have shown that breaking the transient property can lead to severe vulnerabilities. If the transient value set by C/C++ is serialized and used by the receiver without any verification, the receiving application may process data that has no meaning in its current context. Hence, developers have to carefully declare fields as transient when they receive transient data from C/C++. This task is not straightforward and developers sometimes forget to do it. The following section describes an example of such an error.

4.1.1 CVE-2015-3837: Example of a vulnerable transient field in an open source cryptography library

CVE-2015-3837[9] concerns the cryptography Java library named conscrypt[10]. This library provides a Java interface to BoringSSL[11], a fork of OpenSSL[12]. This CVE was patched in May 2015, and is referenced in conscrypt by the bug ID 21437603. The patch commit[13] is very interesting since, besides some new tests, it only adds the `transient` keyword to the field `mContext` of the class `OpenSSLX509Certificate`. This sole addition is enough to remove the vulnerability.

9: `https://nvd.nist.gov/vuln/detail/CVE-2015-3837`
10: `https://github.com/google/conscrypt`
11: `https://boringssl.googlesource.com/boringssl/`
12: `https://www.openssl.org/`
13: Commit: 8d57b9d

Before the patch, the class `OpenSSLX509Certificate` contained a field of type `long` named `mContext`, declared as `private` and `final`. This field is used to store the address of a `X509` instance, an OpenSSL struct allocated in C++. The `OpenSSLX509Certificate` class extends `X509Certificate`, a serializable class which is part of the Java default API[14].

14: `https://developer.android.com/reference/java/security/cert/X509Certificate`

Thus, any application that uses an unpatched version of BoringSSL could receive an Intent containing an instance of `OpenSSLX509Certificate` as an extra object[15]. Such extra objects are automatically deserialized upon reception. Thus, a malicious application could send any forged instance of this vulnerable class. The resulting memory layout is represented in Figure 4.1. Both the sender's and the receiver's address spaces are shown. In the illustrated case, the `mContext` field is not null and points to a C++ allocated area. Because this field is not `transient`, the exact same value is sent to the second process. When deserializing the `OpenSSLX509Certificate` object, the `mContext` field does not point to an `X509` instance since Address Space Layout Randomization (ASLR) may have moved the C++ heap around the memory, as represented in the Figure 4.1. The field `mContext` now points to an arbitrary address that has been chosen by the malicious sender.

15: Intents carry information of non-primitive types through so-called extra objects.

However, injecting an arbitrary address into the targeted process does not lead to any bug or exploitation as long as this value is not used. Since the malicious object is not intended by the targeted application, no usage will be made of it. Unfortunately, when the object get eventually freed by the GC, the `finalize` method shown in Listing 4.1 is called. This method calls an other native method named `NativeCrypto.X509_free`, shown in Listing 4.2. This method takes the `mContext` field in argument and frees it using the OpenSSL function presented in Listing 4.3. This method, if called with an `X509` instance as argument, decrements the field named `references` of the given instances. The `X509` struct is defined as in Listing 4.4. If after decrementing the value becomes zero, the `X509` struct is freed. By sending numerous forged `OpenSSLX509Certificate`, the attacker can decrement a value at a known address in the target application process. This is called a "constrained write what where" primitive. Using this primitive, an attacker can make the targeted application execute arbitrary code, leading, for example, to privilege escalation[16].

16: A complete exploitation chain and several payloads can be found in [36].

Listing 4.1: conscrypt/src/main/java/org/conscrypt/OpenSSLX509Certificate.java

```java
@Override
protected void finalize() throws Throwable {
  try {
    if (mContext != 0) {
        NativeCrypto.X509_free(mContext);
    }
  } finally {
    super.finalize();
  }
}
```

Listing 4.2: conscrypt/src/main/native/org_conscrypt_NativeCrypto.cpp

```cpp
static void NativeCrypto_X509_free(JNIEnv* env, jclass, jlong x509Ref) {
  X509* x509 = reinterpret_cast<X509*>(static_cast<uintptr_t>(x509Ref));
  JNI_TRACE("X509_free(%p)", x509);

  if (x509 == nullptr) {
    jniThrowNullPointerException(env, "x509 == null");
    JNI_TRACE("X509_free(%p) => x509 == null", x509);
    return;
  }

  X509_free(x509);
}
```

Listing 4.3: boringssl/src/crypto/asn1/tasn_fre.c

```c
static void asn1_item_combine_free(ASN1_VALUE **pval, const ASN1_ITEM *it,
      int combine)
  {
  [...]
  if (!asn1_refcount_dec_and_test_zero(pval, it))
    return;
  [...]
```

```
}
```

```c
struct x509_st
     {
     X509_CINF *cert_info;
     X509_ALGOR *sig_alg;
     ASN1_BIT_STRING *signature;
     int valid;
     CRYPTO_refcount_t references;
     [...]
     } /* X509 */;
```

Listing 4.4: boringssl/src/include/openssl/x509.h

Thus, due to a missing `transient` keyword for the `mContext` field of the `OpenSSLX509Certificate` class, every application that links an unpatched version of BoringSSL library is vulnerable to an arbitrary code execution exploit. The `mContext` should have been declared `transient` because its value, a pointer, has been set by the C++ code. This highlights the need for detecting such C/C++ interferences into Java data.

4.1.2 Formal definition of problematic transient fields

As shown in the previous section, fields that are not declared transient but that should be because they store native pointers, may leave severe vulnerabilities inside their application or library. Thus, it is essential to be able to remove them from source code[17]. For this purpose, we need to formally define the fields that are problematic. It is noteworthy that a problematic field is not necessarily exploitable. Indeed, missing `transient` keywords of not-serializable classes cannot be exploited but could bring vulnerabilities if a developer updates these classes into making them serializable.

17: By declaring them transient.

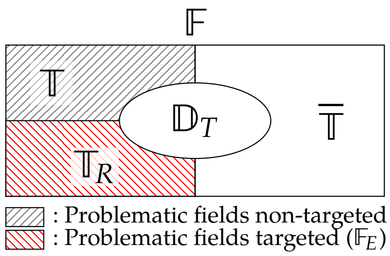


Figure 4.2: Set view of targeted problem

We represent Java fields using a set view, shown in Figure 4.2. Since the `transient` keyword is meaningful only for serializable classes, only fields from such classes are taken into account here. The set of fields ($\mathbb{F}$) is divided between fields that should be transient ($\mathbb{T}$) and those which should not ($\overline{\mathbb{T}}$). Among $\mathbb{T}$, some of the fields should be transient because they store references ($\mathbb{T}_R$, $\mathbb{T}_R \subseteq \mathbb{T}$). Technically, in the source code such references can be encoded inside object references but also in `long` or `int`. If a field is typed as an `int` or a `long`, it is difficult to determine if it stores a reference or simply a value, and thus needs to be declared transient by the developer.

As a consequence, the developer has to declare the fields that are transient ($\mathbb{D}_T$). The fields declared transient should be equal to the fields that should be transient ($\mathbb{D}_T = \mathbb{T}$). However, the developer can forget to declare some transient fields: $\mathbb{T} \setminus \mathbb{D}_T \neq \emptyset$. More dangerous, if the forgotten field should be transient because it stores a reference ($\mathbb{T}_R \setminus \mathbb{D}_T \neq \emptyset$), the programming error might make the application vulnerable to serialization attack [36]. This set of fields is named exploitable fields, $\mathbb{F}_E = \mathbb{T}_R \setminus \mathbb{D}_T$.

[36]: Peles and Hay (2015), 'One Class to Rule Them All: 0-day Deserialization Vulnerabilities in Android'

In Chapter 6, we will introduce a method for detecting missing `transient` keywords in applications composed of Java and C/C++, based on a cross-language taint analysis.

4.2 Modification of Java data by native code

When Java and C/C++ source codes exchange data, they need to use the provided JNI. This interface is in charge of hiding to the developer the differences in data representation between the two languages. This interface contains methods that allow the native code to access the heap, the area where Java objects are stored. Developers use it extensively because, first, the Java language does not constrain how data is stored in the heap. Second, C/C++, which are languages whose data representation are closer to the underlying processor architecture, may handle data storage and manipulation differently depending on the smartphone model for which the code is compiled. Using JNI, developers can avoid to treat all these potential different cases separately.

JNI is well known: analysis tools are able to setup hooks in this interface in order to retrieve the behavior of the assembly part of an application and to model how native code modifies the Java fields [81, 83, 84]. However, for obfuscation purposes, developers may hide how data is modified. For example, for hiding a ciphering-key stored in a Java field, the developer could initialize it with a dummy value and modify it in native code. By hiding this modification, the analyst could be mislead into thinking that the used key is the dummy one.

[84]: Wei, Lin, Ou, Chen, and Zhang (2018), 'JN-SAF: Precise and Efficient NDK/JNI-aware Inter-language Static Analysis Framework for Security Vetting of Android Applications with Native Code'
[83]: Xue, Zhou, Chen, Luo, and Gu (2017), 'Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART'
[81]: Qian, Luo, Shao, and Chan (2014), 'On tracking information flows through jni in android applications'

This section presents Direct Heap Access (DHA), a new obfuscation technique that consists in using native code to modify Java object fields directly on the heap without relying on bytecode or runtime functionalities. Indeed, obfuscation techniques can consist in stealthily modifying values of carefully chosen fields.

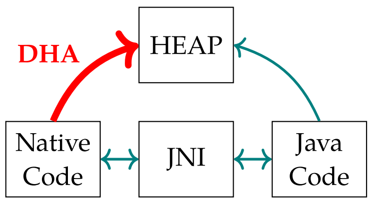


Figure 4.3: Direct Heap Access (DHA) technique

Modifying Java fields without JNI by directly modifying their value allows to bypass the aforementioned state-of-the-art tools. This is the purpose of DHA. The Dalvik virtual machine does not give any guarantee on how fields are stored in the heap. Consequently, directly reading or writing the heap without using JNI is not straightforward. We provide three ways to implement a DHA. They are ordered increasingly on the amount of knowledge required about Android runtime internals to implement them. The first implementation we provide, in Section 4.2.1, describes a solution based on a legitimate use of `DirectByteBuffer`, a specific class provided by Android. Section 4.2.2 gives a naive way of doing a DHA by scanning the whole heap memory. Finally, Section 4.2.3 gives an advanced implementation which is able to navigate through the internal structures of the Dalvik virtual machine.

Each implementation is shown using the same example. In this example, the obfuscation aims at modifying the value of the polynomial used to compute a CRC table. The polynomial is stored in an integer field named "`polynomial`". The field is initialized with a dummy value `0xeb31d82e` and should be changed to `0xedb88320`[18].

18: The same example is used to illustrate Bytecode Free OAT (BFO) with bytecode modification in Section 3.2.2.3

4.2.1 Legitimate implementation: DirectByteBuffer

While implementing a DHA seems technically difficult, it is facilitated by the Java class `ByteBuffer` which provides a way to allocate a buffer directly accessible by the native code. This buffer, named

```c
extern "C" JNIEXPORT void JNICALL bytebuffer(JNIEnv *env, jobject thisObj,
      jobject buf) {
    void* addr = env->GetDirectBufferAddress(buf);
    *(unsigned long*)addr = 0xedb88320;
}
```

Listing 4.5: DHA using `DirectByteBuffer`

```java
Field field;
field = Buffer.class.getDeclaredField("address");
field.setAccessible(true);
long addr = (long) field.get(directBytebuffer);
```

Listing 4.6: Retrieving `DirectByteBuffer` address without JNI

`DirectByteBuffer`[19], has an `address` field, that locates the bytes in the memory heap. It is created using the `ByteBuffer` method `allocateDirect`. `DirectByteBuffer` was introduced for native optimization purposes.

19: `https://developer.android.com/reference/java/nio/ByteBuffer`

However, the `address` field is not visible. That is why JNI provides `GetDirectBufferAddress` to directly access it, as shown in Listing 4.5. To avoid using JNI, which is the goal of DHA, this field can be retrieved using reflection. This is done in Listing 4.6. Using the obtained address, native code can directly access the contents of the `DirectByteBuffer`. In any case, the native code needs to receive or retrieve the byte buffer address which can be detected by state-of-the-art tools[83, 84]. Thus, using `DirectByteBuffer` does not fulfill the obfuscation goal of realizing a stealthy access.

4.2.2 Naive implementation: memory lookup

Native code can avoid the need of receiving the address of the field. Indeed, if the field has a specific unique value, such as `0xeb31d82e` in the CRC example, the native code can scan the memory to retrieve its location. Listing 4.7 shows this process by reading the special `/proc/self/maps` file. This file contains the memory mapping of the process that reads it, including the memory area named "dalvik-main space" which is the one that stores the fields. By searching for the obfuscated field value inside this area, its address can be retrieved.

Even if this memory lookup fulfills the obfuscation goals, which are modifying a Java field value without using anything from the Java code, it still has two main drawbacks. First, the lookup incurs a high time overhead: in order to modify a single field, the native code has to scan the whole heap which can grow to tens or hundreds of megabytes [98] (depending on the Android version). Second, the field has to be initialized to a unique value. This can lead to errors if the whole application code is not obfuscated at the same time. For example, an application can be obfuscated after adding a library that has been already obfuscated by its owner. In this case, fields from both the application and the library have been initialized to magic values. Some of them may be equal because when obfuscating the application the potential library magic values are not known.

[98]: (2020), *Android Compatibility Definition Document*

Listing 4.7: DHA using memory lookup

```
#define SEARCHED_VALUE 0xeb31d82e // Value of the searched field, known at compilation time
extern "C" JNIEXPORT void JNICALL memLookup(JNIEnv *env, jobject thisObj) {
  // Read "/proc/self/maps" file line by line
  FILE *file = fopen("/proc/self/maps", "r");
  if (file == NULL) return;
  char *line = NULL;
  size_t n = 0;
  while (getline(&line, &n, file) > 0) {
    char *path = strchr(line, '/');
    if (!path) continue;

    // Retrieve the \gls{heapL} area
    if (strcmp(path, "/dev/ashmem/dalvik-main space\n")!=0 && strcmp(path, "/dev/ashmem/dalvik-main space (deleted)\n")!=0) continue;

    // And get corresponding addresses
    unsigned long vm_start, vm_end;
    char r, w, x, s;
    if (sscanf(line, "%lx-%lx %c%c%c%c", &vm_start, &vm_end, &r, &w, &x, &s) < 6)
      continue;
    if (r != 'r' || w != 'w') continue;

    // Search for the field value inside the \gls{heapL} area
    for(unsigned long i=0; i < vm_end-vm_start-sizeof(unsigned long) ; i++) {
      if(*(unsigned long*)((unsigned char*) start + i) == SEARCHED_VALUE) {
        unsigned long* field_ptr = (unsigned long*)((unsigned char*) vm_start + i);

        // When found, \gls{dhaL} is realised
        *field_ptr = 0xedb88320;
      }
    }
  }
}
```

4.2.3 Advanced implementation: reflection

Native code can avoid the need of scanning the heap memory of a field by introspecting the obfuscated object itself. This requires to understand how the runtime stores objects and fields in memory and what native code has access to. This layout is presented in Figure 4.4. Native code has access to Java objects and fields through handles, respectively `jobject` and `jfieldID`. These are returned by the JNI and no guarantees are given about their implementation.

However, by looking at the source code of the Android runtime, we observe that they are pointers. A `jfieldID` is a pointer to an instance of the `ArtField` class. This class is used, inside the runtime, to store information about the field such as its declaring class, its access flags (`private`, `public`) or even the offset of the field within an instance object (`offset_`). These pieces of information are set up by the class linker. A `jobject` is a pointer to an another pointer that refers to the actual object[20]. This object stores the addresses of the different fields of the object. Fields are sorted alphabetically, grouped by type. The order is wanted "relatively stable [...] so that adding new fields minimizes disruption of C++ version such as Class and Method."[21].

20: This allows the garbage collector to move objects around the memory without having to change all references to it but only one.

21: AOSP source code, `class_linker.cc` file.

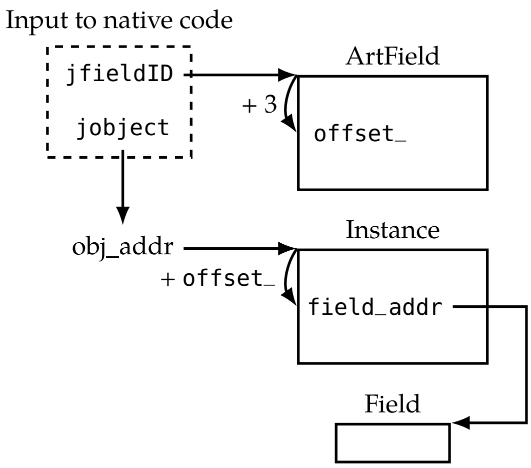


Figure 4.4: Memory layout of Java objects in Android

Listing 4.8: Retrieving field offset

```
1 #define OFFSET_OF_OFFSET_FIELD_IN_ARTFIELD_CLASS 3
2 extern "C" JNIEXPORT void JNICALL retrieve_offset(JNIEnv *env, jobject
     thisObj) {
3   jclass cls = env->GetObjectClass(thisObj);
4   jfieldID fid=env->GetFieldID(cls,"polynomial","I");
5   unsigned long offset = *((unsigned long*)fid+
     OFFSET_OF_OFFSET_FIELD_IN_ARTFIELD_CLASS);
6 }
```

Listing 4.9: DHA using reflection

```
1 #define OFFSET 0x10
2 extern "C" JNIEXPORT void JNICALL reflection(JNIEnv *env, jobject thisObj)
      {
3   unsigned long* thisPtr = *(unsigned long**)thisObj;
4   unsigned long* field_ptr = &thisPtr[OFFSET/4];
5   *field_ptr = 0xedb88320;
6 }
```

Thus, in order to implement a DHA trough reflection-like mechanism, the code has to first, retrieve the value `offset_`, which is the offset of the field address inside an object instance. This operation is realized in Listing 4.8. The `ArtField` instance of the field is retrieved at Line 4 and Line 5 retrieves the `offset_` value by accessing the third `long` of the `ArtField` instance. This offset (3) has been hardcoded at Line 1. Second, using the obtained `offset_`, the code modifies directly the field value. This is realized in Listing 4.9. The instance of the object is retrieved Line 3 by dereferencing two times the calling object. Then, at Line 4, the field address is obtained using the `offset_` value previously retrieved. It has to be noted that the first operation, Listing 4.8, requires to use JNI. To avoid being detected by JNI hooks, the value is computed and hardcoded in the Listing 4.9, at Line 1. This way, the Listing 4.9, which is the code that is finally embed in the application, does not have any calls to JNI. Both listings have been successfully tested from Android 7.0 up to Android 10 without changing neither the value of `offset_` (0x10) nor the offset of `offset_` in `ArtField` class (3).

4.3 Conclusion

In this chapter, we have reviewed the possible interferences of native code over Java data. Two consequences are presented. First, Java fields owned by a serializable class should be declared transient if they store native addresses. Indeed, if not, an attacker can send an arbitrary pointer to the application, potentially leading to a vulnerability exploitation. Second, a field object can be accessed directly by the native code, bypassing the JNI interface. We named this bypassing method Direct Heap Access (DHA). These two problems are solved in Chapter 6. We have also proposed several implementations of DHA to show its practicability. The usage of this technique in the wild is evaluated in Section 6.3 of Chapter 6.

Detection of native interferences in Java Android applications

Detection of native interferences in Java Android applications

	Code interferences	Data interferences
Obfuscation	Packer / AOTC	DHA
Vulnerabilities	-	Missing transient

Table 4.1: Interferences between native code and Java code, in an Android application

In the previous part, we have seen that the presence of assembly code inside Android applications allows interferences between the native code and the Java part of the application. These interferences are summarized in Table 4.1. They can happen on the Dalvik bytecode or on the Dalvik data. This creates two types of issues. First, obfuscation issues: an attacker can use assembly code to modify or replace the Dalvik bytecode itself or the data the bytecode handles and thus, complicating the analysis of the application. Second, vulnerability exploitation issues: a developer can accidentally break Dalvik bytecode security properties by using assembly code and leave vulnerabilities exposed for malicious exploitation.

Four distinct interferences have been presented:

- **Packing**: assembly code can load Dalvik bytecode at execution time, making bytecode unavailable for static analysis.
- **AOTC-based bytecode hiding scheme**: Dalvik bytecode can be compiled into assembly code and then removed, again making bytecode unavailable for static analysis.
- **Missing transient fields**: storing memory addresses coming from assembly into Dalvik serializable fields can make applications vulnerable and potentially exploitable if these field are not properly declared transient.
- **Direct Heap Access**: assembly code can modify Dalvik data without using the standard JNI interface in order to obfuscate data flow.

In this part, we will describe methods to detect these interferences. Indeed, detecting them is the first step towards tackling their corresponding issues. For obfuscation issues, when the interference is detected, a specific tool to deobfuscate the application can be used or the interference can be taken into account by a more generic solution. Such a solution, that deeply analyzes an application, will be presented in Chapter 7. For vulnerability issues, when the problematic interference is detected, the developer can patch the code to remove the issue.

Chapter 5 tackles code interferences. We present detection methods, which consist in statically checking that the bytecode is not altered either in the DEX file or in the OAT file. Then, Chapter 6 targets data interactions. The solutions we propose consist in observing the interface between the assembly and the bytecode data both on the source code, or directly during the execution of an application. For both chapters, detection techniques will be used to determine if the discussed interferences are already present and used in the wild.

5 Static detection of native interferences in Java code

As we have seen in Chapter 3, the assembly part of an Android application can modify or replace its Dalvik bytecode. Since assembly code is harder to understand than Dalvik bytecode, this allows to obfuscate the code. These obfuscation techniques, when used, fool analysis tools into drawing erroneous conclusions. Automatic systems, such as antiviruses, may not properly handle applications if they do not use specific deobfuscation methods. Hence, it is necessary to be able to detect native interferences in Java code.

Due to the tremendous number of applications created each day[1], analysis tools must work in a limited time. It is necessary to automatically determine in a short amount of time if an application is likely to be malicious. If so, a more thorough analysis can be performed. Hence, it is necessary to develop both methods that scale very well, and methods that provide very precise information about how an application behaves.

1: Several applications are released each minute: `https://www.statista.com/statistics/1020956/android-app-releases-worldwide/`

Dalvik bytecode obfuscation, and by extension, native interferences in Java code, should be treated with the same principle. Detecting that an application bytecode is obfuscated needs to be fast and not requiring the execution of the application. However, this task is not trivial. Indeed, the studied obfuscation methods, that is packers and AOTC-bytecode hiding schemes, prevent static analyzers to access the bytecode. Additionally, as shown in Section 3.2, the obfuscation can be realized in a stealthy way: the application under analysis may contain bytecode which is never executed (because quick code for the same methods is available). Thus, relying only on the absence of bytecode is not trustworthy.

In the Android runtime, bytecode is stored in DEX files[2]. When the application is compiled ahead of time, the DEX file is stored in an OAT file. Each file format is targeted by a specific obfuscation technique: packers obfuscate DEXs when Bytecode Free OAT (BFO) targets OATs.

2: The DEX file format does not allow more than 65,536 (`ushort`) methods. Thus, a single application can contain several DEX files.

This chapter presents the detection methods corresponding to these techniques, as well as the results of their usage in the wild. Section 5.1 tackles native interferences in DEX files, that is packer obfuscations, while Section 5.2 deals with native interferences in OAT files, that is BFO obfuscations.

5.1 Detecting native interferences in DEX files

Detection method: A packer ciphers the DEX file of an application, replaces it by a code that deciphers and loads the original code during the execution. This technique was described more precisely in Section 3.1. Numerous packing services exist online[3]. They allow users to submit applications on their platform, so that they receive a corresponding packed application. Each service uses its own packer and thus it is possible to list, for each of them, artifacts that allow to detect their

3: Alibaba Inc., Baidu Inc., Bancle Inc., Ijiami Inc., Qihoo360, Tencent Inc.

Table 5.1: Packer detection for various datasets

	GOOD [4]	MAL [4]	AMD [58]	Drebin [56]
Total	4999	4991	24552	5560
Detection	3 0.06%	542 10.86%	31 0.13%	0 0%

Table 5.2: Packer detection evolution inside AndroZoo

Year	2008-2013	2014	2015	2016	2017	2018
Packed app.	0	1	4	5	7	7

usage. Public and collaborative databases containing such artifacts can be found on the web[4]. For example, specific classes or file names inside an application can reveal that it is packed [78]. This method is very precise since it allows not only to detect that a packer has been used but also to identify which packer was used. However, it cannot detect new packers until they are analyzed.

4: APKiD: https://github.com/rednaga/APKiD

[78]: Zhang, Luo, and Yin (2015), 'Dex-hunter: toward extracting hidden code from packed android applications'

In order to statically detect unknown packers, detectors can use the `Manifest` file [94]. This file, which is read by the Android system when applications are installed, defines the activities and the services of the application. A packer cannot alter this file: the `Manifest` is read by Android before the application execution, *i.e.* before the deciphering routine could decipher it. However, the classes that are referenced (as activities or services) in this file may have been packed and thus, cannot be found by statically analyzing the application's DEX file. Thus, a class referred to in the `Manifest` but not found in the application code is a good indicator that the application under analysis has been packed.

[94]: Jiang, Zhou, Liu, Jia, Liu, and Zuo (2017), 'CrackDex: Universal and automatic DEX extraction method'

Detection in the wild: To determine how much the packing technique is used in the wild, we have searched for common known packing signatures[5] inside four datasets: AMD [58], Drebin [56], GM19 [4] (split into GOOD and MAL datasets) and an extract of 9,041 applications randomly picked from AndroZoo [57]. These datasets are more precisely described in Section 2.2. Results are reported in Table 5.1 and 5.2.

5: Using APKiD

In Table 5.1, when comparing goodware (GOOD dataset) and malware (MAL dataset), it is clear that packing methods are more frequently used in malware samples. Indeed, malicious applications very likely want to prevent analysts from reverting them. Table 5.2 shows that the usage of packers has increased starting from 2014. This explains why Drebin, which is older than 2014, does not contain any packed application, even if it is composed of malware. On the other hand, AMD, which is also a dataset of malicious applications, contains a very low number of packed applications. This can be explained by the fact that AMD is a manually crafted dataset, *i.e.* every application has been manually reversed. This tends to show that packing is a very effective technique to prevent reverse engineering.

5.2 Detecting native interferences in OAT files

Section 3.2 introduced an obfuscation technique called BFO, which consists in compiling the Dalvik bytecode into assembly code and then modifying the bytecode in order to make the bytecode unavailable for analysis. Thus, the original bytecode, which no longer exists, is protected against both static and dynamic analyses.

When Android compiles a DEX file[6], it creates an OAT file which contains both the original bytecode and the obtained assembly code. Both bytecode and assembly code are stored method by method, *i.e.* it is possible that only a subset of the application's methods is compiled to native code. Similarly, the BFO obfuscator can work at the granularity of the method. For each method it can remove, nop or replace the bytecode. This section presents, for each case, a detection method and results when applied in the wild.

6: DEX files contain the bytecode of the application

The experiments will be conducted over two datasets:

- AOSP dataset: all the compiled applications from the AOSP Android 7.0[7] firmware. These applications are used to validate that methods do not generate false positives since they are not obfuscated.
- Firmware dataset: all the compiled applications from 17 firmwares from various brands. This dataset has been described in Section 2.2 and the complete list is available in Appendix A.

7: Nougat, 2016

5.2.1 Detecting removed bytecode

Detection method: The detection of bytecode removal is straightforward since it only consists in searching for methods that have assembly code but no bytecode. We have also tried to develop a naive technique to detect partially removed bytecode. It consists in, first, computing, for each method, the ratio of the length of the bytecode over the length of the assembly code and, then, checks if it exceeds a given threshold. Indeed, one could say that number of assembly instructions used to represent a bytecode instruction is bounded. However, compiler optimizations defeat this relation between bytecode and assembly. For example, compilers can decide that a method should be inlined, or that a condition can be removed because it is always true or false. Thus, this method generates too many false positives to be usable.

Detection in the wild: We have searched for methods containing assembly code while not containing bytecode inside the precompiled applications of the firwmare dataset. This would have been the evidence of BFO usage. However, no such method has been found. This shows that BFO based on removing the bytecode is, at least for studied firmwares, not actively used in the wild.

5.2.2 Detecting nopped bytecode

Detection method: Detection of nopped bytecode can be achieved using statistical properties such as entropy. Since nopping code consists in rewriting bytecode using always the same pattern, it lowers its entropy. By using an entropy threshold, the nopping can be detected. For each method of the tested application, we compute the entropy of the bytecode. A small entropy reveals a nopped bytecode. To determine the threshold that reveals a nopped method, we have computed the entropy of the

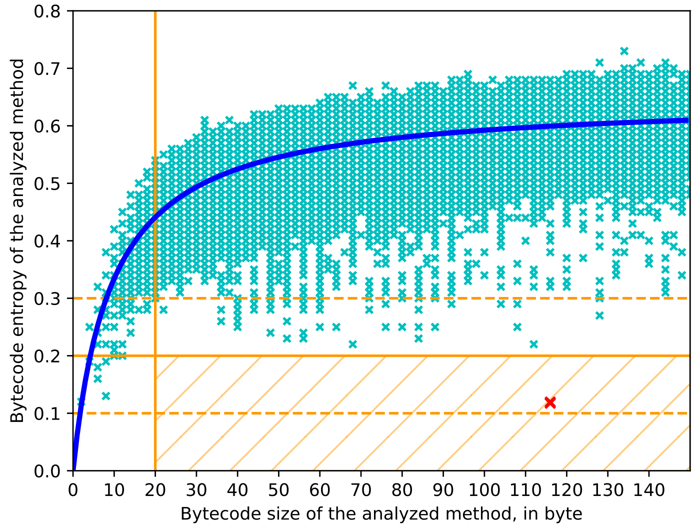


Figure 5.1: Bytecode entropy for methods of AOSP Android 7.0 APKs

Listing 5.1: Example of false positive for nopped bytecode search

```
// Entropy: 0.197
public static final float[] horizontalFlipMatrix() {
  return new float[] { -1.0F, 0.0F, 0.0F, 0.0F, 0.0F, 1.0F, 0.0F, 0.0F,
     0.0F, 0.0F, 1.0F, 0.0F, 1.0F, 0.0F, 0.0F, 1.0F };
}
// Entropy: 0.180
public static final float[] identityMatrix() {
  return new float[] { 1.0F, 0.0F, 0.0F, 0.0F, 0.0F, 1.0F, 0.0F, 0.0F, 0.0
     F, 0.0F, 1.0F, 0.0F, 0.0F, 0.0F, 0.0F, 1.0F };
}
// Entropy: 0.197
public static final float[] verticalFlipMatrix() {
  return new float[] { 1.0F, 0.0F, 0.0F, 0.0F, 0.0F, -1.0F, 0.0F, 0.0F,
     0.0F, 0.0F, 1.0F, 0.0F, 0.0F, 1.0F, 0.0F, 1.0F };
}
```

methods of all applications of the AOSP dataset. This corresponds to 255,309 methods. These applications are not obfuscated, so their entropy should be higher than the threshold. The obtained entropy for each bytecode size is shown in Figure 5.1. For methods whose bytecode size is lower that 20, the entropy does not reveal anything and is too fluctuating to be able to set a threshold. Three thresholds are drawn on Figure 5.1: 0.1, 0.2 and 0.3. Results shows that 0.1 is too strict while 0.3 generates too many false positives. Using a threshold of 0.2, only one method is falsely reported which is completely acceptable. Thus, by considering only bytecode of 20 bytes or more and by setting a threshold of 0.2, we should be able to detect nopped bytecode.

Detection in the wild: We applied this detection technique to the firmware dataset. As shown in Table 5.3, few methods have an entropy less than 0.2. We manually checked these methods by looking at their bytecode. Unfortunately, no true positive has been found. Listing 5.1 shows examples for three methods that are false positives. The code is not a nopped code, but rather the initialization of several arrays. This initialization is composed of many repetitions of the same value, which lowers the entropy. However, this could have been a nopping pattern

Firmwares		Entropy / Total	< 0.1	< 0.2	< 0.3
Alcatel	APKs	338	0 0.00%	13 3.85%	138 40.83%
2 firmwares	Methods	2,716,821	0 0.00%	23 0.00%	614 0.02%
Archos	APKs	110	0 0.00%	2 1.82%	28 25.45%
1 firmwares	Methods	246,962	0 0.00%	2 0.00%	95 0.04%
Huawei	APKs	271	0 0.00%	9 3.32%	87 32.10%
3 firmwares	Methods	1,146,585	0 0.00%	9 0.00%	317 0.03%
Samsung	APKs	795	0 0.00%	6 0.75%	97 12.20%
5 firmwares	Methods	1,817,146	0 0.00%	12 0.00%	667 0.04%
Sony	APKs	1,412	0 0.00%	23 1.63%	341 24.15%
4 firmwares	Methods	5,463,229	0 0.00%	31 0.00%	1,547 0.03%
Wiko	APKs	188	0 0.00%	12 6.38%	81 43.09%
1 firmwares	Methods	1,709,624	0 0.00%	22 0.00%	365 0.02%
Total	APKs	3,114	0 0.00%	65 2.09%	772 24.79%
16 firmwares	Methods	13,100,367	0 0.00%	99 0.00%	3,605 0.03%

Table 5.3: Nopped methods in firmware dataset

used to obfuscate applications. Thus, we believe that this method is able to detect nopped patterns to be confirmed by manual investigations.

5.2.3 Detecting replaced bytecode

Detection method: Detection of bytecode replacement consists in detecting if a given assembly code is the result of the compilation of a given bytecode. This can be done by compiling the bytecode and then comparing the result of this compilation to the given assembly. For each precompiled application (OAT file), we extract the bytecode file (DEX) from the OAT file. Then, we recompile it using the compiler present in the emulator provided by Google. We carefully choose the emulator to reflect the Android version and the processor architecture used by the real smartphone. Compiling using the same environment as the firmware constructor is impossible since applications are cross-compiled on vendor computers and no documentation is available about their build systems. Finally, we compared the obtained assembly code with that of the firmware. If no BFO techniques have been used, they should be equal.

However, in practice, the output of a compiler is highly dependent on the configuration of a particular system and many of them use non-

Table 5.4: Difference percentage for one firmware

	Difference / Total	> 0%	> 25%	> 5%	> 75%
Methods	14629	14578 99.7%	3029 20.7%	51 0.3%	9 0.1%
APKs	43	21 48.8%	15 34.9%	7 16.3%	3 7.0%

[99]: Dullien, Carrera, Eppler, and Porst (2010), *Automated attacker correlation for malicious code*

8: Immediate values directly wrote inside assembly operands.

deterministic algorithms [99]. Thus, the obtained assembly code and the firmware's one are slightly different, for almost all methods.

In order to investigate how much the codes are different, we have disassembled them and removed the hardcoded[8] values, which usually correspond to offsets that are very likely to change between two compilations. Then, we proceeded to perform a textual diff where each assembly instruction constitutes a line. Finally, we computed the following ratio:

$$difference\_percentage = \frac{n_{addition} + n_{deletion}}{2 * n_{line}}$$

This ratio, which is comprised between zero and one, has been calculated for all the pre-compiled applications of one firmware. Table 5.4 shows that more than twenty percent of the compiled methods differs from the native code present in the firmware by at least one instruction out of four.

By manually investigating the differences, we observed that they are due to subtle choices made by the compiler. For example, we saw that multiple `if else` structures are, sometimes, compiled into `switch` structures. That is, instead of having multiple comparisons and jumps, one version of the assembly computes an index and jumps at an address stored in a vector. Also, we saw that the compilers did not choose to store the class fields at the same offsets.

[99]: Dullien, Carrera, Eppler, and Porst (2010), *Automated attacker correlation for malicious code*
[100]: Flake (2004), 'Structural comparison of executable objects'

9: https://www.zynamics.com/bindiff/manual/
10: https://github.com/joxeankoret/diaphora/

That is why, we tried to use state-of-the-art binary diffing tools [99, 100], such as bindiff[9] or diaphora[10], however they did not achieve to detect more accurately if codes are the same. Indeed, these tools rely heavily on the callgraph of the analyzed codes, which is almost nonexistent for assembly code: due to its object and framework oriented compilation, all calls are indirect and cannot be resolved statically.

5.2.3.1 Future work on detecting BFO usage

[101]: Gabel, Jiang, and Su (2008), 'Scalable detection of semantic clones'

Thus, no BFO usage has been found in the wild for BFO consisting in removing or noping the bytecode. For the replacement case, no suitable detection technique is known. Thus, more specific techniques need to be developed. Indeed, we have conducted a syntactic detection: we tried to mimic the compiler used by the studied application and to remove inconsistent part of assembly code in order to minimize differences between studied code and recompiled code. But, even in this case, compilers outputs differ a lot. In such conditions, semantics-based approaches have shown to be more resilient than syntactic ones [101]. Semantics-based approaches work on high-level representations of programs and functions. For example, code can be converted into dependency graph [101] or functions can be summarized as formulas that represent their computation [102]. This allows not to take into account low-level features of the code that are highly volatile and very probably change between two versions of the same code.

[102]: Pewny, Schuster, Bernhard, Holz, and Rossow (2014), 'Leveraging semantic signatures for bug search in binary programs'

Such approaches have be conducted in the Android context [29, 30, 103]. While most of them target only bytecode [29, 103], some handle native code by converting both bytecode and assembly to the same intermediate language and conduct their analysis on this language [30]. However, these works do not aim at matching application functions together but rather match whole applications. Outside the Android context, numerous assembly functions matching methods have been developed [104, 105]. It is noteworthy that assembly code of OAT files differs from the one of classical desktop programs: dependency between functions is almost nonexistent. Thus, state-of-art semantic-based methods should be adapted to this context. This is left as future work.

[29]: Crussell, Gibler, and Chen (2012), 'Attack of the clones: Detecting cloned applications on android markets'
[103]: Crussell, Gibler, and Chen (2013), 'Andarwin: Scalable detection of semantically similar android applications'
[30]: Alam, Riley, Sogukpinar, and Carkaci (2016), 'Droidclone: Detecting android malware variants by exposing code clones'

[104]: Rattan, Bhatia, and Singh (2013), 'Software clone detection: A systematic review'
[105]: Roy, Cordy, and Koschke (2009), 'Comparison and evaluation of code clone detection techniques and tools: A qualitative approach'

5.3 Conclusion

In this chapter, we have measured how much native interferences for obfuscating the code are used in the wild. It is clear that packers are widely used by malicious applications, and we can believe that normal applications will also tend to resort more and more to these obfuscation techniques.

We investigated the use of our new proposed obfuscation method, BFO, that creates an OAT file without leaving the original DEX file intact. Results indicate that the full nopping of bytecode is not used yet. If slight modifications are introduced in the DEX part, detecting an inconsistency between the compiled part and the modified DEX is a difficult problem. We proposed a first approach but did not spot any application in our dataset that performs such a complex obfuscation pattern. This result could indicate that no application in the analyzed firmware have used this technique – which is a reasonable hypothesis – or that our detection method missed such a usage. Building a reliable detection technique remains an open problem.

6 Detection of native interferences in Java data

At stated in Chapter 4, native code can interfere with bytecode data. These data interferences have been classified into two types:

- Data injection[1]: assembly code can unintentionally break properties guaranteed by the Dalvik virtual machine on its data by storing untrusted values inside Java fields. Doing so leaves vulnerabilities inside the application by storing native data into Dalvik ones. Missing `transient` keyword for Java fields storing native addresses is an example of such a vulnerability.
- Data modification[2]: assembly code can circumvent the Java Native Interface (JNI), the interface given by the Android runtime, to stealthily modify bytecode values and bypass reverse-engineering tools. This is called Direct Heap Accesses (DHAs).

1: Section 4.1.

2: Section 4.2

These interferences are the cause of vulnerability and obfuscation issues. The vulnerabilities should be detected before releasing the application to users, in order to avoid spreading unsafe applications which, even if a patch is proposed, might not be updated by users. The obfuscation issues should be taken into account by analysis tools. However, creating and maintaining tools that handle this kind of interferences complicates a lot the analysis of applications. In Chapter 7, we will describe the architecture of *OATs'inside*, a tool we developed that tackles this very issue. Consequently, it is necessary to detect in advance if applications in the wild already actively use this technique.

Thus, both types of interferences should be detected. They consist in dataflow between two languages. of phenomenon is naturally made by analyzing their interface, in this case JNI. Nevertheless, detection methods of these two interferences do not happen at the same place: for data injection, inside the source code and for data modification, inside the compiled application. Indeed, data injection, which leaves vulnerabilities inside the application, should be treated by developers on their source code and data modification, which bypasses JNI at runtime, should be handle by dynamic analysis tools. To date, multiple efforts focusing on observing dataflow in such contexts have already been conducted. However:

- for data injection, already existing dataflow tools do not model the transient property.
- for data modification, already existing methods rely on the JNI interface and are, due to the inherent principle of DHA, bypassed.

This chapter presents the detection methods corresponding to the aforementioned interferences and their associated issues. Section 6.1 reviews the contributions related to taint analysis of native Android applications. Section 6.2 tackles missing `transient` keywords and illustrates the proposed detection method on open-source applications. On the other hand, Section 6.3 describes how to detect DHAs and presents their usage in the wild.

6.1 Taint-analysis across native and Java interface

Conducting taint analysis across the interface between native and Java has already been conducted in the Android context [84, 106–108]. Typically, taint analysis is concerned with finding information flows from so-called *sources* of sensitive information into *sinks*, that leak this information. Android taint analysis are usually used to track privacy leaks made by applications [72, 74]. For example, the `getImei`[3] method is considered as a source while `Socket.writeUTF` and `File.write` are potential sinks.

3: International Mobile Equipment Identity (IMEI), number that uniquely identifies a mobile device

Researchers try to find privacy leaks in applications without the source code in order to be able to vet applications that come from untrusted developers. Thus, contrary to our solution proposed in Section 6.2, they work on bytecode and assembly rather than the source code. Dynamic solutions [106–108] store the taints among the value of the carriers by, for example duplicating the size of the heap and using an integer to represent the taint of each integer stored on the heap. Static solutions [84] follows the code flow to summarize the stores that can happen during the execution of the analyzed application.

Unfortunately, these approaches cannot be used for modeling the transient property: they cannot capture if the value of a field has been computed using a native pointer. Indeed, they work on assembly code and assembly registers are not typed: a register is a value that can be interpreted as any type such as pointer, integer or character.

These approaches, that can also be used to defeat obfuscation, are detailed in greater detail in Section 7.1.

6.2 Analyzing dataflow between native and Java at the source code level

We have seen in Section 4.1 that native and Java data do not have the same nature, they do not carry the same types of data, and that Java data gives more guarantees over its data than assembly. Then, we have seen that when native data is transferred into Java data[4], Java values' properties, such as their type, may be broken and vulnerabilities may be generated inside the application. In particular, Section 4.1 described deeply how breaking the transient property is very dangerous and can lead to arbitrary code execution inside the application scope.

4: Classically through JNI

In order to ensure users' security, this kind of vulnerability should be removed from any application. Since spotting them requires advanced security knowledge and involves analyzing dataflow between different languages, developers may miss some of them. Thus, it is necessary to provide them with tools that automatically detect missing `transient` keywords. Since the tool is intended for developers, it can directly work on the source code and, this way, not suffer from information loss inherent to the compilation process. Additionally, the tool does not need to be run frequently during the development cycle but rather only once before the application is released[5] and thus, its analysis is not time-constrained.

5: As part of the continuous integration tests, for example.

This section presents a method that detects Java fields that stores references without being declared `transient`. After giving the method architecture and its implementation, this section shows its effectiveness by analyzing both a known CVE and the open-source Telegram application.

6.2.1 Statically detecting reference fields in source code

The solution presented in this section is able to detect exploitable fields ($\mathbb{F}_E$) that are reference fields, not declared transient. For the sake of clarity, Figure 4.2 which represents missing transient properties using a set view, has been reproduced here in Figure 6.1. To compute this set, the analysis has to compute $\mathbb{D}_T$ and $\mathbb{T}_R$. The set of declared fields ($\mathbb{D}_T$) is easily obtained by looking at the Java definition of fields. Computing the set of reference fields ($\mathbb{T}_R$) requires for each field to determine if it encodes a memory address. As the type is not enough to decide, the usage of this field in the code needs to be tracked. If it stores a reference then it should be used at some point as a pointer. We propose a static method to track the references manipulated in C/C++ code inside the Java code using taint analysis. The difficulty lies in the duality of an Android application: while the Java code declare fields, the C/C++ part of the application code can manipulate them by writing into the memory of the application.

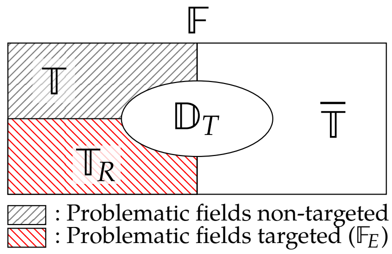


Figure 6.1: Set view of targeted problem, Reminder of Figure 4.2 Section 4.1

The overall architecture of the proposed solution is given in Figure 6.2. First, the C/C++ code is analyzed to list the fields that are manipulated as pointers. This is the first part of the reference fields ($\mathbb{T}_{R_1}$). Moreover, the C/C++ analyzer lists all pointers that interface with the Java code. Using this list, the Java analyzer conducts its own taint analysis to track fields that interact with these pointers. This is the second part of the reference fields ($\mathbb{T}_{R_2}$). Finally, the declared fields ($\mathbb{D}_T$) are extracted and subtracted from the reference fields $(\mathbb{T}_{R_1} \cup \mathbb{T}_{R_2}) \backslash \mathbb{D}_T$, in order to compute the exploitable fields ($\mathbb{F}_E$).

6.2.1.1 Reference field patterns in source code

In order to determine precisely which taint analysis to conduct, we have listed all possible implementations of reference fields ($\mathbb{T}_R$). Like every variable or field, a reference field can be either written or read. Because

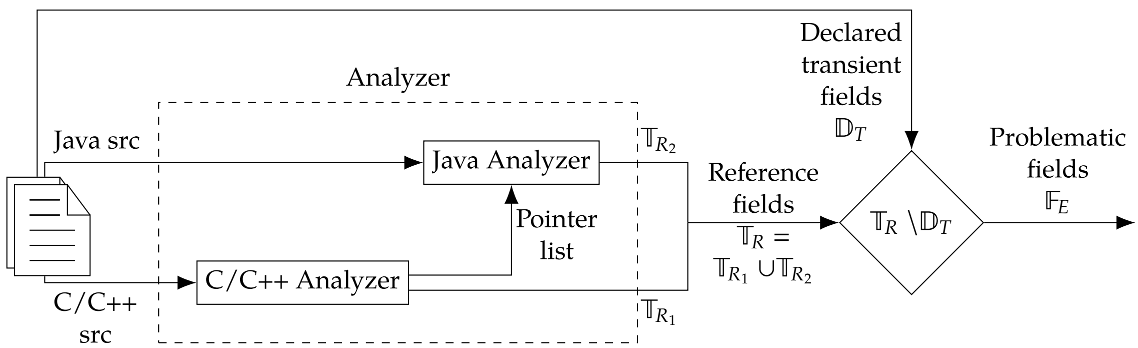


Figure 6.2: Architecture overview

```
public transient long referenceField;
JNIEXPORT void JNICALL native_method(JNIEnv* env,
    jobject thisObj) {
  jclass cid = env->FindClass("ThisClass");
  jfieldID fid = env->GetFieldID(cid, "
    referenceField", "L");
  unsigned long ptr;
  ptr = (unsigned long) malloc(sizeof(char));
  env->SetLongField(thisObj, fid, ptr);
}
```

(a) Native only assignation

```
public transient long referenceField;
JNIEXPORT void JNICALL native_method(JNIEnv* env,
    jobject thisObj) {
  jclass cid = env->GetObjectClass(thisObj);
  jfieldID fid = env->GetFieldID(cid, "
    referenceField", "L");
  unsigned long ptr = env->GetLongField(thisObj,
    fid);
  free((void*)ptr);
}
```

(b) Native only usage

```
public transient long referenceField;
JNIEXPORT jlong JNICALL native_method(JNIEnv* env,
    jobject thisObj) {
  return (unsigned long) malloc(sizeof(char));
}
public void java_method() {
  this.referenceField = this.native_method();
}
```

(c) Native to Java assignation

```
public transient long referenceField;
public void java_method() {
  this.native_method(this.referenceField);
}
JNIEXPORT void JNICALL native_method(JNIEnv* env,
    jobject thisObj, jlong arg) {
  free((void*)arg);
}
```

(d) Java to native usage

Figure 6.3: Reference fields in Android application source code

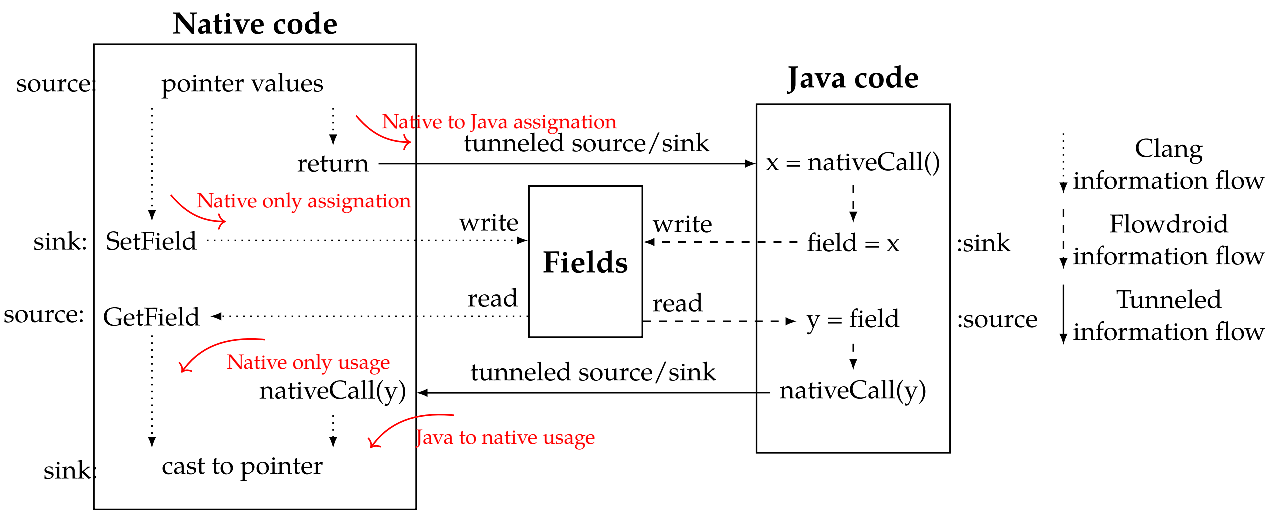


Figure 6.4: Representation of reference flow tracking

Table 6.1: Sources and sinks for detecting reference field patterns

Patterns		Native source code		Java source code	
		Sources	Sinks	Sources	Sinks
Native only	assignation	Pointers	JNI Set fields	∅	∅
	usage	JNI Get fields	Casts to pointer	∅	∅
Native to Java	assignation	Pointers	Return operations*	Native method returns*	Field assignations
Java to native	usage	Method arguments*	Casts to pointer	Fields value	Native method arguments*

only native code is able to handle raw memory addresses, assignation or usage can happen either directly in native code or within a mix of native and Java code. This results in four distinct code patterns. These patterns are shown in Listings 6.3a, 6.3b, 6.3c, 6.3d and are the following:

1. **Native only assignation**: an address, or pointer in C/C++, is used to set a field through JNI. For example in Listing 6.3a, the address retrieved from `malloc` at Line 6, is set to the field `referenceField` of ID `fid` at Line 7 using the JNI `SetLongField` method.
2. **Native only usage**: a value obtained from a field through JNI is cast to a pointer. For example in Listing 6.3b, the field value retrieved at Line 5 using JNI `GetLongField` method, is cast to a pointer at Line 6.
3. **Native to Java assignation**: an address that is returned by a native method is assigned to a field in a Java method. For example in Listing 6.3c, the address returned by the C++ `native_method` at Line 3 is assigned to the field `referenceField` by Java code at Line 6.
4. **Java to native usage**: a field is used, in a Java method, as an argument to a native method that casts this argument to a pointer. For example in Listing 6.3d, the field `referenceField`, is used as an argument when calling the C++ `native_method` at Line 3. During this call, the field is cast to a pointer at Line 6.

Focusing on the aforementioned patterns, we have the guarantee that our approach will exhaustively retrieve all the reference fields ($\mathbb{T}_R$) and therefore successfully build the list of fields whose `transient` keyword is missing ($\mathbb{F}_E$).

6.2.1.2 Static patterns detection using taint analysis

In order to detect the patterns that we just presented, we compute information flows where sources and sinks are associated to fields or memory pointers. All possible sources and sinks, *i.e.* the elements of the native or Java source code that may produce a flow, are summarized in Table 6.1. For example, a “Native only assignation” flow exists if the native code manipulates a pointer (source) and calls a JNI Set field method using this pointer as an argument (sink). For patterns that are composed of Java and C/C++ code, the information flow is more complex as at least two parts of the code have to be analyzed together. For example, a “Native to Java assignation” flow starts in the native code when a pointer is used and returned by a method called in the Java code, and ends in the Java code when the returned value is set to a field. This is symbolized by an asterisk in Table 6.1 and we call this a "tunneled source/sink" because the sink of the native code becomes a source for the Java code. When a taint reaches a tunneled sink, in a given programming language, it does not report a problematic flow but instead creates a new tunneled source, in the other language.

All information flows associated to assignation/usage patterns are illustrated in Figure 6.4. When analyzing the code, the sources and sinks are created using the following rules:

1. **Native only assignation**: all pointers, *i.e.* all values whose type contains “*” or any cast to such a type, are considered as sources. All JNI `Set*Field` methods are considered as sinks. Thus, in Listing 6.3a, Line 6 generates a taint that is propagated until the sink at Line 7.

2. **Native only usage**: all field values retrieved using a JNI `Get*Field` method are sources and every cast to a pointer type is considered as a sink. Thus, in Listing 6.3b, Line 5 generates a taint that is propagated until the sink at Line 6.
3. **Native to Java assignation**: in native source code, every pointer is considered as a source and return operations are tunneled sinks. Then, in Java source code, every return of a method corresponding to a tunneled sink, is a tunneled source. The sinks are all the field assignations that happened in Java. Thus, in Listing 6.3c, Line 3 generates a taint (`malloc` returns a pointer) and sinks this taint since it is a return operation. Then, Line 6 generates an other taint, because `native_method` is a tunneled source. Finally, the same Line 6 sinks the taint during the assignation.
4. **Java to native usage**: in Java source code, all field values are sources generating taints that may be sunk passing through native method arguments. Then, in the native source code of these specific methods, the argument that has sunk a field value is a tunneled source. The taint finally sinks when reaching a cast to the pointer type. Thus, in Listing 6.3d, Line 3 propagates the taint of the field `referenceField` to the first argument of the `native_method`. Thus, the third argument at Line 5 generates a taint that is propagated until the cast at Line 6.

Thus, a tool which correctly conducts all these taint analyses is able to detect any pattern usage, in other words any reference field ($\mathbb{T}_R$).

6.2.1.3 Static analysis limitations

Due to its static nature, the proposed analysis suffers from common static Android taint analysis drawbacks. In particular, when a field is manipulated through reflection, the analysis cannot determine which field is used and so cannot report a potential missing `transient` keyword, leading to false-negative generation.

Moreover, the Java analysis has to match the name of the native method in the C/C++ source code to the one in the Java source code. Even though the Android native method loader uses a convention for the naming of native methods, developers can register their own names by using the JNI method `RegisterNatives`. As this registration is done at execution time, this behavior cannot be retrieved by the static analysis and may generate false positives and negatives. However, the developer could inform about his specific mappings.

6.2.2 Analysis architecture

In practice, our taint analysis has been implemented following the architecture presented in Figure 6.2. The source code is first split between Java and C/C++ and each language is handled by its own analyzer which are described in this section.

C/C++ analyzer For the C/C++ part, the taint analysis is built over clang [109]. Clang provides an event-driven API for developing static analyzers. After converting the C/C++ source code into an Abstract Syntax Tree (AST), clang offers the possibility to call hooks before or after the evaluation of specific expressions. Our hooks are reported in Algorithm 6.1.

[109]: Lattner (2008), 'LLVM and Clang: Next generation compiler technology'

For optimization purposes, all four taint analyses are conducted at the same time with three different types of taints: **A** for the method arguments; **F** for the fields got using JNI; **P** for the pointers. Taints are applied to C/C++ expressions. If no taint has been applied to an expression, then this expression is considered to carry the taints of its sub-expressions (*e.g.* `a+b` carries the taints of `a` and those of `b`).

Finally, the C/C++ analyzer outputs a JSON file that contains:

1. a list of fields that have been detected through "native only assignation" and "native only usage" patterns;
2. a list of methods whose return values are pointers. This corresponds to the native part of the "native to Java assignation" pattern;
3. a list of method arguments that are used as pointers that coincide with the "Java to native usage" pattern.

Java analyzer For the Java part, the taint analysis is built over Flowdroid[6] [68]. Flowdroid provides the same event-driven programming fashion as clang. It also gives an additional class hierarchy lookup mechanism that is used to filter and treat only fields from serializable classes. The Java analyzer takes as input the Java code, the list of methods and the list of method arguments generated by the C/C++ analyzer. Using these lists to setup its taints and sinks, Flowdroid generates a list of fields detected through "native to Java assignation" and "Java to native usage" patterns. Here no special taint management is made since the two taint analyses are not mixed.

6: version 2.7.1

[68]: Arzt, Rasthofer, Fritz, Bodden, Bartel, Klein, Le Traon, Octeau, and McDaniel (2014), 'Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps'

Final reporting Finally, a union is made between the two field lists generated by respectively the C/C++ and Java analyses. This set is an under-approximation of the reference fields ($\mathbb{T}_R$). By parsing the code, the fields declared as transient ($\mathbb{D}_T$) are retrieved and the potentially exploitable fields set $\mathbb{F}_E$ is computed by subtracting the set of declared transient fields to the set of reference fields ($\mathbb{T}_R \setminus \mathbb{D}_T$).

6.2.3 Validation of the detection method

In order to evaluate and validate our novel approach, we need to analyze the full source code of applications. To get a chance to find vulnerabilities related to serialization, these applications should have native code and additionally, should manipulate objects from both sides. Finding such open source applications is very difficult, as most of candidate applications from the Google Play store do not release their source codes. Moreover, by construction of our approach, systematic testing is not easily automatable as the recompilation process needs fine-tuning. Thus, large-scale benchmarking of our approach becomes unrealistic.

Algorithm 6.1: C/C++ analyzer taint management

```
while after function call do
  if called function is JNI Get Field then
    Result expression is tainted with taint F
  end if
end while

while before value declaration do
  if value is an argument of the analyzed function then
    Value expression is tainted with taint A
  else if value is a variable then
    Value expression is tainted with taint P
  end if
end while

while before function call do
  if called function is JNI Set Field then
    if second argument of the called function is tainted with P then
      print native only assignation
    end if
  end if
end while

while before function return do
  if return value expression is tainted with P then
    print native to java assignation
  end if
end while

while before casting expression do
  if expression is cast to a pointer then
    if cast expression is tainted with F then
      print native only usage
    end if
    if cast expression is tainted with A then
      print java to native usage
    end if
  end if
end while
```

Table 6.2: Flows reported for Telegram

Reported fields $\mathbb{F}_E$		Native to Java assignation	Java to native usage	Native only patterns
Class name	Field name			
`tgnet.TLObject`	`ip ipv6 peer_tag`			✓
`messenger.BaseController`	`currentAccount`	✓		
`messenger.NotificationCenter`	`currentAccount`	✓		
`SQLite.SQLiteDatabase`	`sqliteHandle`	✓	✓	
`SQLite.SQLitePreparedStatement`	`sqlite StatementHandle`		✓	
`tgnet.NativeByteBuffer`	`address`	✓		
`ui.Components.RLottieDrawable`	`nativePtr`	✓	✓	

Nevertheless, we performed the following tests that clearly show the benefit of the proposed methodology. We analyzed three cases:

1. Constructive validation: we construct an application that regroups the four patterns described in Listings 6.3a, 6.3b, 6.3c, 6.3d. These cases can be seen as unit tests that confirm that the tool works as expected.
2. Literature confirmation: we review the `OpenSSLX509Certificate` class from the conscrypt[7] library, as according to Peles & Hay [36], this class contains a field named `mContext` which should be vulnerable for not being transient. This test shows that our tool can reproduce previous results automatically, enhancing the results of Peles *et al.* who discovered the vulnerabilities manually.
3. At-scale verification: we select, based on its popularity and its robustness, the Telegram application[8] as it constitutes a large open-source Android application with more that 1 million lines of code spread across Java and C++. This test shows the benefit of our tool when analyzing the full source code of an application.

7: `https://github.com/google/conscrypt`

[36]: Peles and Hay (2015), 'One Class to Rule Them All: 0-day Deserialization Vulnerabilities in Android'

8: `https://github.com/DrKLO/Telegram`, tag: release-5.15.0_1869

In a nutshell, instead of seeking exhaustivity, our experimental protocol focuses on validating our approach and aims at showing that: *a*) our patterns are catchable; *b*) the previously known vulnerabilities are retrieved automatically; and *c*) our solution can be used to check large open-source applications. Finally, on a different dimension, we also discuss the performances of our approach in terms of computational time and resources consumption.

6.2.3.1 Constructive validation of the patterns

As intended, all four transient fields have been detected by their respective pattern. For patterns "native to Java assignation" and "Java to native usage", the analysis has logged the transient field names and their class names. It also logged the name of the native method responsible for setting or using the transient field. For "native only" patterns, the analysis only logged the name of the transient fields. It also recovered the class name for native only assignation but did not manage for the native only usage pattern. This pattern, see Listing 6.3b, uses the JNI method `GetObjectClass` (line 3) instead of `FindClass` whose argument is the class name. As mentioned in Section 6.2.1.3, this method is not handled yet.

6.2.3.2 Catching known errors

When running the C/C++ analyzer over the `OpenSSLX509Certificate` class source code, no field were reported using the native only patterns. For the "native to Java assignation" pattern, the C/C++ analyzer has reported 127 native methods whose return value is a pointer, that is 127 tunneled sinks. On the Java side, the analyzer has not reported any corresponding information flow because the returned values are used for initializing fields that are object references. Object references do not need to be declared transient, *cf.* Section 4.1. For the "Java to native usage" pattern, the C/C++ analyzer has reported 111 native method arguments treated as pointers inside the native code, *i.e.* 111 tunneled sources. The

```
Source: <OpenSSLX509Certificate: long mContext>
Sink: $l4 = staticinvoke <NativeCrypto: long X509_get_notAfter(long,OpenSSLX509Certificate)>($l3, r0)
Source: <OpenSSLX509Certificate: long mContext>
Sink: $l2 = staticinvoke <NativeCrypto: long X509_get_notBefore(long,OpenSSLX509Certificate)>($l1, r0)
Source: <OpenSSLX509Certificate: long mContext>
Sink: staticinvoke <NativeCrypto: void X509_free(long,OpenSSLX509Certificate)>($l2, r0)
```

Figure 6.5: `OpenSSLX509Certificate` Java analysis output

Java analyzer has reported 3 corresponding information flows, all related to the field `mContext`, which is the one that was previously reported vulnerable [36]. Thus, the analysis has reported no false positive and has detected the three vulnerabilities.

[36]: Peles and Hay (2015), 'One Class to Rule Them All: 0-day Deserialization Vulnerabilities in Android'

The source / sink couples are reported in Figure 6.5. As shown in this Figure, the fields are clearly identifiable and the output could be read by any developer even without specific security-oriented knowledge.

6.2.3.3 Checking a large open-source application

We applied our detection method for missing transient keywords to the open-source Telegram application. The warnings generated during the analysis of this application are reported in Table 6.2. In order to assess that the analysis scales with huge code bases, we have analyzed all classes, including non-serializable ones. The C/C++ analysis of Telegram has reported three fields using the "native only" pattern: `ip`, `ipv6`, `peer_tag`. For all these fields the class name was not recovered since the C/C++ code uses `GetObjectClass` to retrieve the class of the accessed fields. By manually looking at the source code, we found that the three fields are declared several times in subclasses of the `tgnet.TLObject` class. The three fields are `String` fields: they are references to objects, which are automatically handled by the serialization process. Thus, they do not need to be declared transient and are *false positives*. For the "native to Java assignation" pattern, the C/C++ analyzer has reported 26 native methods whose return value is a pointer (tunneled sinks). The Java analyzer then reported 5 fields that should be transient. On the other hand, for the "Java to native usage" pattern, the C/C++ analyzer has reported 62 method arguments that are treated as pointers (tunneled sources) which has led the Java analyzer to report 3 fields. Since 2 fields are reported by both patterns, we finally obtained flows of `int` or `long` fields that should be transient. Nevertheless, the 6 corresponding declaring classes are not serializable. As a consequence, the forgotten `transient` keywords do not lead to vulnerabilities, in the current state of Telegram. The results reported are programming errors that could bring vulnerabilities if a developer updates these classes into making them serializable.

6.2.3.4 Analysis time

We have recorded the time elapsed during the analysis of the three cases (non-serializable classes omitted for Telegram analysis). The times and the number of Source Line Of Code (SLOC) analyzed are reported in Table 6.3. Analyses have been run using 26G of DDR4 RAM and an Intel Core i7-8850H[9] processor. Even for the huge Telegram application (encompassing more than 1 million lines of code), the analysis terminates

9: 12 threads, 2.60GHz

Name	Java		C++	
	SLOC	duration	SLOC	duration
Patterns example	26	1.508s	42	0.34s
`OpenSSLX509`	663	5.499s	8,269	356.93s
Telegram	511,519	6min 6s	628,864	5h 02min

Table 6.3: Analysis time

but takes five hours. The proposed method is not intended to be used frequently during the development cycle but rather only once before the application release, as part of the continuous integration tests.

6.3 Monitoring the interface between Java and native code at the execution time

Section 4.2 introduced a technique called DHA, which consists in accessing Java data using assembly code without using JNI. For this purpose, the native code directly accesses or modifies the heap, the memory area where Java objects are stored. While Android provides a specific class named `DirectBytebuffer` to realize this operation[10], we have shown that an application could inspect the memory itself, therefore bypassing state-of-the-art tools.

10: Originally introduced for optimization purposes.

Since DHA is a newly proposed obfuscation technique that relies on already known optimization mechanisms, we would like to determine if and how DHA is used in the wild. This section presents a detection method and its results when applied in the wild.

The experiments will be conducted over two datasets:

- Androzoo [57]: a dataset of about 13,000,000 different applications retrieved from various market including Google Play[11]. Thus, applications can be either malware or goodware. The experiment will be conducted on a subset of 100,000 applications chosen randomly.
- AMD [58]: a ground-truth dataset of 24,552 malware that have been reversed and classified among different malware families.

11: `https://play.google.com/store`

[57]: Allix, Bissyandé, Klein, and Le Traon (2016), 'AndroZoo: Collecting Millions of Android Apps for the Research Community'

[58]: Wei, Li, Roy, Ou, and Zhou (2017), 'Deep Ground Truth Analysis of Current Android Malware1'

Detection method: In order to detect DHA, the analysis tool has to track all reads or writes that are made to the heap. Statically determining the addresses accessed by a piece of assembly code is an open research problem. On the other hand, determining it during an execution of the analyzed application is easier: this is done by disallowing, using `mprotect`, any access to the heap addresses when running native code. Then, when native code tries to access the heap, it generates a `SEGV` signal which can then be caught. By parsing the internal structures of the garbage collector, the tool retrieves the type of the accessed value. Finally, the access is authorized and the execution is resumed.

We intentionally gave an insight of this detection method, which avoids giving implementation details. It may give the feeling that implementing this method is straightforward. In reality, this detection method is part of a more global tool, *OATs'inside*, that is fully described in Chapter 7. Technical challenges have been discussed apart, in Chapter 8.

Table 6.4: Number of DHAs detected

Dataset	Total	ARMv8	DHA	DHA without system libs
Androzoo [57]	100,018	10,661	8,158 (76.5 %)	4,021 (37.7 %)
AMD [58]	24,552	349	194 (55.6 %)	103 (29.5 %)
Total	124,570	11,010	8,352 (75.9 %)	4,124 (37.5%)

[57]: Allix, Bissyandé, Klein, and Le Traon (2016), 'AndroZoo: Collecting Millions of Android Apps for the Research Community'
[58]: Wei, Li, Roy, Ou, and Zhou (2017), 'Deep Ground Truth Analysis of Current Android Malware1'

Table 6.5: Classes and libraries detected to be using DHA

Dataset	System libraries		WebView		Other	
	samples	classes	samples	classes	samples	classes
Androzoo [57]	74.7%	1,797	37.3%	1,424	0.4%	7
AMD [58]	54.7%	154	29.5%	221	0%	0

Detection in the wild: The detection has been implemented for ARMv8 and Android version 7.0[12]. The datasets were first filtered to keep only the compatible APKs, and we checked that these applications can be launched correctly. Column "ARMv8" of Table 6.4 reports the number of applications obtained after applying this filter.

12: Nougat, 2016

We analyzed these filtered datasets and logged all performed DHAs, *i.e.*, each time the heap was accessed from the native code. Note that each application was run from only the main activity and without any user interaction. Consequently, the results presented in Table 6.4 are a lower bound on the actual usage of DHA. For each DHA, we logged the class of the accessed value and the name of the library performing the access, obtained from `/proc/self/maps`. The implementation of this logging mechanism is available in Section 8.1.4.

Globally, between 55% and 76% of the applications performed DHAs. This lower bound shows that DHA cannot be ignored when building an analysis tool. When investigating which libraries perform DHAs, we noticed that most accesses are done by systems libraries[13]. However, we have still detected that 37% of applications perform DHAs using custom libraries.

13: *e.g.* `libc.so`, `boot.oat`, `libandroid_runtime.so`

A comparison of the statistics retrieved for Androzoo and AMD datasets showed that DHA usage does not discriminate a malicious behavior from a benign one. In fact, according to the name of the libraries performing DHA, it seems to be used mostly to increase performance.

We investigated the name of the classes accessed by DHA, the number of unique class names is reported in Table 6.5. As expected, system libraries access a large variety of objects of different classes as these libraries are part of the runtime internals. Additionally, we separated a specific library, `WebView`, because it manipulates a lot of internal objects of the browser. Finally, remaining libraries modify seven different classes. Almost every sample uses `[F`, `String`, `[B` or `ByteArrayInputStream` which confirms that developers mainly use DHA as Google recommends, without bypassing their guidelines [110]. In particular, we notice that one library, `conscrypt`[14], accesses the `OpenSSLX509Certificate` and `OpenSSLX509CertificateFactory` classes using DHA.

[110]: (2019), *JNI tips*

14: https://github.com/google/conscrypt

These results comfort the idea that DHA is not yet used as a way to bypass analysis, even in the security community [97]. However, due to

[97]: Bao, He, and Wen (2018), 'DroidPro: An AOTC-Based Bytecode-Hiding Scheme for Packing the Android Applications'

the high number of benign DHA, a few malicious ones could be hidden and remained undetected. This highlights the need for tools and methods that take into account this kind of accesses, such as the one described in Chapter 7.

6.4 Conclusion

In this chapter we addressed the problem of interferences of native code over Java data, presented before in Chapter 4. Two use cases have been considered: an interference that would bring a vulnerability and an interference that would be used for obfuscating a modification of an object on the heap from the native code. For these problems, we designed two independent solutions.

First, we designed an approach based on data flows, working across the native and bytecode world. Contrary to other tainting approaches of the literature, the proposed solution handles the source code of an application. It enables to detect a missing `transient` keyword that possibly flaws a memory pointer. We carefully designed unitary tests for covering all combinations of flows going from native to native, native to Java and Java to native. We confirmed the already known vulnerability of the `OpenSSLX509Certificate` – this time automatically, and not manually –, and we investigated the Telegram code source. We found programming errors for Telegram that cannot be considered as vulnerabilities, but that would be if a developer decides to serialize the concerned class. Observed performances for Telegram, that contains more than one million lines of code, show that our tool can be used when preparing a release of an application.

Second, we designed a new method for detecting DHA that bypasses the Java Native Interface. DHAs is widely used for optimization purpose and our manual investigation of the suspicious uses show that no malicious or obfuscation usage is present in the analyzed datasets. This is not surprising as such a technique is introduced by this thesis [7].

Nevertheless, as an obfuscation method could be based on DHAs in the near future, we develop a dedicated tool for handling such an obfuscation technique. This is the contribution presented in the next part of this thesis.

Reverse-engineering of multi-language Android applications

Reverse-engineering of multi-language Android applications

In Chapters 3 and 4, we presented three obfuscation techniques: packers, Bytecode Free OAT (BFO) and DHA. In Chapters 5 and 6, we showed that the packers and DHA obfuscation techniques described in the first part are actively used in the wild. No evidence of usages of BFO has been found, but the detection techniques that have been implemented only focus on the simple forms of this obfuscation and may have missed some instances.

Thus, it is necessary for reverse-engineering tools that intend to work on Android applications to handle these obfuscation techniques. However, existing state-of-the-art tools do not work with BFO and DHA, the new obfuscation techniques we introduced. Additionally, generally speaking these tools try to recover the unobfuscated form of the bytecode of the analyzed application. Although when successful, this approach is ideal since it totally removes any benefit from using obfuscation techniques, this forces tools to handle specifically every possible obfuscation. When a new obfuscation technique appears, a corresponding deobfuscation technique has to be developed and integrated inside the tool. This cat-and-mouse game is in favor of the obfuscated applications since it is very hard for analysis tool developers to correctly guess what the future obfuscation techniques will be.

We believe that detecting if an application is a malware or analyzing a malware does not require to have the full code of the application. Indeed, understanding the overall behavior of an application does not require to understand all the effects of all the instructions that compose it. In this part, we focus on building *OATs'inside*, an hybrid tool that is able to retrieve the behavior of an Android application regardless of the potentially used obfuscation techniques. To this extent, *OATs'inside* observes and reports, at execution time, all the effects of the application on the Android system and create, statically after the execution, a graph-based model of the application's behavior. This allows to generically study the application.

Chapter 7 describes the architecture of *OATs'inside*, a new analysis tool that generically handles the new obfuscation techniques presented in this thesis. Chapter 8 describes the implementation challenges encountered when developing *OATs'inside* and their corresponding solutions.

OATs'inside: Retrieving behavior of multi-language applications 7

As shown in the first part of this manuscript, state-of-the-art Android analysis tools can be defeated by obfuscations based on native interferences. Additionally, it has shown that static analysis can be easily hindered. That is why this part presents *OATs'inside*, a new dynamic tool that takes these interferences into account.

This tool is intended to analyze Android applications and retrieve their behavior. It is noteworthy that it does not intend to retrieve the source code of the analyzed application: it outputs graph representations of what the application realizes. These graphs are composed of Java-level behaviors, *e.g.* method invocations or object modifications. While this representation misses low-level native events, it allows to understand how the application manipulates the Android environment. When manually used, *OATs'inside* proposes to the analyst to conduct a symbolic analysis on a method of special interest. This additional static analysis retrieves the data flow between the different element of the output graphs.

Additionally, *OATs'inside* does not modify the analyzed application. That is, all its analysis is conducted inside the Android runtime or on the analysis computer. This prevents applications from crashing due to unstable modifications.

Section 7.1 reviews the contributions in the literature related to native applications analysis and highlights gaps that *OATs'inside* fills. Section 7.2 presents the architecture and the different modules of *OATs'inside*, a new Android hybrid analysis tool. An example of obfuscated application analysis using *OATs'inside* is given in Section 7.3. Then, Section 7.4 shows the overhead induced when analyzing an application using *OATs'inside*. Finally, Section 7.5 discuses the stealthiness of *OATs'inside*.

7.1 Adapting instrumentation system to multi-language applications

As stated in Section 2.4, analysis techniques for native applications have been mainly declined in three fields: unpackers [95], taint analysis [84, 106] and application instrumentation [83, 88].

To assess how these tools would perform their analysis on obfuscated applications, we designed unit tests, as reported in Table 7.1. As these tools are most of the time unavailable, we minutely read the papers describing these fives tools: TIRO [95], ARTist [88], TaintART [106], JN-SAF [84], and Malton [83]. In Table 7.1, each reported column corresponds to a tool.

To build the test cases, we read the Dalvik bytecode specification[1] to enumerate all possible Java source statement behaviors. These behaviors

1: https://source.android.com/devices/tech/dalvik/dalvik-bytecode

Table 7.1: State-of-the-art solutions against native obfuscations

	Original source code	**Bytecode**															**Native**														
	Case	*DEX only*					*Pack DEX*					*Bytecode Free OAT (BFO)*					*JNI*					*JNI+obf*					*Direct Heap Access (DHA)*				
	Evaluated tool	A	Ta	T	J	M	A	Ta	T	J	M	A	Ta	T	J	M	A	Ta	T	J	M	A	Ta	T	J	M	A	Ta	T	J	M
Method	Invoke / Return	●	●	●	●	●	-	-	●	-	○	-	-	-	-	○	-	-	-	●	○	-	-	-	●	○					
Allocation	Object	●	●	●	●	●	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-					
	Primitive variable	●	●	●	●	●	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-					
	Primitive Array	●	●	●	●	●	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-					
Access	Object Field	●	●	●	●	●	-	-	●	-	○	-	-	-	-	○	-	-	-	●	○	-	-	-	●	○	-	-	-	-	-
	Primitive variable	●	●	●	●	●	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-					
	Primitive Array	●	●	●	●	●	-	-	●	-	○	-	-	-	-	○	-	-	-	●	○	-	-	-	●	○	-	-	-	-	-
Operations	Object Field	●	●	●	●	●	-	-	●	-	○	-	-	-	-	○	-	-	-	○	○	-	-	-	-	○	-	-	-	-	-
	Primitive variable	●	●	●	●	●	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-					
	Primitive Array	●	●	●	●	●	-	-	●	-	○	-	-	-	-	○	-	-	-	○	○	-	-	-	-	○	-	-	-	-	-
Condition	Object Field	●	●	●	●	●	-	-	●	-	○	-	-	-	-	○	-	-	-	○	○	-	-	-	-	○	-	-	-	-	-
	Primitive variable	●	●	●	●	●	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-					
	Primitive Array	●	●	●	●	●	-	-	●	-	○	-	-	-	-	○	-	-	-	○	○	-	-	-	-	○	-	-	-	-	-
Typing	Check	●	●	●	●	●	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-					
	Cast	●	●	●	●	●	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-					
Exception	Throw / Catch	●	●	●	●	●	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-					
Monitor	Enter / Exit	●	●	●	●	●	-	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-					

O: ***OATs'inside***; A: ARTist [88]; Ta: TaintART [106]; T: TIRO [95]; J: JN-SAF [84]; M: Malton [83]
Retrieval is ●: fully, ○: partially, -: empty
■: not applicable

are divided into 8 families and 20 categories listed in the two first columns of Table 7.1. For each category of behaviors, we distinguish, when relevant, three Java types: object, primitive variable or primitive array. For example, the condition category represents changing the execution flow (*e.g.* `if` statements), depending on the value of an object field, a primitive variable allocated onto the stack, or primitive array element. Then, the test cases were packaged in a single application obfuscated in five different versions:

- *DEX only*: test cases made in Dalvik bytecode.
- *Pack DEX*: packed[2] version of *DEX only*.
- *BFO*: BFO[3] version of *DEX only*. BFO consists in compiling the DEX file into assembly and then, keeping only the resulting assembly.
- *JNI*: test cases implemented in C++ using JNI.
- *JNI+obf*: resulting from the usage of Obfuscator-LLVM [111] on the *JNI* version.
- *DHA*: DHA[4] version of *JNI*. DHA consists in obfuscating heap accesses, then other tests cases are irrelevant (grayed in Table 7.1).

Unit tests are precisely described in Appendix B.

2: Obfuscation technique described in Section 3.1

3: Obfuscation technique proposed described in Section 3.2

4: Obfuscation technique proposed in Section 4.2

We present below the evaluation of state-of-the-art tools on the designed test cases.

TaintART [106] and ARTist [88] rely on `dex2oat`, the Android compiler responsible for compiling Dalvik bytecode into assembly code. Both approaches add instrumentation instructions during the compilation step by customizing `dex2oat`. Since `dex2oat` can only compile Dalvik bytecode, these tools can only work on this bytecode. That is why they can only retrieve behaviors for the *DEX only* version.

TIRO [95] is an unpacker. Thus, it can output the loaded bytecode of the *Pack DEX* version. Then, the bytecode being available, the analyst can retrieve all the application behaviors. However, excluding the behavior related to code loading, which is not an elementary Java behavior, TIRO does not analyze any native code. That is why it cannot work for other versions.

JN-SAF [84] is a static analysis-based taint tracking tool. It uses angr [112] to statically track information flows. The key idea is to initialize the JNI entry point table with symbolic addresses representing the different JNI methods. Then, JN-SAF can represent their effects symbolically. All these approaches provide all the sensitive information flow between methods, but retrieve only the behaviors involved in the information flow. Therefore, it does not care about allocations, typing, exceptions, or monitoring of events, and do not output them at all. Owing to its static nature, it cannot work with the *Pack DEX* version. Moreover, JN-SAF targets only native methods and does not handle AOTC-compiled code and, thus, it misses the *BFO* version. Because JN-SAF aims at tracking flows, it does not log explicitly the conditions and the operations made by the code. However, these elements are taken into account when computing a data flow. That is why some partial information about operations and conditions is captured. Finally, because JN-SAF relies on classical symbolic execution, obfuscated assembly can overload its analysis by, for example, adding conditional instructions that depend on the application inputs [113]. This prevent the *JNI+obf* version from being handled.

[113]: Banescu, Collberg, Ganesh, Newsham, and Pretschner (2016), 'Code obfuscation against symbolic execution attacks'

Malton [83] is a hybrid analysis platform that performs data taint tracking over framework libraries and system calls. It relies on Valgrind [96] to hook method calls, ART, and framework libraries. It stores the address of every Java and native method and then checks, for every jump, if the destination address is the address of a method. Then, Malton reconstructs the Java objects corresponding to the arguments by parsing the memory. It also hooks framework methods responsible for loading code and the JNI entry points, and intercepts all system calls. Finally, it propagates taints through every assembly instruction. Moreover, Malton leverages concolic execution to trigger or force the execution of specific, manually tagged, code areas. The output only focuses on information flow and thus, does not include allocations, typing, exceptions, or monitoring of events. Moreover, Malton cannot hook methods from the analyzed APK, but only the runtime and framework ones. Therefore, it does not retrieve information about the internal code methods and classes. That is why all its outputs are qualified as partial. Moreover, because Malton is dynamic, it can handle the *Pack DEX* version. The symbolic analysis that is conducted is concolic: it follows the execution and, thus, is not sensitive to obfuscation (unlike JN-SAF) and can tackle the *JNI+obf* version. Finally, Malton works with the *BFO* version because it does not rely on the APK structure but bases all its analysis on executed assembly instructions.

Additionally, all these tools relies, when they monitor native heap accesses[5], on the JNI interface. Thus, as stated in Section 4.2, they all miss the *DHA* version since DHA obfuscation consists in bypassing this interface.

5: Accesses to Java fields

Thus, we design *OATs'inside* such that:

- It handles all forms of Android applications: Dalvik bytecode, compiled bytecode (BFO) and native code.
- It handles all Java instructions, including: operation, condition, typing, exception and monitoring.
- It does not rely on the usage of the Java Native Interface (JNI), hence it handles DHA.

7.2 *OATs'inside* architecture

To address native-based obfuscated Android applications, we describe *OATs'inside*, a deobfuscator that supports every application that performs Java operations, even if it is protected by full-native-based and runtime-based obfuscation techniques. *OATs'inside* combines dynamic analysis with symbolic execution. The dynamic analysis gathers sequences of low-level events, and the symbolic execution is driven by these events. *OATs'inside* outputs a CFG that can be passed to existing security analysis tools such as GroddDroid [49], IntelliDroid [50], or directly to a human analyst. The CFG is said to be *at the object level* because it contains instructions acting on objects such as calling methods or setting object fields. It describes the contents of each method, the conditional expressions involved in the control flow instructions, the data flow between actions, and the interprocedural calls.

[49]: Abraham, Andriatsimandefitra, Brunelat, Lalande, and Tong (2015), 'GroddDroid: a gorilla for triggering malicious behaviors'
[50]: Wong and Lie (2016), 'IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware'

OATs'inside adopts a two-step analysis: first, a dynamic analysis, followed by a concolic analysis. These steps are based on four main modules as described in Figure 7.1.

During the dynamic step, the Runner module executes the application and logs every action dealing with objects. As an application requires external inputs, the execution is either driven manually or via a dedicated exploration tool [49, 50, 52, 114]. The CFG Creator module initializes a first version of the CFG from the actions obtained from the Runner module.

[49]: Abraham, Andriatsimandefitra, Brunelat, Lalande, and Tong (2015), 'GroddDroid: a gorilla for triggering malicious behaviors'
[50]: Wong and Lie (2016), 'IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware'
[52]: Hao, Liu, Nath, Halfond, and Govindan (2014), 'PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps'
[114]: Machiry, Tahiliani, and Naik (2013), 'Dynodroid: An input generation system for android apps'

During the concolic step, the Concolic Analyzer module performs a symbolic execution based on the actions logged by the Runner module and memory snapshots issued by the Memory Dumper. It enriches the CFG by recovering conditional expressions at branching nodes and data dependencies between actions.

To illustrate *OATs'inside*'s methodology, we developed `PINtest`, a PIN verification application written in Java. It runs transparently on an Android 7.0 smartphone. We will use this application as a running example throughout the rest of this section. For the sake of readability, we give a simplified version of its source code in Figure 7.1. `SimpleTestPIN.test` has three possible behaviors. If the `pin` field of the calling object (`this`) is negative, an exception is thrown. It returns true when the `pin` is the correct one (`1337`) and false otherwise. `SimpleTestPIN.test` is obfuscated using BFO technique: the bytecode is compiled into assembly and then removed.

The whole analysis is driven by a human analyst who runs the application twice with two different PINs: a negative (`-42`), which generates an exception, and a wrong positive (`42`). The final objective of *OATs'inside* is

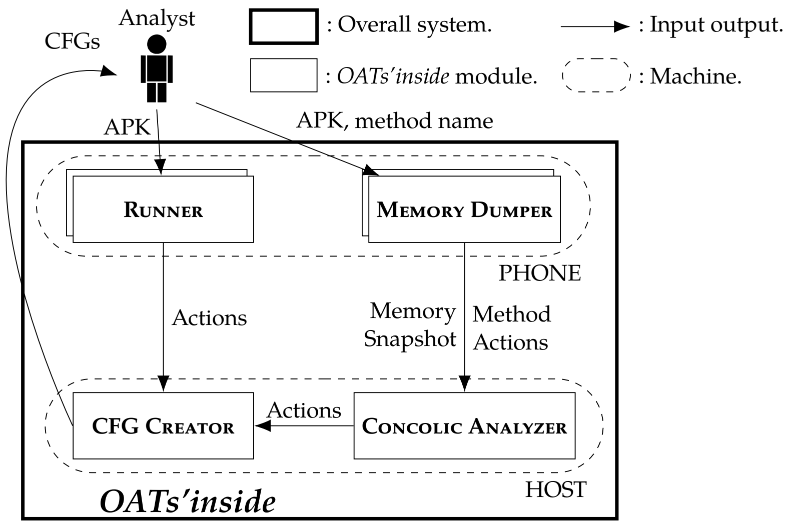


Figure 7.1: *OATs'inside* architecture

```java
// Two executions: test() with pin = -42 and pin = 42
public class SimpleTestPIN {
  public int pin = 0;
  public boolean test() throws Exception {
    if(this.pin < 0) throw new Exception("Negative PIN");
    if((this.pin ^ 0x2323) == 9754) // 1337^0x2323=9754
      return true;
    else return false;
}}
```

Listing 7.1: Simplified PIN test

to compute a CFG that best approximates the complete CFG which is, for this example, given in Figure 7.2.

[95]: Wong and Lie (2018), 'Tackling runtime-based obfuscation in Android with TIRO'
[84]: Wei, Lin, Ou, Chen, and Zhang (2018), 'JN-SAF: Precise and Efficient NDK/JNI-aware Inter-language Static Analysis Framework for Security Vetting of Android Applications with Native Code'
[88]: Backes, Bugiel, Schranz, Styp-Rekowsky, and Weisgerber (2017), 'Artist: The android runtime instrumentation and security toolkit'

7.2.1 Runner module

The Runner module is in charge of running the analyzed application and logging every object-level action performed by the application. There are nine different object-level actions[6]: invoking or returning from a method, reading from or writing to an object field, allocating an object, entering or exiting a monitor session, and throwing or catching exceptions.

6: https://source.android.com/devices/tech/dalvik/dalvik-bytecode

Applications contain three types of code: DEX, OAT, or native. State-of-the-art approaches suffer from one or more of the following limitations: they do not support OAT, arguing that the DEX bytecode is always available [84, 88, 95]; they do not collect all the possible actions [81, 83, 84, 87]; or they are bypassed by DHA obfuscations because they rely on JNI [83, 84]. The Runner module lifts these limitations by using simple monitoring methods inside the ART library when possible and low-level debug methods otherwise.

[87]: Yan and Yin (2012), 'DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis'
[81]: Qian, Luo, Shao, and Chan (2014), 'On tracking information flows through jni in android applications'
[83]: Xue, Zhou, Chen, Luo, and Gu (2017), 'Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART'
[84]: Wei, Lin, Ou, Chen, and Zhang (2018), 'JN-SAF: Precise and Efficient NDK/JNI-aware Inter-language Static Analysis Framework for Security Vetting of Android Applications with Native Code'

Table 7.2 summarizes how each action is monitored, depending on the binary code type. If the action goes through the ART (all actions in the Dalvik bytecode, and object allocation, monitoring of the entry or exit, and exception handling in all code types), then a direct event is generated by adding a call to the logger inside the runtime. Otherwise, the action is retrieved by generating a low-level event based on debugging or memory protection capabilities. In particular, an OAT code that accesses (read or write) an object field is captured by disabling the heap memory: all

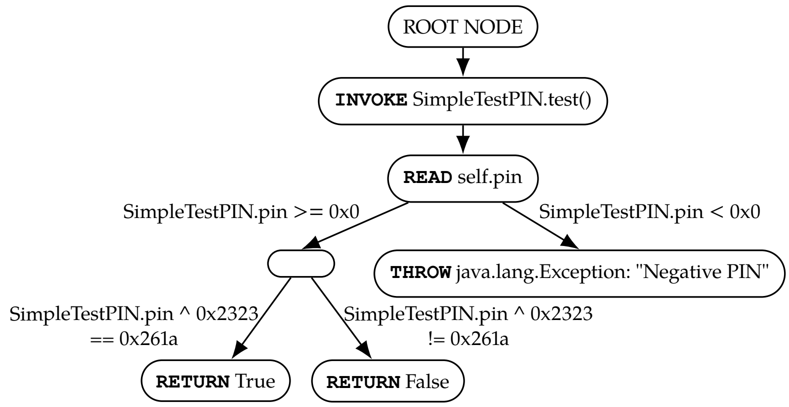


Figure 7.2: Expected output for `SimpleTestPIN.test`

Table 7.2: Monitoring of object actions for different types of executed code

Analyzed binary		DEX	OAT	Native
Invoke return	Method	interpreter	class linker	class linker
	Event type	direct	breakpoint	breakpoint
Field access (read/write)	Method	interpreter	disable heap	disable heap or JNI
	Event type	direct	SEGV	SEGV or direct
Allocation monitor	Method	allocator	allocator	allocator
	Event type	direct	direct	direct
Exceptions throw or catch	Method	exception handler	exception handler	exception handler
	Event type	direct	direct	direct

heap accesses will generate a `SEGV` event. The same applies to native code when bypassing the JNI interface. Additionally, an OAT or a native code that invokes a method without calling the runtime is captured by hooking the address table and generating a breakpoint event.

Consequently, three types of events are generated or captured: 1. direct events: allocating an object, entering or exiting a monitor session, and throwing or catching an exception; 2. breakpoint events: invoking and returning a method; 3. `SEGV` events: reading or writing an object field.

To manage these events, we built the Runner module, a patch of the Android runtime whose main components are represented in Figure 7.3, where the three types of events are annotated as (D) for direct events, (B) for breakpoint events, and (S) for `SEGV` events. The runtime has information about high-level structures such as classes, signatures, and objects, and also knows low-level entities such as register values, heap addresses, and kernel signals. Thus, patching the runtime allows bridging the semantic gap between the assembly and the bytecode world. The patch is divided into two entities: the `ProbeManager` and the `SignalManager`. The `ProbeManager` handles high-level events. It is the interface between the runtime and the output file when actions are logged. It logs object-level

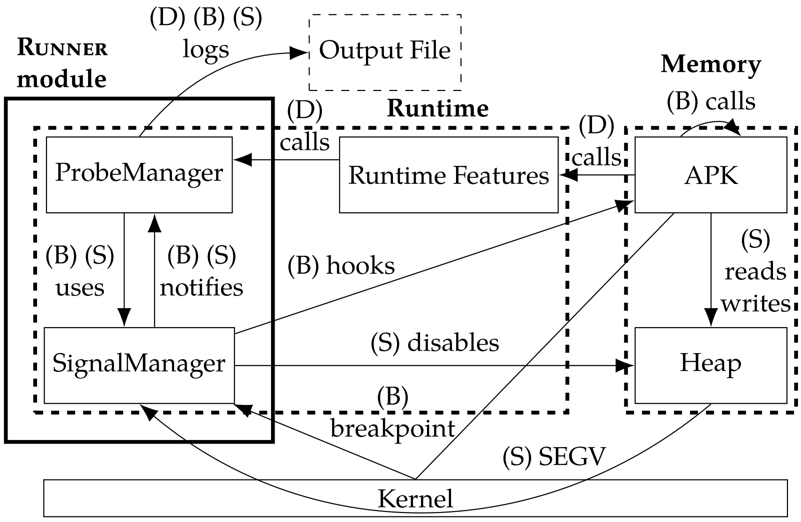


Figure 7.3: Runtime patch architecture

actions when they occur. The `SignalManager` handles low-level events. It sets breakpoints, handles kernel signals, and notifies the `ProbeManager` to log associated actions.

The `ProbeManager` logs each event with its associated instruction address. This allows linking of the bytecode events to the assembly code and will be used by the Concolic Analyzer module (*cf.* Section 7.2.4). The thread identifier from which the event originates is also logged, to avoid concurrent execution issues. In the following, we detail how the three types of events are handled by the Runner module.

Direct events These events, indicated as "direct" in Table 7.2, are generated by the runtime library code. For example, when an object is allocated, the runtime allocator is called. The allocator allocates memory and returns it to the application. A call to the `ProbeManager`, containing the class of the allocated object, is added to the allocator, before returning to the APK code. This part of the Runner module links the assembly world (the allocated address) and the bytecode world (the object class, independently of the executed code type). Entering or exiting a monitor session and throwing or catching an exception are logged using similar mechanisms in the runtime monitor and exception handler.

Breakpoint events These events correspond to invoking or returning from a method. To log the invoke action, the Runner module needs to be notified when the first instruction of the method is executed. The classical way to do this would be to set a breakpoint at this address, catch the breakpoint (`SIGTRAP` signal), log the action, remove the breakpoint, and resume the execution. However, removing the breakpoint would prevent catching of future calls to this method. At first glance, instead of directly resuming the execution, a possible improvement would be to step one instruction, reset the breakpoint, and resume the execution. In this way, the breakpoint would be available for further calls. However, removing and then resetting the breakpoint would generate a concurrency issue if multiple threads execute the same method. In practice, every application runs more than five threads (garbage collector, intents, profiler, *etc.*).

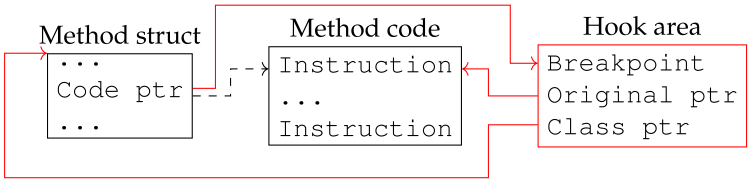


Figure 7.4: Method invocation hooking process

To solve this problem, the RUNNER module modifies the address of all the methods linked by the class linker in the runtime. As shown in Figure 7.4, the real address of the method (`code ptr`) is replaced by the address of a new dedicated area (`hooking area`). It contains a breakpoint instruction, for generating the invoke action; the original address of the method's code (`original ptr`); and a pointer to the runtime internal structure representing the method. This last pointer is used, when the breakpoints is hit, to easily access the information about the hooked method such as its name or the list of its parameters.

The same problem exists for catching the return event and is thus solved similarly. When this type of event occurs, as the breakpoint only carries the information about the executed address, we retrieve the method signature by using the runtime internal structure of the dedicated areas. This bridges the semantic gap between the assembly (instruction address) and the bytecode (method signature) world.

SEGV events These events correspond to accesses, by reading or writing, to object fields stored on the heap. Catching such accesses requires watching every load or store instruction to detect those that target heap addresses. To this end, the `SignalManager` uses the system page protection mechanism: it forbids all accesses to the heap memory pages using `mprotect`, causing any access to object fields to generate a fault, a `SEGV` kernel signal, which is caught by the `SignalManager`, which retrieves the faulty address.

Then, the garbage collector's internal structures are leveraged to map the assembly address to an object field. This is then transmitted to the `ProbeManager`, which in turns logs this heap access.

Finally, for an application to run as expected, the heap access should actually be performed. The heap is re-enabled, and a single instruction is executed before disabling the heap again. To avoid concurrent accesses to the heap in the meanwhile, a thread-oriented `mprotect` has been added to the kernel [115]. More details about this thread-oriented `mprotect` are given in Section 8.1.7.

[115]: Razeen, Lebeck, Liu, Meijer, Pistol, and Cox (2018), 'SandTrap: Tracking Information Flows On Demand with Parallel Permissions'

Running example output Listing 7.2 gives the actions outputted by the RUNNER module for the running example, from lines 2 to 18 for the first execution and lines 21 to 28 for the second one. When these logs and the source code of Listing 7.1 are compared, it shows that most of the elements are retrieved. The access to the `pin` field (lines 5 and 6, and lines 24 and 25) is present for each execution of the method. The `throw` is divided into four events: the string creation (lines 8 and 9), the initialization of the exception object and its associated return (lines 11 and 12, and lines 14 and 15), and the throw itself (lines 17 and 18). Finally, the

```
1  # first run
2  tid: 3520, event_address: 512236427828
3  invoke SimpleTestPIN;test()
4
5  tid: 3520, event_address: 512236429712
6  read SimpleTestPIN;pin => -42
7
8  tid: 3520, event_address: 512236429884
9  newObj String => 315654920
10
11 tid: 3520, event_address: 512236429832
12 invoke java/lang/Exception;<init>((String) 315654920)
13
14 tid: 3520, event_address: 512236429836
15 return void
16
17 tid: 3520, event_address: 512236429844
18 throw java.lang.Exception("Negative PIN")
19
20 # second run
21 tid: 3520, event_address: 512236427908
22 invoke SimpleTestPIN;test()
23
24 tid: 3520, event_address: 512236429712
25 read SimpleTestPIN;pin = > 42
26
27 tid: 3520, event_address: 512236427912
28 return false
```

Listing 7.2: Runner module output on `SimpleTestPIN`

`return false` (lines 27 and 28) is detected. However, some Java actions are missing. The `return true`, which is never executed in our case, is not logged. The conditions are also lacking, as well as the usage of the allocated string (lines 8 and 9), which is a dependency of the `init` call (lines 11 and 12). Obtaining these pieces of information is the purpose of the remaining modules.

7.2.2 CFG creator module

The CFG Creator module is in charge of creating the CFG. In fact, this graph is the union of the interprocedural call graph (iCFG) and the methods' object-level control flow graphs (olCFGs). These CFGs are built sequentially, using the events outputted by the Runner module: first, actions are split by method and the iCFG is created, and then the olCFG of each method is computed.

iCFG computation The logs are split by method. The boundary of a method is defined by two properties of the Dalvik bytecode[7]. First, each method begins with an invoke and ends with a return. Then, each return comes after its corresponding invoke action. An invoke action is not necessarily followed by a return: methods may never return (for example, the main loops of graphical engines are infinite loops). Second, there is no jump across method bodies ("`goto`"-like statement). Thus, if a method m_b is invoked after a method m_a, the return of m_a cannot occur before the return of m_b. Invocations cannot be interleaved. Thanks to this last remark, we can easily split actions by method by reconstructing the call stack. During the call stack computation, the iCFG is made: when an invoke event occurs, an edge is added between it and the last method.

7: https://source.android.com/devices/tech/dalvik/dalvik-bytecode

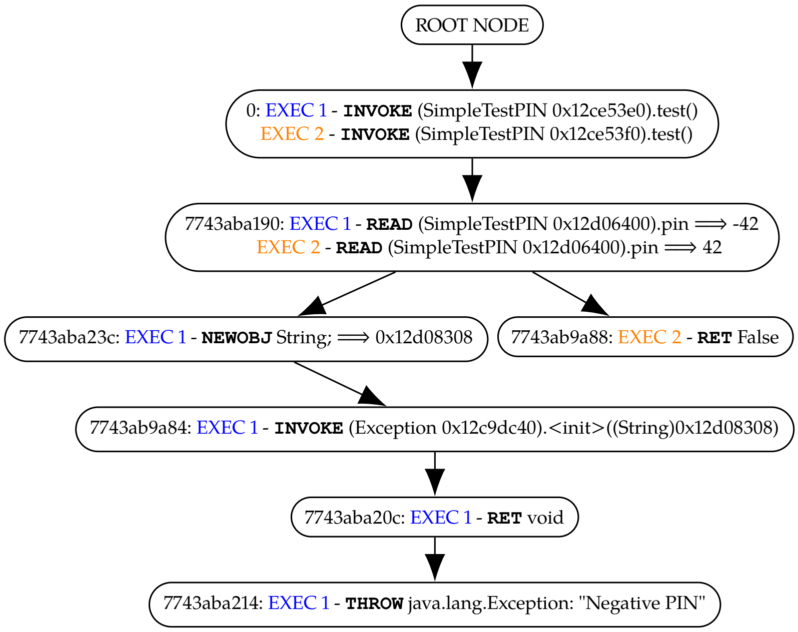


Figure 7.5: Object-level control flow graph of `SimpleTestPIN.test`

Note that actions are mixed between different threads. The inclusion of the thread identifier in logs allows each thread CFG to be built independently.

olCFG computation This algorithm takes as input the sequence of actions of a method. Each node is uniquely characterized by the address of the assembly instruction generating the action. Thus, if the same address is executed multiple times (several executions of the method or loops in the method's body), the node representing this action contains the details of all executed actions. For example, in Figure 7.5, the node `7743aba190` contains two read actions from two different executions: the first read obtains the value `-42`, and the second one, `42`.

A special root node is added to mark the beginning of the method. When iterating over the sequence of actions, the algorithm creates an edge from the current instruction to the next one. When a node holds several actions, several destinations can follow, hence revealing the existence of a condition whose nature is not known yet. Note that, if ASLR[8] is activated, addresses change between two different executions, breaking the node unicity previously mentioned. Nevertheless, using offsets to the base address of the loaded binary solves this problem.

8: Address Space Layout Randomization

Running example output Figure 7.5 shows the olCFG computed for the `SimpleTestPIN.test` method. This is a human-readable representation of Listing 7.2. The same elements are missing: the "return true" case is not present, the condition expressions are missing, and the dependency between the allocation and the invocation is not explicit. Thus, the analyst cannot retrieve the correct PIN number: he/she cannot identify the conditions that need to be satisfied because they are not present.

7.2.3 Memory Dumper module

The Memory Dumper module is responsible for making snapshots of the memory. This module is called just before the execution of a method and dumps the whole memory of the process. These snapshots give the method's code and data to the Concolic Analyzer module in charge of the symbolic execution. In this way, if a method is used as a place holder for several unpacked assembly codes, each snapshot will provide the current version of the code. This module can be either activated by the Runner at each method execution (but it considerably slows down the analysis) or activated on demand by the human analyst.

7.2.4 Concolic analyzer module

The Concolic Analyzer module symbolically executes the dumped assembly code and uses the values observed from the actions logged by the Runner module. The first step allows building a CFG describing the execution paths explored during the dynamic analysis; however, it lacks both conditional expressions and how variables are manipulated by the actions. Such knowledge is important for the analyst because it helps to understand the behavior execution. For example, in Figure 7.5, the parameter (`0x12d08308`) of the invoke (`0x7743ab9a84`) should be linked with the preceding allocation.

The Concolic Analyzer module takes as input the list of actions logged by the Runner module and all the memory snapshots made by the Memory Dumper module, and then it generates the conditional expressions at branching nodes and the data dependencies between variables. This is done in three steps:

1. Assembly breakpoints: in the assembly code returned by the Memory Dumper module, a breakpoint is set for all generated actions. For example, we set a breakpoint at the address `0x7743aba190` (READ `pin` field action), which corresponds to the instruction `ldr w2, [x1, #12]`.
2. Symbolic execution: the symbolic execution is initialized: the `PC` is set to the entry point of the method and a symbolic value is created for each method parameter. The symbolic execution can stop for one of three reasons: a breakpoint, a condition, or the end of the method is reached.
3. Analysis stop and concretization: when the symbolic execution is stopped, the analysis flow is guided and symbolic values are managed. Two types of stops are handled:

    a) Breakpoint: first, if the action type is allocation, read, or return, a new symbolic value is created, named according to the Java class or field name. For example, the read at address `0x7743aba190` creates a symbol "SimpleTestPIN.pin," as shown in Listing 7.3, line 7. The instruction output register `w2` is set to this new symbolic value. Second, if the action type is read, write, or invoke, the read register or memory value is retrieved. If this expression is symbolic, it is outputted. For example, the parameter of the invoke is logged in line 18 with the name created previously in line 14.

b) Condition: the symbolic engine provides the two symbolic conditions corresponding to the two branches. They are concretized: symbolic values are replaced by the concrete value given by the trace. The condition that holds is logged and the symbolic execution is resumed, taking the corresponding path. Note that the concretization happens only for logging and choosing the branch; however, registers and memory stay symbolic to continue tracking data dependencies.

One key point of this symbolic analysis is that no SMT solver is ever called. Instead, only value replacement (concretization) is made. Moreover, the analysis always follows only one path. This saves the analysis from the usual drawbacks of symbolic analysis that could lead to high execution time or memory space overhead [116]. Additionally, we prove in Appendix C that the Concolic Analyzer module terminates and is correct (gives the expected results).

[116]: Baldoni, Coppa, D'elia, Demetrescu, and Finocchi (2018), 'A survey of symbolic execution techniques'

Finally, the results of the Concolic Analyzer module are sent to the CFG Creator module to improve the olCFG. Blank nodes are added, which represent the computation of a condition or branches that have never been taken during the dynamic analysis step. This is the final human-readable output of *OATs'inside*.

Running example output After the Concolic Analyzer execution on the snapshot and the actions retrieved for the running example, the enriched list of actions is given in Listing 7.3. Compared to Listing 7.2, three new condition events have been added (lines 9 and 10, 34 and 35, and 37 and 38). The first two (lines 9 and 10 and lines 34 and 35) are the opposite because they represent the same condition that is taken or not. It corresponds to the check that the `pin` field is positive, which is expressed in the condition expression written in the listing. The third condition (lines 37 and 38) is the comparison to the correct pin value, which is XORed. The symbol `SimpletestPIN.pin` has been concretized by `42` using line 31 to choose the branch to execute.

Moreover, four symbolic annotations have been added (lines 7, 14, 18, and 32). The two lines attached to the read (lines 7 and 32) and the line attached to the allocation (line 14) represent the new symbolic values created. The remaining one (line 18) shows that the symbol representing the output of the allocation (line 14) is directly used as an invocation parameter when calling the constructor exception `<init>`.

The updated olCFG of the `SimpleTestPIN.test` method is shown in Figure 7.6. An analyst can now easily understand how the PIN is handled.

7.2.5 Unit test results

Coming back to the unit tests that we introduced in Section 7.1[9], we evaluated if *OATs'inside* succeeds in capturing all Java source statement behaviors. Results are reported in Table 7.3.

9: Unit tests are precisely described in Appendix B.

OATs'inside retrieved almost every behavior. When the payload of the APK is executed through bytecode (*DEX only* and *Pack DEX* versions), *OATs'inside* passes all the test cases because it hooks the bytecode interpreter in the runtime. For other versions, executed through assembly

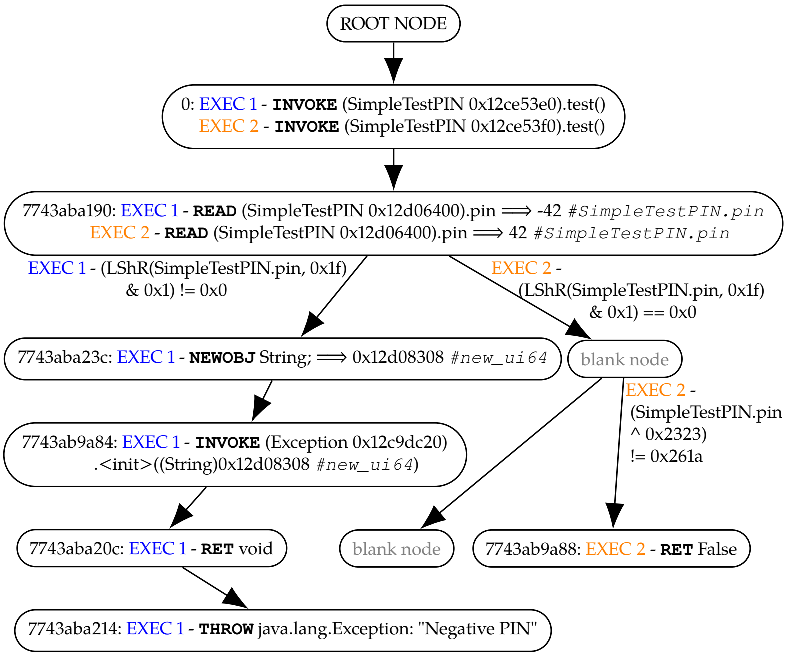


Figure 7.6: Object-level control flow graph of `SimpleTestPIN.test`

Table 7.3: *OATs'inside* against native obfuscations

	Original source code	Bytecode			Native		
	Case	*DEX only*	*Pack DEX*	*BFO*	*JNI*	*JNI+obf*	*DHA*
Method	Invoke / Return	R	R	R	R	R	■
Allocation	Object	R	R	R	R	R	■
	Primitive variable	R	R	-	-	-	■
	Primitive Array	R	R	R	R	R	■
Access	Object Field	R	R	R	R	R	R
	Primitive variable	R	R	-	-	-	■
	Primitive Array	R	R	R	R	R	R
Operations	Object Field	R	R	R+C	R+C	R+C	R+C
	Primitive variable	R	R	-	-	-	■
	Primitive Array	R	R	R+C	R+C	R+C	R+C
Condition	Object Field	R	R	R+C	R+C	R+C	R+C
	Primitive variable	R	R	-	-	-	■
	Primitive Array	R	R	R+C	R+C	R+C	R+C
Typing	Check	R	R	?	R	R	■
	Cast	R	R	?	?	?	■
Exception	Throw / Catch	R	R	R	R	R	■
Monitor	Enter / Exit	R	R	R	R	R	■

R: Retrieved by the RUNNER module
R+C: Retrieved by the CONCOLIC ANALYZER module
?: Retrieving would require more static analyses
■: not applicable

Listing 7.3: Concolic Analyzer module output on `SimpleTestPIN.test`

```
# first run
tid: 3520, event_address: 512236427828
invoke SimpleTestPIN;test()

tid: 3520, event_address: 512236429712
read SimpleTestPIN;pin => -42
symb: "SimpleTestPIN.pin"

tid: 3520, event_address: 512236429716
condition "(LShR(SimpleTestPIN.pin, 0x1f) & 0x1) != 0x0"

tid: 3520, event_address: 512236429884
newObj String => 315654920
symb: "new_ui64"

tid: 3520, event_address: 512236429832
invoke java/lang/Exception;<init>((String) 315654920)
symb: ["new_ui64"]

tid: 3520, event_address: 512236429836
return void

tid: 3520, event_address: 512236429844
throw java.lang.Exception("Negative PIN")

# second run
tid: 3520, event_address: 512236427908
invoke SimpleTestPIN;test()

tid: 3520, event_address: 512236429712
read SimpleTestPIN;pin => 42
symb: "SimpleTestPIN.pin"

tid: 3520, event_address: 512236429716
condition "(LShR(SimpleTestPIN.pin, 0x1f) & 0x1) == 0x0"

tid: 3520, event_address: 512236429736
condition "(SimpleTestPIN.pin ^ 0x2323) != 0x261a"

tid: 3520, event_address: 512236427912
return false
```

instructions, only two classes of operations were missed that we detail in the rest of the section. Nevertheless, we can state that, globally, *OATs'inside* is robust against obfuscation and can analyze any type of APK.

For allocations, accesses, operations, and conditions realized on an assembly variable, *OATs'inside* missed, as expected, these behaviors. Indeed, it corresponds to manipulation of registers and the stack, which are areas that are not monitored by the proposed method. It has to be noted that watching them is not trivial. Stack and registers store numerous different pieces of information such as return addresses, arguments, and clobbered registers, and finely distinguishing between them is a very difficult task. Missing these variable-oriented behaviors for native code is not an important limitation because they are still considered by the symbolic execution. For example, a Java field copied in an assembly variable and copied back to another field would be detected by *OATs'inside* as a data dependency between the two fields, silently dropping the variable.

Second, type checking (for the *OAT only* version) and casting are not retrieved by *OATs'inside* either. Indeed, these behaviors are performed at compile time and are never present in the generated assembly code. However, the lost information could be retrieved by carrying out more static analyses on the obtained CFG. Analyses such as type propagation and checking [117] could be used to detect missing typing operations. This work is left as future improvement for *OATs'inside*.

[117]: Cardelli and Wegner (1985), 'On Understanding Types, Data Abstraction, and Polymorphism'

7.3 *OATs'inside* output on obfuscated application

In order to show *OATs'inside* output we have modified the running example to integrate DHA accesses and native code obfuscation, while still using BFO. Section 7.3.1 presents the resulting application and Section 7.3.2 describes how a user could employ *OATs'inside* to conduct an analyst of the application by playing, our-self, the role of the user.

7.3.1 Obfuscated application presentation

The test application is composed of two classes. The first one, see Listing 7.4, is a simple activity. This activity is composed of an edit area, a button and a text area. When the button is pressed, it retrieves the content of the edit area, uses the `SimpleTestPIN` class to check if this content corresponds to the correct PIN and updates the text area accordingly.

Listing 7.4: Unobfuscated activity code

```
package pg.testpin;
import [...]
public class MainActivity extends Activity {
    static { System.loadLibrary("native-lib"); }
    public SimpleTestPIN pin = new SimpleTestPIN();
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState); setContentView(R.layout.
     activity_main);
        Button bt1 = findViewById(R.id.button);
        bt1.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                TextView tv = findViewById(R.id.textView);
                EditText pinview = findViewById(R.id.editText);
                int entered_pin=-1;
                try {
                    entered_pin = Integer.parseInt(pinview.getText().
     toString());
                    pin.set_pin(entered_pin);
                } catch (NumberFormatException e) { tv.setText("Incorrect
     format"); }
                try {
                    pin.test();
                    if(pin.validated) tv.setText("Good PIN");
                    else tv.setText("Wrong PIN");
                } catch (Exception e) { tv.setText("Incorrect format"); }
            }
        });
    }
}
```

The `SimpleTestPIN` class, shown in Listings 7.5 and 7.6, is composed of two methods: `test` and `set_pin`. `set_pin`, see Listing 7.6, is implemented in C++. It adds 7331 to its paramater value and stores it inside the `pin` field using

DHA. Thus, no JNI call are made. `test`, see Listing 7.5, is implemented in Java. It throws an exception if the `pin` field is lower than `7331` (that is if the entered PIN is negative). Then, it sets `validated` field to false if the `pin` field value is different from 8668 (`0x2ff^0x2323=8668`).

Listing 7.5: SimpleTestPIN unobfuscated Java code

```java
package pg.testpin;
public class SimpleTestPIN {
  public int pin = -1;
  public boolean validated = true;
  public void test() throws Exception {
    if (this.pin < 7331) throw new Exception("Negative PIN");
    if ((this.pin ^ 0x2323) == 0x2ff) validated = true;
    else validated = false;
  }

  public native void set_pin(int pin);
}
```

Listing 7.6: SimpleTestPIN unobfuscated C++ code

```cpp
#include <jni.h>
#define PIN_FIELD_OFFSET 0x8
extern "C" JNIEXPORT void JNICALL
Java_pg_testpin_SimpleTestPIN_set_1pin(JNIEnv *env, jobject thisObj, jint pin) {
  unsigned int* thisPtr = (unsigned int*) *(unsigned int*)thisObj;
  unsigned int* field_ptr = &thisPtr[PIN_FIELD_OFFSET/4];
  *field_ptr = pin;
  *field_ptr += 7331;
}
```

Additionally, when compiling the native code, we use Obfuscator-LLVM [111] compiler to add opaque predicates and control flow flattening to the assembly code. On the other side, the bytecode of the application is compiled and removed from the APK and the OAT file.

[111]: Junod, Rinaldini, Wehrli, and Michielin (2015), 'Obfuscator-LLVM – Software Protection for the Masses'

7.3.2 *OATs'inside* output on the application

First, when analyzing an application using *OATs'inside*, the analyst has to run the application and browse it. In our example, we have run the application and we entered two PINs. First, we have entered `-321` and we saw the application complaining about the PIN format. Then, we entered `123` and saw that the application has rejected the PIN. We stopped our first run here.

After this analysis, *OATs'inside* gave the list of analyzed method. This list is shown in Listing 7.7. Since the PIN verification seems to be triggered by the button, we chose to start our investigation by having a look at `pg.testpin.MainActivity$1.onClick`.

Listing 7.7: List of executed method

```
pg/testpin
|-- MainActivity
|   |-- <clinit>
|   |-- <init>
|   '-- onCreate
|-- MainActivity$1
|   |-- <init>
|   '-- onClick
'-- SimpleTestPIN
    |-- <init>
    |-- set_pin
    '-- test
```

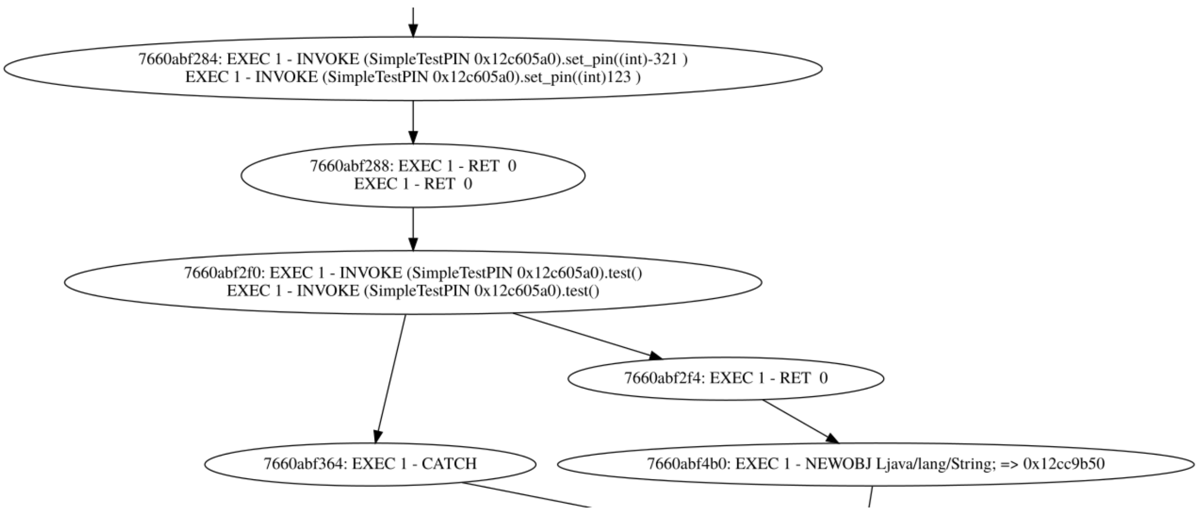


Figure 7.7: Extract of MainActivity$1.onClick olCFG

Thus, we requested *OATs'inside* to build the olCFG of this method. For sake of clarity, we have reported in Figure 7.7 only the relevant part of this graph. In this graph, we noticed that the method `SimpleTestPIN.set_pin` have been called with the PINs we entered as parameter (`-321` and `123`). Straight after, `SimpleTestPIN.test` is called and a branching is generated: either an exception is caught or a String is created. Using this graph we made a preliminary conclusion: `SimpleTestPIN.set_pin` is used to set the entered PIN and `SimpleTestPIN.test` checks if it is correct.

Naturally, we continued our analysis by digging into the olCFGs of `SimpleTestPIN.set_pin` and `SimpleTestPIN.test`, respectively shown in Figures 7.8 and 7.9. Here, the analysis started to become a bit tricky with the only information we had. Indeed, for `SimpleTestPIN.set_pin`, we noticed than the value stored inside the `pin` (`7010` and `7454`) field differs from the PINs we entered (`-321` and `123`) and we had no clue of how and if these values were related. For `SimpleTestPIN.test`, we guessed, using string related to the throw, that the exception is generated when a negative PIN is entered. However, we had no indication on how and if the `validated` field can be set to `True`.

Since the elements that we missed for continuing our analysis were: data flow (for `SimpleTestPIN.set_pin`) and potential conditions (for `SimpleTestPIN.test`), we decided to conduct a symbolic analysis on these two methods. To this extend, the application was re-run and the memory snapshots were made when calling these two methods. During this second run we only entered a positive PIN (`1234`). Indeed, we already knew what happen for negative PIN and thus we did not need to investigate more this case.

After running the symbolic analysis on `SimpleTestPIN.set_pin`, we obtained the olCFG shown in Figure 7.10. This new graph does not contain new nodes but nodes are annotated by symbolic value. In particular, we noticed that the last write value (`8565`) corresponds to the previous PIN value plus 7331 (`0x1ca3`). Since the previous PIN value has been set to the PIN we entered (`1234`), we concluded that the PIN check is made on PIN entered plus 7331. It is noteworthy that no data flow has been detected between the parameter (`1234`) and the first write. Indeed, parameters are

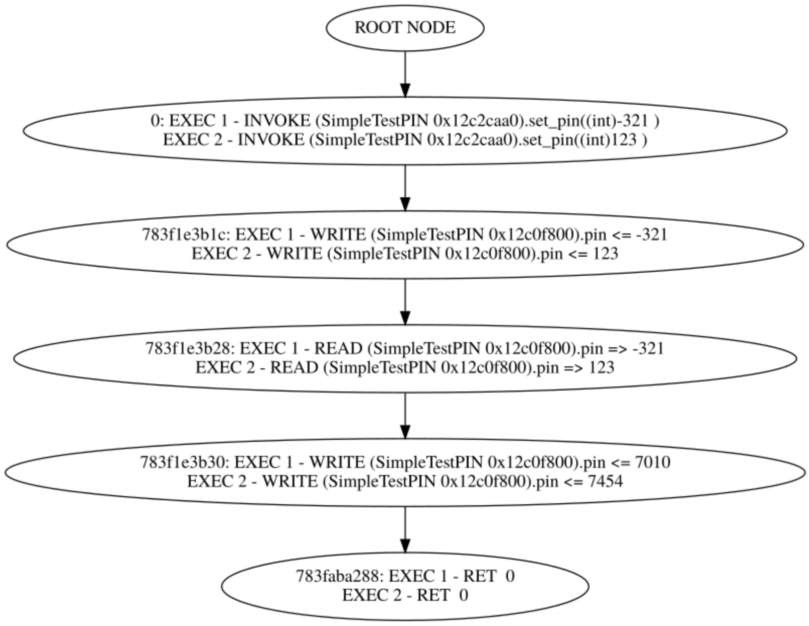


Figure 7.8: `SimpleTestPIN.set_pin` olCFG

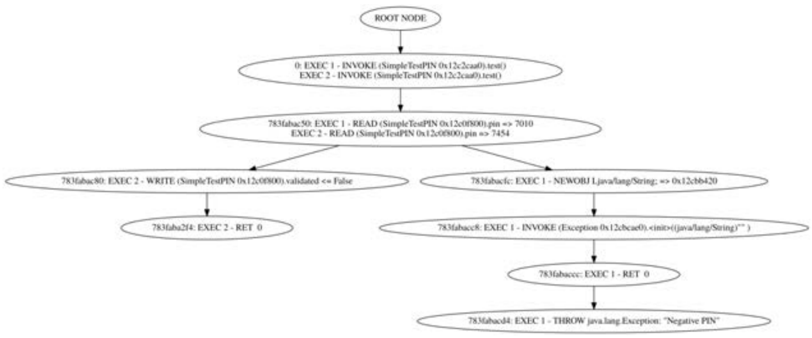


Figure 7.9: `SimpleTestPIN.test` olCFG

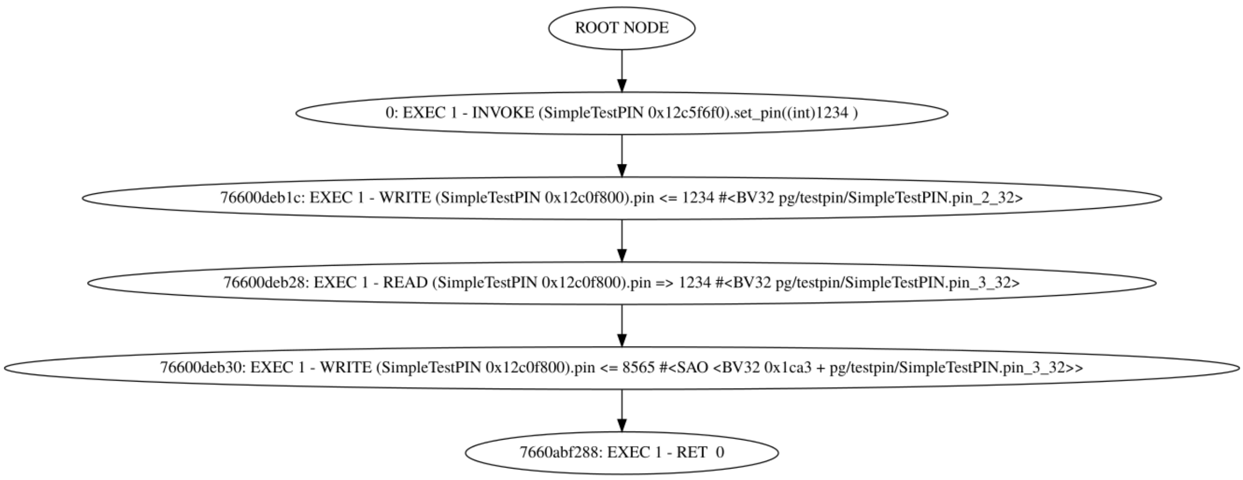


Figure 7.10: `SimpleTestPIN.set_pin` olCFG after symbolic analysis

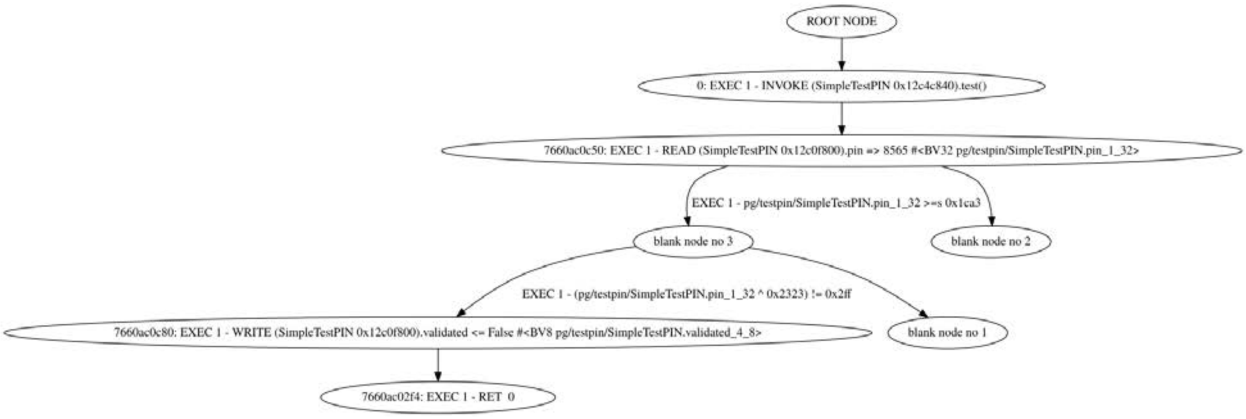


Figure 7.11: `SimpleTestPIN.test` olCFG after symbolic analysis

stored on the heap and thus are not tracked by *OATs'inside*.

After running the symbolic analysis on `SimpleTestPIN.test`, we obtained the olCFG shown in Figure 7.11. As expected, new unexplored branching path have been added. First branching condition (`pg/testpin/SimpleTestPIN .pin_1_32 >=s 0x1ca`), corresponds to the negativity check. Indeed, we saw using the olCFG of `SimpleTestPIN.set_pin` that `pin` field is added to `0x1ca`, so being superior to this value correspond to enter a positive PIN. The second branch condition (`pg/testpin/SimpleTestPIN.pin_1_32 ^ 0x2323 != 0 x2ff`) is about the `pin` field value. By combining this formula with the one of the `SimpleTestPIN.set_pin` method, we deduced that the other branch is taken if we entered `1337` as PIN value (`(0x2ff ^ 0x2323) - 0x1ca3`). Finally, we tested the `1337` value in the application and observed that the PIN is accepted. The analysis was over.

7.3.3 Final words on *OATs'inside* output

Thus, using *OATs'inside*, we have been able to easily understand the behavior of an obfuscated application. DHA, BFO and obfuscation of the native code have been handled without any effort. While the usage of native code prevent us from getting the data flow of the used variables[10], we can still conduct useful analysis.

10: stored on the stack

7.4 Performance overhead

To quantify the overhead of the Runner module, we ran an AES-128 over a 16-byte block of data using *OATs'inside* and a Sony Xperia X under AOSP Android 7.0. We used two implementations: one in full Java that stores intermediate results in Java arrays (hereinafter AES-J), and the other is a native implementation manipulating C variables (hereinafter AES-C). AES-J intensively stresses the heap, either from the interpreted version (AES-J DEX) or from the compiled version (AES-J OAT). Indeed, the runtime was dominated by heap accesses. The results are given in Table 7.4. The overhead was reasonable for the AES-J DEX implementation and non-existent for the AES-C implementation. For these versions, most of the time (around 70%) was consumed by protobuf for sending logs to the host. For AES-C, no performance overhead is observed because, as

Table 7.4: Time overhead and number of actions/events

			AES-J DEX	AES-J OAT	AES-C
Time (ms)	Bare-Metal	Total	5.8	0.199	0.007
	OATs'inside	Total	103 ×18	656 ×3,296	0.007 ×1
		Protobuf	72 70 %	481 73 %	0 0 %
		Runtime	31 30 %	175 27 %	0.007 100 %
No. of actions	*OATs'inside*	Allocation	2	1	0
		Access	5,287	11,384	0
		Methods	6,828	6,828	0
No. of signals	*OATs'inside*	SEGV	0	21,136	0
		BP	2	27,965	1

Table 7.5: Dump size depending on the APK size

APK name	Hello world	7146b3c02f0f4e3420c4471c2034de9d
APK size	174 Kb	140 Mb
Dump size	1.5 Gb	1.7 Gb
Dump time	8 sec. 687 ms	9 sec. 257 ms

expected, no events are generated. The overhead was much higher for the fully compiled version (OAT): a factor of 3,296 was observed because of the generation of the SEGV and BP events.

To quantify the evolution of the overhead depending on the number of actions, we crafted special applications that perform a fixed number of actions of type direct, breakpoint, and SEGV. The results are shown in Figure 7.12: time is in nanoseconds and is represented with a logarithmic scale. We observed that the overhead was linear with the number of actions. The highest overhead was induced by SEGV actions.

To assess the overhead of the Memory Dumper module, we dumped the contents of two applications: a simple "hello world" and the biggest ARMv8-compatible APK from AndroZoo, which was retrieved in 2019 (md5 given in Table 7.5). The results are shown in Table 7.5. The dump time and the size of the dump varied by only 12% for an application that is 1000 times bigger. Indeed, most of the memory contained libraries.

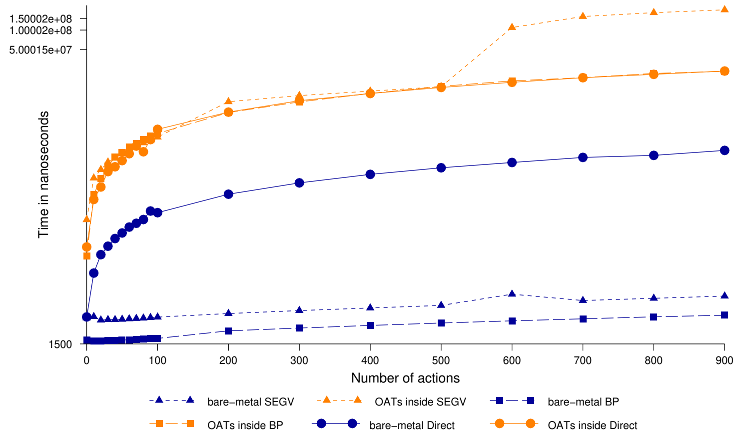


Figure 7.12: Runner module overhead with No. of actions

7.5 *OATs'inside* stealthiness

Obfuscated applications could try avoiding being analyzed [17]. Then, it is important to assess the stealthiness of *OATs'inside*, *i. e.* its capacity of not being detected. One could argue that the behavior of *OATs'inside* is fingerprintable by detecting the generation of `SEGV` and `TRAP` signals. However, these signals can never be caught by the application (*cf.* Section 8.1.5), making them stealthy.

[17]: Tam, Feizollah, Anuar, Salleh, and Cavallaro (2017), 'The evolution of android malware and android analysis techniques'

Additionally, *OATs'inside* induces a time overhead when running the application. Then, an application could fingerprint the time of the execution. A more sophisticated approach would be to measure the difference between the time spent for accessing a variable or a field. Such techniques can be defeated by hooking the syscall `gettimeofday` and changing its return value to a nominal one [80].

[80]: Xue, Luo, Yu, Wang, and Wu (2017), 'Adaptive unpacking of Android apps'

Also, a standard way to avoid being debugged is to check that no breakpoints have been set up, or that the code has not been modified by using checksums. However, *OATs'inside* does not modify the application code but rather modifies call and return addresses to redirect them to breakpoints. One could argue that an application can scan specifically these addresses, trying to detect specifically *OATs'inside*. *OATs'inside* controls the MMU and could disallow read and write accesses to the breakpoint area, and redirect the accesses to the legitimate code area [95], making them stealthy. This is left as future work.

[95]: Wong and Lie (2018), 'Tackling runtime-based obfuscation in Android with TIRO'

Finally, *OATs'inside* is not based on any emulation tool but is run on a real smartphone. Then, all the numerous techniques [47] that detect specific environments are ineffective.

[47]: Petsas, Voyatzis, Athanasopoulos, Polychronakis, and Ioannidis (2014), 'Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware'

7.6 Conclusion

This chapter has described *OATs'inside*, a tool that retrieves Java-level behaviors even if they are obfuscated with native code. By observing very finely the memory operations, *OATs'inside* catches the native code that bypasses the Java Native Interface. By combining the observations collected during a set of executions, with a concolic execution of the code currently in memory, a control flow graph of a specific method can be extracted with the conditions involved in the branching nodes. This is particularly useful for an analyst who investigates an obfuscated application.

Experiments show a high overhead. Being agnostic of the obfuscation technique and relying on the capture of reads and writes to the memory incurs a high cost. However, we put a lot of efforts into optimizing *OATs'inside* but the complexity of AOSP increases this challenge. This it not the only challenge we had to face. In the next chapter we discuss the most difficult technical issues of the implementation that we solved.

OATs'inside: implementation challenges and solutions

8

Chapter 7 presented the overall architecture of *OATs'inside* and the algorithms that we used to build it. *OATs'inside* is composed of four distinct modules: Runner, CFG Creator, Memory Dumper and Concolic Analyzer. This chapter presents the challenges that we faced during the implementation of these modules. We developed our techniques on a Sony Xperia X smartphone, running Android 7.0[1] on an ARMv8 [118] 64-bit processor. It has to be noted that the implementation could be easily ported to am ARMv7[2] or to a newer Android version[3] since the functionalities of the ART runtime that we modify and hook have not changed.

[118]: (2017), *ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile*

1: Nougat, 2016

2: 32-bit ARM architecture

3: Android runtime functionalities have not changed a lot since Android 7.0.

The Runner and the Memory Dumper are implemented inside the Android runtime, which is the `libart.so` library, and thus are installed on the smartphone. The two other modules are executed on the analysis computer and are programmed in Python.

No technical challenges regarding the approach proposed in Section 7.2.4 has been encountered during the implementation of the Concolic Analyzer module. It was developed using angr [112] as a symbolic execution engine. A dedicated angr backend has been created to load memory values from the dump file and a custom symbolic evaluation function is used to replace calls to the SMT by concretization of values, as was explained in Section 7.2.4. Thus, this chapter does not describe in more detail the implementation of the Concolic Analyzer module.

[112]: Shoshitaishvili, Wang, Salls, Stephens, Polino, Dutcher, Grosen, Feng, Hauser, Kruegel, and Vigna (2016), 'SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis'

Section 8.1 describes the challenges implied by modifying the Android runtime library to implement the Runner and the Memory Dumper modules. Section 8.2 shows algorithms involved in the CFG Creator module.

8.1 Runner and Memory Dumper modules: `libart` modifications

The ART runtime implementation is provided by AOSP, the Google open source project associated with Android. Thus the whole source code is available. However, no real documentation is provided. That is, to understand how it works, one has to directly read the source code and guess the role of the entities using their name. Also, very few comments are present in the source code. The `libart` library is mainly written in C++ and in assembly for the processor specific parts, and so are the Runner and Memory Dumper modules.

This section presents the technical challenges for developing the logging of Runner module events, as described in Section 7.2.1.

8.1.1 Analysis initialization

Symptoms: Modifications of the runtime code affects all applications and not only the one we want to observe.

Cure: Indeed, the runtime library is copied into the memory of every application. When an application is launched, a process named `zygote` is forked: this process contains all the basic libraries required to execute an application, including the runtime library `libart`. Thus, the modified version of this library is embedded into every application. If, as soon as an event is captured, it is logged, the events of every application are logged. However, to reduce the time overhead, only the events coming from the application under analysis should be reported.

Inside the runtime, an easy way to distinguish which application the library is running for is to use the process `EUID`[4]. The `EUID`, which is a number, identifies the user that runs a process. In order to sandbox applications, each Android application is run by a different user. Thus, inside `libart`, we start *OATs'inside* only when the library `EUID` matches the user corresponding to the application to be analyzed.

4: Effective User ID

8.1.2 Communication channel

Symptoms: Observed events sent from Android to *OATs'inside* on the PC side overloads the adb connection when streamed as full text.

Cure: When an event is captured, the runtime needs to send it to the analysis computer. The smartphone is connected *via* USB to the computer. Hence, we create, using `adb`[5], a reverse socket connection. Inside the runtime, information is serialized and sent using a socket connection. Using a classical socket allows further extensions of *OATs'inside*, such as remote application analysis.

5: Android Debug Bridge, a command line tool used to manage a smartphone

To reduce the amount of data send, we use protobuf[6], which helps compress the data into a binary form. Protobuf, which is developed by Google, is a language that allows to formally define data structures intended to be sent. A set of libraries for several programming languages[7] is available. Protobuf parses the protocol definition and manages all the communication and memory allocations necessary.

6: Protobuf: https://developers.google.com/protocol-buffers/

7: Java, Python, Objective-C, C++, Dart, Go, Ruby, and C#

Currently, the information is sent synchronously, that is, when an event is captured, the data is encoded and send directly. If the socket is busy, the runtime waits for logging to end before resuming the execution. We plan, as future work, to create a thread dedicated to information logging. When an event is captured, it will be written into a shared memory and the application execution will resume almost immediately. This asynchronous communication, as mentioned in Section 7.4, would improve the time overhead.

8.1.3 Methods white-listing

Symptoms: Android system native libraries generate a huge amount of events on the memory, drastically slowing down the execution.

Cure: Due to its intensive usage of signals, *OATs'inside* suffers from time overhead. To reduce this overhead, we have white-listed all Android system libraries. Indeed, their methods are well-known and common to every application and do not need to be analyzed. When a method is called, we use the `dladdr` function to retrieve the name of the library which contains it. If the name corresponds to a system library[8], the method is white-listed. We also white-list all Java libraries[9] for the same reasons.

8: Starts with "`/system/lib/`" or "`/system/lib64/`"

9: Stored in "`/system/framework/`"

When a method is white-listed, its invocations, and its returns, are still logged since they are part of the calling method's events. However, during its execution, the heap is enabled and no `SEGV` signals are generated. The heap is disabled when a non-white-listed, or "tracked", method is called, or when the white-listed method returns.

To keep track of the current protection applied to the heap, a stack (composed of integers) that represents the state of the heap, is built during the execution. This stack is maintained when *OATs'inside* hooks method invocations and returns inside `libart`. Every time a tracked method is called the number at the top of the stack is incremented. When the method returns, it is decremented. When a white-listed method is called, respectively returns, a zero is pushed onto, respectively popped from, the stack. Hence, when a zero is written on top of the stack, the heap is enabled. When a zero is overwritten, it is disabled.

8.1.4 Garbage Collector internal structures browsing and resolution caching

Symptoms: Finding the object field corresponding to a memory address is very slow, because we need to scan the full garbage collector structure.

Cure: As mentioned in Section 7.2.1, when an object field is accessed, a `SEGV` signal is generated and the garbage collector's internal structures are leveraged to map the assembly address to an object field.

Indeed, in order to being able to free objects when needed, the heap provides a visitor to walk on every allocated object. To determine the the owning object of an address, we can then walk over every object on the heap and check if the address is comprised inside the memory range of the object. When the owner object is found, we use the offset between the accessed address and the object address to determine the accessed field[10].

10: Offset of fields are saved by the runtime to allow object introspection.

While practical, this method induces a huge time overhead: for every field accessed, the whole heap is processed. To reduce this overhead, we cache the mapping between addresses and object fields. This cache is flushed when the garbage collector is triggered because it may move objects around. Using a cache length of 512 entries, we noticed more than

80% of successful cache requests, hence suggesting that the flushes are not too frequent and that caching indeed improves the performance of *OATs'inside*.

8.1.5 Signal handlers management

Symptoms: Application developers are free to overload signal handlers. As we catch the signals `TRAP` and `SEGV`, our handlers may be overwritten by the application's code.

Cure: The Runner and the Memory Dumper modules heavily rely on two signal handlers set up for the `TRAP` and `SEGV` signals. To prevent them from being replaced or removed by the application, we added a new syscall to the Linux kernel. This syscall sets up definitive signal handlers whose addresses are given in the parameter. *OATs'inside* uses it to register its own signal. The `sigaction` kernel syscall has been modified so that when the set up of a new handler is requested, the handlers set up by *OATs'inside* are kept. In order to preserve the behavior of the analyzed application, that might want to set up its own handlers, the new handler passed to `sigaction` is saved and is called whenever a generated signal is not handled by *OATs'inside*.

This implementation allows to set up transparent signals and thus, make *OATs'inside* stealthier, as mentioned in Section 7.5. Additionally, it solves practical problems due to library helpers for native development such as Google Breakpad[11] or Application Crash Reports for Android (ACRA)[12]. These libraries, that are used by many applications, set up their own handlers to show debug information when a crash occurs[13].

11: Google Breakpad: https://github.com/google/breakpad

12: ACRA: https://github.com/ACRA/acra

13: For example after a corrupt memory access that generate a `SEGV` signal.

8.1.6 Single-stepping and atomic instructions management

Symptoms: Some assembly instructions that *OATs'inside* interrupt generate an infinite loop.

Cure: As mentioned in Section 7.2.1, when an analyzed method realizes an access to a field, a `SEGV` signal is generated, the heap is enabled for a single step[14], after which the heap is disabled again. In order to realize this single-step, the next assembly instruction, that is the instruction that follows the one which realizes the access, is replaced by a breakpoint. Thus, when the execution resumes, only one instruction is executed before generating a new `TRAP` that is retrieved by *OATs'inside*. Then, the original instruction is re-written over the breakpoint and the execution can continue. This is the usual way for debuggers[15] to implement single-steps.

14: Execution of one assembly instruction.

15: Such as `gdb`

While this implementation is fully practical with most ARMv8 assembly instructions, it breaks atomicity properties. Indeed, ARMv8 instruction set contains twin instructions: `ldx`, for LoaD eXclusive, and `stx`, for STore eXclusive[16]. The semantics of these instructions are the following: an `stx` instruction succeeds only if no other process or thread has performed a

16: These instructions are defined for several read/write sizes.

more recent store to the address that has been previously read using an `ldx` [118].

[118]: (2017), *ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile*

However, when running a method analyzed by *OATs'inside*, if an address stored by an `stx` instruction is located inside the heap, the store generates a `SEGV`. Then, *OATs'inside* logs the value contained at that address before the store instruction occurs. This breaks the atomicity property and the `stx` fails when single-stepping. Since these instructions are used to create mutexes, this creates a deadlock: the application retries infinitely to perform the `ldx` and then the `stx`. These kinds of operations are used in particular to implement the `synchronized` Java keyword. This keyword indicates that a portion of the code cannot be run concurrently. An associated mutex is in fact stored as a field inside the object that is referred to by the `synchronized` keyword. This is exactly the problematic case we described previously.

To overcome this limitation, *OATs'inside* emulates the semantics of `ldx` and `stx` instructions. When a `SEGV` signal occurs, *OATs'inside* checks if the faulty instruction is a `ldx` or `stx` instruction. If it is a `ldx`, *OATs'inside* saves that the current thread held the faulty address. If it is a `stx`, *OATs'inside* checks if the thread holds the address. If yes, the `stx` is replaced by a classical store instruction and is single-stepped. After the step, the original `stx` is re-written. If the thread does not hold the address, the `stx` is single-stepped. The store fails, which is the correct semantics. We have not encountered other problematic instructions, but in such cases, the same resolution principle could be used: emulating the semantics of the instruction.

8.1.7 Multi-thread management

Symptoms: An Android application contains always more than 7 threads. Monitoring and interrupting threads other than the one under analysis is useless and slow down the execution.

Cure: As stated in Section 7.2.1, the Runner module needs a thread-oriented `mprotect`. In a vanilla Linux kernel, all threads of the same process share the same address space and thus, the same write protections on their memory pages. However, we want *OATs'inside* to be able to disable or enable the heap on a per-thread basis. Hence, we have added to the kernel the possibility for processes to have two address spaces with different write protections and to switch between them, following the approach in [115]. In one of the address spaces, the heap is enabled, while in the other, the heap is disabled. This allows to disable or enable the heap for specific threads. Additionally, this speeds up the process of enabling or disabling the heap: *OATs'inside* no longer has to walk over all the heap pages and change their protections but rather only change the address space pointer.

[115]: Razeen, Lebeck, Liu, Meijer, Pistol, and Cox (2018), 'SandTrap: Tracking Information Flows On Demand with Parallel Permissions'

8.2 CFG Creator module: NetworkX implementation

Symptoms: When detecting a conditional jump, the graph should contain two branches, but only one is executed and we may miss the alternative node.

Cure: The CFG Creator module runs on the analysis computer. It is in charge of creating the iCFG (interprocedural CFG) and the olCFGs (object-level CFGs). It implements, in Python, the algorithms described in Sections 7.2.2 and 7.2.4, using the NetworkX library[17] to perform graph operations efficiently.

17: https://networkx.github.io/

First, it creates the iCFG by splitting the events between methods, as shown in Algorithm 8.1. Since events that occur in different threads might be mixed inside the list of events, the structures (lines 2, 3 and 4) are dictionaries indexed by the TID[18] of the events. By tracking the current method that is executed (line 11), the call-stack is built (lines 10 and 14). During this operation, the iCFG is built incrementally (line 12).

18: Thread ID

Algorithm 8.1: Create an iCFG and separate events between methods.

Input: events
Output: icfg, methods_events

```
1:  icfg ⟵ EmptyDirectedGraph()
2:  call_stack ⟵ {}
3:  current_method ⟵ {}
4:  methods_events ⟵ {}
5:  for all event ∈ events do
6:    tid ⟵ event.tid
7:    methods_events[tid][current_method[tid]].append(event)
8:    if event is "invoke" then
9:      last_method[tid] ⟵ current_method[tid]
10:     call_stack[tid].push(current_method[tid])
11:     current_method[tid] ⟵ event.method
12:     icfg.add_edge( (last_method[tid], current_method[tid]) )
13:   else if event is "return" then
14:     current_method[tid] ⟵ call_stack[tid].pop()
15:   end if
16: end for
```

Second, it creates the olCFG of the method, as shown in Algorithm 8.2. It assumes that actions performed at a specific address originate from the same instruction (line 14). While processing the events, the graph edges are built (line 17). The events retrieved for a given method might describe several executions of this method, the invocations and returns are tracked (line 39) to determine when the next event is not linked with the current one but instead is a new execution of the method (line 45). Then, the olCFG is built from the dummy `ROOT_NODE` (line 46).

Special care is given to condition events (line 20). Indeed, as stated in Section 7.2.4, when a conditional jump occurs, two "blank" nodes are added (lines 25 to 31) to keep the information that a conditional path has been taken. These blank nodes are then removed, according to Algorithm 8.3, when the blank node follows an event and is followed by an other event. Indeed, in this case, the path taken is already represented in the graph by the edge between the two events. The blank nodes that finally remains are the ones that have no successors, that is a path that has not been taken, or the ones that follows an other blank node, that is two conditions that have occurred successively.

Algorithm 8.2: Create olCFG for a given method.

```
Input: method_events
Output: olCFG
olCFG ← EmptyDirectedGraph()
number_of_invoke_without_return ← 0
execution_number ← 1
olCFG.add_node(ROOT_NODE)
previous_node ← ROOT_NODE

// Create olCFG with blank nodes
for all event ∈ method_events do
   addr ← event.event_address

   // General case
   if event is not "condition" then
      // Update the node label
      olCFG.nodes(addr).append( (execution_number, event) )

      // Link the previous event with the current one
      olCFG.add_edge(previous_node, addr)
      previous_node ← addr

   // Handle blank nodes for condition events
   else
      // Create or retrieve blank nodes if they already exist
      cond_value ← event.condition_value
      target ← event.target
      if olCFG.blank_node_exists(target) then
         blank_node_1 ← olCFG.blank_node(target)
         blank_node_2 ← olCFG.blank_node(-target)
      else
         blank_node_1 ← olCFG.create_blank_node(target)
         blank_node_2 ← olCFG.create_blank_node(-target)
      end if

      // Update graph with blank nodes
      blank_node_1.append((execution_number, event))
      olCFG.add_edge(previous_node, blank_node_1)
      olCFG.add_edge(previous_node, blank_node_2)
   end if

   // Track when the last return of the method is reached
   if event is "invoke" then
      number_of_invoke_without_return += 1
   else if event is "return" then
      number_of_invoke_without_return -= 1
   end if
   if number_of_invoke_without_return == 0 then
      previous_node ← ROOT_NODE
      execution_number += 1
   end if
end for
```

Algorithm 8.3: Remove useless blank nodes from an olCFG.

Input: olCFG
Output: olCFG

```
for all node ∈ olCFG.nodes do
    if node.is_blank_node() then
        if len(node.next_nodes) == 1 then
            next_node = node.next_nodes[0]
            if not next_node.is_blank_node() then
                for all previous_node in node.previous_nodes do
                    olCFG.add_edge(previous_node, next_node)
                end for
                olCFG.remove(node)
            end if
        end if
    end if
end for
```

8.3 Conclusion

A lot of challenges have been faced during the implementation of *OATs'inside*. It shows that modifying the Android kernel and the Android runtime is not a straightforward task. Nevertheless, the developed patches are placed in components that should be relatively stable in future versions, according to the modifications that have already occurred in past AOSP releases. Thus, porting *OATs'inside* for new version of Android is possible, unless some major changes are brought by Android developers.

Epilogue

9 Conclusion

9.1 Thesis contribution summary

This thesis has presented the following contributions.

First, we have introduced Bytecode Free OAT (BFO) and Direct Heap Access (DHA), two new obfuscation techniques that are fully applicable to Android native applications. We have proposed, tested and classified several implementations in order to evaluate their practicality. These techniques, until now, were not known by the scientific community and thus are able to bypass state-of-the-art tools.

Since their usage in the wild would be dangerous, we developed corresponding detection techniques. Then, we used them to search for obfuscated applications inside application stores and smartphone firmwares. Results are lukewarm: while we are able to detect obvious obfuscation techniques, such as nopping or deleting the bytecode, the detection of slighter modifications in the code remains an open problem.

We have detected a lot of DHA usage. These usages were legitimate and driven by optimization goals. Thus, we are confident that we would also be able to detect malicious usages of DHA. Then, appears a new problem that is telling malicious DHA apart from benign ones.

Regarding the vulnerability issues, we have developed a tool that is able to detect missing `transient` keywords. We have confirmed automatically the manual results of Peles *et. al.* on the `conscrypt` library. The proposed technique suffers for usual static analysis limitations. These limitations are classically treated by asking the developer to annotate application code, which is possible since the method works on the source code. Our study of the Telegram application has shown the usability of the tool: several flows were found, although none are directly exploitable.

Additionally, even though the proposed method has targeted the specific `transient` keyword problem, it could be applied to more general security issues. For example, leakage of file descriptor numbers to external sockets could be treated similarly. Similarly to DHA detection, the difficulty now resides in distinguishing legitimate from malicious leakage.

Finally, we synthesize the knowledge obtained about Android obfuscation to build *OATs'inside*, a new tool that is independent from potential obfuscation techniques used by analyzed applications. *OATs'inside* combines dynamic and symbolic analysis to retrieve the object-level behavior of obfuscated Android applications. *OATs'inside* outputs an object-level CFG that contains instructions acting on objects such as calling methods or setting object fields, even when these actions are performed by native code. It describes the contents of each method, the conditional expressions involved in the control flow instructions, the data flow between actions, and the interprocedural calls. This information is particularly useful for an analyst who studies a particular obfuscated method. This highly precise analysis is very costly in terms of time overhead. This is the consequence of the very fine granularity of observation. Since we detect instruction-level behaviors, we are forced to stop the execution during the analysis of the involved instruction. The more details we observe, the more the execution is slowed down. We believe that this scientific obstacle cannot be solved at the software level but requires modifications of the underlying hardware.

9.2 Perspectives for future work

We suggest three axes for extending this thesis work.

First, we believe that this thesis work could be applied to other system platforms. Indeed, the Android ecosystem is very wide: connected cars[1], smartwatches[2], TVs[3] and various connected devices[4]. All these platforms have their own specificities but rely on the same Android AOSP core. Thus, the work presented in this thesis, in particular *OATs'inside*, would be usable in these new contexts. These specificities may modify malicious intents: for example, there is no gain in crafting a ransomware for a smart watch that has no personal data. Consequently, we believe that our work should be adapted to different malicious intents.

1: Android Automotive
2: Android Wear
3: Android TV
4: Android Things

Since the devices are usually closed-source, device vendors pre-install a lot of applications and sometimes force users to use them. For example, in modern televisions, a well-known streaming platform[5] is pre-installed due to commercial agreements. The need for protection for these applications would push vendors to use security techniques such as obfuscations and vulnerability detection. This highlights that adapting our work to these new contexts is urgent.

5: Netflix

Second, we suggest that this thesis work could be applied to other languages. Indeed, Android is not the only system that allows developers to mix one high-level language with a lower one. For example, the reference Python interpreter, named CPython, allows to extend Python scripts with assembly code and provides an interface to help Python and C/C++ languages to operate together. Similarly to Android, CPython does not enforce the usage of this interface and let assembly interact freely with bytecode. Obviously, tools' implementations presented in this thesis are not directly usable in this context. We could evaluate if proposed techniques can be adapted to the target context.

Third, we believe that Android should be modified in order to harden the interface between bytecode and native code. Indeed, the challenges described during this thesis are the consequences of an ill-defined and too permissive interface between these two languages. Rebuilding a new interface from scratch, while keeping in mind the issues described in this thesis, will remove these challenges.

We could draw inspiration from web browser implementations of Javascript and WebAssembly. These languages are used for creating interactive web pages. Javascript is a high-level language that is executed, inside the web browser, by a virtual machine. WebAssembly is a low-level language that looks like assembly code, and is used for optimization purposes. Similarly to Android, the web browser virtual machine offers an interface that allows these languages to interact together. However, WebAssembly is run inside a sandbox that is instantiated and controlled by the Javascript code. For example, Javascript code uses the API of the interface to define memory areas that are accessible by WebAssembly. Nevertheless, modifying Android in a such way will lead to backward compatibility issues. Thus, we should design a solution that both defines a new interface and allows to port current applications to this new system transparently for application developers. This solution, even if incompatible with existing applications, should be more restrictive while allowing legitimate usage of native code.

Appendices

A List of tested firmwares

Table A.1 lists all the 17 firmwares that constitute the dataset on which we conducted experiments in Section 5.2. These firmwares have been retrieved on `https://androidmtk.com`, a web site that provides firmwares and drivers for more than fifty brands.

We have chosen 6 brands among the most commonly distributed brands and downloaded the firmwares of all their devices. We have limited our dataset to one firmware by device and kept only firmwares that run Android Nougat (7.x), the Android version targeted by *OATs'inside* [1].

1: Solution presented in Chapter 7

Table A.1: List of tested firmwares

Brand	Phone model	Android version
Alcatel	1T 10	7.0
	OneTouch A3 Plus 5011A	7.0
Archos	50f Neon	7.0
Huawei	Ascend Mate 9 MHA-AL00	7.0
	Enjoy 7 Plus TRT-TL10A	7.0
	P10 VRT-AL00	7.0
Samsung	Galaxy A3 SM-A310M	7.0
	Galaxy C7 Pro SM-C710F	7.1.1
	Galaxy Note 5 SM-N920A	7.0
	Galaxy S6 Edge SM-G925S	7.0
	Galaxy A5 SM-A510M	7.1.1
Sony Xperia	Touch G1109	7.0
	L1 Dual G3312	7.1.1
	M2 Aqua D2403	7.0
	Z5 Premium E6853	7.0
	Z5 501SO	7.1.1
Wiko	Jerry 2	7.0

B Java behavior unit tests

This appendix presents unit tests used in Sections 7.1 and 7.2.5. Section B.1 describes the unobfuscated versions of the unit tests that is the *DEX only* and the *JNI* versions. Section 2 describes the obfuscation techniques used to generate the four other versions: *Pack DEX*, *Bytecode Free OAT (BFO)*, *JNI+obf*, *Direct Heap Access (DHA)*.

B.1 unobfuscated unit tests

Unit tests are gather in two applications. One has implemented the different test in Java, the other in C++. The *DEX only* and the *JNI* versions correspond to a vanilla compilation of the code presented here, using Android studio. Each application is composed of a single activity. This activity class contains a method for each test case and seven fields, one for each primitive type[1]. Test cases (*i.e. methods*) are grouped in eight families, each family standing for a Java behavior. All tests are launched when the activity is created, inside `onCreate` method.

1: `char`, `long`, `byte`, `boolean`, `double` and `float`

In order to help the writing of JNI test cases, two helpers are available in the *JNI* test case application. They are shown in Listing B.1 and allow to easily retrieve the JNI ID of a class or of the primitive fields.

Method behaviour This family is divided in two categories: invoke and return. For each method we verify that the tested tool correctly reports the invocation and the return behavior with the types and the values of the method arguments and return value. Two other marginal tests have been added: one that checks invocations using numerous arguments are also handle[2] and one that checks multi-level of invocations are all handled correctly.

2: Invocations with numerous arguments, in common Application Binary Interface (ABI), are handled differently than “small” invocations.

Allocation behaviour This family is divided in three categories, whether an object, a primitive variable or primitive array variable is allocated. For each method, we verify that the tested tool correctly reports that an entity has been allocated (on the heap for object and primitive array variables, on the stack for primitive variables).

```
#define GET_CLS jclass cls = env->GetObjectClass(instance)

/* Usage: GET_FIELD(int, Int, "I"); for getting intField, GET_FIELD(char, Char, "C"); for getting charField, ...
     */
#define GET_FIELD(type, Type, sig) \
    jfieldID fid = env->GetFieldID(cls, #type "Field", sig); \
    j##type type##Field = env->Get##Type##Field(instance, fid);
```

Listing B.1: JNI test cases helpers

```java
public int testInvokeReturnInt(int intVar) { return intVar; }
public char testInvokeReturnChar(char charVar) { return charVar; }
[...] /* long, byte, boolean, double and float cases removed from snippet */
public int testInvokeManyIntArgs(int arg1,int arg2,int arg3,int arg4,int arg5,int arg6,int arg7,int arg8,int arg9,
    int arg10,int arg11) { return arg1+arg2+arg3+arg4+arg5+arg6+arg7+arg8+arg9+arg10+arg11;}
public long testSubInvokeReturn() { return testInvokeReturnInt(1) + testInvokeReturnLong(2) + testInvokeReturnByte
    ((byte)3);}
```

Listing B.2: Java method test cases

```cpp
extern "C" JNIEXPORT jint JNICALL testInvokeReturnInt(JNIEnv *env, jobject instance, jint intVar) { return intVar;
     }
[...] /* char, long, byte, boolean, double and float cases removed from snippet */
extern "C" JNIEXPORT jint JNICALL testInvokeManyIntArgs(JNIEnv *env, jobject instance, jint arg1, jint arg2, jint
    arg3, jint arg4, jint arg5, jint arg6, jint arg7, jint arg8, jint arg9, jint arg10, jint arg11) { return
    arg1+arg2+arg3+arg4+arg5+arg6+arg7+arg8+arg9+arg10+arg10+arg11;}
extern "C" JNIEXPORT jlong JNICALL testSubInvokeReturn(JNIEnv *env, jobject instance) { GET_CLS; jmethodID mid;
    jlong res = 0;
  mid = env->GetMethodID(cls, "testInvokeReturnInt", "(I)I");
  res += env->CallIntMethod(instance, mid, (jint)1);
  mid = env->GetMethodID(cls, "testInvokeReturnLong", "(J)J");
  res += env->CallLongMethod(instance, mid, (jlong)1);
  mid = env->GetMethodID(cls, "testInvokeReturnByte", "(B)B");
  res += env->CallByteMethod(instance, mid, (jbyte)3); return res; }
```

Listing B.3: JNI method test cases

Access behaviour This family is divided in three categories, whether the access is performed on an object field, a primitive variable or primitive array variable. For each method, we verify that the tested tool correctly reports that a read-access and a write-access has been performed and reports the read value, the written value and the overwritten value. The variables are not local to the method to avoid aggressive optimizations from the compiler.

Operations behaviour This family is divided in three categories, whether the operation uses an object field, a primitive variable or primitive array variable. For each method, we verify that the tested tool correctly reports that an operation has been conducted and reports the formula corresponding to the operation. The variables are not local to the method to avoid aggressive optimizations from the compiler.

Condition behaviour This family is divided in three categories, whether the condition depends on an object field, a primitive variable or primitive array variable. For each method, we verify that the tested tool correctly reports that a conditional path has been taken and reports the formula

```java
/* Object */
public class AllocatedClass {int varI; char varC; long varL; byte varByte; boolean varB; double varD; float varF;}
public AllocatedClass testAllocateObject() {return new AllocatedClass();}
/* Primitive variable */
public void testAllocateVariable() {int varI; char varC; long varL; byte varByte; boolean varB; double varD; float
     varF;}
/* Primitive Array */
public void testAllocateArray() {int[] varI={}; char[] varC={}; long[] varL={}; byte[] varByte={}; boolean[] varB
    ={}; double[] varD={}; float[] varF={};}
```

Listing B.4: Java allocation test cases

```
/* Object field */
extern "C" JNIEXPORT jobject JNICALL testAllocateObject(JNIEnv *env, jobject instance) {
  jclass cls = env->FindClass("pg/nativetests/TestLauncher$AllocatedClass"); return env->AllocObject(cls);}
/* Primitive variable */
extern "C" JNIEXPORT void JNICALL testAllocateVariable(JNIEnv *env, jobject instance) {
  jint varI; jchar varC; jlong varL; jbyte varByte; jboolean varB; jdouble varD; jfloat varF;}
/* Primitive Array */
extern "C" JNIEXPORT void JNICALL testAllocateArray(JNIEnv *env, jobject instance) {
  jintArray varI = env->NewIntArray(1);jcharArray varC = env->NewCharArray(1);jlongArray varL = env->NewLongArray
     (1);jbyteArray varByte = env->NewByteArray(1); jbooleanArray varB = env->NewBooleanArray(1);jdoubleArray varD
      = env->NewDoubleArray(1);jfloatArray varF = env->NewFloatArray(1);}
```

Listing B.5: JNI allocation test cases

```
/* Object field */
public int testAccessObjectInt() { int tmpInt; tmpInt=intField;intField=3000;return tmpInt;}
public char testAccessObjectChar() {char tmpChar; tmpChar=charField;charField='o';return tmpChar;}
[...] /* long, byte, boolean, double and float cases removed from snippet */
/* Primitive variable */
public int testAccessVariableInt(int intVar) { int tmpInt; tmpInt=intVar;intVar=3000;return tmpInt;}
public char testAccessVariableChar(char charVar) {char tmpChar; tmpChar=charVar;charVar='o';return tmpChar;}
[...] /* long, byte, boolean, double and float cases removed from snippet */
/* Primitive Array */
public int testAccessArrInt(int[] intVar) { int tmpInt; tmpInt=intVar[0];intVar[0]=3000;return tmpInt;}
public char testAccessArrChar(char[] charVar) {char tmpChar; tmpChar=charVar[0];charVar[0]='o';return tmpChar;}
[...] /* long, byte, boolean, double and float cases removed from snippet */
```

Listing B.6: Java access test cases

```
/* Object field */
extern "C" JNIEXPORT jint JNICALL testAccessObjectInt(JNIEnv *env, jobject instance) {
  GET_CLS;GET_FIELD(int, Int, "I");jint tmp = intField;env->SetIntField(instance, fid, 3000);return tmp;}
extern "C" JNIEXPORT jchar JNICALL testAccessObjectChar(JNIEnv *env, jobject instance) {
  GET_CLS;GET_FIELD(char, Char, "C");jchar tmp = charField;env->SetCharField(instance, fid, 'o');return tmp;}
[...] /* long, byte, boolean, double and float cases removed from snippet */
/* Primitive variable */
extern "C" JNIEXPORT jint JNICALL testAccessVariableInt(JNIEnv *env, jobject instance, jint intVar) {
  jint tmp=intVar; intVar=3000;return tmp;}
extern "C" JNIEXPORT jchar JNICALL testAccessVariableChar(JNIEnv *env, jobject instance, jchar charVar) {
  jchar tmp=charVar; charVar=3000;return tmp;}
[...] /* long, byte, boolean, double and float cases removed from snippet */
/* Primitive Array */
extern "C" JNIEXPORT jint JNICALL testAccessArrInt(JNIEnv *env, jobject instance, jintArray intVar_) {
  jint *intVar = env->GetIntArrayElements(intVar_, NULL);jint tmp = intVar[0];intVar[0] = 3000;env->
     ReleaseIntArrayElements(intVar_, intVar, 0);return tmp;}
extern "C" JNIEXPORT jchar JNICALL testAccessArrChar(JNIEnv *env, jobject instance, jcharArray charVar_) {
  jchar *charVar = env->GetCharArrayElements(charVar_, NULL);jchar tmp = charVar[0];env->ReleaseCharArrayElements(
     charVar_, charVar, 0);return tmp;}
[...] /* long, byte, boolean, double and float cases removed from snippet */
```

Listing B.7: JNI access test cases

```
/* Object field */
public int testOperationsObjectInt() {return intField + 1;}
public char testOperationsObjectChar() {return (char)((int)(charField) + 2);}
[...] /* long, byte, boolean, double and float cases removed from snippet */
/* Primitive variable */
public int testOperationsVariableInt(int intVar) { return intVar + 1; }
public char testOperationsVariableChar(char charVar) { return (char)((int)(charVar)+2); }
[...] /* long, byte, boolean, double and float cases removed from snippet */
/* Primitive Array */
public int testOperationsArrInt(int[] intVar) { return intVar[0] + 1; }
public char testOperationsArrChar(char[] charVar) { return (char)((int)(charVar[0])+2); }
[...] /* long, byte, boolean, double and float cases removed from snippet */
```

Listing B.8: Java operations test cases

```
/* Object field */
extern "C" JNIEXPORT jint JNICALL testOperationsObjectInt(JNIEnv *env, jobject instance) {
  GET_CLS;GET_FIELD(int,Int,"I");return intField+1; }
extern "C" JNIEXPORT jchar JNICALL testOperationsObjectChar(JNIEnv *env, jobject instance) {
  GET_CLS;GET_FIELD(char,Char,"C");return charField+2; }
[...] /* long, byte, boolean, double and float cases removed from snippet */
/* Primitive variable */
extern "C" JNIEXPORT jint JNICALL testOperationsVariableInt(JNIEnv *env, jobject instance, jint intVar) {return
    intVar+1;}
extern "C" JNIEXPORT jchar JNICALL testOperationsVariableChar(JNIEnv *env, jobject instance, jchar charVar) {
    return charVar+2;}
[...] /* long, byte, boolean, double and float cases removed from snippet */
/* Primitive Array */
extern "C" JNIEXPORT jint JNICALL testOperationsArrInt(JNIEnv *env, jobject instance, jintArray intVar_) {
  jint *intVar = env->GetIntArrayElements(intVar_, NULL);jint res = intVar[0];env->ReleaseIntArrayElements(intVar_
    , intVar, 0);return res+1;}
extern "C" JNIEXPORT jchar JNICALL testOperationsArrChar(JNIEnv *env, jobject instance, jcharArray charVar_) {
  jchar *charVar = env->GetCharArrayElements(charVar_, NULL);jchar res = charVar[0];env->ReleaseCharArrayElements(
    charVar_, charVar, 0);return res+2;}
[...] /* long, byte, boolean, double and float cases removed from snippet */
```

Listing B.9: JNI operations test cases

```
/* Object field */
public boolean testConditionObjectEq() {if( intField == 42) return true; else return false; }
public boolean testConditionObjectInfEq() { if( intField <= 42) return true; else return false; }
public boolean testConditionObjectSup() { if( intField > 42) return true; else return false; }
/* Primitive variable */
public boolean testConditionVariableEq(int i) {if( i == 42) return true; else return false; }
public boolean testConditionVariableInfEq(int i) { if( i <= 42) return true; else return false; }
public boolean testConditionVariableSup(int i) { if( i > 42) return true; else return false; }
/* Primitive Array */
public boolean testConditionArrEq(int[] i) {if( i[0] == 42) return true; else return false; }
public boolean testConditionArrInfEq(int[] i) { if( i[0] <= 42) return true; else return false; }
public boolean testConditionArrSup(int[] i) { if( i[0] > 42) return true; else return false; }
```

Listing B.10: Java condition test cases

corresponding to the condition taken. The variables are not local to the method to avoid aggressive optimizations from the compiler.

Typing behaviour This family is divided in two categories, whether argument type is checked or the argument is cast. For each method, we verify that the tested tool correctly reports the check or the cast and gives the types used.

Exception behaviour This family is divided in two categories, whether an exception is raised or caught. For each method, we verify that the tested tool correctly reports the exception. A supplementary test has been added to test if inner method calls that leave exceptions are also handled.

Monitor behaviour In this family, we verify that the tested tool correctly reports the beginning and the ending of a Java monitored session.

```
/* Primitive variable */
extern "C" JNIEXPORT jboolean JNICALL testConditionVariableEq(JNIEnv *env, jobject instance, jint i) { if( i == 42
      ) return true; else return false; }
[...] /* <= and > removed from snippet */
/* Primitive Array */
extern "C" JNIEXPORT jboolean JNICALL testConditionArrEq(JNIEnv *env, jobject instance, jintArray i_) {
  jint *i = env->GetIntArrayElements(i_, NULL);
  if( i[0] == 42 ) {env->ReleaseIntArrayElements(i_, i, 0);return true;}
  else {env->ReleaseIntArrayElements(i_, i, 0);return false;}}
[...] /* <= and > removed from snippet */
/* Object field */
extern "C" JNIEXPORT jboolean JNICALL testConditionObjectEq(JNIEnv *env, jobject instance) {
  GET_CLS; GET_FIELD(int, Int, "I"); if(intField == 42 ) return true; else return false;}
[...] /* <= and > removed from snippet */
```

Listing B.11: JNI condition test cases

```
public class Toto {}
public class Tata extends Toto {}
public class Tutu extends Toto {}
public boolean testCheckType(Toto t) {return t instanceof Tata;}
public Tata testCastType(Toto t) {return (Tata)t;}
```

Listing B.12: Java typing test cases

B.2 Obfuscated unit tests

Packer The *Pack DEX* version is built from the *DEX only* version. We employed the home-made packer[3] described in [5]:

1. The *DEX only* application is copied. The copied version, once building, constitutes the *Pack DEX* version.
2. The DEX file of the original *DEX only* application is extracted.
3. The extracted DEX is xored with an hardcoded key (`0x42`) and is stored in the resource folder of the copied application using an other name (`butterfly.png`).
4. The DEX file of the copied application is noped by replacing all the DEX instructions by `const/4 v1, 0x1` instruction.
5. A native library is added to the copied DEX file. Before calling each test method, the `decodeMethod` of this library is called to unpack the test. This method, see Listing B.19:
   a) retrieves the location of the DEX file by browsing the `/proc/self/maps` file, see Listing B.18.
   b) retrieves the address (`APK_insns_`) and the length (`APK_insns_size_in_code_units_`) of the noped method by parsing the DEX file.
   c) loads the xored file (`butterfly.png`) and retrieves the address (`PNG_insns_size`) of the xored method by browsing the DEX file structure.
   d) un-xors the xored method and writes the results over the

3: https://gitlab.inria.fr/jlalande/teaching-android-mobile-security

```
extern "C" JNIEXPORT jboolean JNICALL testCheckType(JNIEnv *env, jobject instance, jobject t) {
  jclass cls = env->FindClass("pg/nativetests/TestLauncher$Tata"); return env->IsInstanceOf(t, cls);}

extern "C" JNIEXPORT jobject JNICALL testCastType(JNIEnv *env, jobject instance, jobject t) {return t;}
```

Listing B.13: JNI typing test cases

```java
public void testThrow() { throw new IllegalArgumentException("Testing throw"); }
public void testNoCatch() { testThrow(); }
public void testThrowCatch() { try { testNoCatch(); } catch (Exception e) {} }
```

Listing B.14: Java exception test cases

```cpp
extern "C" JNIEXPORT void JNICALL testThrow(JNIEnv *env, jobject instance) {
  jclass cls = env->FindClass("java/lang/IllegalArgumentException");env->ThrowNew(cls, "Testing throw");}
extern "C" JNIEXPORT void JNICALL testNoCatch(JNIEnv *env, jobject instance) {
  GET_CLS;jmethodID mid= env->GetMethodID(cls, "testThrow", "()V");env->CallVoidMethod(instance, mid, (jint)1);}
extern "C" JNIEXPORT void JNICALL testThrowCatch(JNIEnv *env, jobject instance) {
  GET_CLS;jmethodID mid= env->GetMethodID(cls, "testNoCatch", "()V");env->CallVoidMethod(instance, mid, (jint)1);
     if(env->ExceptionCheck() == JNI_TRUE)env->ExceptionClear();}
```

Listing B.15: JNI exception test cases

noped-method bytecode.

BFO The *BFO* version is built from the *DEX only* version. The *DEX only* application is installed on the test-smartphone (or the test-emulator). Then, all the methods of the application are compiled into OAT using the command presented in Listing B.20. This command is run on the test-smartphone. Finally, the DEX file of the application and the DEX file of the OAT file are noped on the smartphone. They are located inside the `/data/app/unit_tests_package/` folder on the test-smartphone. This folder constitutes the *BFO* version.

Listing B.20: Application compilation command

```
cmd package compile -m speed -f unit_tests_package
```

Native obfuscation The *JNI+obf* version is built from the *JNI* version. Instead of using the classical Android studio compiler (clang), the IDE is set-up to use Obfuscator-LLVM [111]. The parameters passed to Obfuscator-LLVM are shown in Listing B.21. These options activate opaque predicate usage (`-bcf -mllvm -bcf_prob=100`) and control flow flattening (`-mllvm -split -mllvm -fla`)

Listing B.21: Obfuscator-LLVM command line

```
-mllvm -bcf -mllvm -bcf_prob=100 -mllvm -split -mllvm -fla
```

To illustrate the effect of Obfuscator-LLVM, we show here the resulting Control Flow Graphs[4] of `testConditionObjectEq` method. Figure B.1 shows the unobfuscated CFG (*JNI* version). Figure B.2a, resp. Figure B.2b, shows the CFG of `testConditionObjectEq` method when opaque predicates, resp. control flow flattening, are applied to `testConditionObjectEq`.

4: Retrieved using IDA, `https://www.hex-rays.com/products/ida/`.

DHA The *DHA* version is built from the *JNI* version. In fact, DHA obfuscation only applies when an object field or a primitive array[5] is read or written. This happens only for the access, operation and condition behaviors. For these behaviors the read and the write operations that are performed using JNI, are replaced by a direct memory access, see Listing B.22.

5: Primitive arrays are also stored on the heap

```java
public void testMonitor(int i) {synchronized(this){try{Thread.sleep(i);} catch(InterruptedException e){}}}
```

Listing B.16: monitor test cases

```c
extern "C" JNIEXPORT void JNICALL testMonitor(JNIEnv *env, jobject instance, jint i) {
  env->MonitorEnter(instance);

  jclass cls = env->FindClass("java/lang/Thread");
  jmethodID mid = env->GetStaticMethodID(cls, "sleep", "(J)V");
  env->CallStaticVoidMethod(cls, mid, i);

  if(env->ExceptionCheck() == JNI_TRUE)
      env->ExceptionClear();

  env->MonitorExit(instance);
}
```

Listing B.17: JNI exception test cases

```c
void* getDexFileLocation() {
  FILE* fichier;
  fichier = fopen("/proc/self/maps", "r");
  if(fichier != NULL) {
    char* line = NULL;
    size_t n = 0;
    ssize_t nb_read = 0;

    while((nb_read = getline(&line, &n, fichier)) > 0) {
      if(nb_read > 6) {
        if (line[nb_read - 6] == '.' && line[nb_read - 5] == 'o' && line[nb_read - 4] == 'd' && line[nb_read - 3] == 'e' && line[nb_read - 2] == 'x') {
          fclose(fichier);

          void* oat_addr = (void *) strtoll(line, NULL, 16);

          line[nb_read-1] = '\0'; // Remove ending '\n'
          char* oat_location = strchr(line, '/');

          /* Already loaded so it only retrieves the handle */
          void* oat_dl_handle = dlopen(oat_location, RTLD_LAZY);
          void* oatdata_dl_addr = dlsym(oat_dl_handle, "oatdata");
          Dl_info info; dladdr(oatdata_dl_addr, &info);
          unsigned long oatdata_offset = (unsigned long)oatdata_dl_addr - (unsigned long)info.dli_fbase;
          dlclose(oat_dl_handle);

          void* oatdata_addr = (void*)((char*)oat_addr + oatdata_offset);

          unsigned int dex_file_count = *(unsigned int*)((char*)oatdata_addr + 20);
          unsigned int key_value_store_size = *(unsigned int*)((char*)oatdata_addr + 68);
          unsigned int oat_header_size = 72 + key_value_store_size;

          /* We only read the first \gls{dexL} */
          void * oat_dex_header = (char*)oatdata_addr + oat_header_size;
          unsigned int dex_file_location_size = *(unsigned int*)oat_dex_header;
          unsigned int dex_file_pointer = *(unsigned int*)((char*)oat_dex_header + 8 + dex_file_location_size);

          return (void*)((char*)oatdata_addr + dex_file_pointer);
        }
      }
    }
    fclose(fichier);
  }
  return NULL;
}
```

Listing B.18: DEX file location

```
/* Constants than can move between different libart \gls{runtimeL} */
#define OFFSET_OF_CODE_ITEM_OFFSET_ART_METHOD 8
unsigned int getCodeItemOffset(JNIEnv* env, jclass thisClass, const char* methodName, const char* methodSignature) {
    void* art_method = (void*) env->GetMethodID(thisClass, methodName, methodSignature);
    unsigned int code_item_offset = *(unsigned int*) ((char*)art_method + OFFSET_OF_CODE_ITEM_OFFSET_ART_METHOD);
    return code_item_offset;
}

void* GetCodeItemInstructions(const void* dex_addr, unsigned int code_item_offset, unsigned int* code_size /* out */) {
    void* code_item = (void*)((char*)dex_addr + code_item_offset);
    *code_size = *(unsigned int*) ((char*)code_item + 12);
    void* insns_ = (void*) ((char*)code_item + 16);
    return insns_;
}

const void* GetXoredApk(JNIEnv* env, jobject thisPtr, jclass thisClass) {
    jmethodID getAssetsId = env->GetMethodID(thisClass, "getAssets", "()Landroid/content/res/AssetManager;");
    jobject jMgr = env->CallObjectMethod(thisPtr, getAssetsId);
    AAssetManager* mgr = AAssetManager_fromJava(env, jMgr);
    AAsset *asset = AAssetManager_open(mgr, "butterfly.png", AASSET_MODE_STREAMING);
    off64_t start, length;
    int fd = AAsset_openFileDescriptor64(asset, &start, &length);
    return AAsset_getBuffer(asset);
}

extern "C" JNIEXPORT void
JNICALL
decodeMethod(
        JNIEnv *env,
        jobject thisPtr,
        jstring jMethodName,
        jstring jMethodSignature) {
    /* Convert jstring to char* */
    const char *methodName = env->GetStringUTFChars(jMethodName, 0);
    const char *methodSignature = env->GetStringUTFChars(jMethodSignature, 0);

    jclass thisClass = env->GetObjectClass(thisPtr);

    unsigned int code_item_offset = getCodeItemOffset(env, thisClass, methodName, methodSignature);

    void* dex_file_location = getDexFileLocation();

    const void* mmaped_file_location = GetXoredApk(env, thisPtr, thisClass);

    unsigned int APK_insns_size_in_code_units_;
    void* APK_insns_ = GetCodeItemInstructions(dex_file_location, code_item_offset, &APK_insns_size_in_code_units_);

    unsigned int PNG_insns_size_in_code_units_;
    void* PNG_insns_ = GetCodeItemInstructions(mmaped_file_location, code_item_offset, &PNG_insns_size_in_code_units_);

    /* Change right on APK instruction page */
    void* base_addr = (void*)((char*)APK_insns_ - ((unsigned long)APK_insns_ % PAGE_SIZE));
    mprotect(base_addr, (size_t)((char*)APK_insns_ + APK_insns_size_in_code_units_*2 - (char*)base_addr), PROT_READ|PROT_WRITE|PROT_EXEC);

    /* Copy all the instruction */
    unsigned int i;
    for(i=0 ; i < APK_insns_size_in_code_units_; i++) {
        *((char*)APK_insns_ + 2*i) = *((char*)PNG_insns_ + 2*i) ^ 0x42;
        *((char*)APK_insns_ + 2*i + 1) = *((char*)PNG_insns_ + 2*i + 1) ^ 0x42;
    }

    /* Release created char* */
    env->ReleaseStringUTFChars(jMethodSignature, methodSignature);
    env->ReleaseStringUTFChars(jMethodName, methodName);
}
```

Listing B.19: Packer decode method

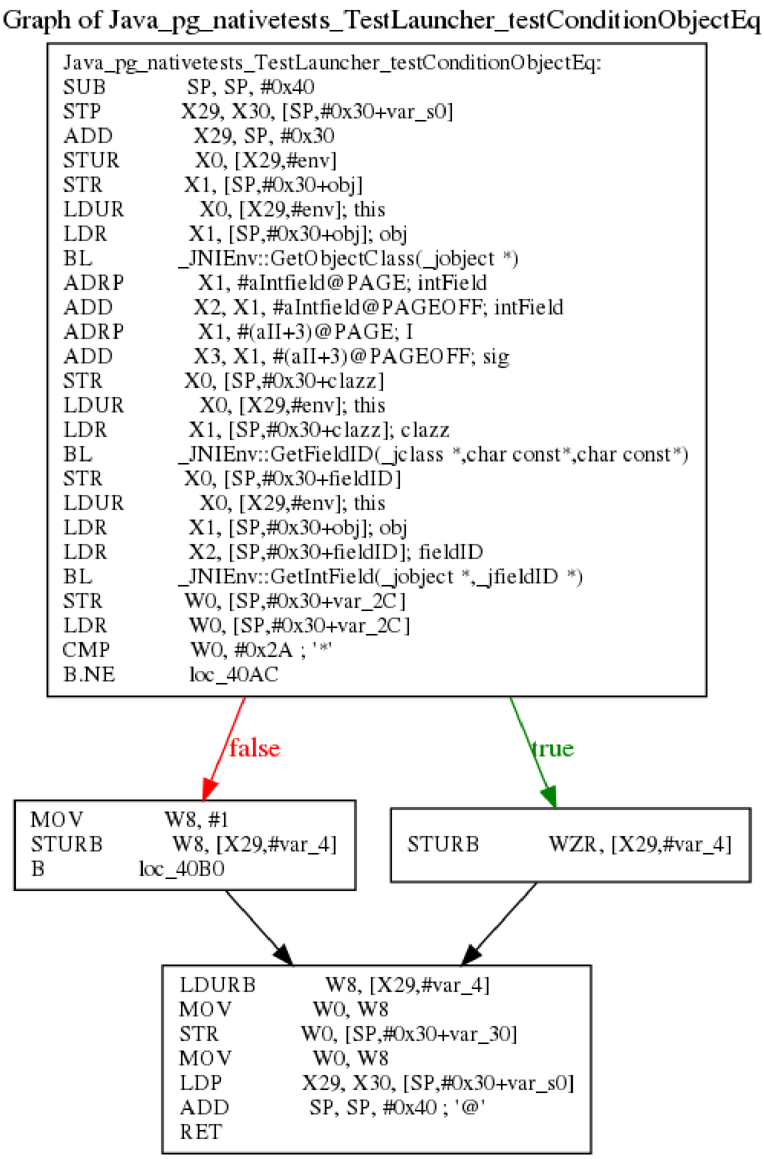
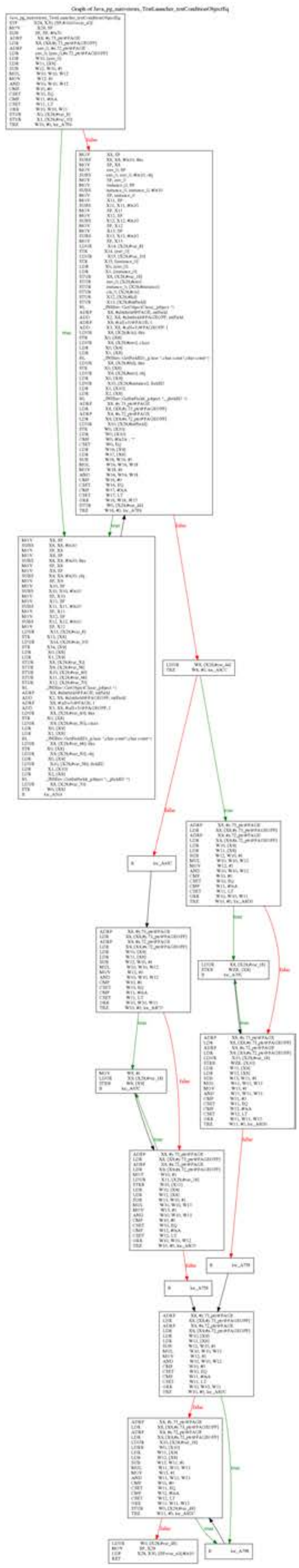


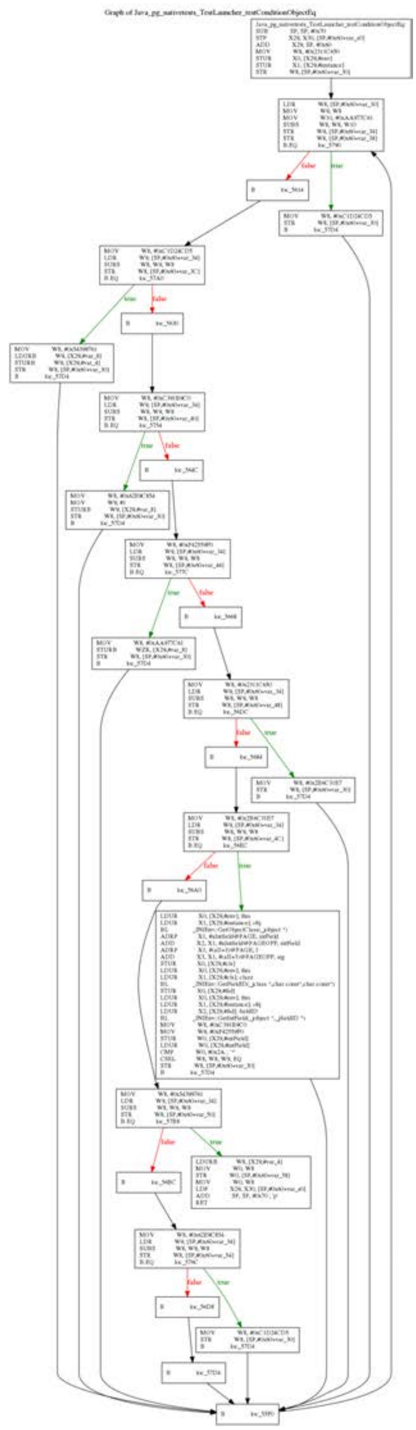
Figure B.1: Non obfuscated `testConditionObjectEq` method

```
#define INTFIELD_OFFSET 0x8
extern "C" JNIEXPORT jint JNICALL testAccessObjectInt(JNIEnv *env, jobject instance) {
  unsigned long * thisPtr = *(unsigned long **)thisObj;
  unsigned long * field_ptr = &thisPtr[INTFIELD_OFFSET/4];
  jint tmp = *field_ptr;
  *field_ptr = 3000;
  return tmp;
}
```

Listing B.22: Example of DHA unit test



(a) Opaque predicate obfuscation



(b) Control flow flattening obfuscation

Figure B.2: Obfuscated `testConditionObjectEq` method

C Concolic analysis functioning proof

This appendix presents a proof of the functioning of the concolic analysis presented in Section 7.2.4.

Definition 1. A dump $d \in \mathbb{D}$ is the whole memory of a process at a given time. It comprises both data and code areas.

Definition 2. A location $l \in \mathbb{L}oc$ is either a CPU register or a memory address. The special register `PC` (Program Counter) is the register that identifies the current instruction address.

Definition 3. A symbolic value $sv \in V\mathbb{S}alue$ is an expression over values ($v \in \mathbb{V}alue$) and symbols ($s \in \mathbb{S}ymbol$).

Definition 4. A symbolic state is a tuple $(\theta, \pi, \rho) \in \mathbb{S}$ where:

- $\theta : \mathbb{L}oc \rightarrow V\mathbb{S}alue$
  θ associates every location to a symbolic value.
- π is the current condition path, *i.e.* the set of conditions needs to be satisfied in order to reach to current instruction.
- $\rho : \mathbb{S}ymbol \rightarrow \mathbb{V}alue$
  ρ is the concretization function that associates symbols to their corresponding concrete value.
  By extension we note $\rho : \mathbb{C}ondition \rightarrow \mathbb{C}ondition$ the function that replaces symbols with their corresponding values in a condition.

Definition 5. A state $s \in \mathbb{S}$ is satisfiable, written $sat(s)$, if all its conditions $s.\pi$ are satisfiable when concretized.

$$sat(s) = \bigwedge_{p \in s.\pi} s.\rho(p)$$

Definition 6. A symbolic engine $\mathbb{E} : \mathbb{S} \times \mathbb{D} \rightarrow \mathbb{S}$ is a function that associates a state and a dump to a new state, resulting of the execution of one instruction.

Definition 7. An action $a \in \mathbb{A}$ is one of the following:

- **read** $r\ symb\ v$, a memory read, returning the symbol $symb$ concretized by value v, stored in register r;
- **write** $l\ symb\ v$, a memory write at location l of the symbol $symb$, concretized by value v;
- **invoke** $n\ (l, symb, v)^*$, an invocation of a method named n, with each potential parameter being a symbol $symb$ written at location l and concretized by value v;
- **ret** $r\ symb\ v$, a return of a method, returning the symbol $symb$ concretized by value v, stored in register r;
- **throw** $l\ symb\ v$, a throw of the symbol (exception object) $symb$ concretized by value v, stored in register l;
- **catch** $l\ symb\ v$, a catch of the symbol (exception object) $symb$ concretized by value v, written in register l.

Contextualized actions are tuples $(addr, act, next\_addr)$ where $addr$ is the address of the corresponding assembly instruction, $act \in \mathbf{A}$ is an action, and $next\_addr$ is the address of the next instruction.

Definition 8. The application of an action to a state, written $apply$: $\mathbb{A} \times \mathbb{S} \rightarrow \mathbb{S}$, reflects the effect of an action on a state.

For a **read**, a **write**, a **ret**, or a **catch**: $a = (\texttt{addr}, (\texttt{l}, \texttt{symb}, \texttt{v}), \texttt{next\_addr})$,

$apply(a, s) = (s.\theta[\texttt{l} \mapsto \texttt{symb},\ \texttt{PC} \mapsto \texttt{next\_addr}],\ s.\pi, s.\rho[\texttt{symb} \mapsto \texttt{v}])$

For an **invoke**, or a **throw**, only the `PC` is updated. Note that the instruction following an **invoke**, resp. a **throw**, is always a **ret**, resp. a **catch**.

Applying an action to a state generates the state in which subsequent actions will be executed.

The algorithm described in Section 7.2.4 is reported in Listing C.1. This algorithm outputs the conditions taken during the execution of a method using the actions and the dump given by *OATs'inside*. This concolic algorithm symbolically executes the instructions corresponding to the specific execution recorded by *OATs'inside*. Conditions are evaluated based on the values recovered from the execution traces, so only one single path is explored.

The correctness of the concolic analysis of Algorithm C.1 is supported by the following theorem:

Theorem 1. Given a dump, a list of actions, the entry point of a method and the list of parameters, the algorithm in Listing C.1 accurately generates the conditions taken by the execution and all assertions always hold (lines 17 and 27).

Under the hypothesis that the implementation of *OATs'inside* is accurate, that is:

Hypothesis. *OATs'inside* gives the complete list of actions (read, write, new, throw, catch, invoke, ret, monitor enter and exit) occurring during the execution of the analyzed method.

Proof. Proving Theorem 1 requires to show that:

- the algorithm generates accurately the conditions associated to the path taken during the concrete execution;
- for any generated symbolic state, the algorithm generates only one satisfiable state (asserts lines 17 and 27 hold).

The proof is achieved by induction over the number of instructions symbolically executed.

Base case: After the initialization (line 2), clearly only one state is generated (S_c). No instruction has yet been executed thus no condition appears.

Induction step: Let S_k the symbolic state generated by the symbolic execution of the k^{th} instruction. Assuming the two previous properties hold until the generation of S_k, we prove that these properties still hold for the generation of S_{k+1}:

- all occurring conditions are accurately logged;

- the next state is unique (asserts lines 17 and 27 hold) and accurate unless the end of the method is reached.

The k^{th} instruction can be either an instruction corresponding to an action outputted by *OATs'inside* or not.

First case the k^{th} instruction corresponds to an action (line 7). This instruction cannot be a conditional branching instruction because *OATs'inside* does not log such actions (note that throw and catch action can generate non-conditional branching). Thus, no condition is generated for the execution of the k^{th} instruction which obviously keeps the generation of conditions accurate.

The next step is generated by skipping the current instruction *i.e.* jumping to the next instruction executed by the concrete execution. Thus, the generated S_{k+1} state is unique. Moreover, S_{k+1} state is accurate because it has been updated accordingly to the actions outputted by *OATs'inside* (line 9), which is, by hypothesis, accurate.

Second case the k^{th} instruction does not correspond to an action observed by *OATs'inside*. The algorithm generates, line 14, the set S of possible next symbolic states.

- If there is only one generated state *i.e.* the executed instruction is not a conditional branching instruction, the next state is unique and logged conditions are still accurate (line 20).
- If there is more than one state generated, the algorithm has to determine which state has been taken during the concrete execution and to log the taken condition.
  To determine the next state, the algorithm obtains all the satisfiable states among the generated ones by replacing the symbols with concrete values inside the state condition (π) and checking that all conditions are still satisfiable (line 24).
  All the symbols added by the algorithm (line 9) have a corresponding concrete value. The others, added by the symbolic engine (line 14), would correspond to registers or memory areas. Such symbol cannot correspond to an unknown memory area because all memory is initialized using the dump. Moreover, this symbol cannot correspond to an uninitialized register: a well-formed method, *i.e.*, one respecting the ABI, only uses registers initialized by itself or by the calling method which is done by the algorithm during the initialization (line 1). Thus, when concretizing, all symbols are replaced with concrete values.
  Because a real execution cannot be in several states at the same time, only one state remains satisfiable when replacing all the symbols. Thus, only one next state is generated (assert line 27 hold).
  The algorithm needs to log the exact condition that has determined why this specific state has been taken rather than the other states generated. In fact, the condition corresponds to the difference between the previous state condition, $S_c.\pi$, and the conditions of the new selected state, $S_{sat}[0].\pi$ (line 28). The previous state being accurate and the symbolic engine being correct, the condition computed by difference is also accurate.

- Because S is accurate, its symbolic execution generates at least the state taken by the concrete execution (assert line 17 hold).
  If there is no generated state, it means that the symbolic engine considers that the program has crashed. Yet, the concrete execution has not crashed, which is contradictory.
  Nevertheless, the only potential case that could crash the symbolic engine is a syscall because its code is not provided to the symbolic engine. However, their results and effects can be easily retrieved by *OATs'inside*. Then, the symbolic analysis can treat them as any other action by applying syscall effects to the symbolic state instead of trying to execute syscalls. This way, crashes are avoided.

In all cases, the algorithm generates an accurate and unique next state and logs the eventual accurate conditions. This proves the induction step and, thus, the overall theorem. ■

Algorithm C.1: Get taken conditions algorithm.

Input: $c \in \mathbb{C}, actions \in \mathbb{A}^*, entrypoint \in \mathbb{V}alue, parameters\_values \in \mathbb{V}alue^*, engine \in \mathbb{E}$

Construct an initial state according to the entrypoint address and the method parameters given by *OATs'inside*.
$S_c = (\theta = \{PC \mapsto entrypoint\}, \pi = \emptyset, \rho = \{regs \mapsto parameters\_values\})$
Get the first action.
$a_c = actions.pop()$
While PC is in analyzed method.
while $S_c.\theta(PC) \in method$ **do**
  If current address corresponds to the next action.
  if $S_c.\theta(PC) == a_c.addr$ **then**
    Execute this action.
    $S_c = apply(S_c, a_c)$
    Get the next action.
    $a_c = actions.pop()$
  else
    Execute symbolically one instruction.
    $S = engine(S_c, c)$
    At least one state must have been generated.
    assert $(len(S) \geq 1)$
    If only one state has been generated.
    if $len(S) == 1$ **then**
      Go to this next state.
      $S_c = S[0]$
      If several states have been generated.
    else
      Get all the satisfiable states.
      $S_{sat} = \{s \mid s \in S \wedge sat(s)\}$
      Exactly one state must be satisfiable
      assert $(len(S_{sat}) == 1)$
      Output the conditions that are present in this satisfiable state but not in the previous state.
      $LOG(S_{sat}[0].\pi \setminus S_c.\pi)$
      The next state is this satisfiable state.
      $S_c = S_{sat}[0]$
    end if
  end if
end while

Résumé substantiel en français

1 Introduction

1.1 Sécurité du système Android

Android est le système d'exploitation le plus utilisé dans les smartphones modernes. C'est pourquoi il constitue une cible de choix pour les personnes malveillantes comme en témoigne le grand nombre de CVEs[1] reportées chaque année, voir Table 1. Google apporte donc un soin tout particulier à développer une architecture sécurisée pour Android. Cette dernière repose sur deux points :

- L'isolation des applications : chaque application est exécutée par un utilisateur UNIX différent. Cela permet d'utiliser les mécanismes d'isolation éprouvés du noyau Linux afin de séparer chaque application.
- Une gestion fine des permissions : les opérations sensibles[2] sont uniquement réalisables par les applications disposant de la permission adéquate. Ces permissions sont accordées par l'utilisateur lui-même.

1: *Common Vulnerabilities and Exposures*, vulnérabilités connues publiquement.

2: Comme l'envoi de SMS ou l'accès à la liste des contacts.

< 2010	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020
18	23	89	169	123	1686	422	872	1191	457	771	528

Source: `https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Android`

Table 1: Nombre de CVE contenant le mot-clef "Android"

Néanmoins, sécuriser le système Android n'est pas suffisant. En effet, les applications installées par les utilisateurs sont potentiellement malveillantes ou vulnérables. Une application est considérée vulnérable si elle peut être détournée par un attaquant afin de réaliser des opérations malveillantes telles qu'envoyer des SMSs à des services payants, contourner le système de permissions, obtenir des informations privées de l'utilisateur, diffuser de la publicité intempestive.

Malheureusement, faire reposer la sécurité d'Android uniquement sur l'isolation et l'emploi de permissions n'est pas fiable. Le système de permissions est notamment mal compris par les utilisateurs, qui peuvent octroyer des permissions dangereuses à des applications [12, 13]. Puisque le système Android ne peut se prémunir contre ce type d'attaque, il est nécessaire:

- de détecter les applications malveillantes ou vulnérables afin de les retirer des plateformes de diffusion d'application[3].
- de comprendre le comportement de ces applications afin de pouvoir évaluer et résorber les dommages commis après une compromission.

[12]: Felt, Ha, Egelman, Haney, Chin, and Wagner (2012), 'Android permissions: User attention, comprehension, and behavior'
[13]: Benton, Camp, and Garg (2013), 'Studying the effectiveness of android application permissions requests'

3: Notamment le Google Play store, plateforme officielle de Google.

S'engage alors un jeu du chat et de la souris entre d'une part les analystes et chercheurs qui tentent de mettre au point des systèmes de détection et d'analyse, et d'autre part les applications malveillantes qui tentent d'échapper à ces derniers en inventant de nouvelles techniques et en trouvant de nouvelles vulnérabilités.

1.2 Applications natives et sécurité

Dans ce contexte, un nouveau type d'applications Android voit, depuis 2014, son utilisation de plus en plus fréquente : les applications natives [16–18]. Ces applications, contrairement aux applications classiques développées en utilisant uniquement les langages Java et Kotlin, contiennent également du code assembleur, résultant de la compilation de code C/C++. Android fournit une interface appelée Java Native Interface (JNI) afin de permettre au code assembleur de communiquer avec le bytecode Dalvik provenant de Java ou Kotlin.

[16]: Afonso, Geus, Bianchi, Fratantonio, Kruegel, Vigna, Doupé, and Polino (2016), 'Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy'
[17]: Tam, Feizollah, Anuar, Salleh, and Cavallaro (2017), 'The evolution of android malware and android analysis techniques'
[18]: Sadeghi, Bagheri, Garcia, and Malek (2017), 'A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software'

Il est donc nécessaire d'adapter les techniques d'analyse fonctionnant sur le bytecode au code assembleur. Ainsi, il convient de:

- Étudier les possibilités offertes par le code natif pour développer de nouvelles techniques d'obfuscation et proposer des méthodes de détection associées.
- Trouver quelles vulnérabilités pourraient être introduites par la présence de code natif dans une application Android.
- Développer, fort des résultats obtenus dans les études précédentes, de nouvelles techniques d'analyse capables de prendre en compte le code natif des applications Android.

Le domaine de l'étude de la sécurité des applications natives n'est cependant pas vierge de recherches.

En effet, Yu [77] a révélé l'utilisation, par les applications malveillantes, d'une nouvelle technique d'obfuscation native appelée *packing*. Depuis, de nombreux travaux [78–80] ont proposé des contre-mesures à cette technique. Dans cette thèse, nous proposons, dans les Sections 2.1 et 2.2, de nouvelles techniques d'obfuscation utilisant le code natif.

[77]: Yu (2014), 'Android packers: facing the challenges, building solutions'

Par ailleurs, des vulnérabilités impliquant le code natif ont également été trouvées [36, 40]. Dans la Section 3.2 nous proposerons une méthode ainsi qu'un outil, qui détecte automatiquement les vulnérabilités présentées par Peles [36].

[36]: Peles and Hay (2015), 'One Class to Rule Them All: 0-day Deserialization Vulnerabilities in Android'

Enfin, différentes techniques d'analyse d'applications Android ont été adaptées aux applications natives [83, 84, 88, 95, 106]. Cependant, nous montrerons que ces outils ne parviennent pas à analyser des applications natives obfusquées à l'aide des méthodes que nous proposons et nous finirons par présenter un outil, appelé *OATs'inside*, capable de gérer de telles applications dans la Section 4.

1.3 Contributions

Cette thèse présente:

- Deux nouvelles méthodes d'obfuscation pour les applications Android utilisant le code natif [1, 3, 7].
- Le résultat des expérimentations évaluant l'utilisation de ces techniques dans la nature [2, 7].
- Un nouveau framework d'analyse, appelé *OATs'inside*, qui combine analyses dynamique et symbolique afin de récupérer le comportement des applications Android obfusquées.
- Une nouvelle méthode de détection de vulnérabilité portant sur l'oubli du mot-clef `transient` au sein des applications Android natives.

[1]: **Graux**, Lalande, and Viet Triem Tong (2018), 'Etat de l'Art des Techniques d'Unpacking pour les Applications Android'
[3]: **Graux**, Lalande, and Viet Triem Tong (2019), 'Obfuscated Android Application Development'
[7]: **Graux**, Lalande, Wilke, and Viet Triem Tong (2020), 'Abusing Android Runtime for Application Obfuscation'

[2]: Lalande, Viet Triem Tong, Leslous, and **Graux** (2018), 'Challenges for reliable and large scale evaluation of android malware analysis'
[7]: **Graux**, Lalande, Wilke, and Viet Triem Tong (2020), 'Abusing Android Runtime for Application Obfuscation'

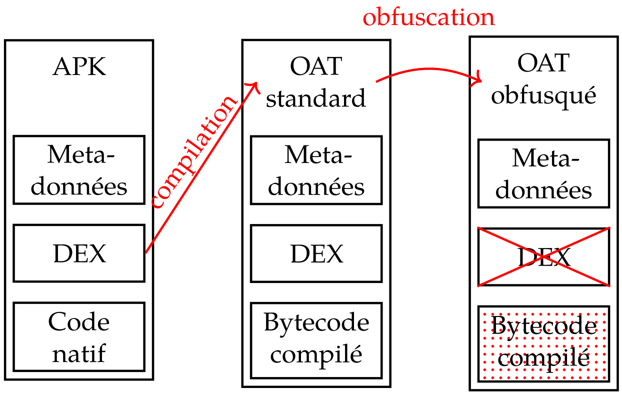


Figure 1: Bytecode Free OAT

2 Problèmes de sécurité induits par les interférences natives dans les applications Android Java

Afin de lister l'ensemble des problèmes que peut poser la présence de code natif dans une application Android, nous allons, dans cette section, lister de manière systématique l'ensemble des interférences que le code natif peut produire sur le code Java. Seront alors mis en lumière les différents points faibles des analyses préexistantes. Nous présenterons d'abord les interférences qui concernent le code Java, dans la Section 2.1, pour ensuite étudier celles qui concernent les données, dans la Section 2.2.

2.1 Problèmes portant sur le bytecode

Le bytecode Dalvik est plus simple à analyser que le code assembleur. En effet, le bytecode Dalvik[4] est un langage haut-niveau puisqu'il utilise la programmation orientée objet pour définir les objets sur lesquels il travaille. À contrario, le code assembleur est très proche de l'architecture du téléphone puisqu'il change en fonction du processeur qui l'exécute. C'est pourquoi, un développeur qui souhaite protéger son application peut vouloir remplacer le bytecode qu'elle contient par du code assembleur. L'écriture d'applications en assembleur étant fastidieuse et sujette aux erreurs, il est nécessaire d'employer des techniques d'automatisation.

4: `https://source.android.com/devices/tech/dalvik/dalvik-bytecode`

Nous avons vu dans l'état-de-l'art une telle technique appelée *packing* [77]. Elle consiste à chiffrer le bytecode de l'application et à le remplacer par une routine de déchiffrement appelée *unpacker*. Cet *unpacker*, au moment de l'exécution de l'application, déchiffre le bytecode original de l'application et l'exécute. Ainsi, une analyse statique[5] ne peut pas accéder au code de l'application et est rendue caduque. Une évaluation de l'utilisation de cette méthode dans la nature est proposée dans la Section 3.1.

[77]: Yu (2014), 'Android packers: facing the challenges, building solutions'

5: Une analyse qui n'exécute pas l'application.

Cependant, l'emploi de cette technique ne prémunit pas contre une analyse dynamique[6]. La technique que nous proposons, appelée Bytecode-Free OAT (BFO) consiste à compiler l'application en utilisant le compilateur d'Android, comme montré dans la Figure 1. Ce dernier, à des fins d'optimisation, transforme le bytecode de l'application en code assembleur, stocké dans un fichier au format OAT[7]. Le bytecode original est

6: Une analyse qui exécute l'application.

7: Aucune explication officielle n'est fournie pour cet acronyme.

alors modifié. Le code assembleur étant exécuté à la place du bytecode, le comportement de l'application n'est pas modifié mais un analyste ou un outil qui ne prend en compte que le bytecode obtient un résultat faussé : il ne se base pas sur le bon code. Une méthode de détection associée à cette technique est proposée dans la Section 3.1.

2.2 Problèmes portant sur les données du bytecode

Les données décrites par le code Java offrent plus de garanties de sécurité que celle décrites par les langages C/C++. En effet, le code Java est fortement typé et la machine virtuelle qui l'exécute vérifie que les données qu'elle manipule sont cohérentes[8]. D'un autre coté les langages C/C++ sont très permissifs et permettent de manipuler librement les données qui sont vues comme des suites de bits. Malheureusement un développeur peut, par inadvertance, injecter des données assembleur dans les données bytecode qui ne respectent pas les garanties offertes par ces dernières. Peles [36] a notamment montré que le stockage de pointeurs mémoires (données assembleurs) dans un champ (donnée bytecode) d'une classe sérialisable peut rendre l'application vulnérable si le champ n'est pas déclaré `transient`. Une solution à ce problème est proposée dans la Section 3.2.

8: Par exemple, ajouter un entier et une chaîne de caractère n'a pas de sens.

[36]: Peles and Hay (2015), 'One Class to Rule Them All: 0-day Deserialization Vulnerabilities in Android'

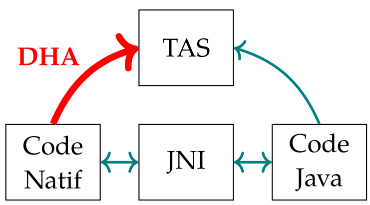


Figure 2: Direct Heap Access (DHA)

En plus de ces injections de données, un développeur d'application peut volontairement modifier les données Java depuis le code natif à des fins d'obfuscation. Classiquement, cela est réalisé en utilisant l'interface JNI fournie par Android. Tout naturellement, les différents outils de l'état-de-l'art reposent sur cette interface pour prendre en compte les effets du code natif sur les données bytecode. Afin d'outrepasser ces analyses nous proposons une technique, appelée Direct Heap Access (DHA), représentée par la Figure 2. Elle consiste à accéder directement au tas[9] afin de modifier les données bytecode sans utiliser JNI. Une méthode de détection de cette technique est proposée dans la Section 3.2.

9: Espace mémoire où sont stockés les objets bytecode.

3 Détection des interférences natives dans les applications Android Java

Dans la section précédente, nous avons montré que les interférences entre le code assembleur et le bytecode peuvent être utilisées à des fins d'obfuscation ou peuvent introduire de nouvelles vulnérabilités. Dans cette section, nous nous attachons à donner de nouvelles méthodes pour détecter ces interférences et à évaluer leur utilisation dans la nature.

3.1 Détection des interférences sur bytecode

Concernant l'utilisation de *packing*, des solutions de détection sont déjà proposées [78, 94]. Afin de montrer l'utilisation de cette technique dans la nature, nous avons utilisé l'outil APKiD[10] sur trois datasets: un comprenant des malwares (MAL), un comprenant des goodwares (GOOD) et un dernier dont les APKs sont plus ou moins datés. Comme

10: `https://github.com/rednaga/APKiD`

[78]: Zhang, Luo, and Yin (2015), 'Dexhunter: toward extracting hidden code from packed android applications'
[94]: Jiang, Zhou, Liu, Jia, Liu, and Zuo (2017), 'CrackDex: Universal and automatic DEX extraction method'

	GOOD [4]	MAL [4]
Total	4 999	4 991
Détection	3 0,06%	542 10,86%

Table 2: Détection de *packing* dans plusieurs datasets

Année	2008-2013	2014	2015	2016	2017	2018
App. *packée*	0	1	4	5	7	7

Table 3: Détection de *packing* selon les années

le montrent les Tables 2 et 3, l'utilisation du *packing* est prévalente chez les malwares et est en progression.

BFO étant une nouvelle technique, l'état-de-l'art ne propose aucune solution. Nous avons donc proposé une nouvelle méthode pour détecter son utilisation. Elle consiste à compiler le bytecode de l'application et comparer le code assembleur obtenu à celui déjà présent dans l'application. Si les codes sont différents alors c'est que la technique BFO a été utilisée. Nous avons utilisé cette technique de détection sur les applications de 17 firmwares sans trouver aucune utilisation de BFO.

3.2 Détection des interférences sur les données du bytecode

Total	ARMv8	DHA	DHA sans libs. systèmes
100 018	10 661	8 158 (76,5 %)	4 021 (37,7 %)

Table 4: Nombre de DHAs détectés

Afin de détecter l'utilisation de DHA au sein d'une application, nous proposons de l'exécuter tout en interdisant le code natif d'accéder au tas. Pour cela nous modifions la machine virtuelle d'Android qui exécute les applications pour interdire, à l'aide de `mprotect`, l'accès au tas lorsque du code natif est exécuté. Nous avons utilisé cette méthode sur une partie (100 000 applications) du dataset Androzoo [57] et trouvé qu'une majorité des applications utilisent du DHA. Les résultats sont reportés dans la Table 4. Cependant, après investigation, nous n'avons pas été capables d'isoler un cas d'utilisation à des fins d'obfuscation. En effet, DHA semble n'être, pour l'instant, utilisé que pour optimiser l'application en évitant d'utiliser JNI qui est une interface coûteuse en temps.

[57]: Allix, Bissyandé, Klein, and Le Traon (2016), 'AndroZoo: Collecting Millions of Android Apps for the Research Community'

Pour détecter la vulnérabilité proposée par Peles [36], il faut trouver tous les champs, non déclarés `transient`, d'objets bytecode qui peuvent recevoir un pointeur mémoire. Pour cela nous proposons de conduire sur le code-source une analyse de teinte, comme représentée dans la Figure 3. Deux analyses sont conduites. Durant la première, les pointeurs mémoires sont considérés comme des sources, l'écriture dans un champ comme un puits. Cela permet de détecter tout pointeur qui est enregistré dans un champ. Durant la seconde, les lectures de champs sont considérées comme des sources, et l'utilisation d'un pointeur (cast en pointeur) est considérée comme un puits. Cela permet de détecter tout champ qui est utilisé comme un pointeur. L'utilisation de cette technique nous a permis de retrouver une vulnérabilité connue[11] dans la librairie SSL d'Android et de trouver, au sein de l'application Telegram, des mot-clefs `transient` manquants mais non exploitables.

[36]: Peles and Hay (2015), 'One Class to Rule Them All: 0-day Deserialization Vulnerabilities in Android'

11: CVE-2015-3837: https://nvd.nist.gov/vuln/detail/CVE-2015-3837

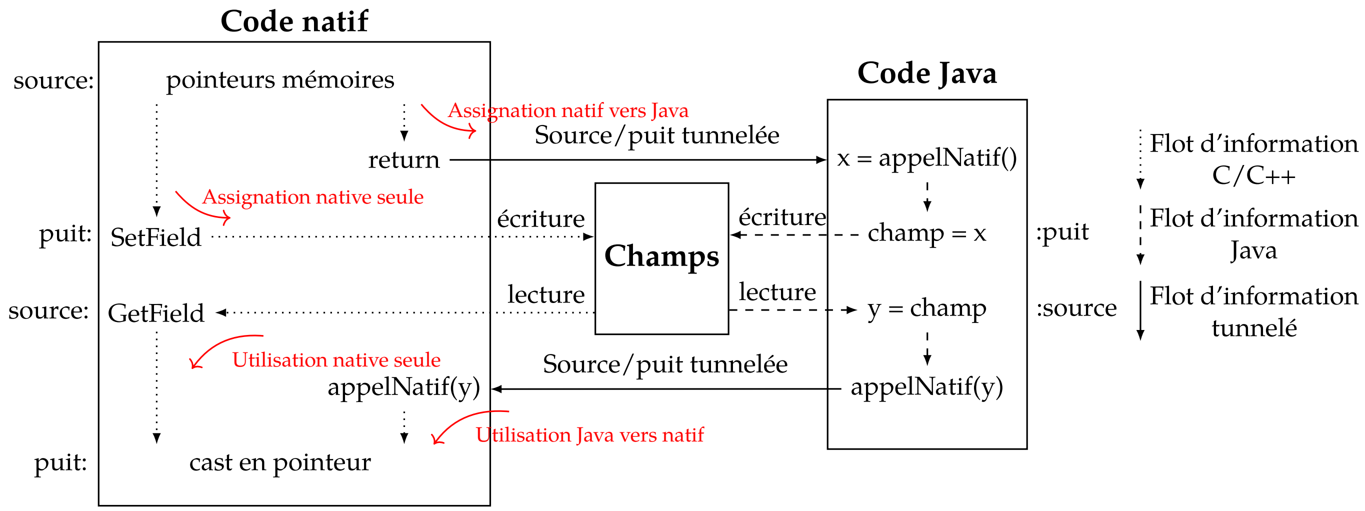


Figure 3: Représentation du suivi des flux de références mémoires

4 *OATs'inside*: rétro-ingénierie des applications Android natives

Comme nous avons vu dans la Section 2, les outils de l'état-de-l'art ne sont pas capables de prendre en compte tous les problèmes que le code natif introduit dans les applications Android. En particulier, ils ne sont pas capables de prendre en compte correctement BFO et DHA.

C'est pourquoi nous proposons un nouveau framework d'analyse appelé *OATs'inside*. L'analyse qu'il conduit est composée de deux phases représentées par la Figure 4. Durant la première, l'application est exécutée sur un téléphone dont la machine virtuelle est modifiée pour enregistrer toutes les actions objets[12] réalisées par l'application. Sont notamment enregistrés les accès directs au tas (DHA), comme décrit en Section 3.2, et les actions réalisées par le bytecode compilé (BFO). Ces actions, qui constituent l'ensemble du comportement objet de l'application, sont ensuite envoyées à un ordinateur d'analyse qui les présente sous forme de graphe[13] à un analyste. Ce graphe montre de quelle manière les actions de l'application s'enchaînent. L'analyste peut alors choisir une méthode du code qu'il trouve particulièrement intéressante à investiguer. Une seconde exécution est alors effectuée pour récupérer l'état de la mémoire (snapshot) lors de l'exécution de cette méthode. À l'aide de ce snapshot, *OATs'inside* conduit une analyse symbolique qui enrichit le graphe précédant en indiquant comment les données sont transmises entre les différentes actions. Ce graphe final permet à un analyste de comprendre correctement le comportement d'une application Android native même obfusquée.

12: Modification de champs, appels de méthodes, levées d'exceptions, ...

13: Ces graphes sont des *Control Flow Graph* (CFG).

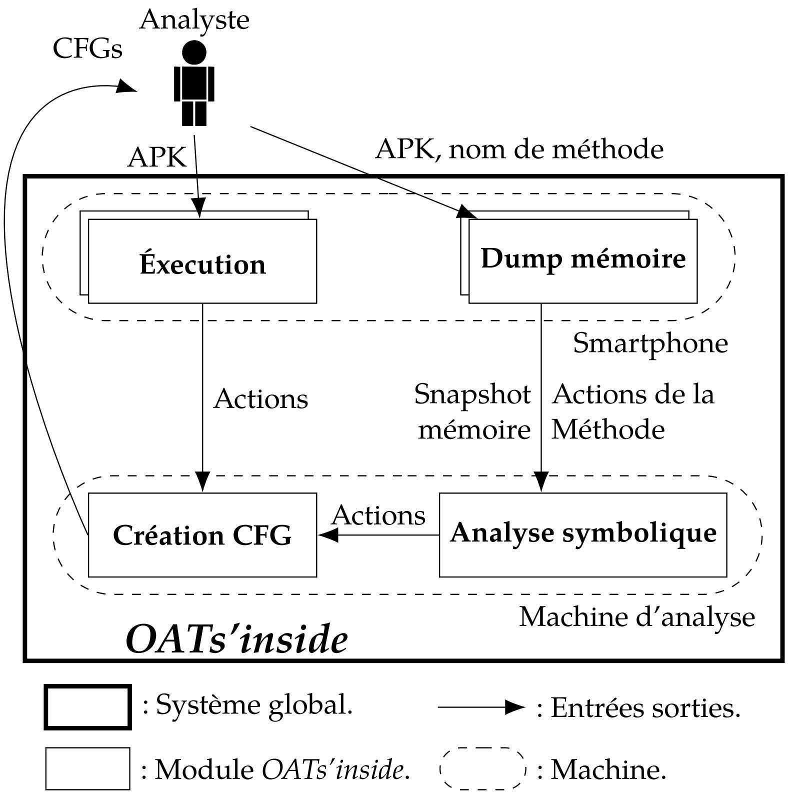


Figure 4: Architecture de *OATs'inside*

5 Travaux futurs

Au cours de cette thèse, nous avons montré que la présence de code natif au sein d'une application Android permet l'utilisation de nouvelles techniques d'obfuscation et introduit de nouvelle vulnérabilités. Nous avons également proposé des méthodes de détection correspondant à ces deux problèmes ainsi qu'un framework d'analyse d'applications Android natives obfusquées.

Dans un premier temps, nous pensons que ces travaux de thèse pourraient être appliqués à d'autres systèmes Android, comme Android Automotive[14] ou Android TV[15]. Ils pourraient également être adaptés à d'autres langages comme Python qui est également capable d'exécuter du code natif.

Dans un second temps, nous pensons qu'il est nécessaire de redéfinir l'interface entre le code natif et le bytecode afin de mieux contrôler leurs interactions. Pour cela, nous pourrions nous inspirer de l'implémentation que font les navigateurs web de Javascript et de WebAssembly. En effet, ces deux langages fonctionnent de la même manière que Java/Kotlin et C/C++. Cependant, au sein des navigateurs web, ils sont fortement isolés et c'est Javascript, le langage haut-niveau, qui déclare explicitement les points d'interface avec WebAssembly. Cela permet de mieux contrôler et d'analyser cette interface.

14: Android pour voitures connectées.

15: Android pour télévisons connectées.

Bibliography

[1] **Pierre Graux**, Jean-François Lalande, and Valérie Viet Triem Tong. 'Etat de l'Art des Techniques d'Unpacking pour les Applications Android'. In: *Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information*. La Bresse, France, May 2018.

[2] Jean-François Lalande, Valérie Viet Triem Tong, Mourad Leslous, and **Pierre Graux**. 'Challenges for reliable and large scale evaluation of android malware analysis'. In: *2018 International Conference on High Performance Computing & Simulation*. Orléans, France: IEEE, July 2018, pp. 1068–1070.

[3] **Pierre Graux**, Jean-François Lalande, and Valérie Viet Triem Tong. 'Obfuscated Android Application Development'. In: *3rd Central European Cybersecurity Conference*. Munich, Germany: ACM, Nov. 2019, pp. 1–6.

[4] Valérie Viet Triem Tong, Cédric Herzog, Tomàs Concepción Miranda, **Pierre Graux**, Jean-François Lalande, and Pierre Wilke. 'Isolating Malicious Code in Android Malware in the Wild'. In: *14th International Conference on Malicious and Unwanted Software*. Nantucket, MA, USA: IEEE, Oct. 2019.

[5] Jean-François Lalande, Valérie Viet Triem Tong, **Pierre Graux**, Guillaume Hiet, Wojciech Mazurczyk, Habiba Chaoui, and Pascal Berthomé. 'Teaching android mobile security'. In: *50th ACM Technical Symposium on Computer Science Education*. Minneapolis MN, USA: ACM, Feb. 2019, pp. 232–238.

[6] Jean-François Lalande, **Pierre Graux**, and Tomás Concepción Miranda. 'Orchestrating Android Malware Experiments'. In: *27th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. Rennes, France: IEEE, Oct. 2019, pp. 1–2.

[7] **Pierre Graux**, Jean-François Lalande, Pierre Wilke, and Valérie Viet Triem Tong. 'Abusing Android Runtime for Application Obfuscation'. In: *Workshop on Software Attacks and Defenses*. Genova, Italy: IEEE, Sept. 2020, pp. 616–624.

[8] **Pierre Graux**, Jean-François Lalande, Valérie Viet Triem Tong, and Pierre Wilke. 'Preventing Serialization Vulnerabilities through Transient Field Detection'. In: *36th ACM/SIGAPP Symposium On Applied Computing*. Gwangju, Korea: ACM, Mar. 2021, pp. 1598–1606.

[9] Ibtisam Mohamed and Dhiren Patel. 'Android vs iOS security: A comparative study'. In: *12nd International Conference on Information Technology - New Generations*. Las Vegas, NV, USA: IEEE, Apr. 2015, pp. 725–730.

[10] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. 'Android security: a survey of issues, malware penetration, and defenses'. In: *IEEE Communications Surveys & Tutorials* 17.2 (2014).

[11] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. 'A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software'. In: *IEEE Transactions on Software Engineering* 43.6 (2016).

[12] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. 'Android permissions: User attention, comprehension, and behavior'. In: *8th Symposium on Usable Privacy and Security*. Washington, DC, USA: ACM, July 2012, pp. 1–14.

[13] Kevin Benton, L Jean Camp, and Vaibhav Garg. 'Studying the effectiveness of android application permissions requests'. In: *IEEE International Conference on Pervasive Computing and Communications Workshops*. San Diego, CA, USA: IEEE, Mar. 2013, pp. 291–296.

[14] Ali Aydın Selçuk, Fatih Orhan, and Berker Batur. 'Undecidable problems in malware analysis'. In: *12th International Conference for Internet Technology and Secured Transactions*. Cambridge, UK: IEEE, Dec. 2017, pp. 494–497.

[15] Philippe Beaucamps and Éric Filiol. 'On the possibility of practically obfuscating programs towards a unified perspective of code protection'. In: *Journal in Computer Virology* 3.1 (2007).

[16] Vitor Monte Afonso, Paulo L de Geus, Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, Giovanni Vigna, Adam Doupé, and Mario Polino. 'Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy'. In: *23rd Network and Distributed System Security Symposium*. San Diego, USA: The Internet Society, Feb. 2016.

[17] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. 'The evolution of android malware and android analysis techniques'. In: *ACM Computing Surveys* 49.4 (2017).

[18] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. 'A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software'. In: *IEEE Transactions on Software Engineering* 43.6 (2017).

[19] Mengtao Sun and Gang Tan. 'Nativeguard: Protecting android applications from third-party native libraries'. In: *ACM conference on Security and privacy in wireless & mobile networks*. Oxford, UK: ACM, July 2014, pp. 165–176.

[20] Elias Athanasopoulos, Vasileios P Kemerlis, Georgios Portokalidis, and Angelos D Keromytis. 'NaClDroid: Native Code Isolation for Android Applications'. In: *21st European Symposium on Research in Computer Security*. Heraklion, Greece: Springer, Sept. 2016, pp. 422–439.

[21] Shuai Zhao, Xiaohong Li, Guangquan Xu, Lei Zhang, and Zhiyong Feng. 'Attack tree based android malware detection with hybrid analysis'. In: *13th International Conference on Trust, Security and Privacy in Computing and Communications*. Beijing, China: IEEE, Sept. 2014, pp. 380–387.

[22] Yousra Aafer, Wenliang Du, and Heng Yin. 'Droidapiminer: Mining api-level features for robust malware detection in android'. In: *International conference on Security and Privacy in Communication Cystems*. Sydney, NSW, Australia: Springer, Sept. 2013, pp. 86–103.

[23] Wen-Chieh Wu and Shih-Hao Hung. 'DroidDolphin: a dynamic Android malware detection framework using big data and machine learning'. In: *Conference on Research in Adaptive and Convergent Systems*. Towson, Maryland, USA: ACM, Oct. 2014, pp. 247–252.

[24] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 'Hey, you, get off of my market: detecting malicious apps in official and alternative android markets.' In: *19th Annual Network & Distributed System Security Symposium*. San Diego, CA, USA: The Internet Society, Feb. 2012, pp. 50–52.

[25] Gianluca Dini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. 'MADAM: a multi-level anomaly detector for android malware'. In: *6th International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*. St. Petersburg, Russia: Springer, Oct. 2012, pp. 240–253.

[26] Martina Lindorfer, Matthias Neugschwandtner, and Christian Platzer. 'Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis'. In: *39th Annual Computer Software and Applications Conference*. Taichung, Taiwan: IEEE, July 2015, pp. 422–433.

[27] Radoniaina Andriatsimandefitra and Valérie Viet Triem Tong. 'Capturing android malware behaviour using system flow graph'. In: *9th International Conference on Network and System Security*. Nwe York City, USA: Springer, Nov. 2015, pp. 534–541.

[28] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 'Apposcopy: Semantics-based detection of android malware through static analysis'. In: *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Hong Kong, China: ACM, Nov. 2014, pp. 576–587.

[29] Jonathan Crussell, Clint Gibler, and Hao Chen. 'Attack of the clones: Detecting cloned applications on android markets'. In: *17th European Symposium on Research in Computer Security*. Pisa, Italy: Springer, Sept. 2012, pp. 37–54.

[30] Shahid Alam, Ryan Riley, Ibrahim Sogukpinar, and Necmeddin Carkaci. 'Droidclone: Detecting android malware variants by exposing code clones'. In: *6th International Conference on Digital Information and Communication Technology and its Applications*. Konya, Turkey: IEEE, July 2016, pp. 79–84.

[31] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 'Detecting repackaged smartphone applications in third-party android marketplaces'. In: *2nd ACM conference on Data and Application Security and Privacy*. San Antonio, Texas, USA, Feb. 2012, pp. 317–326.

[32] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. 'Towards a scalable resource-driven approach for detecting repackaged android applications'. In: *30th Annual Computer Security Applications Conference*. New Orleans, Louisiana, USA, Dec. 2014, pp. 56–65.

[33] Ying-Dar Lin, Yuan-Cheng Lai, Chien-Hung Chen, and Hao-Chuan Tsai. 'Identifying android malicious repackaged applications by thread-grained system call sequences'. In: *Computers & Security* 39 (2013).

[34] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. 'Obfuscation-resilient privacy leak detection for mobile apps through differential analysis'. In: *24th Network and Distributed System Security Symposium*. San Diego, CA, USA: The Internet Society, Mar. 2017.

[35] Min Zheng, Mingshen Sun, and John CS Lui. 'DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability'. In: *International Wireless Communications and Mobile Computing Conference*. Nicosia, Cyprus: IEEE, Aug. 2014, pp. 128–133.

[36] Or Peles and Roee Hay. 'One Class to Rule Them All: 0-day Deserialization Vulnerabilities in Android'. In: *9th USENIX Workshop on Offensive Technologies*. Washington, D.C., USA: USENIX, Aug. 2015.

[37] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 'Chex: statically vetting android apps for component hijacking vulnerabilities'. In: *ACM conference on Computer and Communications Security*. Raleigh, North Carolina, USA: ACM, Oct. 2012, pp. 229–240.

[38] Mu Zhang and Heng Yin. 'AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications.' In: *21st Network and Distributed System Security Symposium*. San Diego, CA, USA: The Internet Society, Feb. 2014.

[39] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. 'Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps'. In: *21st Network and Distributed System Security Symposium*. San Diego, CA, USA: The Internet Society, Feb. 2014.

[40] Yacong Gu, Kun Sun, Purui Su, Qi Li, Yemian Lu, Lingyun Ying, and Dengguo Feng. 'JGRE: An Analysis of JNI Global Reference Exhaustion Vulnerabilities in Android'. In: *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Denver, CO, USA: IEEE, June 2017, pp. 427–438.

[41] Chenxiong Qian, Xiapu Luo, Yu Le, and Guofei Gu. 'Vulhunter: toward discovering vulnerabilities in android applications'. In: *IEEE Micro* 35.1 (2015).

[42] R Dhaya and M Poongodi. 'Source Code Analysis for Software Vulnerabilities in Android based Mobile Devices'. In: *International Journal of Computer Applications* 93.17 (2014).

[43] Kun Yang, Jianwei Zhuge, Yongke Wang, Lujue Zhou, and Haixin Duan. 'IntentFuzzer: detecting capability leaks of android applications'. In: *9th ACM symposium on Information, Computer and Communications Security*. Kyoto, Japan: ACM, June 2014, pp. 531–536.

[44] Raimondas Sasnauskas and John Regehr. 'Intent fuzzer: crafting intents of death'. In: *Joint International Workshop on Dynamic Analysis and Software and System Performance Testing, Debugging, and Analytics*. San Jose, CA, USA: ACM, July 2014, pp. 1–5.

[45] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. 'Patchdroid: Scalable third-party security patches for android devices'. In: *29th Annual Computer Security Applications Conference*. New Orleans, Louisiana, USA: ACM, Dec. 2013, pp. 259–268.

[46] Timothy Vidas and Nicolas Christin. 'Evading android runtime analysis via sandbox detection'. In: *9th ACM symposium on Information, computer and communications security*. Kyoto, Japan: ACM, June 2014, pp. 447–458.

[47] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 'Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware'. In: *7th European Workshop on System Security*. Amsterdam, Netherlands: ACM, Apr. 2014, pp. 1–6.

[48] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 'Automated Test Input Generation for Android: Are We There Yet?' In: *30th IEEE/ACM International Conference on Automated Software Engineering*. Lincoln, NE, USA: IEEE/ACM, Nov. 2015, pp. 429–440.

[49] Adrien Abraham, Radoniaina Andriatsimandefitra, Adrien Brunelat, J-F Lalande, and V Viet Triem Tong. 'GroddDroid: a gorilla for triggering malicious behaviors'. In: *10th International Conference on Malicious and Unwanted Software*. Fajardo, Puerto Rico: IEEE, Oct. 2015, pp. 119–127.

[50] Michelle Y Wong and David Lie. 'IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware'. In: *23rd Network and Distributed System Security Symposium*. San Diego, California: The Internet Society, Feb. 2016, pp. 21–24.

[51] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. 'Efficiently, effectively detecting mobile app bugs with appdoctor'. In: *9th European Conference on Computer Systems*. Amsterdam, The Netherlands: ACM, Apr. 2014, pp. 1–15.

[52] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. 'PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps'. In: *12th Annual International Conference on Mobile Systems, Applications, and Services*. Bretton Woods, USA: ACM, June 2014, pp. 204–217.

[53] Franz-Xaver Geiger and Ivano Malavolta. 'Datasets of Android Applications: a Literature Review'. In: *arXiv preprint arXiv:1809.10069* (2018).

[54] Yajin Zhou and Xuxian Jiang. 'Dissecting android malware: Characterization and evolution'. In: *IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE, May 2012, pp. 95–109.

[55] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 'Are your training datasets yet relevant?' In: *7th International Symposium on Engineering Secure Software and Systems*. Milan, Italy: Springer, Mar. 2015, pp. 51–67.

[56] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 'Drebin: Effective and explainable detection of android malware in your pocket'. In: *21st Network and Distributed System Security Symposium*. San Diego, CA, USA: The Internet Society, Feb. 2014, pp. 23–26.

[57] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 'AndroZoo: Collecting Millions of Android Apps for the Research Community'. In: *13rd International Conference on Mining Software Repositories*. Austin, Texas: ACM, May 2016, pp. 468–471.

[58] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 'Deep Ground Truth Analysis of Current Android Malware1'. In: *14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Bonn, Germany: Springer, July 2017, pp. 252–276.

[59] Joydeep Mitra and Venkatesh-Prasad Ranganath. 'Ghera: A repository of android app vulnerability benchmarks'. In: *13th International Conference on Predictive Models and Data Analytics in Software Engineering*. Toronto, Canada: ACM, Nov. 2017, pp. 43–52.

[60] Yuru Shao, Jason Ott, Yunhan Jack Jia, Zhiyun Qian, and Z Morley Mao. 'The Misuse of Android Unix Domain Sockets and Security Implications'. In: *ACM SIGSAC Conference on Computer and Communications Security*. Vienna, Austria: ACM, Oct. 2016, pp. 80–91.

[61] Rohit Bhatia, Brendan Saltaformaggio, Seung Jei Yang, Aisha Ali-Gombe, Xiangyu Zhang, Dongyan Xu, and Golden G Richard III. '"Tipped Off by Your Memory Allocator": Device-Wide User Activity Sequencing from Android Memory Images'. In: (Feb. 2018).

[62] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 'Pscout: analyzing the android permission specification'. In: *ACM conference on Computer and communications security*. Raleigh, North Carolina, USA: ACM, Oct. 2012, pp. 217–228.

[63] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 'WHYPER: Towards Automating Risk Assessment of Mobile Applications.' In: *USENIX Security Symposium*. Washington, D.C, USA: USENIX, Aug. 2013, pp. 527–542.

[64] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. 'Droidmat: Android malware detection through manifest and api calls tracing'. In: *7th Asia Joint Conference onInformation Security*. Tokyo, Japan: IEEE, Aug. 2012, pp. 62–69.

[65] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. 'Slicing droids: program slicing for smali code'. In: *28th Annual ACM Symposium on Applied Computing*. Coimbra, Portugal: ACM, Mar. 2013, pp. 1844–1851.

[66] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 'Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques'. In: *23rd Network and Distributed System Security Symposium*. San Diego, USA: The Internet Society, Feb. 2016.

[67] Raja Vallee-Rai and Laurie J Hendren. 'Jimple: Simplifying Java bytecode for analyses and transformations'. In: (1998).

[68] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 'Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps'. In: *Acm Sigplan Notices* 49.6 (2014), pp. 259–269.

[69] Corina S Păsăreanu and Neha Rungta. 'Symbolic PathFinder: symbolic execution of Java bytecode'. In: *IEEE/ACM international conference on Automated software engineering*. Antwerp, Belgium: IEEE/ACM, Sept. 2010, pp. 179–180.

[70] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. 'Appintent: Analyzing sensitive data transmission in android for privacy leakage detection'. In: *ACM SIGSAC conference on Computer & communications security*. Berlin, Germany: ACM, Nov. 2013, pp. 1043–1054.

[71] Kristopher Micinski, Jonathan Fetter-Degges, Jinseong Jeon, Jeffrey S Foster, and Michael R Clarkson. 'Checking interaction-based declassification policies for android using symbolic execution'. In: *European Symposium on Research in Computer Security*. Vienna, Austria: Springer, Sept. 2015, pp. 520–538.

[72] Radoniaina Andriatsimandefitra and Valérie Viet Triem Tong. 'Detection and identification of Android malware based on information flow monitoring'. In: *2nd International Conference on Cyber Security and Cloud Computing*. New York, USA: IEEE, Jan. 2015, pp. 200–203.

[73] Radoniaina Andriatsimandefitra, Stéphane Geller, and Valérie Viet Triem Tong. 'Designing information flow policies for Android's operating system'. In: *IEEE International conference on communications*. Ottawa, ON, Canada: IEEE, June 2012, pp. 976–981.

[74] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 'TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones'. In: *11th USENIX conference on Operating systems design and implementation*. Vancouver, Canada: USENIX, Oct. 2014, pp. 393–407.

[75] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. 'Copperdroid: Automatic reconstruction of android malware behaviors.' In: *22nd Network and Distributed System Security Symposium*. San Diego, CA, USA: The Internet Society, Feb. 2015.

[76] Fabrice Bellard. 'QEMU, a fast and portable dynamic translator'. In: *USENIX Annual Technical Conference, FREENIX Track*. Berkeley, CA, USA: USENIX, Apr. 2005.

[77] Rowland Yu. 'Android packers: facing the challenges, building solutions'. In: *24th Virus Bulletin International Conference* (2014).

[78] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. 'Dexhunter: toward extracting hidden code from packed android applications'. In: *20th European Symposium on Research in Computer Security*. Vienna, Austria: Springer, Nov. 2015, pp. 293–311.

[79] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. 'Appspear: Bytecode decrypting and dex reassembling for packed android malware'. In: *18th International Symposium on Recent Advances in Intrusion Detection*. Kyoto, Japan: Springer, Dec. 2015, pp. 359–381.

[80] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. 'Adaptive unpacking of Android apps'. In: *39th International Conference on Software Engineering*. Buenos Aires, Argentina: IEEE, May 2017, pp. 358–369.

[81] Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin TS Chan. 'On tracking information flows through jni in android applications'. In: *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Atlanta, USA: IEEE, Sept. 2014, pp. 180–191.

[82] Hongliang Liang, Yudong Wang, Tianqi Yang, and Yue Yu. 'AppLance: A Lightweight Approach to Detect Privacy Leak for Packed Applications'. In: *23rd Nordic Conference on Secure IT Systems*. Oslo, Norway: Springer, Nov. 2018, pp. 54–70.

[83] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. 'Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART'. In: *26th USENIX Security Symposium*. Vancouver, Canada: USENIX, Aug. 2017, pp. 289–306.

[84] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. 'JN-SAF: Precise and Efficient NDK/JNI-aware Inter-language Static Analysis Framework for Security Vetting of Android Applications with Native Code'. In: *25th ACM SIGSAC Conference on Computer and Communications Security*. Toronto, Canada: ACM, Oct. 2018, pp. 1137–1150.

[85] Valerio Costamagna and Cong Zheng. 'ARTDroid: A Virtual-Method Hooking Framework on Android ART Runtime'. In: *1st International Workshop on Innovations in Mobile Privacy and Security co-located with the International Symposium on Engineering Secure Software and Systems*. London, UK: CEUR Workshop Proceedings, Apr. 2016, pp. 20–28.

[86] Nikolaos Totosis and Constantinos Patsakis. 'Android Hooking Revisited'. In: *IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress*. Athens, Greece: IEEE, Aug. 2018, pp. 552–559.

[87] Lok-Kwong Yan and Heng Yin. 'DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis'. In: *21th USENIX security symposium*. Bellevue, USA: USENIX, Aug. 2012, pp. 569–584.

[88] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp von Styp-Rekowsky, and Sebastian Weisgerber. 'Artist: The android runtime instrumentation and security toolkit'. In: *IEEE European Symposium on Security and Privacy*. Paris, France: IEE, Apr. 2017, pp. 481–495.

[89] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. 'SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers'. In: *IEEE Symposium on Security and Privacy*. San Jose, USA: IEEE, May 2015, pp. 659–673.

[90] Tim Strazzere and Jon Sawyer. 'Android hacker protection level 0'. In: *DEF CON 22* (Aug. 2014).

[91] Dongwoo Kim, Jin Kwak, and Jaecheol Ryou. 'Dwroiddump: Executable code extraction from android applications for malware analysis'. In: *International Journal of Distributed Sensor Networks* 11.9 (2015).

[92] Yeongung Park. 'We can still crack you! general unpacking method for android packer (no root)'. In: *Black Hat Asia* (Mar. 2015).

[93] Avi Bashan and Slava Makkaveev. 'Unboxing Android: Everything You Wanted To Know About Android Packers'. In: *DEF CON 25* (July 2017).

[94] Zhongqing Jiang, Anmin Zhou, Liang Liu, Peng Jia, Luping Liu, and Zheng Zuo. 'CrackDex: Universal and automatic DEX extraction method'. In: Shenzhen, China: IEEE, July 2017, pp. 53–60.

[95] Michelle Y Wong and David Lie. 'Tackling runtime-based obfuscation in Android with TIRO'. In: *27th USENIX Security Symposium*. Baltimore, USA: USENIX, Aug. 2018, pp. 1247–1262.

[96] Nicholas Nethercote and Julian Seward. 'Valgrind: a framework for heavyweight dynamic binary instrumentation'. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. San Diego, USA: ACM, June 2007, pp. 89–100.

[97] Judong Bao, Yongqiang He, and Weiping Wen. 'DroidPro: An AOTC-Based Bytecode-Hiding Scheme for Packing the Android Applications'. In: *IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ IEEE International Conference On Big Data Science And Engineering*. New York, USA: IEEE, Aug. 2018, pp. 624–632.

[98] *Android Compatibility Definition Document*. https://source.android.com/compatibility/cdd.html. 2020.

[99] Thomas Dullien, Ero Carrera, Soeren-Meyer Eppler, and Sebastian Porst. *Automated attacker correlation for malicious code*. Tech. rep. Mar. 2010.

[100] Halvar Flake. 'Structural comparison of executable objects'. In: *Detection of Intrusions and Malware & Vulnerability Assessment*. Dortmund, Germany: Gesellschaft für Informatik, July 2004, pp. 161–173.

[101] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 'Scalable detection of semantic clones'. In: *30th International Conference on Software Engineering*. Leipzig, Germany: IEEE, May 2008, pp. 321–330.

[102] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. 'Leveraging semantic signatures for bug search in binary programs'. In: *30th Annual Computer Security Applications Conference*. New Orleans, Louisiana, USA, Dec. 2014, pp. 406–415.

[103] Jonathan Crussell, Clint Gibler, and Hao Chen. 'Andarwin: Scalable detection of semantically similar android applications'. In: *European Symposium on Research in Computer Security*. Springer, 2013, pp. 182–199.

[104] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 'Software clone detection: A systematic review'. In: *Information and Software Technology* 55.7 (July 2013), pp. 1165–1199.

[105] Chanchal K Roy, James R Cordy, and Rainer Koschke. 'Comparison and evaluation of code clone detection techniques and tools: A qualitative approach'. In: *Science of computer programming* 74.7 (May 2009), pp. 470–495.

[106] Mingshen Sun, Tao Wei, and John Lui. 'TaintART: A practical multi-level information-flow tracking system for Android RunTime'. In: *23rd ACM SIGSAC Conference on Computer and Communications Security*. Vienna, Austria: ACM, Oct. 2016, pp. 331–342.

[107] Wei You, Bin Liang, Wenchang Shi, Peng Wang, and Xiangyu Zhang. 'Taintman: An art-compatible dynamic taint analysis framework on unmodified and non-rooted android devices'. In: *IEEE Transactions on Dependable and Secure Computing* 17.1 (2017).

[108] Lei Xue, Chenxiong Qian, Hao Zhou, Xiapu Luo, Yajin Zhou, Yuru Shao, and Alvin TS Chan. 'NDroid: Toward tracking information flows across multiple Android contexts'. In: *IEEE Transactions on Information Forensics and Security* 14.3 (2018).

[109] Chris Lattner. 'LLVM and Clang: Next generation compiler technology'. In: *The BSD conference*. Ottawa, Canada, May 2008.

[110] *JNI tips*. `https://developer.android.com/training/articles/perf-jni#primitive-arrays`. Android. 2019.

[111] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 'Obfuscator-LLVM – Software Protection for the Masses'. In: *1st International Workshop on Software Protection*. Florence, Italy: IEEE, May 2015, pp. 3–9.

[112] Yan Shoshitaishvili et al. 'SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis'. In: *IEEE Symposium on Security and Privacy*. San Jose, USA: IEEE, May 2016, pp. 138–157.

[113] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 'Code obfuscation against symbolic execution attacks'. In: *32nd Annual Conference on Computer Security Applications*. Los Angeles, CA, USA: ACM, Dec. 2016, pp. 189–200.

[114] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 'Dynodroid: An input generation system for android apps'. In: *9th Joint Meeting on Foundations of Software Engineering*. Saint Petersburg, Russia: ACM, Aug. 2013, pp. 224–234.

[115] Ali Razeen, Alvin R. Lebeck, David H. Liu, Alexander Meijer, Valentin Pistol, and Landon P. Cox. 'SandTrap: Tracking Information Flows On Demand with Parallel Permissions'. In: *16th Annual International Conference on Mobile Systems, Applications, and Services*. Munich, Germany: ACM, June 2018, pp. 230–242.

[116] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 'A survey of symbolic execution techniques'. In: *ACM Computing Surveys* 51.3 (July 2018).

[117] Luca Cardelli and Peter Wegner. 'On Understanding Types, Data Abstraction, and Polymorphism'. In: *ACM Computing Surveys* 17.4 (Dec. 1985).

[118] *ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile*. ARM. Dec. 2017.

UNIVERSITÉ DE RENNES 1

Titre : Défis pour les Applications Android Natives : Obfuscation et Vulnérabilités

Mot clés : sécurité, Android, natif

Résumé :

Android est le système d'exploitation le plus utilisé et donc, assurer la sécurité des applications est essentiel. Sécuriser une application consiste à empêcher les attaquants potentiels de corrompre le comportement attendu de l'application. En particulier, l'attaquant peut s'appuyer sur des vulnérabilités laissées dans le code par le développeur, mais aussi voler la propriété intellectuelle d'une application existante. Pour ralentir le travail de l'attaquant qui essaie de *reverser* la logique applicative, le développeur est incité à chercher les vulnérabilités potentielles et à introduire des contremesures dans le code. Parmi les contremesures possibles, l'obfuscation de code est une technique qui cache l'intention réelle du développeur en faisant en sorte de rendre le code non disponible à l'adversaire qui utilise des outils de *reverser*. Avec l'augmentation des applications soit malveillantes, soit manipulant des informations sensibles, obfusquer le code et chercher ses vulnérabilités devient essentiel.

Cette thèse présente l'impact du code natif sur, à la fois le *reversing* et la recherche de vulnérabilités, appliqué à des applications Android. Premièrement, en listant les interférences possibles entre l'assembleur et le bytecode, nous mettons en évidence des nouvelles techniques d'obfuscation et vulnérabilités logicielles. Ensuite, nous proposons de nouvelles techniques d'analyse combinant des blocs d'analyse statiques et dynamiques, tels que la propagation de teintes ou la surveillance du système, afin d'observer le comportement du code qui a été obfusqué ou de révéler de nouvelles vulnérabilités. Ces deux objectifs nous ont menés à développer deux nouveaux outils. Le premier cible une vulnérabilité spécifique due à l'interaction du natif et des données Java. Le second extrait le comportement d'une application au niveau objet, que l'application contienne du code natif d'obfuscation ou non. Enfin, nous avons implémenté ces nouvelles méthodes et les avons évaluées expérimentalement. En particulier, nous avons trouvé automatiquement une vulnérabilité dans la librairie SSL d'Android et nous avons analysé plusieurs firmware Android pour détecter l'usage d'une classe spécifique d'obfuscation.

Title: Challenges of Native Android Applications: Obfuscation and Vulnerabilities

Keywords: security, Android, native

Abstract:

Android is the most used operating system and thus, ensuring security for its applications is an essential task. Securing an application consists in preventing potential attackers to divert the normal behavior of the targeted application. In particular, the attacker may take advantage of vulnerabilities left by the developer in the code and also tries to steal intellectual property of existing applications. To slow down the work of attackers who try to reverse the logic of a released application, developers are incited to track potential vulnerabilities and to introduce countermeasures in the code. Among the possible countermeasures, the obfuscation of the code is a technique that hides the real intent of the developer by making the code unavailable to an adversary using a reverse engineering tool. With the growing amount of malware and applications carrying sensitive information, obfuscating the code and searching vulnerabilities becomes essential.

This thesis presents the impact of native code on both reverse-engineering and vulnerability finding applied to Android applications. First, by listing the possible interferences between assembly and bytecode, we highlight new obfuscation techniques and software vulnerabilities. Then, we propose new analysis techniques combining static and dynamic analysis blocks, such as taint tracking or system monitoring, to observe the code behaviors that have been obfuscated or to reveal new vulnerabilities. These two objectives have led us to develop two new tools. The first one spots a specific vulnerability that comes from inconsistently mixing native and Java data. The second one extracts the object level behavior of an application, regardless of whether this application contains native code, embedded for obfuscation purposes. Finally, we implemented these new methods and conducted experimental evaluations. In particular, we automatically found a vulnerability in the Android SSL library and we analyzed several Android firmware to detect usage of a specific class of obfuscation.